© 2012 by David M. Kunzman. All rights reserved.

RUNTIME SUPPORT FOR OBJECT-BASED MESSAGE-DRIVEN PARALLEL APPLICATIONS ON HETEROGENEOUS CLUSTERS

BY

DAVID M. KUNZMAN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kale, Chair
Professor William Gropp
Associate Professor Craig Zilles
Professor David A. Bader, Georgia Institute of Technology

Abstract

In the last several years, there has been an increased interest in using various accelerator technologies in the realm of high performance computing (HPC). Some of these technologies include the Cell processor (Cell), many integrated cores (MIC), the single chip cloud (SCC), field programmable gate arrays (FPGAs), and graphics processing units (GPUs). Considerable effort has been put forth in harnessing the computing power of these technologies for more general purpose programming. However, making use of these technologies is typically considered hard for various reasons, including their parallel nature and the architecture-specific details involved in programming them.

In this work, we would like to explore (1) how the various pieces of accelerator hardware can be abstracted within a programming model and (2) how an underlying runtime system can assist programmers in making use of accelerators. In other words, how can we make accelerators more accessible to programmers and allow them to focus on the problem at hand, rather than focusing on the hardware itself? What role can/should runtime systems play in making accelerator technologies more accessible to programmers?

[[TODO: Dedication goes here]]

Acknowledgments

TODO : Acknowledgments go here

Table of Contents

List of Tables . . . . **viii**

List of Figures . . . . **ix**

List of Abbreviations . . . . **xii**

Chapter 1 Introduction **1**
1.1 Send in the Cores 1
1.2 Heterogeneity 1
1.2.1 Advantages of Heterogeneous Systems 3
1.2.2 Challenges of Heterogeneous Systems 4
1.3 The Scope of This Work 5
1.3.1 Considering the Cell Processor 7
1.3.2 Considering GPGPUs 8
1.4 Contributions of this Work 9

Chapter 2 Background **10**
2.1 The Cell Broadband Engine Architecture (CBEA or Cell) 10
2.1.1 Architecture 11
2.1.2 The Offload API 12
2.2 CUDA-Based GPGPUs 13
2.2.1 Architecture 14
2.2.2 GPU Manager 17
2.3 Charm++ Basics 18
2.3.1 Interface Files 20
2.3.2 Projections 21

Chapter 3 Approach **23**
3.1 Building Upon Charm++ 23
3.1.1 Advantages of Charm++ 25
3.1.2 Disadvantages of Charm++ 27
3.1.3 Our Work Withing the Larger Charm++ Context 28
3.2 Programming Model Extensions 31
3.2.1 Accelerated Entry Methods (AEMs) 31
3.2.2 SIMD Instruction Abstraction (SIMDIA) 41
3.3 Runtime System Modifications 44
3.3.1 Accelerator Manager 44
3.3.2 Architecture-Specific Code Generation 49

Chapter 4 Static Load Balancing . . . 57
4.1 Description of Molecular Dynamics Code . . . 58
4.2 Heterogeneous Cluster Description . . . 59
4.3 Results . . . 61
4.3.1 Establishing a Performance Baseline . . . 62
4.3.2 Homogeneous Scaling . . . 65
4.3.3 Heterogeneous Scaling . . . 66
4.4 Static Load Balancing Summary . . . 68

Chapter 5 Dynamic Load Balancing . . . 70
5.1 Accelerator Load Balancing Strategies and the Accelerator Manager . . . 71
5.1.1 Modeling Workload Balance . . . 72
5.1.2 Base Class for Accelerator Load Balancing Strategies . . . 75
5.1.3 List of Accelerator Load Balancing Strategies . . . 77
5.2 Across Host Core Load Balancing *(AccelEvenLB)* . . . 93
5.3 Description of Hardware . . . 95
5.3.1 Forge (GPGPU Cluster) . . . 96
5.4 Applications . . . 96
5.4.1 5-Point Weighted Stencil . . . 96
5.4.2 Molecular Dynamics . . . 106
5.5 Adapting to Interference . . . 127
5.5.1 Overlapping Execution . . . 128
5.5.2 Partial Interference . . . 131
5.6 Effects of Host Splitting . . . 135
5.7 Revisiting the Cell Processor . . . 137
5.7.1 Sampling *(centralized)* . . . 138
5.7.2 AdjustBusy . . . 143
5.8 Summary of Dynamic Load Balancing . . . 146

Chapter 6 Data Management . . . 147
6.1 Automatic Modification of Data Crossing Processor Boundaries . . . 147
6.1.1 Programmer Impact . . . 148
6.1.2 Method . . . 149
6.2 Data Within GPGPU Batch Sets . . . 150
6.2.1 Data Alignment of Streamed Parameters on GPGPUs . . . 151
6.2.2 Shared Buffers . . . 153
6.3 Persistent Data . . . 155
6.3.1 Description . . . 156
6.3.2 Application Code Example (5-Point Stencil Modification) . . . 157
6.3.3 Results . . . 158
6.4 Integration with Fault Tolerance Support . . . 167
6.5 Summary of Data Management . . . 170

Chapter 7 Related Work . . . 171
7.1 *Native* Programming Models . . . 171
7.2 Extensions to OpenMP . . . 172
7.3 Extensions to MPI . . . 174
7.4 Other Approaches . . . 175
7.5 Previous Work Within Charm++ . . . 176

Chapter 8 Conclusions . . . 178
8.1 Summary of Content . . . 178
8.2 Concluding Remarks . . . 180
8.3 Future Work . . . 182
8.4 The Future of Heterogeneity . . . 183

References 186

List of Tables

1.1 Number of supercomputers including accelerators on the top 10 systems on the Green500 list by year [21]. 4

4.1 Differences between the core types in the cluster. 61
4.2 Performance of the simple MD code scaled to multiple Cell-based compute nodes. 64

5.1 Scaling performance of the simple MD application from one to six nodes on Forge with GPGPU sharing. 122

8.1 The top 10 systems on the Green500 list by year [21]. 183

List of Figures

2.1 Architecture overview of the Cell Processor. 11
2.2 Architecture overview of a CUDA-based GPGPU. 15
2.3 The anatomy of a Projections timeline. 22

3.1 Overview of the runtime system structure, along with programming abstractions. 29
3.2 Structure of an accelerated entry method (AEM). 31
3.3 The basic code structure of an accelerated entry method (AEM) within the context of a Charm++ interface file. 33
3.4 Effects on the DAG resulting from the usage of accelerated entry methods. 35
3.5 Example code for a splittable accelerated entry method 39
3.6 A comparison of a scalar addition to a SIMD addition. 42
3.7 Example code illustrating the use of SIMDIA 43
3.8 The two tiers of load balancing for heterogeneous systems that include accelerator devices. . . . 47
3.9 Overview of the generated code created by charmxi. 50

4.1 The interaction of objects, or flow of data between those objects, in a single timestep of the simple MD code. 58
4.2 Integration code from the simple molecular dynamics code. 60
4.3 Performance of the MD application as it is scaled on QS20 Cells, PS3 Cells, and x86 cores. . 64
4.4 Performance of the simple MD application scaled to use all the nodes within a heterogeneous cluster, making use of x86-based and Cell-based processors. 67

5.1 Balancing a workload across the host and accelerator. 72
5.2 Balancing a workload across the host and accelerator. 75
5.3 An example breakdown of one timestep in an arbitrary application 80
5.4 An example illustrating the construction of the workload balance graph by the Sampling accelerator load balancing strategy. 84
5.5 5-Point weighted stencil pattern used in the 5-point stencil application. 98
5.6 Interaction of tiles in the 5-point stencil application. 98
5.7 Stencil code that acts upon a single tile, written as an accelerated entry method. 100
5.8 The 5-Point stencil application executing on a single host core and accelerator device pairing on Forge using the Step accelerator strategy. 101
5.9 The 5-Point stencil application executing on a single host core and GPGPU pair on Forge using the AdjustBusy strategy. 102
5.10 The 5-Point stencil application executing on a single host core and GPGPU pair on Forge using the sampling accelerator strategy. 103
5.11 The 5-Point stencil application executing on a single host core and GPGPU pair on Forge using the Greedy strategy, with and without the Crawler strategy. 104
5.12 The 5-Point stencil application executing on a single host core and GPGPU pair on Forge using the various accelerator strategies available within the Charm++ runtime system. Note that the vertical axis does not start at zero. 106

5.13 The molecular dynamics application scaled to a single node of Forge and using the Step accelerator strategy. . . . 108
5.14 The molecular dynamics application scaled to a single node of Forge and using the Step accelerator strategy. . . . 110
5.15 The molecular dynamics application scaled to a single node of Forge and using the AdjustBusy accelerator strategy. . . . 111
5.16 Zoomed in views of the data in figure 5.15, along with corresponding percent device values. . 112
5.17 The final percent device values for the various configurations used in figure 5.15. . . . 114
5.18 The molecular dynamics application scaled to a single node of Forge and using the AdjustBusy accelerator strategy along with the AccelEvenLB load balancing strategy. . . . 115
5.19 The molecular dynamics application scaled to a single node of Forge and using the BaseLoad profiler, AdjustBusy accelerator strategy, and the AccelEvenLB load balancing strategy. . . . 115
5.20 The molecular dynamics application scaled to a single node of Forge and using the Sampling-Centralized accelerator strategy. . . . 116
5.21 Illustration explaining the "humps" in figure 5.20. . . . 117
5.22 The molecular dynamics application on a single host core and a single GPGPU using the Sampling strategy. . . . 118
5.23 The molecular dynamics application scaled to a single node of Forge and using the Sampling-Centralized accelerator strategy along with the AccelEvenLB load balancing strategy. . . . 119
5.24 The MD application using the Greedy strategy, with and without the Crawler strategy. . . . 120
5.25 The MD application executing for 30720 timesteps using the Greedy strategy, with and without the Crawler strategy. . . . 121
5.26 The MD application executing for 30720 timesteps using a variety of accelerator load balancing strategies. . . . 122
5.27 Timeline of the molecular dynamics application executing on six nodes of Forge. . . . 124
5.28 The object distribution and performance of the 6 node execution, with and without GPGPU sharing. . . . 126
5.29 Two instances of the MD application with the same configuration overlapping in time. . . . 129
5.30 An instance of the MD application executing with interference. . . . 132
5.31 An instance of the MD application executing with interference. . . . 134
5.32 Performance of the MD code using various progress call frequencies, with and without splitting accelerated entry methods on the host core. . . . 135
5.33 The MD application executing on QS20 blades in the cell cluster, using the Sampling (centralized) accelerator load balancing strategy. . . . 138
5.34 The MD application executing on 1 QS20 blade in the cell cluster, using the Sampling (centralized) accelerator load balancing strategy. . . . 139
5.35 The MD application executing on 4 QS20 blades and 2 PS3s in the cell cluster, using the Sampling (centralized) accelerator load balancing strategy. . . . 141
5.36 The balance of objects as set by AccelEvenLB for the MD application executing on 4 QS20 blades and 2 PS3s in the cell cluster, using Sampling (centralized). . . . 141
5.37 The MD application executing on QS20 blades in the cell cluster, using the AdjustBusy accelerator load balancing strategy. . . . 144

6.1 Modifications to the send/receive process in a heterogeneous builds of the Charm++ runtime system. . . . 149
6.2 Example of using shared local parameters (PairCompute from MD code). . . . 154
6.3 The data patterns of the 5-point stencil application. . . . 157
6.4 The calculation accelerated entry method in the 5-point stencil application modified to use persistent buffers. . . . 159
6.5 The 5-point stencil application executing on a single host core and GPGPU pair on Forge, with and without the use of the persistent data. . . . 160
6.6 The 5-point stencil application using persistent buffers and dynamic assignment. . . . 161
6.7 The 5-point stencil application using persistent buffers and static assignment. . . . 163

6.8 The 5-point stencil application using persistent buffers, static assignment, and the Step strategy. 164
6.9 The 5-point stencil application using persistent buffers, static assignment, and the Sampling strategy. 165
6.10 The 5-point stencil application using persistent buffers, static assignment, and the Sampling strategy. 166
6.11 Performance of the MD application as it recovers from a host core fault. 167

8.1 The top 50 systems from the June 2011 and November 2011 Green500 lists. 184

List of Abbreviations

accelerator core	A core, other than a host core, available to the application
accelerator device	A set of one or more accelerator cores used collectively
AccelManager	Accelerator Manager
AEM	Accelerated Entry Method
Cell	The Cell Processor
DMA	Direct Memory Access
EM	Entry Method
flop	Floating Point Operation
flop/s	Floating Point Operations Per Second
GPGPU	General Purpose Graphics Processing Unit
GPU	Graphics Processing Unit
host core	A core that is executing a complete version of the runtime system
HPC	High Performance Computing
Hz	Hertz
ISA	Instruction Set Architecture
MD	Molecular Dynamics
MIC	Many Integrated Core
OS	Operating System
pairing	One host core and one accelerator device
PCI-E	Peripheral Component Interconnect Express (or PCI Express)
PE	Processing Element
PPE	Power Processing Element
PS3	Playstation 3
PUP Routines	Pack-UnPack Routines
SDK	Standard Development Kit

SIMD	Single Instruction Multiple Data
SM	Streaming Multiprocessor (in the context of CUDA-based GPGPUs)
SP	Streaming Processor
SPE	Synergistic Processing Element
SSE	Streaming SIMD Extensions

Chapter 1

Introduction

1.1 Send in the Cores

In the not so distant past, each successive generation of a processor saw an increase in clock frequency. This, along with other architectural enhancements, increased the serial performance of single threaded applications running on commodity processors year after year. However, this trend came to a halt as the associated power and cooling requirements became unreasonable. As power and cooling costs placed ceilings on processor frequencies, the number of transistors that can be placed on a die has continued to increase [20]. Historically, the additional transistors available with each generation of processor were typically used to increase the serial performance of individual cores. Over time, this general approach began seeing diminishing returns, with increasingly complicated mechanisms requiring more transistors while providing relatively smaller benefits. Within the last decade, the trend of focusing on the serial performance of a single thread has, to some degree, been reduced. Instead, processor designers have begun using these extra transistors for creating more cores on a single die. This is not to say that single thread performance is no longer important. Rather, there is a recognition that parallelism must be embraced if hardware designs are to continue pushing the performance of processors to higher and higher levels in the face of current power limitations. As a result of this shift, most commodity processors now include multiple cores, with research designs containing (or considering) tens, hundrends, and even thousands of cores [44, 43, 50, 25, 90, 15]. Thus began the multicore era.

1.2 Heterogeneity

Moving beyond just the simple duplication of cores to create multicore processors, some specialized hardware designs went a step further by introducing heterogeneity. Specialized hardware designs typically include even more cores than commodity designs, with the individual cores typically being fairly simple in comparison (e.g. in-order pipelines, simple or no branch prediction, etc.). In addition to simply increasing the number

of cores, other architecture specific features are usually included within specialized designs, such as direct memory access (DMA) controllers and specialized memories.

Examples of such specialized hardware includes graphics processing units (GPUs) [94], the Cell processor (Cell) [84], many integrated core (MIC) (based on the previous Larrabee design [90]), field programmable gate arrays (FPGAs) [73], single chip cloud (SCC) [26, 22], Intel Sandy Bridge [95], AMD Fusion [31], and others. Throughout this document, we will collectively refer to the specialized components of these processors and/or devices as "accelerators." Naturally, the exploration of these specialized architectures, along with their success in accelerating certain types calculations, has further lead to the exploration of heterogeneous clusters. Several supercomputers built within the last several years have utilized accelerators to augment the computational power of the cluster's host cores, including Roadrunner at Los Alamos National Lab (LANL) [59, 7], Forge at the National Center for Supercomputing Applications (NCSA) [78], HokieSpeed at Virginia Polytechnic Institute [70], Cray XK6 [30], Tianhe-1A at the National Supercomputing Center in Tianjin [47], and the list goes on. In fact, several heterogeneous systems have begun showing up on both the Top500 [76] and the Green500 [21] cluster rankings. In June of 2008, the Roadrunner cluster, which includes Cell processors, was the first supercomputer to achieve a sustained flop rate of greater than one petaflop/s for the LINPACK benchmark [57, 76]. The latest Top500 list (November 2011, at the time of this writing) includes 39 heterogeneous supercomputing clusters, which is more than double the number a year prior (17 clusters in November of 2010) and over five times the number two years prior (7 clusters in November 2009). Of the top 50 systems on the current Top500 list, 8 systems include accelerators. Of the top 50 systems on the current Green500 list, 29 systems include accelerators.

One way in which heterogeneity has been explored is through the inclusion of specialized hardware in an otherwise standard computing node. The most well-known example of this is the use of graphics processing units (GPUs). GPUs were originally introduced as specialized hardware designed specifically to accelerate video and graphics processing. However, as the flop rates of these specialized processors began increasing, programmers writing computationally intensive applications began searching for ways to exploit these specialized processors [88, 71, 18]. Currently, this trend has become more mainstream through the use of programming languages/models designed to target such specialized hardware, such as CUDA [80], OpenCL [40], and OpenACC [82]. Designers of specialized hardware have embraced general purpose computing on specialized hardware, in so far as some hardware mechanisms are being included in the hardware designs that wouldn't otherwise be required by the computations that the specialized hardware was originally intended for, such as IEEE floating-point support within general purpose GPUs (GPGPUs), which is not generally

required for graphics.[1]

1.2.1 Advantages of Heterogeneous Systems

Generally speaking, there are two main advantages to using accelerators, and thus heterogeneous systems: flop rates and power usage. First, the peak theoretical flop rates of accelerators are typically higher than the flop rates of standard hardware cores. For example, all of the cores in the Cell processor (which will be described later in section 2.1), regardless of core's type, actually have the same peak flop rate. However, because the specialized cores are smaller, there are more of them per die, and thus provide more flops in aggregate. As such, depending on how one decides to do the flop rate comparison, one could correctly come to either conclusion. For our purposes, we say that the accelerator cores in aggregate provide a higher peak flop rate. The same holds true of CUDA-based GPUs, which have more functional units per streaming multiprocessor (SM), but slower clock speeds. As a result, SMs are more or less comparable to commodity host cores in terms of their peak flop rates. A modern Intel Sandy Bridge host core can perform 16 flops per cycle, for a peak flop rate of 48 Gflop/s with a clock speed of 3.0 GHz [24, 69]. A Fermi SM can execute 64 flops per cycle, for a peak flop rate of 96 Gflop/s with a clock speed of 1.5 GHz (a factor of 2 greater than a host core). However, also note that GPGPU cores place some restrictions on the code making use of them, such as requiring the use of a SIMD execution model, and are not well-suited for all types of calculations. One should be very cautious about making a direct comparison of these fairly different archtectures as it is not a straight forward comparison, especially if one also considers harder to quantify factors such as programmability and portability of the resulting code. That said, once again, the simple GPGPU cores are smaller and designed to be used in large groups, giving them a higher peak flop rate in aggregate. Currently, host processors typically include 4-6 cores, while GPGPUs include 14-16 SMs, creating peak flop rate for GPGPUs that is approximately a factor of 8 greater than host processors (the actual clock speeds and core counts vary between specific processor models).

Second, the flop per Watt ratio is typically lower for accelerators and/or heterogeneous systems when compared to standard commodity cores. At the high-end, we can see this reflected within the top 10 systems listed on the Green500 list over the past several years, as shown in table 1.1. Note that ties are allowed within the Green500 list, allowing more than 10 systems to occupy the top 10 ranks. The Green500 list is released twice annually, once in November (N) and once in June (J). The column headings for table 1.1 indicate the date of the list by month (letter) and year (20*xx*, where *xx* represents the number). Table 1.1

[1] Please note that some consider the term GPGPU to be a bit of a misnomer, implying that GPGPUs are well-suited for any type of calculation. This is not the case, as GPGPU designs tend to be well-suited to particular types or classes of calculations. Throughout this document, we will use the term GPGPU, with the understanding that GPGPUs may not be well-suited for all types of computation.

	N11	J11	N10	J10	N09	J09	N08	J08
Cell		3	3	6	6	4	7	3
GPGPU (nVidia)	4	2	4	2				
GPGPU (ATI)	1	2	1		1			
GRAPE-DR			1		1	1		
Accelerator	5	7	9	8	8	5	7	3
Total Systems	10	10	11	10	14	13	14	11

Table 1.1: Number of supercomputers including accelerators on the top 10 systems on the Green500 list by year [21].

shows that, for the last four years, clusters that include accelerator technologies make up at least half the top 10 systems each year, with the exception of June 2009 and June 2008. We also point out that of the clusters that do include accelerators, the great majority of those clusters either include Cell processors or GPGPUs.

In smaller systems, the inclusion of accelerators into otherwise standard multicore systems may increase the flop rate to power ratio, allowing applications to perform more work per unit energy. For example, the inclusion of one or more GPGPUs within a workstation can significantly increase the peak flop rate provided by the workstation (as previously pointed out by the peak flop rate comparisons above). The power of a modern Sandy Bridge processor is on the order of 95W ("high-end desktop" version) [95]. As a point of comparison, the power usage of a Fermi-based M2070 Computing Module from NVIDIA is approximately 225W [29]. Note that the power increase is less than the peak flop rate increase. However, again, a direct comparison is not truly fair, since the characteristics of these processors are fairly different and since the power usage of specific models vary. Additionally, there is the question of programmability and how far programmers are required to (or willing to) go in order to achieve a significant portion of the peak flop rates provided by any architecture (i.e. the portion of the peak flop rate that is commonly achieved using standard practices, rather than what many refer to as *heroic efforts*). We will return to this topic in section 8.4.

1.2.2 Challenges of Heterogeneous Systems

In addition to the advantages presented by accelerators and heterogeneous systems, there are also several challenges that come along with using such systems. The clearest challenge of heterogeneous systems is realized by any programmer as they begin to write application code for such systems. Put simply, writing application for a heterogeneous system is often noticeably more difficult when compared to writing application code for a homogeneous system. There are a variety of reasons for this, some of which are more or less true depending on the particular system in question. In the case of the Cell processor, data move-

ment between the general cores and the specialized cores must be done explicitly within the application code itself. Further, the accelerator cores within the Cell processor have specialized scratchpad memories which the programmer must also explicitly manage as data is streamed through the accelerator cores. In the case of GPGPUs, data must often be aggregated explicitly by application code, in addition to explicitly being transfered to and from the accelerator device by the application code. This is required so that the throughput-oriented GPGPUs have a sufficiently large collection of data parallel tasks to execute, as required by their execution models (more on this below). The explicit manipulation of application data required by accelerators is less than ideal for two reasons. First, it decreases the portability of application code by making the code more architecturally specific. Further, the code used to accomplish this explicit data movement commonly needs to be interleaved throughout the application code. This interleaving of architecture specific code throughout the application code not only makes the code less portable by its mere presence, but also complicates any porting effort by making the architecture specific code harder to isolate from the application code. Note that interleaving is as much a requirement for performance reasons, as it is a requirement of the programming models themselves.

Another challenge of accelerators is that accelerators also tend to have a particular execution model tied to the design of the hardware. The Cell processor is designed to be well-suited for stream processing, while GPGPUs are designed to be well-suited for SIMD processing. As a result, it is often the case that the structure of a given application must be manipulated by the programmer manually, at least to some degree, as to reflect the execution model of the given accelerator that the programmer is targeting. The warping of the application model to a more specific hardware model tends further complicate the process of porting an application to another platform for reasons similar to those discussed in the previous paragraph.

1.3 The Scope of This Work

Processor and system architects find themselves with an interesting balancing act on their hands. Having more *simpler* cores (i.e. more functional units and thus shifting focus towards overall throughput, rather than solely focusing on serial performance) can make more efficient use of transistor counts and power (i.e. higher Gflop/s and Gflop/s per Watt rates via increasing the overall throughput of the processor, but not necessarily decreasing the latency of any given calculation). On the other hand, serial performance is also quite important for many applications and typically increases with *heavier* cores that devote more transistors to speculation, out-of-order processing, and so on. Specialized hardware can perform certain types of calculations with high levels of performance, but general cores are still required to handle the wide

variety of computing tasks that might be encountered. This is particularly true of applications that have many branches, indirect memory accesses, and/or irregular data structures. It's not hard to begin seeing how a mixture of core types may be advantageous for some applications where the overall workload will vary. However, as the variety of cores increases in a given system, the burden on the programmer also grows by complicating the process of writing codes that make use of all the different cores. Examples of complexities that typically arise include architecture differences between the cores, execution model differences, load balancing issues, explicit data movement between the various cores, and the inclusion of architecture specific code throughout the application code (i.e. portability).

Despite their difficulties, with their higher performance to power ratios, it's not unreasonable to think that heterogeneous systems will become more commonplace. As this occurs, we would like to retain ease of programming and portability. If possible, we would also like to have a runtime system handle many aspects of a parallel heterogeneous application for the programmer, leaving the programmer to focus on the problem they wish to solve, rather than the more focused task of implementing a particular algorithm on a particular hardware configuration. In other words, it would be advantageous if we could use a single programming model to program all of the cores, relieving the programmer of the burden of learning multiple programming models, interfacing those models, learning specific details of the hardware, and so on. While we do not believe that all of the difficulties of heterogeneous systems can be masked from programmers, *it is the goal of this work to understand some of the ways in which the programming model and runtime system can at least partially ease such burdens on programmers.* At the same time, knowing that our abstractions will not solve all of the issues that programmers face when programming heterogeneous systems, we do not want our approach to get in the way of programmers accomplishing what they want to accomplish, even if that means stepping outside the bounds of our abstractions from time to time. For example, as part of our approach we include a SIMD abstraction called SIMDIA (see section 3.2.2) that programmers can use to help make *SIMDized* code in a more portable manner. However, the functionality of SIMDIA is continually growing and may not be appropriate for all situations (yet or ever). As such, we provide a mechanism for programmers to easily check what supported platform their code is being compiled for and include architecture specific code if required.

Since there is a significant amount of knowledge and work that goes into adding support for a particular type of accelerator into our abstraction, we must necessarily restrict ourselves to only a small subset of the available accelerator technologies in existence today. In particular, we chose to include support for the Cell processor and CUDA-based GPGPUs. The remainder of this document will focus on applying our approach to these accelerators. It should be noted that we have already done some initial exploration for

supporting many integrated core (MIC) technologies from Intel within our approach, but current restrictions surrounding this technology stop us from discussing it as part of this document. As such, support for MIC is still considered to be future work at this point in time.

There are two reasons for choosing to support the Cell processor and GPGPUs within our approach. First, both of these accelerator technologies have been used in production clusters and commercial hardware systems, and thus it is fair to say that both of these accelerator technologies have managed to reach mainstream usage at some point in the recent past (last 6 years). Other accelerator technologies, such as FPGAs, still have relatively limited scopes of application in their usage outside of research systems. Note that we are not trying to say that research systems do not have impact, but rather we just want to point out the popularity of the Cell processor and GPGPUs for a variety of applications relative to other techologies. Further, as previously pointed on in section 1.2.1, both Cell processors and GPGPUs have appeared as components in some of the top supercomputers in the world according to both the Top500 and Green500 lists.

A second reason to choose both the Cell processor and CUDA-based GPGPUs as accelerator technologies to support is because they are quite different from one another, as well as being fairly different from commodity host cores. As such, demonstrating that the same programming model extensions and runtime system modifications can be applied to both types of accelerator technologies demonstrates a certain amount of flexibility and generality in our approach. In particular, host cores tend to make use of several (1-32) relatively long-lived threads (i.e. it is common for a thread to exist throughout the entire lifetime of an application's execution when using pthreads [79] or similar models). The Cell processor streams data through a set of on-die, specialized cores with limited amounts of working memory. GPGPUs make use of thousands of very light-weight, short-lived threads in a highly data parallel manner (i.e. a SIMD execution model).

1.3.1 Considering the Cell Processor

The Cell processor has been incorporated into several clusters, including the Condor, Roadrunner, QPACE, Cerrillos, and MariCel clusters. Playstation 3s (PS3s), which also contain Cell processors, have also been used within smaller clusters, since PS3s are less expensive when compared to typical cluster nodes. Despite these successes, the Cell has seen limited adoption relative to other accelerator technologies, most notably GPGPUs. However, we still consider the Cell to be relevant in the study of heterogeneous systems for a few reasons. First, if history is to be a teacher, architectures that more closely resemble commodity hardware tend to be preferred to specialized hardware as long as performance is comparable. Commodity hardware is well understood and requires no architecture specific code to be included within application code. Second,

of the popular contemporary accelerator architectures, Cell is unique in that it has scratchpad memories (referred to as “local stores”). That is, the programmer has complete control over (and responsibility for) the memory used by the Synergistic Processing Elements (SPEs), giving the programmer a degree of control over the data’s location and movement that is not seen in other accelerator architectures (see section 2.1). Third, Cell represents a case where the host cores are similar to the accelerator cores (i.e. the code execution and performance characteristics of various core types can be reasoned about in a similar manner, though there are important differences). This is unlike GPGPUs, since GPUs are programmed using a different, more-restrictive execution model (i.e. SIMD). The Cell is also unlike GPGPUs in that all the Cell’s cores, regardless of type, are on the same die and communicate through an on-die network. Though, recently, we are beginning to see graphics processing units being incorporated on the same die as host cores, including recent processor designs from Intel [96] and AMD [17], removing the need to transfer data off-die and/or requiring separate address spaces be used by the various cores. With these considerations in mind, we believe that exploring various aspects of heterogeneous systems using Cell processors remains an interesting and relevant area of exploration, despite the Cell’s recent decline in popularity.

Background information related to the Cell processor will be discussed in section 2.1. The initial application of our approach to the Cell processor using static load balancing will be discussed in chapter 4. Throughout the remainder of this document, the term “accelerator device,” when used in reference to Cell processors, will refer to the set of all physical SPE cores available to application code for the particular *version* of the Cell processor being discussed. Note that different versions of the Cell processor have a different number of physical SPE cores. The PPE cores will be referred to as the “host cores.” These PPE and SPE cores will be described further in section 2.1.

1.3.2 Considering GPGPUs

Unlike the Cell, which is decreasing in popularity, GPUs are becoming increasingly popular. This is particularly true for programmers writing applications that require high Gflop/s rates, since GPUs currently provide the highest peak flop rates of any accelerator technology. As we have already mentioned, this added performance comes at the cost of being highly specialized. In particular, GPGPUs target calculations that exhibit a high degree of data parallelism. This is a result of being originally designed to handle heavy graphic workloads, which are themselves highly data parallel in nature. While the architecture of the Cell processor is closer to commodity processors, the higher peak performance offered by GPUs is enticing and should not be ignored. With both the opportunities and challenges in mind, we include GPGPUs as one of the target platforms for our work.

Background information about GPGPUs will be covered in section 2.2. Chapters 5 and 6 will cover some of our more recent efforts involving GPGPUs.

1.4 Contributions of this Work

In this work, we will demonstrate the use of a message-driven object-oriented programming model on various hardware architectures, including host cores, Cell processors, and CUDA-based GPGPUs. In particular, the Charm++ programming model will be extended to include *accelerated entry methods* (see section 3.2.1) and SIMDIA (see section 3.2.2). The intent of including these extensions within Charm++ is to create a *unified programming model* in which a programmer can express their application code using a single programming model, regardless of the hardware used to execute the application, and thus provide portability. Further, we wish to explore the use of a message-driven (i.e. similar to event-driven) programming model to program both Cell processors and GPGPUs. For GPGPUs in particular, one does not typically think of using non-data-parallel programming models to express application code that targets accelerators.

By taking advantage of the portability provided by a unified programming model, we also plan to explore various ways in which an active, adaptive, and intelligent runtime system can assist the application, and thus the programmer. By performing a variety of tasks automatically (or perhaps with some minimal guidance), the runtime system can actively perform various tasks that a programmer would otherwise be responsible for supporting explicitly within the application code. A runtime system can help address a variety of concerns shared by a wide variety of applications, ranging from high-level tasks such as dynamic load balancing and fault tolerance support, through very detail-oriented and tedious tasks such as endian correction. The two main areas of focus related to runtime system support will revolve around load balancing (see chapters 4 and 5) and data management (see chapter 6).

Chapter 2

Background

This chapter will review background material that will help the reader better understand the following chapters. There are three main topics that will be covered. First, an overview of the Cell Broadband Engine Architecture (CBEA, or simply "Cell") will be given. The Cell architecture is one of the target architectures that this work targets. Second, an overview of CUDA-based general purpose graphics processing units (GPGPUs) will be given. These GPGPU devices are also a target of our work. Third, an introduction to the Charm++ programming model will be given. This work builds upon the Charm++ programming model and runtime system, so an understanding of how Charm++ operates is important in understanding how our approach operates.

2.1 The Cell Broadband Engine Architecture (CBEA or Cell)

The Cell Broadband Engine Architecture (CBEA), commonly referred to simply as *Cell*, was jointly developed by IBM, Toshiba, and Sony [51]. The architecture was popularized with the inclusion of the Cell processor in Sony's Playstation 3 (PS3), but has also been included in other products, including the QS20 and QS22 blade servers from IBM, along with the SpursEngine from Toshiba. IBM designed a supercomputer cluster which included Cell Processors, named Roadrunner, which is located at Los Alamos National Laboratory (LANL). In June of 2008, the Roadrunner cluster topped the Top500 list and was the first supercomputer to achieve a sustained rate of 1 Pflop/s while executing the Linpack benchmark. Based on the Green500 list from the same month, the top 3 systems, in terms of their flop rate per Watt, were also Cell-based systems, providing further support for the idea that heterogeneous systems are one possible avenue for creating power-efficient, large-scale computing systems.[1]

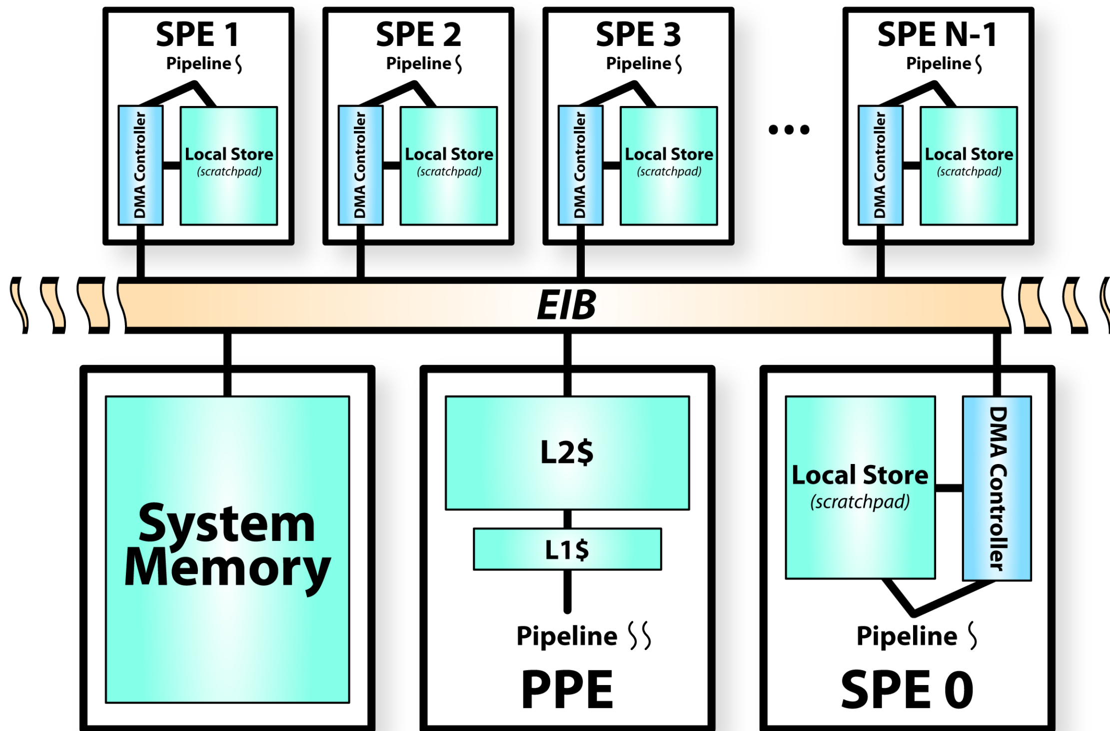


Figure 2.1: Architecture overview of the Cell Processor.

2.1.1 Architecture

The CBEA defines two types of processing cores, the Power Processing Element (PPE) and the Synergistic Processing Element (SPE) [51, 41]. Please refer to figure 2.1 for an overview of the Cell architecture. Note that figure 2.1 does not include all components of the Cell processors, just the components related to this work. Both types of cores are located within a single processor, though in different ratios depending on the particular processor in question.

Generally speaking, the PPE can be thought of as a standard core similar to those included in many commodity multicore processors. In particular, the PPE has direct access to system memory via load/store instructions and full support for virtual memory. The system memory is accessed through a cache hierarchy, similar to other commodity processors.

Unlike the PPEs, the SPEs are specialized cores. In particular, the SPEs are designed to execute computationally intensive code. Typical hardware implementations of the CBEA include several SPEs per PPE. For example, the Cell processors in the IBM blade systems include 1 PPE and 8 SPEs [27], the Sony Playstation 3 includes 1 PPE and 7 SPEs (though, one is dedicated to the OS, so it is effectively 6 SPEs for

[1] http://www.green500.org/lists/2008/06/top/list.php

application code) [65, 19], and the SpursEngine includes 1 PPE and 4 SPEs [42].

While the SPEs resemble typical multicore processor cores in many ways, there are several important differences that are worth noting. First, SPEs do not have direct access to system memory. Instead, each SPE has a dedicated scratchpad memory, called the *local store*, which is accessed via load/store instructions issued by the SPE's pipeline. Local stores are 256KB in size and must contain all data required by the SPE, including the application code, the stack, and the heap. The programmer has complete control over the local store (e.g. dereferencing NULL (0x00) is a valid operation on an SPE). Second, each SPE also has a direct memory access (DMA) engine that is used to move data back and forth between system memory and the SPE's local store. The DMA transactions are issued directly by the application code, leaving the programmer to explicitly interleave DMA related code throughout their application. Well established techniques, such as tiling, code overlays, and streaming, can make use of the DMA engine to overcome the challenge of having such a small amount of physical memory per SPE. However, the task of implementing such techniques is left to the programmer and/or library developers. Third, the ISA of the SPEs is not identical to the PPE. As such, code that is compiled for the PPE will not execute on the SPE, and vice versa. Instead, the SPE code is compiled separately and then *embedded* into the PPE executable. Using functions provided in the Cell SDK, the programmer can then transfer the embedded SPE executable down to an SPE's local store and initiate the execution of that executable on the given SPE. Fourth, the different ISAs also result in different SIMD extensions provided by each core. While having different ISAs has only a limited effect on the programmer when compiling the C/C++ code for either the PPE or the SPE, the different SIMD extensions require different intrinsics to be used by the programmer, depending on which core they are targeting. Further, the types of instructions supported by each SIMD extension are not identical [51].

Each of the cores, regardless of type, are connected to one another via an on-chip network called the Element Interconnect Bus (EIB). Other components, including memory controllers and I/O components, are also connected to the cores via the EIB. The EIB is a bi-directional ring network, with a peak bandwidth of 204.8 Gbytes/s [58].

2.1.2 The Offload API

As part of this work, we developed the Offload API to help ease the burden of programming the Cell Processor [61, 62]. The basic idea behind the Offload API is to allow the programmer to stream *work requests* (think *chunks of work*) to the SPEs. In particular, there should be several active work requests per SPE at any given moment. This allows the SPE runtime to overlap the computation of a ready-to-execute work request with the data transfers of other work requests moving data into and out of the same SPE. This overlap of

computation with data transfers helps to hide the latency of the data transfers themselves.

To make use of the Offload API, the programmer provides a set of *offloaded functions*, each of which takes a programmer-specified number of input and output data buffers (i.e. each buffer can be specified as input and/or output). These functions define the calculations that are available for offloading, from the point-of-view of the PPE. The programmer explicitly creates work requests on the PPE by filling in an associated data structure. Each work request is associated with one of these functions and a number of data buffers, as required by the specific offloaded function. Once the work request is finished being created (code and data specified), the work request is issued to the Offload API. Eventually, at some point in the future, the work request will complete and a programmer specified callback function is called on the PPE to notify the application that any output data that was generated by the work request is ready. The application code can take further actions, such as sending messages, from within the callback function.

As application code issues work requests to the Offload API, the work requests are distributed evenly across the SPEs. Once assigned to a specific SPE, the data associated with a given work request is transferred to the SPE's local store by the SPE's DMA engine. Once all of the data has been transferred to the SPE, the work request is queued for execution. After execution, the output data is transferred back to system memory. Once the output data has been copied to system memory (i.e. is available to the PPE) the Offload API calls the programmer specified callback function. Except for issuing the work request, which occurs directly from within an entry method, and the execution of the callback function, which occurs in-between entry method executons, all of the operations associated with a work request are asynchronous to the application code executing on the PPE (host core). Note, that the explicit use of the Offload API is only required if a Charm++ application *does not* make use of the programming model extensions discussed in chapter 3.

2.2 CUDA-Based GPGPUs

Graphic processing units (GPUs) were originally designed to handle the large amount of number crunching associated with real-time graphics for computer games and other graphic intensive applications. However, as the technologies associated with GPUs have matured over time, they have also been extended into more general purpose computations (i.e. not just graphics). This is where the term general purpose GPU (GPGPU) originated from. Some consider this term to be a bit of a misnomer, since it suggests that GPUs are appropriate for all types of computing tasks. In reality though, GPGPUs work very well at accelerating some types of computations (i.e. computations with very regular data structures, regular memory access patterns, and that are highly data parallel in nature) and are not particularly well-suited for others (i.e. computations

with irregular data structures, irregular or indirect memory access patterns, or are not particularly data parallel in nature).

In this work, we will focus on CUDA-based GPGPUs in particular. In using the phrase "CUDA-based GPGPUs," we simply mean a GPGPU that can be programmed using the CUDA programming model [80]. There is no technical reason that these particular types of GPGPUs were chosen over any other. The decision mainly stems from existing infrastructure related to CUDA within Charm++ (i.e. the GPU Manager, previously known as the Hybrid API, as described in section 2.2.2) [93, 55], mixed with the limited time and resources to build the required infrastructure for several types of GPGPUs by the time of this writing. In other words, CUDA-based GPGPUs are simply the first type of GPGPUs supported within the context of our work, but are not the only type that could be supported.

2.2.1 Architecture

CUDA-based GPGPUs are fairly different from conventional host processors. Most notably, there are many more hardware threads available to the programmer, and subsets of those threads are executed in a single instruction multiple data (SIMD) manner. Before describing how the threads operate relative to one another, we first give the reader an overview of the architecture itself. In particular, we will be describing the Fermi architecture/generation of CUDA-based GPGPUs since these are the type of GPGPUs that we will using in later chapters for conducting experiments [28].

Figure 2.2 gives an overview of CUDA-based GPGPUs based on the Fermi architecture. Note that the Fermi architecture actually has more components than shown in figure 2.2. However, figure 2.2 covers the portions of the architecture that are relevant to our work. Overall, there are many relatively small cores, each of which is capable of executing a single hardware thread. These cores are grouped together into sets of 32 cores which, along with additional resources, are collectively referred to as a streaming multiprocessor (SM). Each SM has a set of resources that are shared/divided amongst the various threads that are executing on the SMs, depending on the exact resource in question. 32 software threads are grouped together into sets called *warps*, with several warps being resident on an SM at any given moment. The exact number of warps an SM can contain depends on the amount of resources that warp requires. Since the resources of an SM are fixed (defined by the particular hardware implementation), the more resources each warp requires, such as registers, shared memory, and so on, the fewer the number of warps that will be able to fit on a given SM at the same time. The threads within the warp are executed using a single instruction multiple data (SIMD) execution model. That is, all of the threads within the warp share the same instruction stream, and the threads within a given warp execute in lockstep with one another. Branches within the

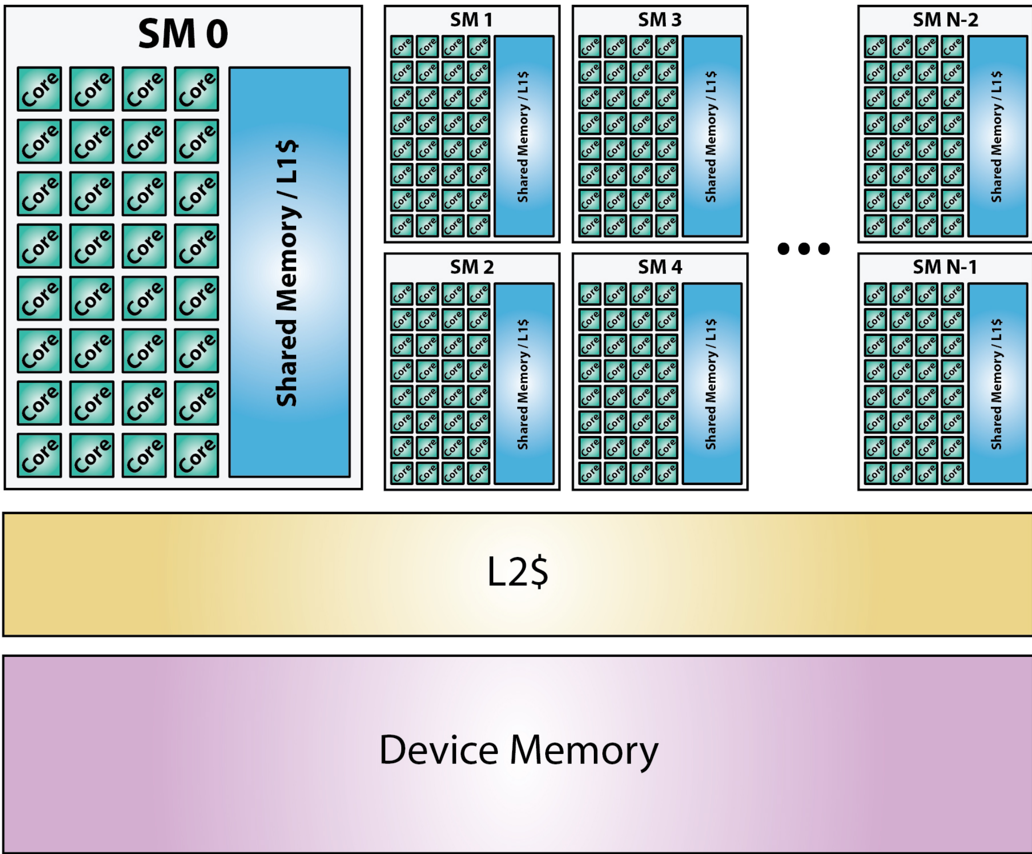


Figure 2.2: Architecture overview of a CUDA-based GPGPU.

instruction stream are handled by causing a subset of the threads to become inactive (think, "executing the instructions, but not committing the results of those instructions" or predication) during the portions of the instruction stream that a particular thread does not need to execute. For example, if at least one thread needs to execute the true block associated with an *if* statement, then all of the threads within the block will executed the true block. The threads that did not evaluate the *if* statement's condition to be true will be marked as inactive during the duration of the *if* statement's true block. In situations where some threads are active and some are inactive, the threads are referred to as being divergent. Threads within a warp being divergent is problematic since it causes hardware cores associated with inactive threads to go unused (i.e. not performing useful work from the point-of-view of the application) as long as those threads remain inactive. This SIMD execution model is a result of the hardware design of GPGPUs. Any application executing code on a GPGPU (of this nature) must adhere to this execution model. The accelerator device as a whole, is made up of several SMs, with current high-end models containing on the order of 16 SMs per device. Across the various generations of hardware, the number of cores has been varied, along with the number of SMs per GPGPU.

Another characteristic of GPGPU hardware is that the threads from one warp can be swapped with the threads in another resident warp within a single cycle. This allows for the instruction latencies experienced within one warp to be overlapped with the execution of instructions from another warp. The overall effect of this fast context switch is that the function units within the cores remain busy for a larger number of cycles (assuming multiple warps per SM). What would otherwise be gaps or bubbles in a core's pipeline cause by data dependencies between instructions from one warp's instruction stream are filled in by instructions from another warp's instruction stream, allowing the application to reach a higher percentage of the accelerator device's peak flop rate that it would otherwise be able to achieve if only a single warp were active per SM. To take advantage of this hardware characteristic, the application typically needs to decrease the granularity of the work associated with each hardware thread, along with keeping the amount of resources required per thread at a minimum, so that the number of warps per SM is maximized (the number of warps per SM is referred to as occupancy). The greater the number of warps per SM, the greater the chance that at least one warp is not stalled on a data dependency, and thus, the greater the chance that an instruction can be issued from one of the warps in the current cycle (i.e. the functional units are kept busy).

However, the SIMD execution model, the fine granularity of the threads, and the separate address space (i.e. device memory) presents extra burden on the programmer. The separate address space is likely a burden that may (or may not, depending on the model of GPGPU being used) be an issue. Some modern processor designs are beginning to include the host processor and the GPGPU cores on a single die, giving

both direct access to and a coherent view of system memory (i.e. the same address space). The other two issues, SIMD execution model and thread granularity, in the most general sense, tend to force programmers to create two versions of their code, one version for the host cores and one version for the accelerator device. Otherwise, the programmer's code is not portable between various platforms. There are, of course, ways around this through the use of various third party libraries and/or the use of programming models designed to abstract these hardware differences away from the application code. Our approach will be discussed in chapter 3.

This description of GPGPU architecture is only intended to provide the reader with a high-level overview of the architecture. There are many resources from a variety of sources providing information related to GPGPUs [80, 28, 94].

2.2.2 GPU Manager

The GPU Manager [93, 55, 49] serves a purpose similar to the purpose the Offload API serves for the Cell processors, which is to allow programmers to easily offload pieces of work to the accelerator device. Like the Offload API, the programmer creates work requests. There is a one-to-one correspondence between work requests and CUDA kernel invocations, which means the granularity of a work request is much greater for the GPU Manager compared to work requests within the Offload API.

The programmer is responsible for writing the actual CUDA kernels. As a result, the code needs to be duplicated by the programmer if using the GPU Manager directly, one copy for the host core and one copy for the GPGPU. In addition to the kernel functions, the programmer also has to write a *kernel select* function. The kernel select function is called on the host core by the GPU Manager as input data for a given work request becomes ready for use on the device. The kernel select function does the work of invoking the kernel's execution on the GPGPU device.

To issue a work request, the programmer fills in a data structure describing the work request from within the application code. The data structure includes all the information required by the GPU Manager to perform the work request on the GPGPU, including but not limited to, which kernel is to be executed, pointers to the data associated with the kernel execution, and a callback function to be called once the kernel has finished executing (i.e. the output data is available to the host core). Similar to the Offload API, each kernel is assigned a unique number, which is specified during work request creating and used by the kernel select function to invoke the appropriate device kernel. Once the work request's data structure has been filled out, the programmer issues the work request to the GPU Manager, referred to as *enqueuing the work request*. At some point in time after the work request has been issued, the GPU Manager will

begin transferring the input data required by the work request from system memory to the GPGPU's device memory. Upon completion of the input data transfer, the GPU Manager calls the kernel select function provided by the application programmer. Using the unique kernel index originally specified in the work request's data structure, the kernel select function invokes the CUDA kernel associated with the index. At this point, the kernel is executed on the GPGPU. Once the kernel completes, the output data is transferred back to system memory. Finally, after the output data has successfully been transferred back to system memory, the GPU Manager triggers the callback function that is associated with the work request. The GPU Manager performs each of these tasks associated with a work request (i.e. transferring input/output data and executing the kernel) asynchronously with the code executing on the host core, with the exception of enqueuing the work request (which occures from within an entry method) and the execution of the callback function (which occurs between entry method executions). This overall process, when applied to multiple work requests, allows the GPU Manager to pipeline the work requests through the GPGPU device, overlapping data transfers with computation on the device.

2.3 Charm++ Basics

The Charm++ programming model is an object-based message-driven parallel programming model [52]. Applications written in Charm++ make use of an adaptive runtime system that controls and modifies the execution of the application. Charm++ has been in use for approximately 20 years and has been used to write several production codes. As such, it is a mature programming model that has a variety of features for application developers to take advantage of. Charm++ is implemented as a collection of C++ libraries which are linked to Charm++ applications. In addition to the libraries, Charm++ makes use of a tool called *charmxi* to generate various pieces of code which serve a variety of roles.

There are three programming constructs at the heart of the Charm++ programming model: chare objects, entry methods, and messages. Chare objects, also called *chares* for short, can be thought of as special C++ objects. Like standard C++ objects, chares have member variables and member functions. However, unlike C++ objects, chare objects also have entry methods (explained below) and can migrated between the host cores available to an application (assuming a Pack-UnPack routine has been created for them, as discribed below). Chare objects can be created and deleted dynamically as an application executes.

Entry methods are special member functions that are asynchronously invoked. That is, when an asynchronous function is invoked on a chare object, it returns immediately and with no return value from the point-of-view of the calling code. At some point in the future, the entry method will be executed on the

target chare object. The invocation of entry methods is how concurrency is expressed within the application code. Like member function in C++, entry methods have a parameter list, allowing values to be passed from the caller to the target. From the point-of-view of any given chare object, the chare may receive more than one entry method invocation at the same time. Should this happen, the messages are *executed on the chare object* in a serial manner, each from start to finish without interruption. As such, there at most one entry method executing on a given chare object at a time, so the number of chare objects within a Charm++ application bounds the amount of parallelism that is achievable within an application. Further, entry methods only have access to the member variables within the chare object they are executing on and to the variables in the message object that triggered their execution (i.e. the parameters passed to the entry method from the caller). These factors remove the need to have locks protecting member variables accessed by more than one entry method in a given chare class.

Charm++ applications make use of a concept referred to as *over-decomposition* or *virtualization*, which basically refers to the idea of having multiple chare objects per physical processing element (PE). The underlying Charm++ runtime system will map the chare objects to the PEs available to an application. In most cases, there are many more objects than there are physical processing elements, and thus, any given PE will have many chare objects mapped to it at any given moment. When programmers write their applications, they write the program as though each chare object has its own PE (and thus write their code independently of the actual number of PEs). However, in reality, several chare objects will be time-sharing a single PE. On any given PE, only a single entry method will be executing at any given moment, and it will complete without interruption.

As entry methods are invoked and messages are created, the Charm++ runtime system routes the messages to the PE that contains the target chare object. Each PE has a message queue that queues all of the messages that target chare objects that are located on the given PE. This leads to the automatic overlap of communication and computation. As some chare objects on the given PE are waiting to have messages arrive, other objects will have messages waiting to be executed in the PE's message queue. This helps keep each of the PEs busy all of the time, maximizing the utilization of the processing elements by the program, along with the amount of parallelism realized at any given moment. Of course, the amount of parallelism realized by the application also relies on how the application is structured. However, this overlap of communication and computation that results from over-decomposition and the nature of the Charm++ programming model influences the programmer to write their application in such a way as to maximize the amount of parallelism.

Charm++ also makes use of Pack-UnPack (PUP) routines, which are used to serialize and deserialize the

state of an object (or any data structure with an associated PUP routine) into and out of data buffers. One use of PUP routines is to pack and unpack the content of a chare object into and out of message objects. As an application executes, the Charm++ runtime system can measure the load of each of the PEs and associate the work being performed to each of the chare objects. If the workloads of the various PEs are unbalanced, the runtime system can use the PUP routines to migrate chare objects between the PEs in an attempt to balance the overall workload across all of the PEs. Exactly how the runtime system decides which chare objects are to be migrated between which PEs is determined by a load balancing strategy. There are several load balancing strategies included within the Charm++ runtime system that decide when to migrate objects using a variety of schemes, such as greedy-based algorithms, and using various pieces of information, such as which objects communicate with one another and the topology of the network connecting the processing elements [12]. Programmers can also define application specific strategies that take application-specific knowledge into account. PUP routines also enable a variety of other features within the Charm++ runtime system, including automatic checkpointing and various fault tolerance mechanisms. Automatic checkpointing can be achieved by migrating all of the chare objects and pending messages to disk via the PUP routines, effectively saving the state of the application. Fault tolerance can also be achieved by duplicating chare objects on multiple PEs (object checkpointing with message logging) to be used for recovery in the event that one of the PEs (or an entire node) suffers a unrecoverable hardware fault. If such a fault occurs, the runtime system can simply recreate the past state of any chare object that has been lost and replay the messages that were sent to it since its last checkpoint [75].

This is only a basic introduction to the main concepts behind the Charm++ programming model and the Charm++ runtime system, with particular emphasis on features relevant to this work. More details can be found at http://charm.cs.illinois.edu/.

2.3.1 Interface Files

In addition to the standard C++ source files and header files, Charm++ also introduces *interface files*. Essentially, one can view the interface files as being similar to header files (conceptually speaking), but only including the portions of the application that make use of Charm++ programming model constructs. Interface files indicate which C++ classes represent chares, chare arrays, groups, and so on. Further, within each class, the programmer also uses the interface files to identify which member functions are also entry methods. Interface files are specific to Charm++. The requirement to include interface files in addition to the header files is a direct result of the Charm++ programming model being implemented as a library that is linked to Charm++ applications.

During the compilation process, a Charm++ specific tool called *charmxi* is used to process the interface files provided by the programmer. Charmxi is a source-to-source translator that reads in the contents of the interface files associated with a Charm++ application and generates additional code. The additional code serves a variety of purposes, including the creation of *proxy* and *index* classes for each type of chare, code related to serializing and deserializing the passed parameters of entry methods, and so on. Generally speaking, the code generated by charmxi provides the necessary hooks required by the runtime system to call into application code, and vice versa, which simplifies the application code. In this work, the charmxi tool is used to generate code related to the programming model extensions we have introduced into Charm++ (see section 3.3.2).

2.3.2 Projections

Projections [56] is a tool used to visually analyze the performance of Charm++ applications. As a Charm++ application executes, the Charm++ runtime system automatically collects various performance related information related to the application's execution. For example, the runtime system tracks the amount of data being transmitted, the execution times for individual entry methods, which entry methods are being invoking by which entry methods, and so on. This information is saved into a set of log files (one per host core) which can then be analyzed using the Projections visualization tool.

Projections includes a wide variety of graphs showing various aspects of an application's execution. For the purposes of discussing our approach to incorporating support for accelerator technologies and heterogeneous systems within the Charm++ programming model, there is one type of Projections graph that will be referred to in this document, the *timeline*. The timeline gives detailed information related to the execution of a Charm++ program on a per host core basis.

Figure 2.3 breaks down the anatomy of a timeline graph. For each host core (PE) in the application, there is a set of rows. Time increases from left to right. Within a set of rows, the bottom row represents the activity on the host core itself (visually speaking, this row is a bit thicker than the others). The remaining rows (which are visually a little bit thinner) are used to display *user events*. Since there can only be one entry method active on the host core at any given moment, per the semantics of the Charm++ programming model, only a single row is needed to display the entry methods executing on a given host core. However, user events are added by the programmer and can occur at arbitrary times. There are two types of events, events that occur instantaneously and events that occur over a period of time with both a starting and stopping time, referred to as *bracketed user events*. Since the programmer can start and stop a bracketed user event at any time, bracketed user events can span one or more entry methods and can overlap with one

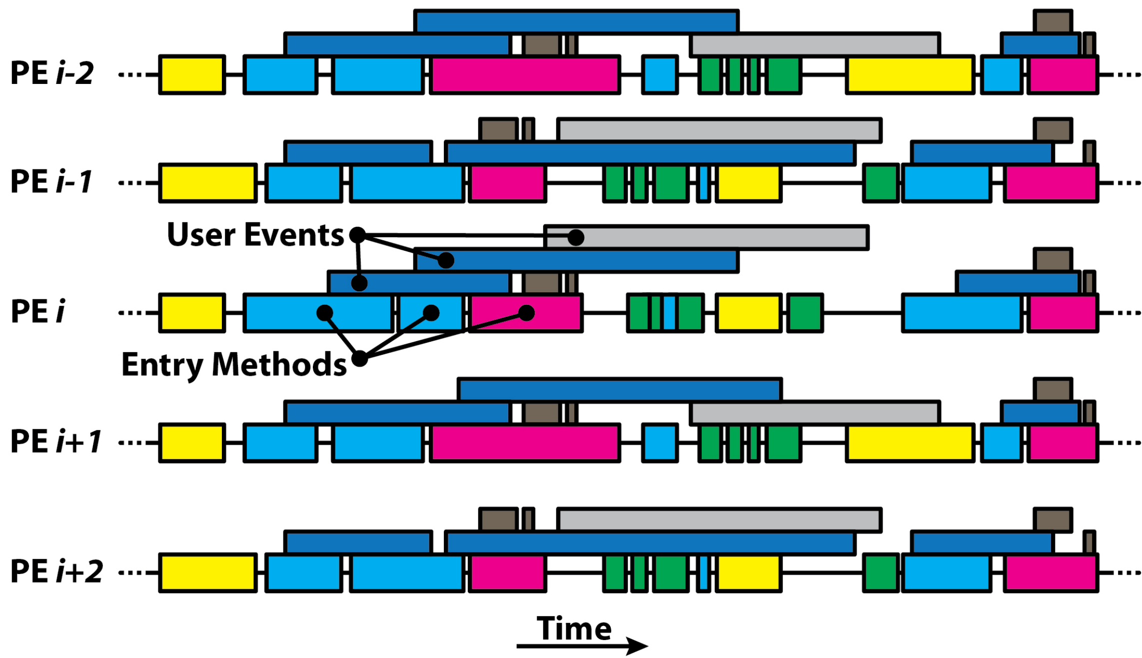


Figure 2.3: The anatomy of a Projections timeline.

another. As such, multiple rows, as many as is required to display them all clearly, are used to display them within a timeline graph. The row in which a particular user event appears in is arbitrary, chosen simply for display purposes so that all user events can be seen by the viewer.

As part of this work, we extend support within the runtime system to collect performance information related to accelerator devices. This additional data can then be viewed within Projections along with the host core data to assist with the performance analysis of applications.

Chapter 3

Approach

Rather than design a programming model and associated runtime system from scratch, we build upon of the existing Charm++ programming model and runtime system. First, the reader will be introduced to the Charm++ programming model in section 3.1. Then, some of the reasons for choosing the Charm++ programming model as the basis for this work be discussed in section 3.1.1. In addition to the advantages, the drawbacks of using Charm++ as the basis of this work will also be discussed in section 3.1.2.

Once the reasons for choosing Charm++ have been covered, the extensions made to the Charm++ programming model as part of this work will be presented. The most significant extension to Charm++ within the context of this work is the addition of accelerated entry methods (AEMs), which will be discussed in more detail in section 3.2.1. In addition to the programming model extensions, there are also several runtime system modifications that have been implemented to support both the programming model extensions and to ease the burden placed on the programmer when using the extensions. The modifications to the Charm++ runtime system will be covered in section 3.3.

3.1 Building Upon Charm++

The basic ideas behind the Charm++ programming model and runtime system were discussed in section 2.3. There are several advantages provided by the Charm++ model that suggest it would serve as a good basis for this work (discussed in section 3.1.1 below). However, beyond just attempting to choose an appropriate base programming model for our work, Charm++ also serves as a platform where we can explore how a message-driven (or perhaps more generally, event-driven) programming model can be applied to accelerator technologies. For example, when one typically thinks of the programming paradigms used for programming SIMD processors such as modern GPGPUs, one does not typically think of using an event-driven model. The selection of Charm++ also creates an opportunity to explore how programming models similar to Charm++ (and, obviously, Charm++ itself) could be applied to a variety of hardware architectures with a variety of characteristics.

With heterogeneity becoming increasingly common, it is important to explore and understand what types of programming models can be applied to heterogeneous architectures. Several other approaches have explored programming models that are explicitly data parallel (see chapter 7). In Charm++, an application is built upon a set of entry methods (i.e. tasks), chare objects (i.e. entities that store data and on which tasks can be performed), and the data dependencies between the entry methods (i.e. messages). If some of these entry methods can have the contents of their function bodies restricted (i.e. just basic code, without system calls, entry method invocations, etc.), these entry methods would be portable between a greater number of architectures, such as accelerator cores. As such, these *restricted entry methods* could be executed on either the host core or an available accelerator core, giving an underlying runtime system the opportunity to offload work from the host core to the accelerator core.

More generally speaking, the overall point is to extend the Charm++ programming model in such a way as to allow the programmer to simply express their application code using a single method of expression. In other words, the point is to create a *unified programming model*, instead of using a separate programming model for the host core with specific subsets of the application code being explicitly offloaded to an available accelerator, such as using MPI and CUDA. The long term goal of this work is to minimize, if not completely remove, the barriers between and the ability to differentiate regions of code that will be executed on either a host core or an accelerator device, or even the type of accelerator device. For example, when using MPI and CUDA code, it is clear that the CUDA kernels execute on the GPGPU(s) (i.e. versus an SPE on the Cell) and that the MPI code executes on the host.

Taking this idea forward yet another step beyond the idea of a single, unified programming model, making the use of the Charm++ runtime system gives us the opportunity to explore how an active runtime system can positively (or negatively) affect an application. Particularly, given a unified programming model, the workload of the application becomes flexible in terms of where it could be executed. Using runtime measurements, a runtime system can apply a load balancing strategy that makes use of that flexibility to balance the workload between the host cores and the available accelerator devices. The end result is that a programmer is in a situation where they need only to express their application code using a single, unified programming model, whether or not an accelerator will actually be available at runtime, but the programmer is still in a position that their application can take advantage of an available accelerator device if one is present.

Finally, it is worth noting that the Charm++ programming model and runtime system has been used in the creation of several production scientific applications, including NAMD [53, 74], ChaNGa [48], OpenAtom [13], and others. As such, the Charm++ programming model and runtime system are mature, providing a

proven foundation for our work.

3.1.1 Advantages of Charm++

There are several advantages in using the Charm++ programming model and existing runtime system infrastructure for the basis of this work. First, an application written in the Charm++ programming model is naturally broken up into a collection of entry methods (tasks) executing on a large set of chare objects. Since computations that are typically offloaded to accelerator devices are short-lived and require a fairly high-degree of parallelism, tasks are a natural fit, assuming there execution can be made to fit with the execution model of the accelerator itself. Second, the Charm++ programming model presents clear and natural communication boundaries that can be used as points where the runtime system can take action, if required. Third, Charm++ already has an existing intelligent, adaptive runtime system infrastructure that can be built upon. Fourth, within the runtime system, there is a message queue that can be thought of as a list of ready to execute tasks (i.e. entry methods). Fifth, Charm++ already makes use of the idea of over-decomposition, which can be useful for hiding latency of various data transfers, whether that data transfers are internal or external to any given node.

Task-Based Nature of Charm++: The Charm++ programming model is an object-based, message-driven programming model, as described in section 2.3. The result of the task-based nature of Charm++ is that there typically are several data independent tasks ready to execute on any given processor at any given moment (i.e. the contents of the message queue). If some of these tasks are suitable for execution on either the host or the accelerator device, then the runtime system is in a position where it has a chance to push data to either the host core or an available accelerator device based on runtime measurements and a given load balancing strategy. Further, since entry methods (i.e. tasks) typically only access the data contained within the chare object the entry method is executing on or the data contained within the message that triggered the entry method (i.e. passed parameter lists), the runtime system is in a situation where the data that will be accessed by the entry method is identified, and thus, the runtime system is able to proactively move the data as needed before the execution of the entry method itself. The overall result is that the execution of the entry methods (tasks) relative to one another is fairly flexible, allowing the runtime system to map their execution to a variety of devices, including accelerators.

Existing Runtime System: As we have already alluded to, Charm++ applications execute on top of an adaptive, intelligent runtime system. In this work, we will be making use of the Charm++ runtime system to assist the programmer in taking advantage of heterogeneous systems. By building upon Charm++, we start with an existing infrastructure in which to explore techniques for heterogeneous systems. Some examples

of how an adaptive runtime system may be extended to assist programmers with heterogeneous systems include assisting with load balancing, adjusting granularity via task splitting or batching, and automatic modification of application data as it crosses processing element boundaries.

Over-Decomposition: One of the main ideas within the Charm++ programming models is the idea of over-decomposition (also referred to as processor virtualization or simply virtualization). The idea is to allow the programmer to write their application as if each chare object has its own physical processing element. However, in reality, the Charm++ runtime system will map many chare objects to each of the physical processing elements available to the application, forcing the individual chare objects to time share the processing element. A benefit of this is that as some chare objects are executing, others are waiting on messages to arrive, causing an overlap of computation with communication as natural result of over-decomposition. This overlap is not only useful for hiding network latencies, but also useful for hiding data movement between a host core and an associated accelerator device. Further, the overlap is useful for hiding the execution of work on an accelerator device itself. If there are many chare objects on a given processing element, that processing element can offload some of its work to an accelerator. To hide the latency of this workload offloading, the host can keep some of the work for itself, overlapping the retained work with the offloaded work.

A second benefit of over-decomposition is that there are opportunities to batch pieces of work from separate chares together. For example, CUDA-based GPGPUs use *kernels* as the unit of work to be offloaded from the host to the device. Kernels typically have a fairly significant amount of work associated with them to amortize the cost of overheads associated with issuing the kernel to the device. However, the amount of work within a single entry method is unlikely to be enough work to required to efficiently execute a kernel. Because there are typically many chare objects mapped to each of the processing elements within a Charm++ application, there is an opportunity to flexibly batch accelerated entry methods (i.e. tasks) invoked on separate chare objects into a single kernel for execution on the GPGPU.

Communication Boundaries: All of the data within a Charm++ application is contained within the chare objects that make up the application. Data is passed between the chare objects through parameter lists in entry method invocations, which creates a natural communication boundary between the chare objects. This boundary gives the runtime system a chance to intercept the data as it is being passed between the chare objects and modify it as needed within a heterogeneous environment. An example of a useful operation that the runtime system may provide is adjusting the endianness of data as it passes from a small-endian architecture to a big-endian architecture, or vice versa. Such modifications are mundane adjustments that a programmer must account for when targeting a heterogeneous system, but also serve

as a source of programming errors that can be quite difficult to track down, depending on how such an error manifests itself. If such an operation could be automated by the runtime system, it would provide a benefit to the programmer by allowing them to focus on their application code without worrying about architecture-specific details, and at the same time, removing a possible source of programming errors.

Message Queue: Within the Charm++ runtime system, there is a message queue for each processing element (more specifically, for each host core). As messages arrive at a given processing element, the messages are executed one-by-one, time-sharing the host core. Messages are accumulated into this message queue, which effectively becomes a *queue of ready-to-execute tasks*. Such a queue potentially gives the runtime system the ability to look-ahead in time to see what tasks will be executing in the near future. If one or more of those tasks is destined for an accelerator device, the runtime system has the opportunity to begin moving data related to the task since it is known that the task *will execute* in the near future.

3.1.2 Disadvantages of Charm++

Some of the main disadvantages of using Charm++ within the context of this work is that Charm++ is implemented as a linked library, rather than having Charm++ specific extensions built into a C++ compiler. This is a disadvantage in the sense that it does not allow data dependency analysis of the application code to be preformed at compile time. There are a few ways in which such analysis could prove helpful.

Lack of Data Analysis: In cases where a particular piece of data contained within a given chare object is accessed by multiple entry methods, at least one of them being an accelerated entry method, it might helpful to know how the entry methods are accessing the data (reading and/or writing). Such information could then be used by the runtime system in making decisions about where the data associated with an AEM should be stored (e.g. are AEMs or EMs accessing the data more frequently), what cores (host or accelerator) are best suited to execute an AEM (i.e. potential costs of data movement compared to the expected runtime of the AEM invocation as a function of its parameters, when possible), and so on.

Correctness or Enforcement of Feature Use: Another disadvantage that stems from not having a Charm++ specific compiler, is that some aspects of the programming model extensions cannot be strictly enforced. For example, there is no compiler to enforce the local parameter's access keywords (e.g. readonly) used in the declarations of accelerated entry methods (discussed in later in section 3.2.1). As such, a programmer could potentially mark a buffer as readonly and then write values into that buffer. The runtime system will likely not transfer the data back to the associated host buffer, leaving the programmer to determine why values written by an accelerator did not make it to host memory. Given compiler support, such access to readonly data buffers could be flagged as errors at compile time.

Lack of Automatic Device-Specific Code Customization: Another disadvantage of not having a custom compiler is that the expression of the code within an accelerated entry method cannot be optimized by a compiler in a device-specific manner. For example, an AEM executing on a GPGPU may be able to take advantage of the device's *constant* memory for local parameters marked as *readonly* or local variables marked as *const*. However, such an optimization would make no sense on an SPE since SPEs do not have *constant* memory. As another example, a compiler could view the contents of an AEM and determine if (and how) it would be safe to automatically convert it to a *splittable* AEM (see section 3.2.1). In this sense, lack of compiler support causes our current implementation of accelerated entry methods to be somewhat limited in providing a truly unifying programming model. Instead, our current implementation limits our AEMs so that they can only provide basic support to the programmer in terms of the AEM's function body (i.e. standard C/C++ code that does not include systems calls, *printf* statements, and so on).

3.1.3 Our Work Withing the Larger Charm++ Context

With both the advantages and disadvantages of building upon the Charm++ programming model and runtime system in mind, figure 3.1 presents how this work fits within the greater context of research surrounding Charm++. Figure 3.1 certainly does not include all of the other abstractions, components, physical hardware support, and so on within Charm++. Rather, the figure is only meant to provide the reader with some context in which to understand how our work relates to other efforts within Charm++ in general.

At the highest level in figure 3.1, there is ongoing research related to language design, programming models, and abstractions, collectively referred to as *programming abstractions*. Each of these programming abstractions are mapped to a *common substrate* that provides the base set of mechanisms used to implement the higher-level abstractions, such as *entry methods*. The elements within the base set of mechanisms can also be used directly by programmers (e.g. entry methods and accelerated entry methods), and as such, they are also included as part of the higher-level programming abstractions provided to programmers. There are too many such elements to be listed, so only a subset of the elements are listed, such as the various mechanisms provided by Charm++ and AMPI [46, 54]. Examples of other mechanisms include CCS and CkDirect [14]. As part of this work, we introduce *accelerated entry methods* to both the programming abstractions and common substrate levels, giving programmers a way of expressing their application code so that it can be mapped to available accelerator devices. The elements of the common substrate are implemented on top of and supported by an active, intelligent, and adaptive *runtime system* made up of various components, such as a scheduler, a load balancing framework, fault tolerance management, an accelerator manager, and so on. As part of this work, we add the accelerator manager component, along with the accelerator

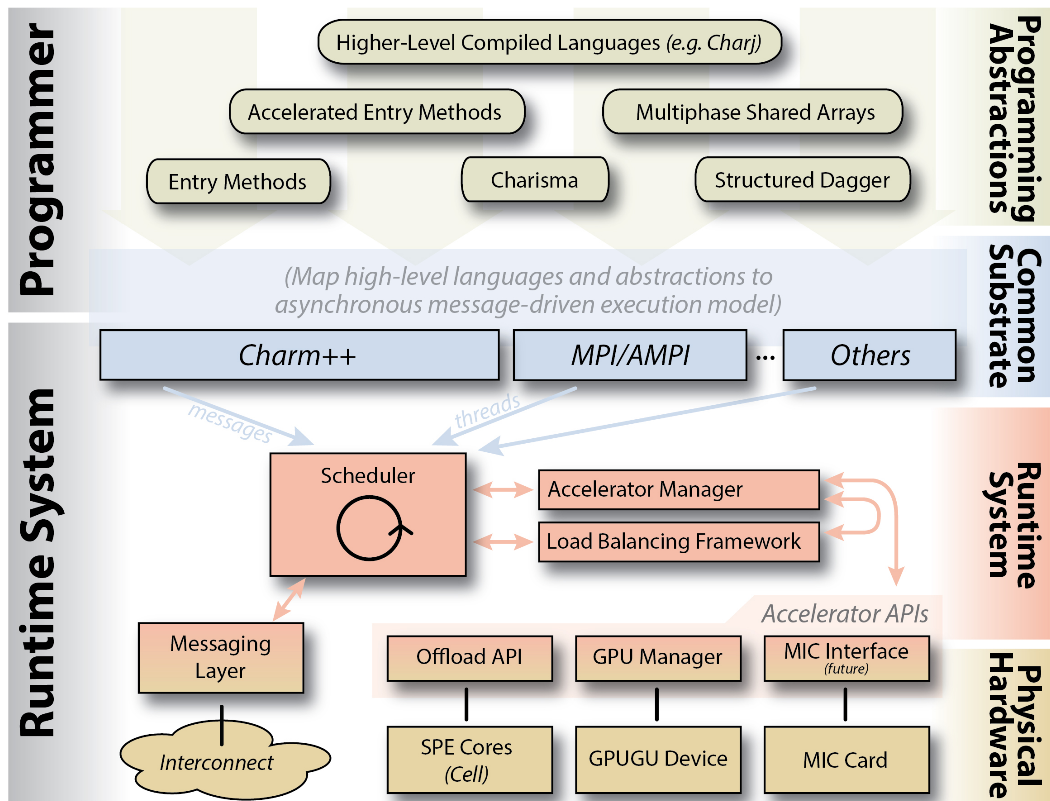


Figure 3.1: Overview of the runtime system structure, along with programming abstractions.

manager's integration and interactions with other components of the runtime system. At the lowest level is the physical hardware itself and the low-level APIs that directly interact with that hardware. As part of this work, we add support for accelerator hardware directly into the Charm++ programming model (both Cell and CUDA-based GPGPUs), along with including a mechanism that allows different builds of the runtime system (i.e. versions compiled for different architectures) to interact with one another at runtime, which enables Charm++ applications to execute across a heterogeneous set of host cores. When our work is considered within this larger context, there are three ways in which other areas of research within Charm++ relates to and interacts with this research, as discussed below.

Addressing Disadvantages: The first way is that other areas of research being explored could potentially address some of the disadvantages of Charm++ in relation to this work (from section 3.1.2). In particular, there currently is an effort to create a compiled language that encompasses the Charm++ programming model called Charj [8]. In particular, Charj presents a path for addressing the disadvantages of Charm++ mentioned above that directly (or indirectly) follow from the lack of a Charm++ specific compiler.

Composability (Applications): There are several abstractions that have been created within the context of the Charm++ programming model, such as multiphased shared arrays [77] and Charisma [45]. Each of these abstractions, along with others, specifically address some aspect of parallel programming in an attempt to ease programmer burden. As such, there are some situations where a portion of a Charm++ application may be written using one of these abstractions, while other portions of the application may make use of accelerated entry methods as presented in this work. Generally speaking, the Charm++ programming model and runtime system support composability of abstractions so that multiple abstractions can be used within a single application for the portion(s) of the application that each abstraction is appropriate for. In this sense, our work fits within the set of interoperable abstractions available to Charm++ programmers.

Modularity (Runtime): At the level of the runtime system, functionality is often compartmentalized with the Charm++ runtime system based on the functionality provided. For example, there are scheduler, load balancing, and fault tolerance, components or modules, just to name a few, within the Charm++ runtime system. As part of this work, an accelerator manager (discussed in section 3.3.1) has been added to this list of components and interacts with the other components of the system in various ways. For example, the scheduler issues AEMs invocations, which eventually requires input from the accelerator manager indicating where the AEM should be executed (host or accelerator). Additionally, the scheduler notifies the accelerator manager when the host core becomes idle or busy. The accelerator manager is also notified of the occurrence of significant events related to load balancing and fault tolerance so that it can take appropriate actions.

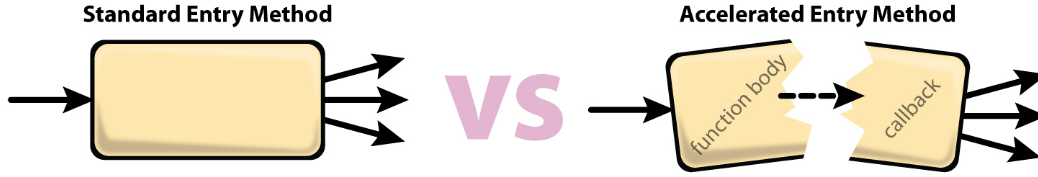


Figure 3.2: Structure of an accelerated entry method (AEM).

By interacting with the various other components, the accelerator manager extends support for the various features and services provided by those other components to include hardware configurations that include accelerator devices.

3.2 Programming Model Extensions

We have added a couple programming model extensions to the Charm++ programming model in order to support heterogeneous systems. The main programming model extension, which is central to this work, is the introduction of accelerated entry methods (AEMs). In addition to AEMs, we have also introduced the SIMD instruction abstraction (SIMDIA), which allows programmers to take advantage of a variety of SIMD instruction extensions while still allowing their code to remain portable between various hardware architectures.

3.2.1 Accelerated Entry Methods (AEMs)

One of the main programming model constructs within Charm++ is the entry method construct. Recall that entry methods are essentially member functions as in C++, except that they are asynchronously invoked and do not return a value from the point-of-view of the caller. There are a variety of types of entry methods, with each type introducing a slight variation from the semantics of a standard entry method. For example, a standard entry method normally executes from start to finish, only returning once it has completed. As such, another entry method can only execute after the current standard entry method has completed. Threaded entry methods, however, are capable of suspending mid-execution and yield the processor until another entry method awakens them. There are also types of entry methods that create other variations to the semantics, such as being able to return values and skipping ahead in the message queue.

As part of this work, we have introduced accelerated entry methods (AEMs). While we have introduced AEMs into Charm++ within the context of accelerator support, it should be pointed out that there is nothing about their structure or nature that is specific to accelerators. As such, AEMs could possibly be applied in other ways in the future. An AEM is essentially a standard entry method that has been divided

into two stages. The left half of figure 3.2 illustrates the execution of a standard entry method, which can be broken down into three main components: a single incoming message triggers the execution of the entry method, the execution of the entry method's function body, and zero or more messages being sent as a result of entry method invocations within the entry method's function body. The right half of figure 3.2 illustrates the execution of an accelerated entry method. The main difference is that the body of the entry method is divided into two stages, the *function body* and the *callback*, with an implicit data dependency in between the two stages to represent any data transfers that might be required if the two stages are executed in different locations. Additionally, the content of the function body is restricted in that it cannot contain system calls, invoke other entry methods, etc. The function body is only intended to contain computation code. The callback stage, on the other hand, may have arbitrary code within it (i.e. it can invoke entry methods, make system calls, etc.). Essentially, the AEMs play the role of the *restricted entry methods* mentioned at the beginning of section 3.1.

Structure

Figure 3.3 illustrates how an accelerated entry method is declared within a Charm++ interface (.ci) file. There are several differences between the structure of a standard entry method and accelerated entry methods.

First, in addition to the the passed parameters used by standard entry methods, accelerated entry methods also declare local parameters. Local parameters are simply a mechanism used to declare which member variables will be accessed by the function body of the AEM. Additionally, each passed parameter can be marked with an access modifier that indicates if the member variable will be accessed in a read-only, read-write, or write-only manner from within the function body. The reason for having the programmer explicitly list the local parameters in this manner mainly stems from the lack of a Charm++ specific compiler, which could otherwise analyze the function body and create a list of local member variables accessed within the AEM's function body. Since the creation of a Charm++ specific compiler is outside the scope of this work, the automated creation of the local parameter list is left as future work. There are also other keywords that can be used in addition to the access modifiers, such as *shared* and *persist*. These modifiers, along with their effects, will be discussed in more detail in chapter 6.

Second, the function bodies of the accelerated entry methods are included within the interface file directly, rather than being located in a source code file. Again, this requirement is basically a implementation detail associated with not having a Charm++ specific compiler that could extract the application code from the C/C++ source file(s) that contain application code. Charm++ does have a tool called *charmxi* that is

```
module myModule {

  array [1D] myChareArray {

    // Constructor
    entry myChareArray();

    // Standard Entry Method
    entry void myEntryMethod(
      type passedParameter1,                    // scalar
      type passedParameter2[passedParameter1],  // array w/ length specified
      ...
    );

    // Accelerated Entry Method
    entry [accel] void myAccelEntryMethod(
      type passedParameter1,
      type passedParameter2[passedParameter1],
      ...
    )[
      modifier : type localParameter1 <impl_obj->memberVariable1>,
      modifier : type localParameter2[localParameter1] <impl_obj->memberVariable2>,
      ...
    ] {

      // ... Function body code goes here ... //

    } callback_function_name;

    // ... other entry method declarations for this array here ... //

  };

  // ... other chare, chare array, group, etc. declarations here ... //

};
```

Figure 3.3: The basic code structure of an accelerated entry method (AEM) within the context of a Charm++ interface file.

used to process the interface files. By including the function body of accelerated entry methods into an application's interface files, the charmxi tool gains access to the application code and can be used to perform various tasks, such as code generation, related to AEMs at compile time.

Third, the name of the callback function that will be executed once the function body has completed is listed at the end of the accelerated entry method declaration (after the function body itself). This requirement is result of AEMs being executed in two stages, rather than a single stage in the case standard entry methods. As such, the presence of a Charm++ specific compiler would not remove this requirement. The callback is used simply as a way of continuing the flow of execution once the AEM has completed. The callback function can be viewed as a separate entry method that is implicitly invoked (with no passed parameters) once the function body of the AEM has completed.

Fourth, the charmxi tool creates multiple classes associated with each chare class within the application. One of these classes is the *CkIndex_xxxx* class, where *xxxx* is the name of the chare class. Some of the generated code, particularly the code that relates to the processing of the parameters associated with an accelerated entry method, is attached to this class. As a result, the programmer is required to declare this *CkIndex_xxxx* class as a *friend* (in the generic C++ sense) of the *xxxx* chare class. Again, the introduction of a Charm++ specific compiler would be one way of removing this requirement.

More generally speaking, the generated code creates a variety of hooks used by the runtime system, such as the *CkIndex_xxxx* class and its related member functions, along with generating some processing code related to the parameters of the entry methods. In the case of accelerated entry methods, the charmxi tool is also responsible for creating the architecture-specific code related to the function body of the AEM. Section 3.3.2 will go into more detail surrounding the generated code within charmxi.

Within the overall discussion about our approach, we wish to clearly distinguish between instances of accelerated entry methods and the *types* of AEMs with an application. For the purposes of this discussion, an *instance* or *invocation* of an AEM refers to the actual execution of an AEM function body (and callback) on a chare object. Each AEM declaration within the application code refers to a distinctive *type* of AEM. We use the term "type" instead of "declaration" since there are actually multiple declarations of each type of AEM, including the declaration of the AEM within the application code, the generated declaration of the AEM for the host core, and any additional generated declarations for each type of supported accelerator devices.

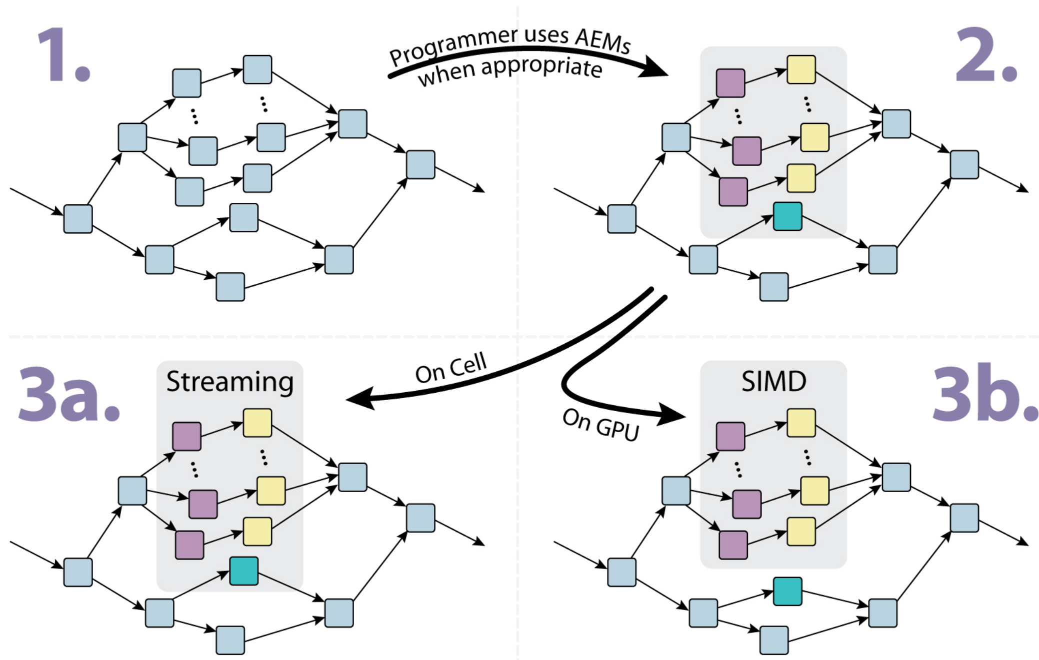


Figure 3.4: Effects on the DAG resulting from the usage of accelerated entry methods.

General Execution

The overall execution of a Charm++ application can be viewed as a directed acyclic graph (DAG). In this graph, each node is an instance of an entry method executing (standard, accelerated, or otherwise). The edges in the DAG represent the data dependencies between the entry methods, realized by the messages carrying the data (e.g. parameters of the entry methods) between the various processing elements. Section 1 of figure 3.4 shows a portion of one such DAG from a hypothetical Charm++ application.

In section 1 of figure 3.4, the entry methods represented by the nodes are simply standard entry methods. If the programmer, instead, uses accelerated entry methods for some of the entry methods, the structure of the DAG remains basically the same. The only difference is that the nodes that represent AEM executions are actually the execution of the function body directly followed by the execution of the callback (i.e. in two parts). These nodes are represented by different colors in section 2, when compared to section 1. Note that only a single node is being used to represent both the function body and the callback of the AEM, since they are executed one after the other anyway, as previously shown in figure 3.2.

At compile time, the build of the Charm++ runtime system determines what accelerators will be supported by the Charm++ runtime system and any applications linked to it. For example, if the application is built and linked against the *net-linux-amd64* build of Charm++, then the runtime system will not attempt to make use of any available accelerator devices. Instead, all accelerated entry methods will be executed

on the host cores. However, if the *net-linux-amd64-cuda* build is used, the runtime system will attempt to make use of any CUDA-based GPGPUs that are available to the application at runtime. In this case, the AEMs *may or may not* be executed on the available accelerators, at the discretion of the runtime system. Typically, a significant fraction of the AEMs will be executed on the accelerators. However, as the host processors have idle time to spare or are better suited for executing the AEMs within an application, the runtime system may use the host core(s) to perform a larger fraction of the computation within the AEMs. In either case, executing on a host core or an accelerator device, the AEMs are still broken up into two stages, the function body followed by the callback.

Exactly how the runtime decides to execute the accelerated entry methods will depend on which build of the runtime system is in use by the application. For example, consider section 3a of figure 3.4. In this case, perhaps the *net-linux-cell* build of the Charm++ runtime system is in use. With each AEM being treated as a *chunk of work*, the runtime system will stream the AEMs through the SPEs on the Cell processor. The direct memory access (DMA) engines for the SPEs will be used to shuttle data back and forth between the host core (PPE) and the accelerator cores (SPEs). Building upon the idea of over-decomposition and with many AEMs active at any given moment, the runtime system has the opportunity to overlap the DMAs from some AEMs with the execution of other AEMs, effectively hiding a significant fraction of the DMA latency.

However, if the *net-linux-amd64-cuda* build of the Charm++ runtime system is in use, the runtime system may do something entirely different with the execution of the accelerated entry methods. Unlike the SPEs of the Cell processor, which are designed to have data and work streamed through them, GPGPUs are designed to work on relatively large amounts of data using a SIMD execution model which utilizes thousands or tens of thousands of very cheap threads actively working on the same kernel at the same time. When such an accelerator is available to the application, the Charm++ runtime system may choose to execute the AEMs in a SIMD-like manner, with one or more GPGPU threads assigned to each instance of an AEM (see section 3b of figure 3.4). Further, sets of AEMs may be grouped together to form *batches* in order to increase the granularity of work offloaded from the host core(s) to the accelerator device(s).

Other accelerator technologies, such as the many integrated cores (MIC) device, may be treated in similar ways or in yet other ways as support for them is added to Charm++. However, from the point-of-view of this document, such support is considered future work.

Triggered AEMs

Accelerated entry methods can also be marked with the *triggered* keyword as part of their declaration in the interface file. The triggered keyword essentially serves as a hint to the runtime system that the AEM will be executed under certain conditions, allowing the runtime system to have more knowledge, and thus make certain assumptions, during the execution of the application.

In particular, the triggered keyword implies two things. First, that the accelerated entry methods will be invoked in a one-to-one manner (for a given AEM type) within a give chare collection. Second, all of the chare objects in the given collection will have the AEM triggered on them before any of them have the same AEM triggered on them a second time. For example, if the programmer were to declare a chare array (i.e. a collection) with an AEM as one of the entry methods, then the programmer could mark the AEM as a triggered AEM as long as the AEM were going to be used in the manner described above. Specifically, all elements within the array would have the AEM triggered the same number of times throughout the lifetime of the application, and all of the elements of the chare array will have the AEM triggered before any one element has the same AEM triggered again.

Knowing that an accelerated entry method of a given type are triggered on a one-to-one basis, the runtime system is able to more easily reason about the execution of the AEMs. In particular, when a GPGPU is present, the runtime system will batch AEM invocations together in an effort to increase granularity of work being offloaded to the GPGPU. If the programmer is able to communicate the intended execution pattern to the runtime system, then the runtime system will be able to make certain simplifying assumptions as it batches together the individual AEM invocations. Certainly, the runtime system can batch together AEMs regardless of whether or not they are triggered. For non-triggered AEMs, however, the runtime system has no *hints* about what to expect during an application's execution and resorts to applying a timeout period before issuing AEM batches that have not yet reached the maximum batch size. Given a triggered AEM, the runtime system *knows* that the number of invocations will be equal to the number of local chare array elements and thus *knows* when batches related to a given triggered AEM should be issued.

Splittable AEMs

In some cases, it may be advantageous to decrease the granularity of individual accelerated entry methods for more efficient execution. A clear example of where this might be advantageous is when the AEMs are being executed on GPGPUs. GPGPUs are throughput-oriented devices, where thousands or even tens of thousands of threads are actively executing on single kernel at any given moment. While the AEMs to be executed are grouped together into one or more kernels to amortize the overhead costs of issuing work to the

GPGPU, the granularity of any individual AEM invocation may be too large to execute on the GPGPUs such that one AEM is mapped to one GPGPU thread. Instead, it would be better if a singe AEM could be executed using many GPGPU threads. Unlike other types of entry methods in the Charm++ programming model, splittable AEMs are unique in that they require only a single message to trigger the execution of several independent *tasks*. That is to say, typically there is a one-to-one relationship between the number of messages a program sends and the number of entry methods that are executed. Some communication mechanisms, such as reductions and broadcasts using spanning trees, break this one-to-one relationship, but it largely holds true. However, in the case of splittable AEMs, a single message can spawn several independent tasks, which are referred to as *splits* in the case of splittable AEMs.

Once again, it is up to the runtime system to determine the actual number of splits in which a splittable AEM will be broken up into at runtime. Within the application code, when the programmer specifies that a given AEM is splittable, the number of *splits* is also specified. The number of splits can be a function of the parameters passed into the entry method. However, the runtime system will ultimately decide whether or not to actually split the splittable AEM into a collection of smaller tasks. As such, the value specified by the programmer is treated as a guiding heuristic by the runtime system (essentially a maximum number of splits, as well as a suggested number of splits). Most likely, the splittable AEM will either be split into the number of splits specified by the programmer or not split at all. In some cases, the runtime system may decided to create a number of splits less than the number specified by the programmer, but also greater than one. Further, as is the case with completely separate entry methods, it is assumed that the individual splits are data independent of one another and require no synchronization between them. The runtime system may execute each of the splits one-by-one (serial execution), in parallel with one another, or in a SIMD-like manner where some of the splits are executed in lockstep relative to one another. Since it is possible that the individual splits may be executed in parallel with one another, splittable AEMs represent one of the few cases where the programmer must be diligent in ensuring there are not data dependencies, true or otherwise, between the splits, which is different from standard entry methods.

Figure 3.5 illustrates the use of splittable accelerated entry methods. The splittable AEM in figure 3.5 performs a simple 5-point stencil calculation on two-dimensional (2D) tile of data. The 2D tile data is embodied within the one-dimensional (1D) arrays *inMatrix* and *outMatrix*, using a row-major data layout (thus *numCols* being used as part of the index calculations into both of these arrays). The main thing to notice within the code example is the declaration of the *for* loop. Since each element of the tile can be calculated independently (i.e. no two iterations write to the same element of *outMatrix*), there are no data dependencies between iterations of the *for* loop and thus the AEM can make use of the splittable keyword.

```
entry [ triggered splittable(numElements) accel ] computeStencilTile() [
  readonly  : int numElements <impl_obj->numElements>,
  readonly  : int numCols     <impl_obj->numCols>,
  readonly  : float inMatrix[numElements] <impl_obj->inMatrix>,
 writeonly  : float outMatrix[numElements] <impl_obj->outMatrix>
] {

  // NOTE: For splittable accelerated entry methods, the variables
  //    'splitIndex' and 'numSplits' are available in the function body.

  // Since the number of splits actually created might not match the number
  //    requested by the code, include a loop that causes this 'split' to loop
  //    over all of the matrix elements assigned to it, with elements being
  //    assigned in a straight forward round-robin manner.
  for (int index = splitIndex; index < numElements; index += numSplits) {

    // Read this element from the input matrix
    float sum = inMatrix[index];
    int count = 1;

    // If not on the top row, add the element directly up
    if (index >= numCols) { sum += inMatrix[index - numCols]; count++; }

    // If not on the bottom row, add the element directly below
    if (index < numElements - numCols) { sum += inMatrix[index + numCols]; count++; }

    // If not on the left edge, add the element directly to the left
    if (index % numCols != 0) { sum += inMatrix[index - 1]; count++; }

    // If not on the right edge, add the element directly to the right
    if (index % numCols != numCols - 1) { sum += inMatrix[index + 1]; count++; }

    // Write the average to this element's index in the output array
    outMatrix[index] = sum / count;

  } // end for (rowIndex < numRows)

} callback_function_for_this_AEM;
```

Figure 3.5: Example code for a splittable accelerated entry method

In making use of the splittable keyword, the *for* loop, that would otherwise just loop over the number of elements (i.e. would simply be *for (int index = 0; index ¡ numElements; index++)*), must be modified to account for multiple splits being created by the runtime system. This is accomplished by simply using the *splitIndex* and *numSplits* values, which are defined by the runtime system as the AEM is invoked, as shown in figure 3.5. Note that the number of splits actually used to execute the AEM is determined at runtime, as reflected in the *for* loop's declaration. If one split is created at runtime, the AEM simply loops over the elements as a non-splittable version of the AEM would have in the first place. If the maximum number of splits are created at runtime, each iteration of the loop will be executed by a separate split, meaning that each split will execute just a single iteration of the *for* loop, resulting in a small amount of overhead. We view this overhead as the cost of giving the runtime system the flexibility of determining the number of splits at runtime. If some number of splits between one and *numElements* is created at runtime, the *for* loop will cause the iterations to be *assigned* to each split in a round-robin manner. Each split will execute its assigned iterations, one-by-one.

Note that the round-robin iteration-to-split assignment we have used in figure 3.5 may not be the most cache friendly assignment scheme possible on a host core because of caching effects, but may be a decent assignment on a GPGPU device because of memory coalescing effects. As a result, this example also illustrates a way in which other approaches, such as threaded building blocks (TBB) [87] which provides abstractions for dividing the iteration space of a *parallel-for* loop amongst a set of threads, may have an advantage compared to the current implementation of our approach. However, techniques such as those used in TBB could be applied within splittable AEMs in the future. As such, we largely see these two approach as orthogonal efforts that could be applied in unison. Our approach mainly involves taking advantage of the existence of a *parallel-for* loop to decrease the granularity of individual tasks, while TBB defines a way in which the the iterations within an iteration space for a *parallel-for* loop can be manipulated. If a TBB-like iteration space mechanism where used within splittable AEMs in the future, the runtime system may choose a different pattern for moving through the iteration space of the *parallel-for* loop based on the target device the AEM invocation is mapped to, such as round-robin if executing on a GPGPU device or a blocked mapping if executing on a host core. As a side note, since architecture of modern CUDA-based GPGPUs are becoming more host-core-like some some ways, such as the inclusion of caches in the Fermi architecture, such iteration-to-split mapping considerations may or may not be a consideration when using future hardware architectures. Within our current implementation, the programmer may use the predefined macros for detecting the particular hardware target the AEM is being compiled for to adjust the iteration-to-split mapping.

While the clearest example of a situation where splittable accelerated entry methods may be useful comes from GPGPUs, there are other situations where splittable AEMs may be useful. For example, when executing an AEM on the host core within a build of the Charm++ runtime system that supports GPGPUs, the runtime system will divide the execution of the AEM between a few splits. Each of these splits will be executed one after the other, with control passing back to the runtime system between splits. While this does not reduce the overall time to execute the work associated with the particular AEM instance, it does reduce the amount of time between the moments in which the runtime system has access to the host core, and thus can take actions. Since the host core must take action to both issue kernels and detect their completion (i.e. making progress calls into the GPU Manager), splitting up AEMs on the host core gives the runtime system a chance to make those progress calls at a higher frequency that it would otherwise be able to if the splittable AEMs were not split at runtime (discussed further in section 5.6). Generally speaking, making use of splittable AEMs on the host core allows the runtime system to be more responsive and reactive when managing the accelerator resources, as required.

It isn't hard to imagine using the the splittable accelerated entry methods in other situations as well. For example, in the case of cores with simultaneous multithreading (SMT) [91], it may be advantageous to create multiple splits, one per SMT thread on a core that an AEM is assigned to. In this case, such a mapping may create situations of constructive cache interference amongst the hardware threads executing the splits, since the splits from the same AEM are likely to access the same memory locations (i.e. the parameters to the AEM invocation).

3.2.2 SIMD Instruction Abstraction (SIMDIA)

Another challenge faced when writing applications so that they are portable to a variety of cores is expressing code related to the single-instruction multiple-data (SIMD) instruction extensions, which have become fairly commonplace in modern commodity hardware. There are several different SIMD instruction extensions that have been introduced in various processor architectures, including SSE [86] (and more recently AVX [24, 69]) in x86-based processors and AltiVec/VMX [92] in Power-based processors. Not only have SIMD instructions become commonplace, making use of SIMD instructions has become a requirement for any application that needs to achieve a significant fraction of a processor's peak flop rate, since using these instructions is usually the only way to completely utilize the functional units of a processor pipeline. To understand why, one first needs to understand how SIMD instructions operate.

SIMD instructions, also known as vector instructions, operate on SIMD registers, also known as vector registers. SIMD registers are simply registers that contain several values at the same time. The exact number

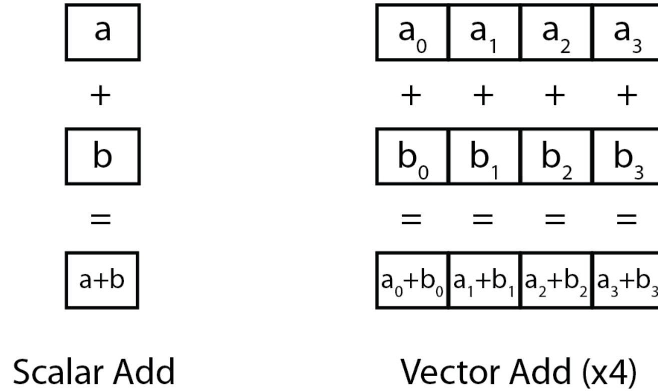


Figure 3.6: A comparison of a scalar addition to a SIMD addition.

of values contained within a SIMD register is architecture dependent. As an example, SSE SIMD registers are 128 bits in size, and thus can contain up to four 32-bit values or up to two 64-bit values. Many SIMD instructions are similar to the standard scalar instructions, performing mathematical operations such as addition, subtraction, multiplication, and so on. The difference is that SIMD instructions perform the given operation on SIMD registers (input and output) and, thus, perform that operation on the values within the SIMD registers in parallel. Figure 3.6 illustrates how a basic SIMD addition occurs, compared to a scalar add.

Because various architectures have differing SIMD instructions, the programmer is burdened with expression their application code in different ways just to perform the same operations as the code is ported between architectures. To address this issue in the context of heterogeneous systems and the inclusion of accelerator technologies, we have introduced the SIMD instruction abstraction (SIMDIA) within the Charm++ programming model.

There were a few observations that influenced the creation, design, and focus of the SIMD instruction abstraction (SIMDIA). (1) In many cases, SIMD code does not need to be modified, except in trivial ways, such as a change in the width of the SIMD register changes. For example, when moving from one architecture to another, the programmer may have to change the stride of a loop iterator to accommodate a different SIMD register width. (2) Even though SIMD instruction extensions vary from architecture to architecture, the extensions are being used to implement the same calculations. As a result, there are many commonalities between the various SIMD extensions. In particular, mathematical operations are very similar. (3) However, there are also several operations that tend to differ between the various processor architectures, which mainly revolve around data operations, such as shuffle instructions. As such, SIMDIA support is weaker for operations related to data shuffling. However, because the SIMD instruction extensions

```
// C = A + B
void addArrays(float *A, float *B, float *C, int len) {

  // Ensure A, B, and C are aligned properly
  assert(((unsigned int)A) % simdia_vecf_numElems == 0);
  assert(((unsigned int)B) % simdia_vecf_numElems == 0);
  assert(((unsigned int)C) % simdia_vecf_numElems == 0);

  // Cast the pointers from scalar to vector types
  simdia_vecf *A_vec = (simdia_vecf*)A;
  simdia_vecf *B_vec = (simdia_vecf*)B;
  simdia_vecf *C_vec = (simdia_vecf*)C;
  int len_vec = len / simdia_vecf_numElems;

  // Loop through the elements, adding as many as possible using vectors
  for (int i = 0; i < len_vec; i++) {
    C_vec[i] = simdia_vaddf(A_vec[i], B_vec[i]);
  }

  // If len is not a multiple of the vector width, add the last few as scalars
  for (int i = len_vec * simdia_vecf_numElems; i < len; i++) {
    C[i] = A[i] + B[i];
  }
}
```

Figure 3.7: Example code illustrating the use of SIMDIA

within various processor architectures are being applied to the same set of calculations, it is likely the various SIMD instructions may begin to converge in the future. Further, common data operations, such as rotating elements, inserting/removing elements, and so on, can be expressed in SIMDIA even if the underlying SIMD instruction extension that a given processor uses does not directly support them or use different approaches. (4) If a mechanism is provided that allows architecture-specific code to be interleaved with the SIMDIA code, the programmer can still benefit from SIMDIA by using SIMDIA when possible and only being required to write architecture-specific code when it is truly required. As SIMDIA support grows and/or hardware implementations converge, such regions will likely be reduced, if not eliminated.

To illustrate the use of SIMDIA, figure 3.7 contains a simple example function. The function, *addArrays*, adds the corresponding elements of arrays A and B and places the result into the corresponding element of array C. All three arrays have *len* elements. The codes starts by verifying the arrays are properly aligned to an address that is a multiple of the native vector width. Such an alignment requirement isn't always required, however, it is usually beneficial for performance reasons, and thus, usually encouraged, even if not required. As a side note, the Charm++ distribution provides functions for allocating and deallocating aligned memory buffers. The next few lines simply cast the scalar pointers passed into the function via parameters A, B, and C into pointers to vector arrays *A_vec*, *B_vec*, and *C_vec*. Additionally, the value

len_vec is calculated, which because of the integer divide being used, is 'the largest multiple of the vector width that is less than or equal to *len*' divided by 'the native vector width.' In other words, it's the maximum number of non-overlapping vector instructions that can be used in place of the equivalent scalar instructions. The first *for* loop proceeds to loop over the arrays, adding a vector width's worth of elements per iteration via a single SIMD instruction (an *add* in the case of the code in figure 3.7). As many of the array elements are added using the vector instructions. However, the value in *len* may not be a multiple of the native vector width, *simdia_vecf_numElems*. In such cases, *len_vec * simdia_vecf_numElems* will be less than *len*, causing the second for loop to finish adding the extra elements at the end of the arrays using scalar instructions. Note that the number of iterations performed in the second *for* loop will always be less than the native vector width.

At this point, it should be noted that the way in which the code within figure 3.7 is implemented in a manner that is similar to how it would be implemented using any particular SIMD instruction extension within a given hardware architecture. That is, the general way in which the code is written using the SIMDIA abstraction or any specific SIMD instruction extension that SIMDIA maps to is the same, such as the two *for* loops, one for vector instructions followed by one with scalar instructions. Also note that our immediate goal is to achieve portability for the purpose of creating a unified programming model, not to necessarily to determine the optimal abstraction SIMD/vector operations.

3.3 Runtime System Modifications

In addition to the extensions to the Charm++ programming model, as described in section 3.2, several modifications were required within the Charm++ runtime system. The most notable change is the creation of the accelerator manager and its related components. The actual execution of accelerated entry methods are controlled by the accelerator manager, whatever that entails given a particular accelerator architecture.

3.3.1 Accelerator Manager

As part of this work, a new component has been introduced into the Charm++ runtime system to help control and coordinate the execution of accelerated entry method (AEMs). This component is called the *accelerator manager* (or *AccelManager*). The main role of the accelerator manager is to direct the execution of the accelerated entry methods to either an available accelerator or the host core for the purposes of load balancing work between the various core types available to an application. To accomplish this task, the accelerator manager tracks certain performance metrics and uses those metrics in applying various

accelerator load balancing strategies that specifically focus on balancing the workload between the host core and its associated accelerator device. Such load balancing strategies will be referred to as *accelerator load balancing strategies* within this document and should not be confused with the typical load balancing strategies included within Charm++.

Performance Measurements

There are currently three performance metrics that are tracked by the accelerator manager: an accelerator's busy and idle time, the PE's busy and idle time, and a periodic measurement.

The first performance metric that is tracked by the accelerator manager is an estimate of an accelerator's busy and idle time. An accelerator, or collection of accelerator cores, is said to be busy if there is at least one accelerated entry method that has been dispatched to the device and has yet to complete. For the purposes of this definition, "dispatched" means actually issued to the device, or an API/library acting as an interface to the device, for execution. "Complete" means the moment the accelerator manager becomes aware that the AEM's function body has completed, which occurs prior to the callback function being executed. In cases where AEMs are processed in batches, those AEMs are only considered to be dispatched when the batch as a whole is dispatched, since prior to that point the accelerator cannot begin executing any of the AEM invocations within the batch set. A simple counter is used to track the number of outstanding AEMs, or batches of AEMs when the AEMs are batched. Each time an AEM is dispatched, the counter increments, and each time an AEM completes, the counter is decremented. Whenever the counter transitions from zero to one, the accelerator is considered to transition from idle to busy. Whenever the counter transitions from one to zero, the accelerator is considered to transition from busy to idle. While this is not an exact measurement of the time it takes for a given piece of code to execute on a given accelerator device, it is a fairly good measurement of how long a set of AEMs, whether streamed individually or batched into one or more sets, takes to complete on a given accelerator device from the point of view of the host processor. It is unlikely that the function relating the number of AEMs issued and the amount of time it takes to complete them will be linear in nature, especially when executing only a few AEMs. The non-linear relationship is a result of a variety of factors that are architecture-specific, including the presence of multiple accelerator cores, context switching between threads on the accelerator itself, and the overlap of data transfers to/from the accelerator device with code executing on the accelerator. There are also second order effects that may affect the performance of the accelerator device, such as the presence of a second accelerator device that somehow shares a resource with the first accelerator device, such as a shared bus located somewhere between the host and the device. For example, consider the case of a single-socket multicore workstation

with GPGPUs attached. A limiting factor may be the rate at which data can be pushed through the pins of the host processor socket, despite the fact that the GPGPUs themselves are located on separate PCI-E connections. Even worst, perhaps while one process is making use of a given GPGPU, another process comes along and also begins making use of the same GPGPU, causing the original process to suffer a performance degradation. For these reasons, we attempt to provide realtime measurements of the accelerator's busy and idle times, rather than model the performance. Further, these performance measurements are taken from the point of view of the host, not only because such measurements might not always be possible on the device itself, but because measurements taken from the host will include the other effects mentioned above, such as streaming effects and interference.

The second performance metric that is tracked by the accelerator manager is the host core's busy and idle time. The existing Charm++ runtime system already provides a mechanism that notifies application code as to when the PE transitions from busy to idle, or vice versa. The accelerator manager simply makes use of this mechanism to accumulate the busy and idle time for the host, as each type of transition occurs.

The third performance metric that can be tracked by the accelerator manager is a periodic measurement. Unlike the first two measurements which are always enabled, the periodic measurement is disabled by default. The accelerator manager exposes a public method, *takePeriodicSample*, that allows application code to periodically provide a value. If no value is specified, the time between calls to this method is used by default (useful for timestep-based applications). The load balancing strategies used by the accelerator manager can then use this information, in conjunction with the busy and idle times discussed above, to guide decisions about where to execute the AEMs. For example, a strategy may adjust the workload balance between the host core and the accelerator in an attempt minimize this value. There is also an option in the accelerator manager to automate this measurement (using time intervals as the value). Enabling this option will cause the accelerator manager to periodically call this method based on the frequency of AEM invocations. However, this configuration may not always be effective, particularly in cases where the AEMs are not invoked at regular intervals, resulting in inaccurate time interval measurements. In any case, a given accelerator load balancing strategy may make adjustments to the workload distribution between the host and accelerator core, while using the values provided by the *takePeriodicSample* function as a way of testing if those adjustments had a positive or negative effect on the performance of the application.

Accelerator Load Balancing Strategies

There are also several accelerator load balancing strategies that can be applied to Charm++ applications from within the accelerator manager. While several strategies have already been defined within the

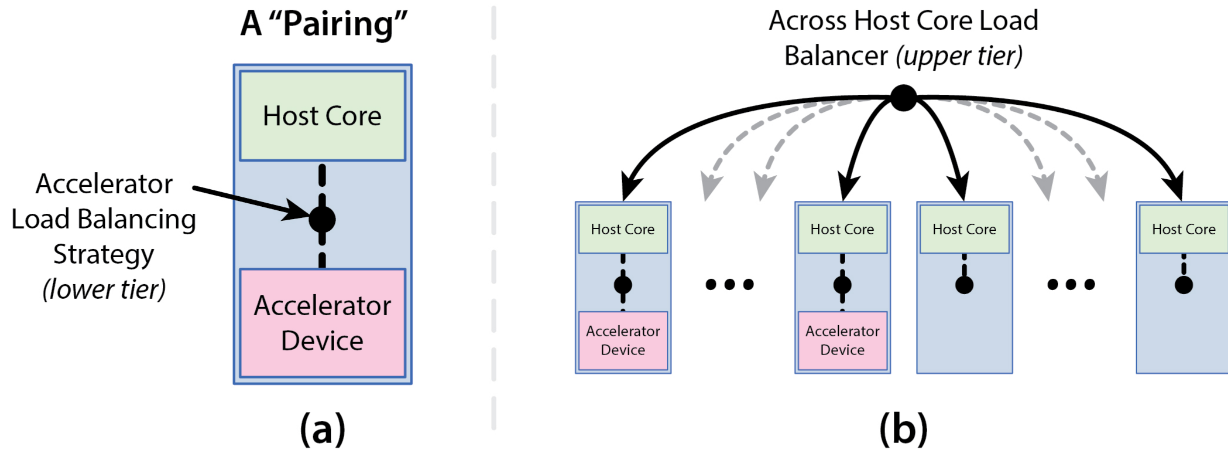


Figure 3.8: The two tiers of load balancing for heterogeneous systems that include accelerator devices.

Charm++ code base, others can be added fairly easily. This is accomplished by creating a child class of the abstract base class, *AccelStrategy*, which will be covered in more detail, along with descriptions of the accelerator load balancing strategies themselves, in section 5.1.

To be very clear, the accelerator load balancing strategies are not the same as the load balancing strategies that are typically talked about within the context of the Charm++ load balancing framework. Instead, the accelerator load balancing strategies are meant to supplement the standard load balancing strategies that are typically talked about within the context of the Charm++ load balancing framework. At a high level, one can think of the runtime system having a two tier load balancing system when accelerators are in use (refer to figure 3.8). The upper tier (figure 3.8(b)) is made up of the existing Charm++ load balancing framework and is responsible for shifting workloads between the various host cores available to a given application. In the upper tier, the presence of the accelerator isn't directly known to the load balancing framework, but rather the host cores with associated accelerator devices are simply viewed as being "faster" or "more powerful" compared to cores without accelerator devices. This tier is the load balancing mechanism that has been traditionally applied to Charm++ applications executing on homogeneous systems. The lower tier (figure 3.8(a)) is made up of the accelerator load balancing strategies, which are responsible for load balancing work between a given host core and its associated accelerator device. As was already mentioned, these strategies will be discussed in further detail in section 5.1.

This division of load balancing into a two tier system is done for a couple of reasons. First, the majority of clusters in use today do not include accelerator devices of any kind, and as such, are not heterogeneous in nature. When designing a load balancing strategy, a programmer should be able to focus on the algorithm the strategy implements, rather that the numerous hardware combinations that might be possible. At a high

level, a load balancing strategy developer only need be aware that some host cores execute certain pieces of calculations (i.e. entry methods or tasks) faster than others, without needed to know the underlying cause of the speedup (i.e. the nature of the hardware itself and why a particular task executes faster on one host core versus another). In some cases, the underlying cause may be as simple as differences in host core clock frequencies, the amount of system memory available to each host core, or architecture differences between the nodes or host cores. In other cases, the cause may be related to the presence of one or more accelerator devices. The goal is to allow the upper level load balancing strategies to remain very general in nature and devoid of architecture details. In the case of a homogeneous cluster, the situation a load balancer faces is then just simplified since all of the host cores (ideally) provide an equal level of performance. Though, it should be noted that even in the case of a seemingly homogeneous cluster, secondary effects, such as non-uniform memory access (NUMA) latencies, may in fact cause the cluster to behave as a heterogeneous cluster to a small degree (i.e. non-uniform performance of the host cores) if those effects are not accounted for properly (e.g. by mapping tasks to locations near the data those tasks operate on).

Second, the accelerator strategies may have different requirements/challenges associated with them compared to the higher tier load balancing strategies. For example, the application of higher tier load balancing strategy to moving objects between host cores can be a fairly costly operation, causing a significant delay in the execution of an application, depending on the strategy used. As such, load balancing between host cores is typically performed on an infrequent basis. However, in the case of an accelerator being present, it is important to find a balance between the host core and the accelerator device quickly, especially in cases where the accelerator device offers a significant performance improvement. As will be shown in chapter 5, even a small misbalance in the division of work between a host core and an accelerator core can result in a significant degradation in performance. Further, some accelerator load balancing strategies, such as the Sampling strategy discussed in section 5.1.3, require several load balancing adjustments to take place before settling in on a particular balance point. Since the local mapping of AEMs to either the host core or the accelerator device can be adjusted with little to no overhead, decoupling the two tiers of load balancing can allow the runtime system to converge on a near optimal balance point in less time. Therefore, the load balancing between a host core and an accelerator device benefits by occurring at a higher frequency.

There are also a couple of potential downsides to having a multi-tier load balancing mechanism acting within a Charm++ application. Since the different tiers act independently of one another to a large degree, there is the potential for feedback cycle to occur which causes a decrease in the performance of the application. If one tier makes a bad decision, it could cause the other tier to also make a bad decisions, which in turn causes the first tier to make an even worst decision, and so on. As such, care should be taken when creating

and applying the load balancing strategies. In particular, it seems as though a best practice is to allow the lower, higher-frequency tier (i.e. the accelerator strategies) to settle into a stable state between each load balancing step in the upper-level tier (i.e. the standard load balancing strategies that move chare objects between host cores). Additionally, having a multi-tier load balancing system means that the upper-tier is acting without complete knowledge. Instead, the host cores with accelerator devices simply seem to have higher performance levels that host cores without access to accelerator devices. As we will point out in section 5.7.1, this can lead to situations where the across host core load balancing strategy is incapable of making an optimal decision on some platforms.

3.3.2 Architecture-Specific Code Generation

Some generated code is also produced for each of the accelerated entry methods, which works in conjunction with the accelerator manager. Speaking in general, whenever an interface file is processed by the charmxi tool, charmxi is generating code that serves as the glue between the runtime system and the application code. For example, for each of the chare classes (or chare collections) declared within an interface file, some auxiliary classes are also generated, including the CBase_xxxx, CProxy_xxxx, and CkIndex_xxxx classes, where "xxxx" is replaced by the name of the chare class. These classes serve a variety of purposes and exist regardless of whether or not AEMs are being used within the application or not.

As an example, the CProxy_xxxx class serves as a proxy for an actual instance of a chare object, or an instance of an entire chare collection. When a chare object is created, the application code receives an instance of the chare's proxy class rather than the chare object itself. This proxy object can then be transferred between PEs via entry method parameters and used to invoke entry methods on the actual chare class, regardless of where that chare class physically resides (i.e. on the same core or a different core). To invoke an entry method on a chare class, one simply invokes that entry method on the proxy object, using the same entry method name and passing the same parameters as would be used as if the proxy were the chare object itself. The proxy object then processes the actual data being passed through the parameter list and interacts with the Charm++ runtime system to initiate the transmission of the message object containing the parameters.

In a similar manner to standard entry methods, code is generated to help support the execution of accelerated entry methods. In terms of invoking an accelerated entry method, the process is almost identical to that of a standard entry method. From the point of view of view of the application code, invoking an accelerated entry method is exactly the same as a standard entry methods. From the sender's point-of-view, the sender simply invokes the AEM on the local copy of the chare's proxy object, passing in any passed

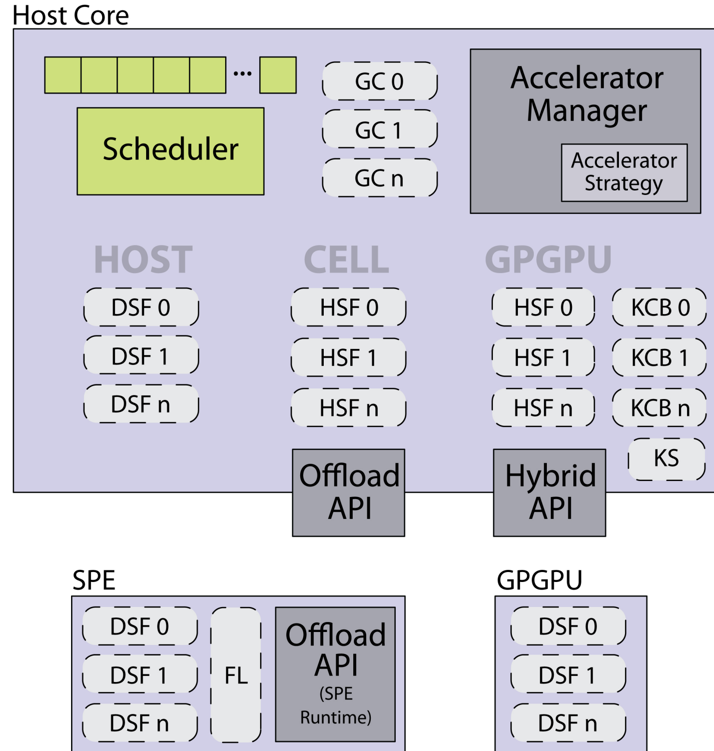


Figure 3.9: Overview of the generated code created by charmxi.

parameters required by the AEM, just as would be done with a standard entry method.

On the receiving side, as a message that targets an accelerated entry method is pulled from the message queue, the chare object is identified and a generated function within the *CkIndex_xxxx* class is used as a hook for the runtime system to call into the application code. Conceptually, this process is the same for an AEMs as well. Where the process for standard entry methods and AEMs diverge is the content of the generated code within *CkIndex_xxxx*. In the case of an AEM, this function is actually broken up into two stages, a general stage and a device-specific stage. As the names suggest, there is a single version of the general stage, and there is a different version of the device-specific stage for each of the supported accelerator devices. First, in the general stage, the parameters are located within the payload of the message and a request for a *decision* is made of the accelerator manager. The result of the decision is determined by the particular accelerator load balancing strategy in use at the moment a decision was requested. The decision returned by the accelerator manager indicates, amongst other things, to which device the entry method should be directed (i.e. the host or an available accelerator). At this point, the appropriate version of the device-specific stage (function) is called, depending on the target device.

The device-specific stage performs any tasks that are necessary to dispatch the execution of the AEM to the associated device. Obviously the contents of the device-specific stages (functions) are dependent on the

type of device being used. For example, there is a basic host-specific function which basically just executes the function body of the AEM, after ensuring any persistent data is available to the host core, and then immediately calls the associated callback function. In the case of a splittable AEM, the host-specific function may also break the function body's execution across a few splits, executing each split one-by-one, and may perform some task(s) in between the splits, such as progress calls to the libraries being used to interface with the accelerators. As another example, the CUDA-based GPGPU-specific function uses the GPU Manager to issue work to the GPGPU, handles persistent data and shared data as required, and batches the AEM invocation into a batch set. As part of the decision result returned by the accelerator manager, whether or not the generated code should "issue" or "dispatch" the current batch set is also indicated. If the AEM invocation is not to be issued, it will be added to the currently accumulating batch set. If the AEM is to be issued, it will first be added to the currently accumulating batch set and then the batch set as a whole will be issued to the indicated location. These device-specific functions will be discussed in greater detail below.

Before moving on to discuss the generated code for each type of device, it should be noted that our approach currently does not include any compile time optimizations for the code included in an accelerated entry method's function body. By this, we mean that an AEM's function body is basically used as is on each type of device. Any compiler optimizations that are performed by the device-specific compiler (e.g. NVCC for CUDA-based GPGPUs) will be performed. However, there are no compile time optimizations included within or approach directly. Such optimizations are considered future work.

Host Core Code

If an AEM is directed to the host core, a generated device-specific function that targets the host core is invoked. This host function executes the function body of the accelerated entry method directly on the host core. Immediately after the function body is finished, it invokes the callback associated with the AEM. This results an accelerated entry method being executed in a way that is very similar to he way in which a standard entry method is executed.

Typically, if an AEM is splittable, the device-specific function generated for the host processor will not actually cause the AEM to be split. However, there are some cases where the host core's device-specific function may split an AEM. For example, in cases where the library being used to interface with an available accelerator device requires that a *progress* function to be periodically called. Long running entry methods on the host core prevent the runtime system from making these progress calls during the duration of the entry method. Should the lack of a progress call cause the accelerator device to become idle, then a long running entry method (task) will be problematic. In the case of splittable AEMs, the runtime system can

use the splittable nature of the AEM to help reduce the time between progress calls by creating multiple splits and issuing them one-by-one, effectively increasing the frequency with which host core control returns to the runtime system. While the time to execute the individual AEM invocation may increase (depending on the nature of the splits, cache locality, etc.), the overall performance of the host core and accelerator device pairing may increase since the host core is more responsive to the accelerator's needs (discussed in section 5.6).

In addition to just immediately executing each AEM invocation dispatched to the host core immediately, the accelerator manager may also indicate that the AEM invocation should be delayed. This is particularly useful when it is know that more AEM invocations will be dispatched to an accelerator device in the case of static assignment (will be covered more in section 6.3.3). To accomplish this, another function is called, who's purpose is to grab the data required to execute the AEM and queue it within the accelerator manager. At an appropriate point in the future, the accelerator manager will call a continuation function which will execute the delayed AEMs using their host-specific function.

Cell Processor Code

In the case of an accelerated entry method being directed to an SPE on the Cell processor, the generated code makes use of the Offload API to simplify the code generation process. The Offload API allows a user to specify a function to be executed, the input and output data to that function, and a callback function that will be called when the output data is available on the host core. Since this functionality already exists within a library available within the Charm++ runtime system, this work will make use of this library rather than duplicate the functionality and using the Cell SDK directly. Even though our work can make use of the Offload API to handle the mechanical details of offloading a function call to one of the SPEs, the code generation process is still required to map an AEM invocation to a work request (i.e. the code would be fairly similar to code generated if the SDK were used directly, minus a few additional functionalities provided by the Offload API).

In a basic sense, the generated Cell-specific code translates the accelerated entry method invocation into work request, as required by the Offload API. More specifically, there are four pieces of code generated in relation to each accelerated entry method.

Registration Code (RC): A requirement of the Offload API is that each of the device functions have a unique index value that can be used to identify the device functions. From the point-of-view of the Offload API, the assignment of these indexes is left to the application code making use of the Offload API library. As part of the code generation process for AEMs within charmxi, a registration function is

generated that enumerates the accelerated entry methods. There is a single registration function for each of the Charm++ modules specified within the interface file(s). Additionally, the registration functions of any external modules specified within the interface file are also invoked within a given module's registration function. The registration function of main modules are invoked at the beginning of the application's execution, allowing any and all modules that include AEMs to dynamically determine their indexes during application startup, and thus share the SPEs. The result of the registration process is a lookup table that is indexed by the AEM's function indexes (a unique function index value for each AEM) and contains the actual device function pointers associated with those indexes. This table is then used to implement the lookup function (*funcLookup*), also required by the Offload API, which is used by the SPE Runtime executing on each available SPE to find the appropriate code to execute for each work request issued to the SPE.

Since the registration process occurs during the initialization process of the runtime system, dynamically assigning identifiers to the accelerated entry methods, the various modules used within an application can all make use of an attached accelerator device. Further, any module that makes use of AEMs is not required to be *aware* of whether or not other modules are also making use of AEMs. This allows for composability within applications, allowing different portions of an application to be written independently of one another and eventually linked together.

Host-side Function (HSF): For each accelerated entry method, a host-side function is generated. The purpose of the host-side function is to setup the work request and issue it to the Offload API. Input and output data is identified, along with the function index associated with the AEM to be executed on the device and the callback function to be executed when the work request as been completed. This work request is then issued to the Offload API, which does the mechanical work of scheduling the work request on one of the SPEs, transferring the input data, scheduling a call to the generated device function associated with this AEM, moving the output data back into system memory, and then triggering the callback function associated with the AEM.

Device-side Function (DSF): For each accelerated entry method, a device-side function is also generated. This function can be thought of as a wrapper function for the function body of the AEM. It performs some initial parameter processing in preparation for executing the function body of the AEM, along with executing the AEM's function body itself. This function is executed on the SPE device directly, whenever an AEM invocation is mapped to an available SPE core.

Function Lookup (FL): For the Offload API's SPE runtime to call into the device function, it requires a lookup function, called *funcLookup*. A single global lookup function is generated for each application. This global lookup function uses the unique function index value associated with each AEM to index into

the table of device function pointers generated during initialization as part of the AEM registration process described above. This lookup process is triggered by the SPE runtime (part of the Offload API) after all of the input data required by the AEM (work request using Offload API terminology) has been transferred into the local store of the SPE to which the AEM was assigned.

CUDA-Based GPGPU Code

The device-specific function that is generated in the case of CUDA-Based GPGPUs makes use of the GPU Manager in a manner that is similar to the way in which the device-specific function for the Cell makes use of the Offload API. The GPU Manager takes care of the mechanical processes of moving data between the accelerator device and the host core (except in the case of persistent data buffers), triggering the execution of the generated kernel code when the input data is ready, and calling a specified callback function when the output data is available within the host core's memory. In addition to the process of issuing the work requests, the generated code also performs the batching operations required to increase the granularity of the work requests. Note that the callback function referred to here is not the callback function specified within the application code, but rather a generated callback function which, amongst other things, is responsible for calling the individual application code callbacks associated with the AEMs that were batched together into a single kernel. The device-specific code for CUDA-based GPGPUs can be broken down into five basic pieces.

Registration Code (RC): Just as it is with the Cell-specific code, the GPGPU-specific code requires a unique index to be generated for each of the accelerated entry methods. This is done in a manner consistent with the Cell-specific code, as described in section 3.3.2.

Host-side Function (HSF): There are two main objectives for the generated host-side code, collecting multiple AEMs together into batch sets and issuing those batches to the GPU Manager for processing.

First, as an AEM is passed to the host-side code, the host-side code adds the AEM to the current batch set. If this happens to be the first AEM in the set, the batch set data structures are also initialized. The data related to the AEM is that packed in to the batch set's data buffer, along with other values used to identify the AEM's data within the buffer and the object on which the AEM is operating. The data buffer will be covered in more detail in section 6.2.

Once the AEM has been incorporated within the current batch set, the output of the accelerator's decision process is used to determine if the current batch set should be dispatched to the device. If so, details of the batch set are passed to a general function which uses the batch set's information to create a work request and issues that work request to the GPU Manager. The GPU Manager is then responsible for transferring

the input data to the device, triggering the device-side code by calling the kernel select function on the host core (required by the GPU Manager), transferring the output data back to the host, and finally triggering the generated kernel-level callback function.

Kernel Select Function (KSF): The GPU Manager requires a kernel selection function, which is used to select and execute the device-side function (or kernel function in CUDA terms) using the unique identifier assigned to each AEM. The kernel select function is called by the GPU Manager only after a work request has been issued to the GPU Manager (host-side function) and the input data associated with that work request has been transferred to the GPGPU device. This generated function does the work of actually triggering the kernel on the device itself (i.e. causing the generated device-side function to execute).

Device-side Function (DSF): The device-side function serves the same purpose as was described for the Cell-specific device-side function in section 3.3.2. This function is the actual kernel function (in CUDA terminology) that is invoked on the host within the kernel select function and executed on the accelerator device. When this function is invoked, it starts by calculating a unique thread index for each of the GPGPU threads associated with the kernel's execution. This thread index is then used to calculate the element index and, if applicable, the split index that the given GPGPU thread is responsible for. The element index simply refers the "array" of chare objects represented by the kernel's data buffer, with each chare object being one element in that array. It should be noted, each of these operations (index calculations and parameter lookups) tend to be quick index calculations taking constant time. In the case of a different number of splits for each AEM invocation, the element calculation may linear time (in terms of the number of chare objects). As a batch set is created, it is assumed that all AEMs within the set will have the same number of splits. However, as the batch set is being filled in, if the runtime system detects that a different number of splits is used for at least one of the AEMs, this alternate (more expensive) element index calculation that accommodates the varying number of splits is then used. Using the element index, which is specific to the current batch set, the GPGPU thread then determines the locations of the various parameters associated with the specific element it is operating on. With the parameters located, the thread moves on to executing the AEM's function body code. Once the function body has completed, a couple of the threads (from the kernel as a whole) are used to set some diagnostic variables within the batch set's data buffer. These variables are used for tasks such as verifying that the kernel code was actually executed, passing an error value back to the host processor in the event that something goes wrong and/or for debugging purposes, and other tasks as well. Once the kernel has completed executing and the GPU Manager will transfer the batch set's data buffer back into system memory, where the output of the AEM can be accessed by the host core.

Kernel Callback (KCB): Since multiple AEMs are being grouped together in batch sets, with each

batch set being executed using a single kernel, the GPU Manager will trigger a single callback function when the batch set has finished processing. However, in terms of the application code, each of the AEMs that were included within the batch set requires its individual callback function to be called. As a result, a kernel callback function is generated for each of the AEMs and performs two main tasks. First, it transfers any output data back into the chare objects associated with the batch set. Second, it triggers the AEM callback functions on the individual chare objects as specified within the application code.

Chapter 4

Static Load Balancing

The Cell Processor was the first accelerator for which an implementation of the extensions presented in section 3.2 were created. This chapter presents a demonstration of our programming model extensions and runtime system modifications on a cluster that includes both x86-based and Cell-based nodes. To illustrate an example code executing using our extensions, a simple molecular dynamics (MD) application will be discussed. The workload contained within the code will be load balanced using a static load balancing method implemented within the application code. This static load balancing work was only intended as a first attempt at load balancing, to give us insight into how the process might be automated and what considerations should be made by the runtime system and load balancing mechanisms within the context of a heterogeneous cluster. Dynamic load balancing will be discussed in chapter 5.

To demonstrate the programming model extensions and the runtime system modifications that have been presented in this work, we created a simple molecular dynamics (MD) code that makes use of accelerated entry methods (AEMs). We choose a simple MD code for a couple of reasons. First, particle interaction codes are an importance class of codes in HPC. Second, we specifically create a simple molecular dynamics code so that we can easily isolate cause and effect when analyzing performance. As such, it is not our goal to demonstrate that we can write an MD code with a high measured flop rate, but rather to demonstrate the application of our approach and to identify various considerations that should be kept in mind when applying our approach within the context of Charm++ applications and/or other programming models. That said, the improvement of the implementation is a continuing process, so we hope to continue increasing performance in the future beyond what is presented here. Before discussing the application of our extensions along with the static load balancing scheme, we first describe the application to which they will be applied.

A summary of the work is presented here both for the reader's reference and because its inclusion helps give a more complete picture of our approach/work. For a more in-depth discussion of the static load balancing experiments, please refer to the original publications [63, 64].

Within this chapter, the term "load balancing" will refer to across host core load balancing. All of the accelerated entry methods will be executed on the accelerator device (i.e. the SPEs within a Cell processor).

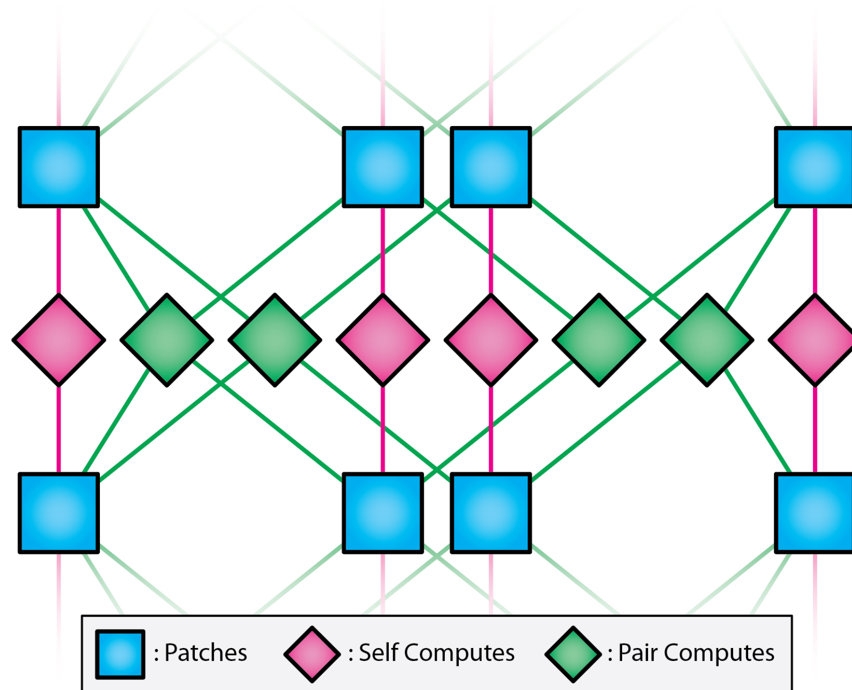


Figure 4.1: The interaction of objects, or flow of data between those objects, in a single timestep of the simple MD code.

As such, no load balancing between the host cores (PPEs) and the accelerator device is occurring. Chapter 5 will cover dynamic load balancing between host cores and accelerator devices, along with across host core load balancing, in greater detail.

4.1 Description of Molecular Dynamics Code

The simple molecular dynamics code used in this work is loosely modeled after the nonbonded force computation in the production molecular dynamics code, NAnoscale Molecular Dynamics (NAMD) [53, 74]. However, NAMD makes use of a cutoff based algorithm, while the simple MD code used in this work does not. Instead, it uses an all-to-all algorithm. As stated above, the regular nature of the all-to-all algorithm will make isolating the causes of various effects seen in the execution of the application easier to identify. The simple MD code does not implement all of the force computations that NAMD implements. In particular, only Coulomb forces are calculated between the particles in the simple MD application.

There are several types of chare objects used within the simple MD code. The overall list of particles is divided evenly into chare objects called *patches*. Additionally, there are two types of chare objects that do force calculation, *self compute* objects and *pair compute* objects. See figure 4.1 for an overview of how

the various objects within the simple MD code interact with one another. There is exactly one self compute object for each patch objects. The self compute objects take a single list of particles, namely the list of particles within the corresponding patch object, and calculates the forces between the particles within that list. There is one pair compute object for each pair of patch objects. The role of the pair compute object is to calculate the forces that the particles in the first patch induce on the particles in the second patch, and vice versa. There is one other type of object, the *patch proxy* objects. Since each host core is likely to have multiple compute objects that require a given patch object's particle data, the patches communicate their particle data to the compute objects through these patch proxy objects. The patch proxy objects are an application-specific construct and not to be confused with the idea chare object proxies within Charm++ as a whole. For any given patch object, there will be a patch proxy object on each host core. The patch proxies are implemented via Charm++ groups. This allows a single message to be sent from a patch object to any remote host core containing one or more compute object that requires that patch object's particle data. In a similar manner, only a single message is required when returning the resulting force data from the host core containing the compute objects (i.e. producing force data) back to the patch object.

While we would like to include the code for the simple MD code to give the reader a more complete idea of how applications are written using our extensions, there is simply too much code in the simple MD code to include it all here. However, to give the reader at least a sense of how accelerated entry methods are used in application code, the accelerated entry method *Patch::integrate* is shown in figure 4.2. The integrate function is invoked once all of the force data has been passed back to the patch object for a given timestep. the integrate function applies the force data calculated by the compute objects to the particles within a patch object. By the time this function is called, all of the forces from all of the compute objects that interact with the given patch object have been accumulated into a single list of force vectors, with one force vector per particle (in particular, *f_x*, *f_y*, and *f_z* in figure 4.2). The integrate function simply loops through the particles in the patch and applies the forces to the particles, updating both the velocities and the positions of the particles.

4.2 Heterogeneous Cluster Description

The cluster being used to run the simple MD code in this situation is a cluster comprised of 4 Cell-based IBM QS20 blade servers ("*blades*"), 4 Cell-based Playstation 3s ("*PS3s*"), and a single dual-core x86-based node ("*x86s*"). Each blade server contains two Cell processors, each having a single Power Processing Element (PPE) and 8 Synergistic Processing Elements (SPEs). Both core types have a clock rate of 3.2 GHz and can

```
entry [ accel ] void integrate()[
   readonly : int numParticles <impl_obj->numParticles>,
  readwrite : float p_x[numParticles] <impl_obj->particleX>,
  readwrite : float p_y[numParticles] <impl_obj->particleY>,
  readwrite : float p_z[numParticles] <impl_obj->particleZ>,
   readonly : float p_m[numParticles] <impl_obj->particleM>,
   readonly : float f_x[numParticles] <impl_obj->forceSumX>,
   readonly : float f_y[numParticles] <impl_obj->forceSumY>,
   readonly : float f_z[numParticles] <impl_obj->forceSumZ>,
  readwrite : float v_x[numParticles] <impl_obj->velocityX>,
  readwrite : float v_y[numParticles] <impl_obj->velocityY>,
  readwrite : float v_z[numParticles] <impl_obj->velocityZ>
] {

  int i;
  simdia_vecf delta_time_vec = simdia_vspreadf(TIME_PER_STEP);

  simdia_vecf* p_x_vec = (simdia_vecf*)p_x;
  simdia_vecf* p_y_vec = (simdia_vecf*)p_y;
  simdia_vecf* p_z_vec = (simdia_vecf*)p_z;
  simdia_vecf* p_m_vec = (simdia_vecf*)p_m;
  simdia_vecf* f_x_vec = (simdia_vecf*)f_x;
  simdia_vecf* f_y_vec = (simdia_vecf*)f_y;
  simdia_vecf* f_z_vec = (simdia_vecf*)f_z;
  simdia_vecf* v_x_vec = (simdia_vecf*)v_x;
  simdia_vecf* v_y_vec = (simdia_vecf*)v_y;
  simdia_vecf* v_z_vec = (simdia_vecf*)v_z;
  const int numParticles_vec = numParticles / simdia_vecf_numElems;

  // For each particle (vector)...
  for (i = 0; i < numParticles_vec; i++) {
    // Update the velocity : v_new = v_old + (F/m)dt
    v_x_vec[i] += delta_time_vec * (f_x_vec[i] / p_m_vec[i]);
    v_y_vec[i] += delta_time_vec * (f_y_vec[i] / p_m_vec[i]);
    v_z_vec[i] += delta_time_vec * (f_z_vec[i] / p_m_vec[i]);
    // Update the position : p_new = p_old + (v)dt
    p_x_vec[i] += delta_time_vec * v_x_vec[i];
    p_y_vec[i] += delta_time_vec * v_y_vec[i];
    p_z_vec[i] += delta_time_vec * v_z_vec[i];
  }

  // For each particle (scalar)...
  for (i = numParticles_vec * simdia_vecf_numElems; i < numParticles; i++) {
    v_x[i] += TIME_PER_STEP * (f_x[i] / p_m[i]);
    v_y[i] += TIME_PER_STEP * (f_y[i] / p_m[i]);
    v_z[i] += TIME_PER_STEP * (f_z[i] / p_m[i]);
    p_x[i] += TIME_PER_STE * v_x[i];
    p_y[i] += TIME_PER_STE * v_y[i];
    p_z[i] += TIME_PER_STE * v_z[i];
  }
} integrate_callback;
```

Figure 4.2: Integration code from the simple molecular dynamics code.

Core Type	Peak Gflop/s	Memory Type	System Memory Access	SIMD Extension
x86	14.88	Cache Hierarchy	Yes	SSE
PPE	25.6	Cache Hierarchy	Yes	AltiVec
SPE	25.6	Scratchpad	No	Modified AltiVec

Table 4.1: Differences between the core types in the cluster.

complete up to 8 flops per clock cycle, resulting in a peak rate of 25.6 Gflop/s per core or a total peak rate of 230.4 Gflop/s for an entire blade Cell processor. Each PS3 contains only a single Cell processor, also with a clock rate of 3.2 GHz. Unlike the blade Cells, the PS3 Cells have only 6 SPEs available to user applications (there are actually 7 SPEs, but one of them is dedicated to the operating system). Each core has the same peak flop rate of 25.6 Gflop/s, resulting in a total peak flop rate of 179.2 Gflop/s available to the user for the entire processor. Each of the x86 cores have a clock speed of 2.0 GHz and are able to complete 8 flops per clock cycle, giving each x86 core a peak flop rate of 16 Gflop/s and the x86 processor a total peak flop rate of 32 Gflop/s.

In total, the cluster that we executed the simple MD code on has three different types of processing cores, x86s, PPEs, and SPEs. Table 4.1 summarizes the differences between the various cores in the cluster. There are two types of memory configurations, scratchpad memories (local stores in the SPEs) and typical memory hierarchies containing several levels of cache between the pipeline and system memory (the x86s and the PPEs). Further, each of the core types presents a separate SIMD extension. The x86 cores use SSE. The PPE cores use AltiVec/VMX. The SPE cores present a set of instructions that are similar to the AltiVec instructions on the PPEs, however, the actual set of instructions available differ. More generally speaking, the instruction sets differ between the SPE and PPE cores (i.e. not just the SIMD instructions), even though these core are located within the same processor chips.

4.3 Results

To ensure that the results are presented clearly, this section is broken down into several sub-sections. First, we establish a performance baseline by executing the simple MD code on a single x86 processor and on a single Cell processor. From there, we will move on to scale the application using homogeneous sets of nodes. To be clear, the nodes will internally be heterogeneous (i.e. using Cell processors), but the nodes themselves will be identical to one another. This simplifies load balancing in that each node should receive an equal amount of work. Finally, we will execute the application on a heterogeneous set of nodes, making use of all the nodes in the heterogeneous cluster.

Please note that the same application code is being used in all cases, despite the differences between

the various cores that will be used, demonstrating that the extensions in this work do increase code portability. Further, figure 4.2 shows that the level of complexity required of the programmer to achieve this portability does not increase significantly, and thus, our programming model extensions and runtime system modifications decrease the burden placed on the programmer.

4.3.1 Establishing a Performance Baseline

The application configuration we chose for creating a baseline measurement uses 144 patches with 640 particles per patch, for a total of 92160 particles. This problem size is chosen because it resembles the ApoA1 particles systems that is commonly used as a benchmark system for measuring the performance of the production MD code NAMD. The ApoA1 particle system contains 92224 particles spread across 144 patches. Further, the simple MD application will create less than a factor of 2 more computes that NAMD does when using 64 host cores on a homogeneous cluster. We cite the 64 host core case for NAMD, since we will be using a total of 66 cores on the test heterogeneous cluster described above once the simple MD code is scaled to use each node in the cluster (more on this in section 4.3.2). The number of compute objects that either application creates is directly related to the amount of parallelism within the application. Therefore, the simple MD code is creating less than a factor of 2 more parallelism on 66 cores compared to the NAMD application executing a similar particle system configuration on 64 cores. The simulation will be executed for 128 timesteps on all hardware configurations reported below.

As a first step, the simple MD code is executed using only the x86 cores. The measured performance when using both x86 cores is approximately 10.09 Gflop/s, or approximately 31.5% of the combined peak flop rate.

Knowing the performance of the simple MD application executing on the two x86 cores, we now move on to measure the performance using a single Cell processor. For this purpose, we use a single QS20 Cell processor, which has 8 SPE cores, giving the program access to a total of 9 processing cores. The performance on a single QS20 Cell processor is approximately 24.46% of the peak flop rate (50.09 Gflop/s). This calculation only includes the SPEs since the SPEs are doing all the physics calculations. In the next section, table 4.2 will include calculations with and without the PPEs.

To place this percentage of peak number within a meaningful context, the generated assembly code from the pair compute object's accelerated entry method that performs the force computation was evaluated. This AEM was chosen since it accounts for the great majority of the total work being performed by the application. The AEM has two loops, one nested within the other, referred to as the *outer* and *inner* loops. Each loop iterates over one of the lists of particles provided by the associated patch objects, calculating

force vectors for all combinations of atoms between the two lists. The inner loop contains 54 instructions, 29 of which are SIMD instructions. A total of 124 flops occur per inner loop iteration. According to the SPE timing tool provided within the Cell SDK, the 54 instructions of the inner loop take approximately 56 cycles to execute. Note that the time to access an SPE's local store from within the same SPE's pipeline is a constant value, so load and store instructions do not take a variable amount of time, as is the case when cache hierarchies are being accessed. From these numbers, we can calculate that the flop rate for a single inner-loop iteration is 124 flops in 56 cycles, or approximately 2.214 flops per cycle on average. With a peak flop rate of 8 flops/cycle, this calculated rate of a single inner-loop iteration represents 27.68% of an SPE's peak performance. 27.68% represents a good estimate for an upper bound on the expected performance since the great majority of the work performed by the application results from the execution of iterations of this inner loop. Further, 27.68% represents an unobtainable upper bound since it assumes only the inner loops are being executed. This upper-bound estimate does not account for any unavoidable overheads such as outer loop instructions, required DMA transactions (which, even if perfectly overlapped, still need to be issued and completed within the instruction stream), and so on. The measured rate of 50.09 Gflop/s (24.46% of peak) is 78.55% of the calculated upper-bound derived from the pair compute object's inner loops (63.77 Gflop/s or 27.68% of peak on a single QS20 Cell processor, including the PPE core).

We would like to point out that the upper bound on the SPEs is lower than the measured rate on the x86 cores. The x86 cores are capable of issuing multiple SIMD vector instructions per cycle, for example both a SIMD add and a SIMD multiply which are not data dependent on one another. However, in the case of the SPE cores, achieving peak performance requires that an SIMD fused-multiply-add be issued every cycle. Further, if SIMD fused-multiply-add instructions are not being issued, then the measured performance of an application cannot achieve greater than half the peak performance of an SPE core. As stated above, the inner loop of the pair compute object's main compute AEM has 54 instructions, 29 of which are SIMD instructions. Of this 29 SIMD instructions, only 2 are fused-multiply-adds. Although, there are several multiply and add instructions that are not fused. The x86 cores are capable of executing closer to peak with just the presence of SIMD add and multiply instructions, while the SPE cores are not. This is not the only reason we see a performance difference between the two core types. However, we believe that this effect is worth noting since it points out that the design difference between the core types allows the x86 cores to have a wider variety of instruction mixes within an instruction streams that are capable of obtaining a higher percentage of the core's peak performance.

# Nodes	# PPEs	# SPEs	Peak Gflop/s	Measured Gflop/s	% Peak (All)	% Peak (SPEs)
1 PS3	1	6	179.2	21.53	12.01%	14.02%
2 PS3	2	12	358.4	77.33	21.58%	25.17%
3 PS3	3	18	537.6	111.03	20.65%	24.10%
4 PS3	4	24	716.8	146.61	20.45%	23.86%
1 QS20	1	8	230.4	50.09	21.74%	24.46%
2 QS20	2	16	460.8	96.63	20.97%	23.59%
3 QS20	3	24	691.2	142.01	20.54%	23.11%
4 QS20	4	32	921.6	178.22	19.34%	21.76%
# Nodes	# x86 Cores		Peak Gflop/s	Measured Gflop/s	% Peak	
1	2		16.0	5.07	31.69%	
2	2		32.0	10.09	31.53%	

Table 4.2: Performance of the simple MD code scaled to multiple Cell-based compute nodes.

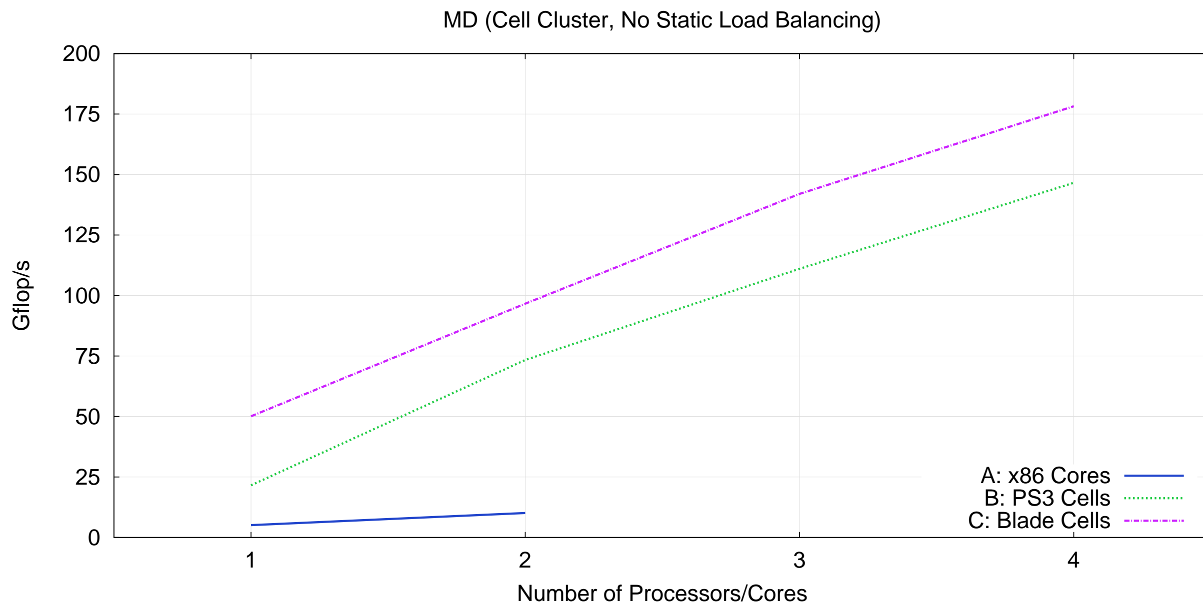


Figure 4.3: Performance of the MD application as it is scaled on QS20 Cells, PS3 Cells, and x86 cores.

4.3.2 Homogeneous Scaling

Table 4.2 shows the performance of the simple MD application as it is scaled from one to four Playstation 3s (PS3s) and from one to four QS20 Cell blades. Figure 4.3 plots the measured flop rates presented in table 4.2. At a high level, the results presented in table 4.2 show that the MD's application code, written using our programming model extensions, is portable between each type of node within the heterogeneous. In the case of the QS20 Cell blades, only a single Cell processor is being used per node (this will be discussed further in section 4.3.3). The workload of the application is being evenly distributed between the nodes. Further, all of the accelerated entry methods are being directed to the SPEs.

There are two columns for % peak in table 4.2, *all* and *SPEs*. The *all* column shows the percentage of peak that is being achieved when all of the cores, both PPEs and SPEs, are being considered. The *SPEs* column shows the percentage of peak that is being achieved when only the SPE cores are being considered. This column as been included since all of the AEMs are being directed to the SPEs, meaning all of the physics calculations are being performed on the SPEs.

The single PS3 case is a bit of an anomaly. The simple MD code executing on a single host core (even without accelerator support added in) requires approximately 200MB of memory. This value alone is already approaching the amount of physical memory on a single PS3 (256MB). When also considering other sources of memory usage, such as our runtime system additions to support Cell, the operating system, the Cell SDK, and any other applications/services executing on the PS3 at the same time, this serves as a strong indication that the amount of physical memory available in the single PS3 case is the main factor contributing to the decreased performance.

To verify that memory was indeed the issue for these runs, we executed the 1 QS20 Cell, 1 PS3 Cell, and 2 PS3 Cell configurations using the 'time' command, which reports page faults. For the 1 QS20 Cell case, there were approximately 50.0 thousand page faults (all reported as minor). For the 2 PS3 Cell case, the page fault counts were 34.7 thousand for the first PS3 and 26.7 thousand for the second PS3 (again, all reported as minor). For the 1 PS3 Cell case, the page fault count was approximately 119.9 thousand (12.5 thousand report as major and 107.4 thousand reported as minor). According to the manual entry (also known as the "man page") for the 'time' command, major page faults are those "where the page has to be read in from disk." Minor page faults "are faults for pages that are not valid but which have not yet been claimed by other virtual pages. Thus the data in the page is still valid but the system tables must be updated." From these measurements, the single PS3 case is the only configuration experiencing major page faults and experiences more significantly more minor page faults per Cell processor.

For the remaining entries, the performance levels seen are what one might expect from the simple MD

application as it is scaled to multiple nodes. Recall that the 27.68% value previously given as an upper bound on the performance expected from the MD code, based on the serial code within the pair computes' inner loops executing on a single SPE, is an unobtainable upper bound. One should expect that value to be even further removed from the actual measured performance as other overheads are taken into account, such as the coordination of multiple SPEs per Cell processor, network communication between multiple Cell processors, and other factors. While there are certainly optimizations remaining for our implementation, the performance levels seen in table 4.2 suggests that our approach for creating portable, parallel code in the presence of heterogeneous hardware allows the serial portions of the code to achieve a high level of their potential performance levels.

Until now, the term "heterogeneous" within this chapter has been used to referred to hardware configurations in which the nodes have multiple types of processor cores, but the nodes themselves are identical. We also wish to explore cases where the nodes are heterogeneous relative to one another, thus increasing the level of heterogeneity even further.

4.3.3 Heterogeneous Scaling

Before discussing the application's performance using a heterogeneous set of nodes, it is worth noting that, through the use of our programming model extensions, there is no architecture specific code included within the MD application code. The runtime system is taking care of directing work and moving the data associated with that work between the various processing cores, host and accelerator alike. In fact, there are several differences between the processing cores, including three instruction set architectures (ISAs), two types of memory structures (cache hierarchies and private scratchpad memories with the use of direct memory accesses), three different SIMD instruction extensions, varying amounts of system memory, differing numbers of SPEs between Cell-based nodes, and so on. Despite all these differences, the MD code does not include any architecture specific code mixed in with the application code itself to account for these differences.

Figure 4.4 shows the performance of the simple MD application as it is scaled to use all of the cores within the heterogeneous cluster described in section 4.2. Recall that this cluster is made up of four Playstation 3s, four QS20 Cell blades, and one dual core x86-based node. Data passed between the x86 and Cell processors is being automatically modified by the runtime system, as described in section 6.1, to account for the architecture differences between the processors.

There are two reference plots in figure 4.4. The first reference plot, $B*2$, simply doubles the measured values from plot B ("PS3 Cells") in figure 4.3. The idea behind including this reference plot is to give the reader an idea (just an estimate) of what one might expect the performance to be if the the number

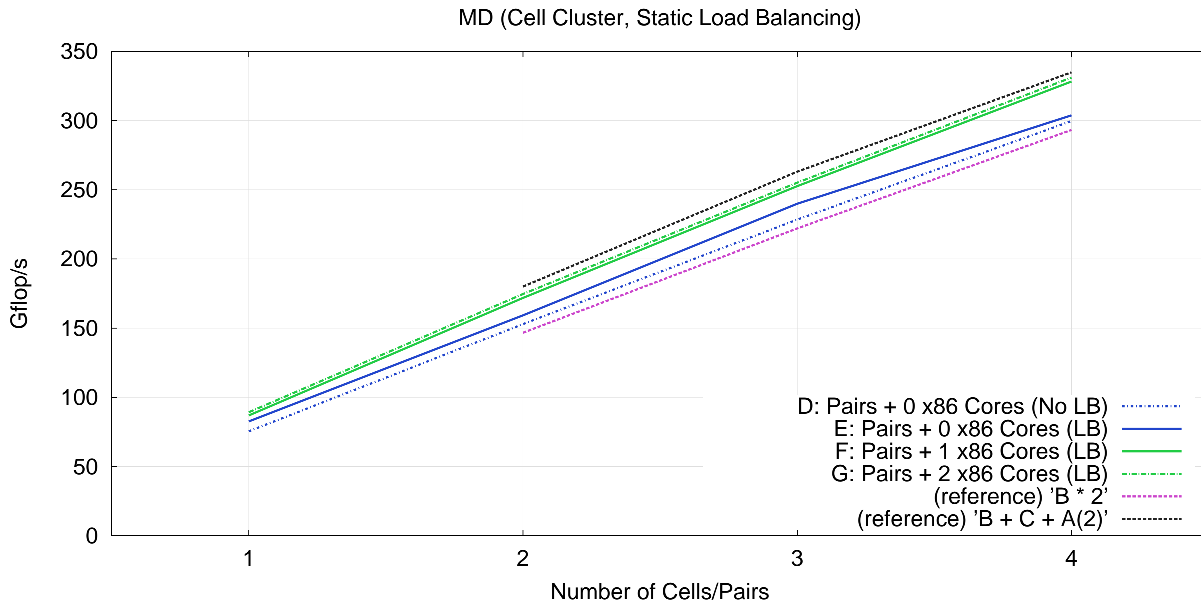


Figure 4.4: Performance of the simple MD application scaled to use all the nodes within a heterogeneous cluster, making use of x86-based and Cell-based processors.

of PS3s used at each point were doubled (ideally). The second reference plot, $B + C + 2 * A(2)$, adds the corresponding points from plots B ("PS3 Cells") and C ("Blade Cells") in figure 4.3, along with adding the constant value of $A(2)$ (i.e. the performance on two x86 cores). The idea behind including the second reference plot is to give the reader an idea of what to expect as the MD application is scaled to use each of the nodes within the heterogeneous cluster.

The first plot, D, shows the performance of the MD application as it is scaled from one to four *Cell pairs* without using any load balancing, where a Cell pair includes one PS3 Cell and one blade Cell. Notice that the performance of D is not much greater than the performance of the reference plot plot $B * 2$. This indicates that the performance of configuration D is likely not benefiting from the extra SPEs available to the blade Cells. Since the workload is not load balanced, all of the Cells, blade and PS3 alike, receive an equal portion of the overall workload. As such, the extra SPEs available to the blade Cells allow the blade Cells to finish their portion of the overall workload faster. However, since the overall performance is limited by the slowest processors, the PS3 Cells limit the application's overall performance since they take longer to execution their portions of the overall workload.

The second performance plot in figure 4.4, E, shows the performance of the MD application as it is scaled from one to four *Cell pairs* and applying static load balancing. Notice that the performance of the MD application isn't much better that the performance in plot D. In particular, notice that the performance difference between plots D and E is very low when using four Cell pairs. Generally speaking, the application

of static load balancing is helping performance. However, as more and more Cells are used, the benefit of static load balancing diminishes.

Plots F and G add one and two x86 cores, respectively, to the Cell pairs used in plot E (a constant number of x86 cores are added, regardless of the number of Cell pairs in use). In particular, notice that the performance of plot F is noticeably greater than the performance of plot E, especially as the application is scaled to a greater number of Cell pairs. The reason for this stems from the fact that the x86 cores are better at communication that the PS3s or the blades. In fact, the blades are noticeable worst at communication than either the PS3s or the x86 cores, in terms of the time it takes to send a given amount of data. Our previous publication details the cause for this difference [64]. As part of this discussion, we will simply say that communication is the cause of the difference. Since the objects have different communication to computation ratios, the static load balancer can take advantage of the x86 cores' increased communication performance for the chare objects with higher communication to computation ratios (the patch objects). The objects with a lower communication to computation ratio (the compute objects) are then mapped to the PS3s and blade Cells, which are better at performing computation. The performance difference between the points $E(4)$ (303.86 Gflop/s) and $G(4)$ (331.13 Gflop/s) is 27.27 Gflop/s, which is significantly greater than (nearly triple) the measured performance on two x86 cores (10.09 Gflop/s). The application is capable of scaling better when x86 cores are also used compared to only using Cell processors. The measured performance of point $G(4)$ is 22.59% of the combined peak rate of all the cores in use (8 PPEs, 56 SPEs, and 2 x86 cores, for a total of 66 cores with a combined peak rate of 1465.6 Gflop/s). Recall that an unachievable upper bound of the application executing on a single SPE (without including any DMA, communication, or other overheads) is 27.68% of peak. While a direct comparison cannot be made between a single SPE and the hardware configuration used for point $G(4)$, this upper-bound should provide some context in which to access the performance achieved at point $G(4)$ since the majority of cores in use are SPE cores.

4.4 Static Load Balancing Summary

In this chapter, the application of the programming model extensions and runtime system modifications discussed in chapter 3 were applied to a simple MD application. The MD application was then executed using various combinations of nodes within a heterogeneous cluster made up of QS20 blades (blades), Playstation 3s (PS3), and a dual-core x86-based node (x86 cores). In section 4.3.2, homogeneous executions were made (i.e. a single run only used nodes that were alike one another, even though each node may or may not have been internally made up of a heterogeneous set of cores). In section 4.3.3, heterogeneous executions

were made up to, and including, all of the nodes within the heterogeneous cluster. Various aspects of these executions were discussed, including pointing out that the performance differences between the x86 cores and the Cell processors allowed the application to scale better.

When using all of the nodes within the the heterogeneous cluster, the application is making used of a total of 66 cores (8 PPEs, 56 SPEs, and 2 x86 cores). In this configuration, the application achieves a measured performance rate of 331.13 Gflop/s, or 22.59% of the combined peak rates of all the cores in use. When executing on this hardware configuration, the application is making use of 3 different types of cores (PPE, SPE, and x86), 3 different ISAs, 3 different SIMD instruction extensions, and 2 types of memory structures (cache hierarchy and scratchpad memories via DMA transactions). Despite all the differences between the cores being used by the application, the application does not include any architecture specific code. Rather, it makes use of the programming model extensions and runtime system modifications presented in chapter 3. Further, the set of AEMs that were issued to the host core(s) where executed sequentially (one-by-one), while the AEMs there were issued to the SPEs were streamed through the SPEs, with data movement overlapping with computation within each SPE and parallelism occurring across SPE cores.

Again, the contents of this chapter were previously discussed in much greater detail in our previous publications [63, 64]. If the reader wishes to have more insight into this topic, please refer to our previous publications. The results have been summarized here since they are highly related to the overall body of work surrounding support for accelerator technologies and heterogeneous computing with Charm++.

Chapter 5

Dynamic Load Balancing

This chapter will discuss the load balancing of workload across a heterogeneous set of processing cores. In the previous chapter, we discussed using a static load balancing technique to a molecular dynamics (MD) application. Using the same programming model extensions, which allow pieces of work to be mapped to various types of cores, a natural next step is to understand what a runtime system can do in terms of automating the load balancing process at runtime.

This chapter will be referring to two types of load balancing strategies. The first type of load balancer is the type that already exists within the Charm++ runtime system. In this document, these load balancers will be referred to as *across host core* load balancers. The second type of load balancer balances work between a given host core (PE) and its associated accelerator device and will be referred to as *accelerator* load balancers. Together these two types of load balancers form a multi-tier approach to load balancing structure that will be discussed in further detail as this chapter continues.

In chapter 4, the experiments were performed on a cluster that contained both x86-based and Cell-based nodes. However, in recent years, the Cell processor has seen a decline in its popularity. Due to the declining popularity of the Cell processor, along with our desire to demonstrate our approach on multiple types of accelerator devices that are fairly different from one another, we will make use of CUDA-based GPGPUs within this chapter.

Chapter 4 used a static approach to load balancing, where the application code included the load balancing mechanism directly. In this chapter, we will discuss our efforts to push the load balancing mechanism into the runtime system with the goal of automatically load balancing a workload across a given set of host cores and accelerator devices. First, we plan to consider cases where a given workload will be shared between cores, host and accelerator alike, within a given node. We will also explore incorporating multiple nodes within a heterogeneous cluster. We also plan to consider cases involving interference, since workstations that incorporate accelerators are sometimes used as shared resources. If multiple applications are making used of the same hardware resources (i.e. accelerator cores) they will likely interfere with one another in terms of their performance. We begin by discussing the mechanisms that have been included within the runtime

system that are related to our dynamic load balancing efforts.

5.1 Accelerator Load Balancing Strategies and the Accelerator Manager

Several accelerator load balancing strategies have been included within the accelerator manager. These accelerator load balancing mechanisms act locally to a given host core and accelerator device pairing, though they may or may not coordinate across host cores. Some of the load balancing strategies are fairly simple and mainly intended to provide basic functionality and/or to be useful for diagnostics, measuring, and testing purposes, while other strategies dynamically balance the workload with the goal of increasing the performance of the application. The choice of which accelerator load balancing strategy the runtime system will use for a given application is specified on the command line when the application is launched. In addition to allowing the use of a single accelerator load balancing strategy, the accelerator manager also has the ability to queue several strategies, allowing later strategies to make use of the output of earlier strategies. This is particularly useful in implementing profilers, which measure various aspects of the application for the explicit purpose of allowing later strategies to make use of the information.

All of the accelerator load balancing strategies within this section make use of *dynamic assignment*. By "dynamic assignment" we mean that the location of any given accelerated entry method will be determined as the AEM is being invoked on the target chare object based on current circumstances, rather having the AEMs for each object have a fixed location for execution. For example, consider the case of an application that has timesteps, and recall that the order of message delivery is not guaranteed in the Charm++ programming model (i.e. if message A was sent before message B, there is no guarantee that A will arrive before B, even if both messages are sent from processor X and received by processor Y). Consider a situation where the same AEM is invoked on chare object X once per timestep, and that it just so happens that the AEM is invoked near the beginning of timestep i and at the end of timestep j. Since the conditions under which the AEM is being invoked on chare object X in each timestep are different (i.e. the start versus the end of the timestep), under dynamic assignment, the location where the AEM is actually executed (i.e. host core versus accelerator device) may be different for timestep i and timestep j. In our approach, it is also possible to use *static assignment*. The subject of static assignment will be discussed within the context of persistent data in section 6.3.

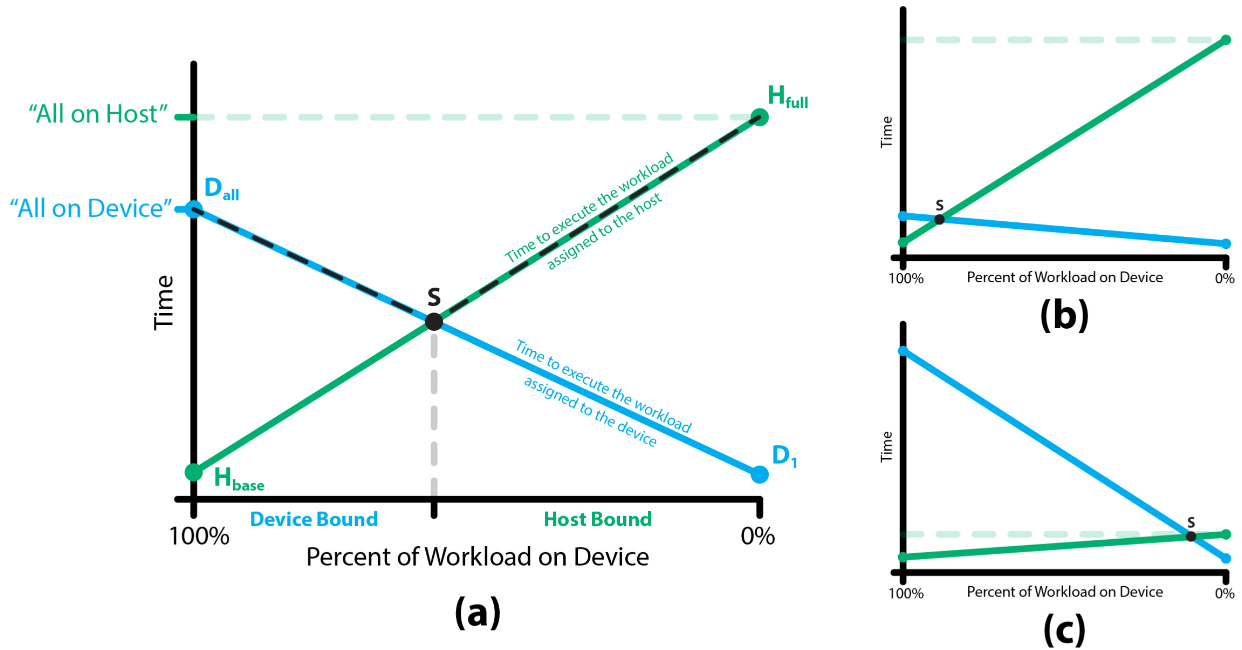


Figure 5.1: Balancing a workload across the host and accelerator.

5.1.1 Modeling Workload Balance

While different load balancing strategies go about balancing the workload of an application in different ways, there is a single model that they all used to *understand* what the balance point should be. For the most part, the discussion of the load balancing strategies will be framed in terms of *the percentage of the workload that is offloaded to the accelerator device*, or just the *percent device* value for short. Since the accelerator load balancing strategies act locally on a single host core and accelerator device pairing, we present a model that describes the effects of dividing a given workload between two different types of processing elements.

Comparing Workloads Between Two Processing Elements

Figure 5.1(a) illustrates the expected outcome of dividing a given workload between a single host core and a single accelerator device. The horizontal axis represents the percentage of the workload that is being passed to the available accelerator device (i.e. the percent device value). Starting from the right edge of the graph and moving to the left, a larger and larger portion of the workload is being assigned to the accelerator device. Conversely, starting on the left edge of the graph and moving towards the right, a greater and greater portion of the workload is being assigned to the host processor. The vertical axis represents execution time. Note that a lower value indicates higher performance since the performance of an application is inversely proportional to its execution time, assuming a fixed workload. There are two plots, one for the host core

(green) and one for the accelerator device (blue).

Figure 5.1(a) is drawn in such a way as to represent a case where the accelerator device performs a given workload only slightly faster than the host core. As such, the best performance (lowest execution time) would be achieved when the workload is almost evenly divided between the host and accelerator device. Figures 5.1(b) and 5.1(c) show how one would expect the graph to look if the accelerator device were significantly more powerful and if the host core were significantly more powerful, respectively. Further, there is a black dotted line that overlaps with both plots that represents the maximum of the two plots for each percent device value. Note that the intersection of these two lines represents the minimum execution time for the given calculation and moves in such a way as to indicate a higher percentage of the workload should be placed on the processing element that provides the greatest performance relative to the other processing element. This intersection point is referred to as S (the *sweet spot*). Also note that the absolute magnitude of the slope of the plot associated with the more powerful processing element approaches zero (i.e. is more horizontal) as the performance of that processing element increases relative to the other processing element.

There are two points on the host core's plot that we used to define it: H_{full} and H_{base}. The H_{full} point represents the time to execute the given workload when the entire workload is being executed on the host core, leaving the accelerator device unused (i.e. percent device equals 0%). Conversely, the H_{base} value represents the amount of time that the host core takes to execute its portion of the workload when all of the accelerated entry methods within the workload are being executed on the accelerator device (i.e. percent device equals 100%). Note that, even when all of the AEMs are being executed on the accelerator device, there are still non-accelerated entry methods that host core is required to execute, which we refer to as the base load of the host core.

In a similar manner, there are two points that we used to define the plot for the accelerator device: D_{all} and D_1. The D_{all} point represents the time to execute all of the accelerated entry methods on the accelerator device (i.e. percent device equals 100%). The D_1 point represents the time it takes to execute a single AEM on the accelerator device. Note that the time to execute zero AEMs on the device will always result in zero execution time for the accelerator device, since unlike the host core, the accelerator device does not have any duties beyond executing the AEMs that have been assigned to it. However, the time to execute a single AEM represents the minimum amount of time required to execute any number of AEMs on the accelerator device. Referring only to the execution of AEMs, the speedup of the accelerator device relative to the host core can be defined as $Speedup_{D/H} = D_{all}/H_{full}$.

Note that we are idealizing the plots in figure 5.1(a) in order to illustrate the model. However, in reality, these plots may not be straight lines for a variety of reasons. The most obvious reason is that

the accelerated entry methods discrete sizes, and as such, the plots are likely to have a stepping nature when viewed closely. However, assuming a significant number of chare objects, and thus AEM invocations associated with them, the plots will appear sufficiently smooth at a macro level. For the accelerator device's plot, there are also device specific considerations that may further increase the stepping nature of the plot. For example, GPGPUs include many cores. As such, shifting a single AEM from the host core to the accelerator device may have a negligible effect on the execution time of all the AEMs because the new AEM's execution completely overlaps with one or more AEMs that were previous assigned to the device ("negligible" since a small amount of overhead may occur, even if the execution completely overlaps). Other sources of non-smoothing effects include the overlap of computation on the accelerator device with data transfers to and from the device, the interleaving of multiple threads on a single core (for GPGPUs in particular), the batching process performed by the runtime system, having multiple types of AEMs present in the application (potentially also a non-linear effect), and so on. However, we assume that there is a sufficient large number of AEMs, and as such these lines will appear to be smooth at the macro level, as our experiments later in this chapter will demonstrate. We point these non-smoothing effects out so that the reader is aware of them, even though we have drawn the plots in figure 5.1(a) as smooth lines.

We further assume that the time to execute a subset of the overall workload increases monotonically as the size of that subset increases, regardless of where the subset is being executed (host core or accelerator device). In other words, we assume that the execution time of a given amount of work on a processing element will always be greater than or equal to the execution time of a smaller amount of work on the same processing element. Given this assumption, there will be a performance sweet spot (S) where the time required to execute the overall workload across both cores is minimized. To the left of S (i.e. a higher percent device value), the accelerator device is overloaded and the workload referred to as "device bound." To the right of S (i.e. a lower percent device value), the host core is overloaded and the workload is referred to as "host bound." Figure 5.2 illustrates that such a point S exists even if the relationship between the amount of work and the time to execute that work is not a linear relationship. When looking at figure 5.2 (or figure 5.1), recall that the workload of the accelerator device increases from right to left (blue plot) and the workload of the host core increases from left to right (green plot). Not all possible combinations are shown (e.g. host being sub-linear and the device being super-linear). However, we believe that these are the likely situations since the same code (i.e. accelerated entry method function bodies) is being used for both the host core and the accelerator device. In cases where the programmer wishes to include architecture specific code, such as mismatch may occur. Regardless, the overall point remains the same. As long as the time taken to execute a workload grows monotonically relative to the size of the workload for both the host

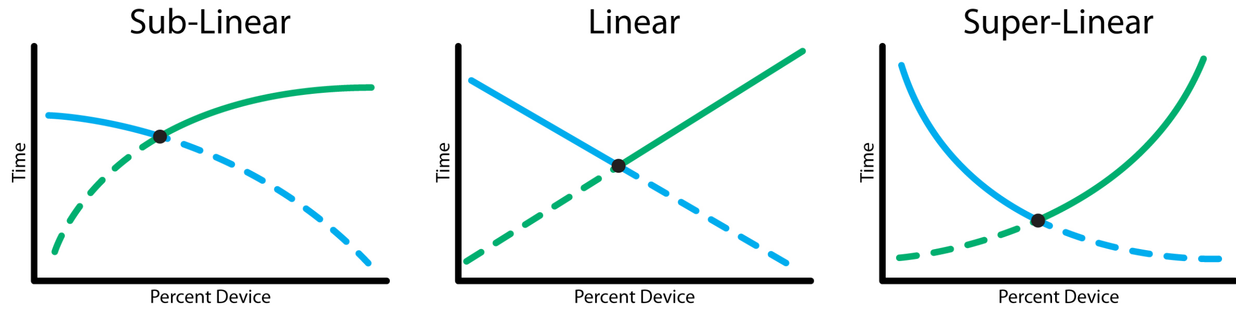


Figure 5.2: Balancing a workload across the host and accelerator.

core and the accelerator device, such a performance sweet spot (S) will exist. In cases where the plots may have flat regions (i.e. are not *strictly* monotonic), S may actually be a contiguous set of values rather than a single value.

It should also be noted that, while we are talking about the graphs in figures 5.1 and 5.2 in terms of having a host core and an accelerator device, the same type of graph would be applicable between any two processing elements. While this discussion has been framed in terms of having a host and accelerator, it is trivial to extrapolate it to compare any two processing elements, regardless of their type.

5.1.2 Base Class for Accelerator Load Balancing Strategies

Before describing the various accelerator load balancing strategies included within the accelerator manager, the base class used to implement the strategies will be discussed. The base class that all accelerator strategies derive from is called *AccelStrategy*. AccelStrategy is an abstract base class that defines an interface (set of virtual functions) that the accelerator manager uses to notify the strategy of events and to request that a strategy make decisions about where to execute each of the accelerated entry method invocations. The following list of virtual functions is intended to give the reader an idea of how the accelerator manager and a strategy interact, but is not an exhaustive list.

- **AccelError decide(int funcIndex, AEMReord *record, AccelDecision &decision, void *objPtr, AccelManager *manager) :** The *decide* function is called by the accelerator manager each time an accelerated entry method is invoked. The funcIndex is the unique value assigned to each type of AEM. The AEMRecord is a data structure that contains information related to each type of AEM, including a *void* strategyVars* parameter that can be used by the strategy to store data between calls to *decide* and information about the number of local objects that are associated with the AEM type. A *decision* data structure is filled in by the *decide* function and represents the decision about where

the particular AEM invocation should be executed. The *objPtr* parameter is a pointer to the actual chare object that the AEM is being invoked on. The *manager* is a pointer to the accelerator manager calling the *decide* function.

- **void notifyIdle(AccelManager *manager) :** The runtime system passes notifications to the accelerator manager to indicate when the host core transitions between idle and busy states. Additionally, a periodic notification occurs each second (approximately). Once the total time (idle plus busy) has accumulated to a given minimum amount, the accelerator manager will pass a notification that idle and busy time statistics are ready to be consumed by the strategy by calling this *notifyIdle* function on the strategy. The notify function can then consume the data and/or reset the statistics to start the accumulation process over again. If reset, the *notifyIdle* function will not be called again until the total time has accumulated pass the minimum point once again. If not reset, the *notifyIdle* function will continue to be called each time a idle/busy transition occurs (or at a periodic rate if such transitions do not occur). The *manager* parameter is a pointer to the accelerator manager.

- **void periodicSample(double value, AccelManager *manager) :** Each time the *takePeriodicSample* function is called on the accelerator manager (refer to section 3.3.1), the accelerator passes this measured value on to the strategy via the *periodicSample* function. The *value* may either be the value passed to the *takePeriodicSample* function or a generated value if no such value was supplied. Generally speaking, the strategies will use the passed in value as a way of gauging if their adjustment(s) to the percent device value(s) are having positive or negative effects on the application's performance. For the strategies that we have included within our implementation, it is assumed that the value passed in is inversely proportional to the performance of the application (i.e. the strategy should be attempting to minimize the value and a lower value indicates better performance than a larger value). However, future strategies may be written to maximize, minimize changes in, or otherwise interpret the passed in value as they see fit. The *manager* parameter is a pointer to the accelerator manager.

- **void notifyLBFinished() :** The *notifyLBFinished* member function is called immediately after across PE load balancing occurs, so that the accelerator strategy can take any appropriate actions. For example, the Sampling strategy, which will be discussed later, resets its history of performance samples since the load balancing process may or may not affect the balance point. If the balance point were to change and the history of old performance samples were retained, then the old values would hinder the Sampling strategy's ability to find the appropriate balance point until the old performance values were replaced by updated values. Further, in some cases, the presence of the old values may

cause the Sampling strategy to avoid resampling certain percent device values altogether by causing the strategy to not reconsider them as possible improvements relative to other percent device values.

- **int safeToRemove(), void aboutToAdd(AccelManager *manager), void aboutToRemove(AccelManager *manager) :** These functions are used by the accelerator manager when attempting to swap strategies. If there are multiple strategies in the queue of strategies, then the *safeToRemove* function is used to indicate to the accelerator manager that the current strategy is in a safe state to be removed from *active duty*. In most cases, this involves the strategy having evacuated all of the AEMs off of the accelerator device and having no batch sets in the process of being created. Except in cases when a strategy can pass percent device values to the next strategy, this requirement of evacuating the accelerator device ensures that application will not hang because of an outstanding and uncompleted AEM batch set is never issued to the accelerator device and not being detected by the next strategy. Since strategies are exchanged infrequently, if at all, within an application's execution, the overhead imposed by this requirement will be minimal over the course of the entire application. The *manager* parameters are pointers to the accelerator manager.

- **int getStrategyType() :** This function returns a unique value associated with each type of accelerator strategy. It is used by the accelerator manager to identify which strategy is in use.

- **void* allocStrategyVars(), void freeStrategyVars(void* vars) :** The *allocStrategyVars* and *freeStrategyVars* functions are used to allocate and free, respectively, the strategy variables associated with each AEMRecord (one per type of accelerated entry method). This mechanism allows for a persistent state associated with each type of AEM to be retained between calls to the *decide* function, as well as the other functions within the strategy.

Again, the list of virtual member functions associated with the AccelStrategy abstract class is just meant to give the reader a sense of how the accelerator manager interacts with the accelerator strategies. It is not meant to be an exhaustive list of all the functionality provided. Further, the declarations of these functions may change over time as more functionality is added. The various accelerator strategies that will be covered in this document have all been implemented on top of the abstract class AccelStrategy, as a direct or indirect child, and make use of the virtual functions listed above.

5.1.3 List of Accelerator Load Balancing Strategies

This section will descrive the various accelerator load balancing strategies that have been developed and added to the Charm++ runtime system as part of this work.

Basic Utility Strategies

There are three basic utility strategies included within the accelerator manager: HostOnly, DeviceOnly, and PercentDevice. The first two strategies are straight forward. The HostOnly strategy, as the name implies, executes any and all accelerated entry methods on the host core. The DeviceOnly strategy does exactly the opposite, executing any and all AEMs on an available device. If no device is available and the DeviceOnly strategy is specified, the host core will be used to execute the accelerated entry methods.

The PercentDevice strategy is used to set a percent device value, as specified on the command line. This specified value is used throughout the execution of the application, allowing the user to select a balance point for dividing the overall workload between the host core and an attached accelerator device. In this strategy, the same percent device value is used for each type of accelerated entry method, which may lead to less than optimal results. However, a benefit of this strategy is that it is simple (i.e. it does not require a lot of overhead to calculate the balance point in an automated way), which can be useful for applications that have short running times, giving any adaptive strategy too little time to adapt. While other strategies may be able to seek a good balance point based on actual measurements of the application, those measurements require the application to be executing for some amount of time while the measurements are being taken.

Stepping Percent Device (*Step*)

The Step strategy begins executing all of the accelerated entry methods within the application on an available device. When this strategy is specified on the command line, a *step size* value is also specified. At regular intervals, the Step strategy will reduce the percentage of AEMs that are executing on the device by the step size specified. As time goes on, the percent device value will continue to drop, until all the AEMs are being executed on the host (i.e. percent device value of 0%). The point of this strategy is to allow a programmer to determine the performance *sweet spot (S)*, or at least a point close to the sweet spot. Once this point is known, the programmer will have a better understanding of how other strategies are performing relative to their application and/or will have an idea of how to set fixed strategies such as the PercentDevice strategy mentioned above. However, because the Step strategy is meant to be a simple diagnostics strategy that gives the programmer an idea of a good balance point between the host cores and the accelerator cores for their application, the same percent device value is used for each type of accelerated entry method. Since the ideal balance point between the host and accelerator cores may involve having a different percent device value for each type of accelerated entry methods, the results of the Step strategy may not be ideal and should only be used as a reference to give the user a general sense of how to balance the overall workload (i.e. to have an idea of how the performance achieved with another strategy is good or bad, in a general sense). Clearly, the

Step strategy is not meant to be used in an actual production setting, as it will spend most of its time at settings other than the *sweet spot (S)*. If the performance of an application executing using the Step strategy were to be plotted over time, one would except that graph to have a single plot resembling the maximum of the two plots presented in figure 5.1(a) from section 5.1.1.

Balancing Busy Time Heuristic (*AdjustBusy*)

The first accelerator load balancing strategy intended to optimize performance is a simple heuristic that will be referred to as *AdjustBusy*. The basic idea is that the runtime system will keep track of the busy times of both the host core and the associated accelerator device and periodically attempt to adjust the *percent device* value so that the host core and accelerator device have the same amount of busy time. The motivation for balancing the busy times comes from figure 5.1(a) in section 5.1.1. Essentially, the plots for the two processing elements meet at the point *S*, where both processing elements are taking the same amount of time to execute their respective fractions of the overall workload. As such, the host core and accelerator device should have an equal amount of busy time, assuming there is no unavoidable idle time (discussed below).

A nice feature of using this heuristic is that it can be implemented locally, without requiring coordination between the host cores. As such, the interference with the application should be minimal. However, a downside is that the heuristic isn't basing the decision of how to divide the workload between the host core and the accelerator device on the actual performance of the application. Obviously, measuring the busy time of processor is a measurement of the application's performance, but it is only a single aspect of the application's performance (both in that each host core only *know's* its own busy time (partial information) and in that idle time is ignored). It is possible for the performance of an application to decrease significantly while, at the same time, a single pairing continues to measure the same amount of busy time. It is also possible for communication latencies to incrase significantly, causing a slowdownin the application, increasing each core's idle time, but not increasing their busy times. It is in this sense that we draw a distinction between measuring the busy time and measuring an application's performance. As we have just eluded to, there are a couple of reasons why simply using busy time without measuring actual application performance could prove to be problematic.

First, in cases where a certain amount of idle time on the host is unavoidable, this idle time is not counted as part of the busy time, but does play a part in defining the lower bound of the application's execution time. To put it a bit more precisely, there may be situations where the value of the host plot at some given point (or set of points) in figure 5.1(a) may include idle time that cannot be avoided. Having the runtime

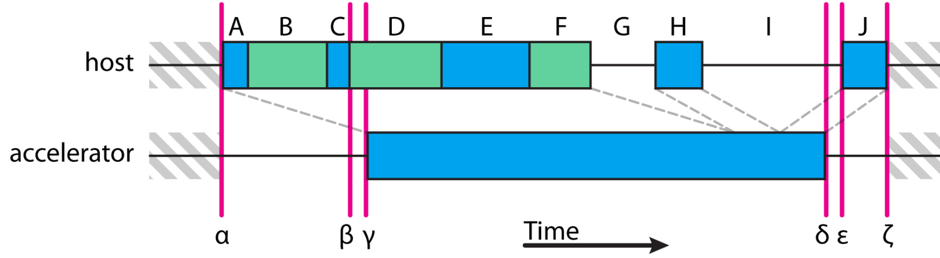


Figure 5.3: An example breakdown of one timestep in an arbitrary application

system adjust for such induced idle time can get very complicated very quickly. In particular, because the data dependencies related to the AEMs execution on the accelerator may or may not be the cause of the idle time. Furthermore, those data dependencies may indirectly cause the idle time, via a chain of data dependencies that crosses node boundaries.

Second, there may be some *startup period* in applications that have timesteps. In this context, the term "startup period" refers to a period of time near the start of each timestep where the host core will be active but the accelerator core will not be active. Such startup periods can be induced for various reasons, including (1) the accelerator device waiting for some data to arrive from a remote node before the first AEM assigned to the accelerator can execute, (2) the inherent latency related to triggering AEMs on the accelerator device, and (3) the batching technique (used by the runtime system to increase granularity) artificially delaying the first several AEMs assigned to the accelerator device until the first batch set is formed. To account for this startup period, and other non-overlapping execution occurring on the host core, the BaseLoad profiler can be used to measure *overlapping* and *non-overlapping* base loads on the host core. The BaseLoad profiler will will be discussed further in section 5.1.3.

To make both the unavoidable idle time and startup period idea a bit more concrete, figure 5.3 illustrates these concepts via an imagined timeline of some arbitrary application executing on system with a GPGPU accelerator device. Since one of the effects is related to applications with timesteps, this example will assume the application has well defined timesteps. The scenario in figure 5.3 is a simple one, but illustrates both of the concepts discussed above. The blue colored blocks represents the work associated with accelerated entry method, while the green blocks represent *other* work that can only be executed on the host core. The timestep starts at time α with the host core beginning the batching process for the AEMs (A) that will be pushed down to the accelerator device. However, if multiple nodes are being used, not all of the data may be present. Therefore, the host may not have enough work accumulated within the current batch set and must wait for the incoming data to show up from the remote location (i.e. C does not occur directly after A). In the meantime, the host core can move on to other pending work while waiting for the remaining AEM

invocations to occur (B). Once the required data arrives, the host core is able to complete the batching process and issue the work to the accelerator device at time β. Note that there is an unavoidable delay between the host triggering the execution of the AEMs and the batch set beginning to execute on the accelerator device at time γ. The time period from α to γ represents the startup period associated with the timestep, where the accelerator device will be idle and the host core will be busy. In this case, there are two contributing factors to the startup period, both the batch process used for GPGPUs and the inherent latency associated with offloading work from the host core to an accelerator device. This scenario is just one example of how the startup period might be structured. In another scenario, the runtime system may create multiple AEM batch sets to be offloaded to the accelerator device, allowing the first batch set to be issued shortly after A completes, before time β, and allowing the AEMs being invoked as part of C to be issued in a later batch set. Moving on, some of the AEMs will execute on the host core, blocks E and H. However, again, there may be some delay in the arrival of the incoming data associated with block H causing a block of idle time, G, to occur on the host processor. This delay could be caused by a number of reasons, including a remote core being overloaded or some other delay in a chain of data dependencies leading up to the AEM being invoked within H. There is also another idle block of time on the host core, I, that is caused because the host core has run out of work to perform and the accelerator device is still busy processing its portion of the workload. Note that, the idle times G and I have occurred for different reasons. In the case of G, the idle time may not be avoidable, depending on the nature of the application itself (e.g. data dependencies delaying a message send). However, in the case of I, the idle time is caused by a workload imbalance between the host core and the accelerator device causing the accelerator device being overloaded relative to the host core. Finally, at time δ, the accelerator finishes its work and notifies the host. After some small, unavoidable delay, the host is notified at time ε and the callback associated with the batch set is triggered on the host core. Once the callback has completed, at time ζ, the timestep is complete.

Keep in mind that this scenario is just one of many possible cases and only meant to give an example of how the scheme of balancing the host core's and accelerator device's busy times within the *AdjustBusy* strategy may fall short. Note the dashed lines relating the busy times between the host and the accelerator device in figure 5.3. In particular, note that the amount of busy time on the host and the amount of busy time on the accelerator are equal to one another. However, it is fairly clear that the block of idle time labeled I could be filled in with some of the work that has been assigned to the accelerator. In other words, there is an opportunity to further reduce the time per timestep, and thus, improve the performance of the application. However, the AdjustBusy strategy, once it has reached this point, will not attempt to the adjust the workload balance between the host core and the accelerator device any further since the busy times of

the host and accelerator device are already equal. One straight forward way to improve on the AdjustBusy time strategy would be to identify and measure the amount of busy time on the host that does not overlap with accelerator. In this particular example, the time periods from α to γ (startup period) and from ε to ζ (non-overlapping callback time, which would only occur for the last callback if there are multiple batches on the accelerator device). Measuring these time periods will be discussed further in section 5.1.3. If the overall busy time of the host is reduced by the amount of busy time contained within these periods (adjusted busy time), then the adjusted busy time of the host is clearly less than the busy time of the accelerator which will result in the AdjustBusy strategy moving work from the accelerator to the device as it attempts to balance the two busy time values.

Sampling Timestep Timing (*Sampling*)

Since the AdjustBusy heuristic discussed in section 5.1.3 does not actually take the performance of the application into account, we developed another accelerator load balancing strategy, called the *Sampling* strategy. The basic idea behind the Sampling strategy is to periodically take note of the time it is taking to complete an equal amount of work (e.g. the time to complete N timesteps) in an effort to directly relate changes in the percent device value to changes in the performance of the application. The assumption is that a decrease in performance will result in an increase in the time to execute each timestep, which in turn will affect the value passed to the accelerator manager via the *takePeriodicSample* function.

Usage: The Sampling accelerator strategy requires the application to have a periodic point in the application (explicitly or implicitly) that occurs at regular intervals such that the work between these points remains constant (or at least as constant as possible). An example of such a point would be the beginning of a set of timesteps where the amount of work performed by the application within each set of timesteps remains fairly constant. The programmer can explicitly mark this point in the application by calling the *takePeriodicSample* function from within their application code at a location that fits the requirements of the periodic point. If this explicit method is used and the programmer generates a custom performance value to be passed into the *takePeriodicSample* function, the Sampling strategy will assume that lower values related to better performance. If no performance value is passed into the *takePeriodicSample* function, then the function will automatically generate a performance value based on the time intervals between successive calls to the function, again assuming that less time per interval relates to an increase in performance. Alternatively, a command line option can be used to indicate that the runtime system should implicitly makes calls to the *takePeriodicSample* function. If this method is used, the runtime system will internally make these periodic calls based on the frequency of AEM invocations. In other words, the runtime system

assumes that the performance of the application increases as the frequency of AEM invocations increases (regardless of location), and vice versa. However, this method is not always reliable. For example, the number of AEMs being invoked may change over the course of the application's lifetime, based on the current conditions within the application. As such, the strategy may be fooled into thinking that there has been a performance increase or decrease, when no such change has actually occurred. As such, this method is mainly useful for applications where the number of AEMs executed per unit of time remains fairly constant.

Internal Construction of the Workload Balancing Graph: Regardless of the method used (implicit or explicit sampling), the samples are essentially used by the Sampling strategy to internally construct the maximum of the host core and accelerator device plots. The maximum is visually marked using black dashes in figure 5.1(a), which overlaps with each of the original plots to some degree. Each time the *takePeriodicSample* function is called, it is considered to be a single sample taken for the current percent device value. Once enough samples have been taken, the runtime system will adjust the percent device value. The exact decision process for determining how the percent device value will be modified is based on the overall set of samples taken so far. Essentially, the decision process seeks to find a valley in the workload balancing graph (figure 5.1(a)) by constantly trying to move *downhill* (towards zero on the vertical axis) as the percent device is adjusted. For any given percent device value, there is a maximum number of samples that will be retained by the runtime system before older samples will be dropped as newer samples are taken. This allows the Sampling strategy to retain a history of the measured performance as it seeks a minimum execution time (maximum performance). However, if the environment in which the application is executing changes over time, affecting the performance of the application, the Sampling strategy is capable of adjusting to those changes over time as old samples are discarded by more recent measurements.

Since the workload balancing graph is be created over time by sampling the performance of the application at different percent device values, the generated graph may contain only a sparse number of data points (especially near the beginning of the application's execution) as well as stale measurements (especially as the application's execution continues on over time). In constructing the graph, the range of possible percent device values (i.e. from 0% to 100%, inclusive) is discretized using a given resolution that is specified as part of the accelerator manager's configuration when the runtime system is compiled (currently set to 0.5%). At any point in time during an application's execution, many of the possible percent device values may not have performance samples associated with them. In these cases, the performance value for that percent device value is assumed to be minimum of the values of the two closest percentages (one greater and one lesser) which do have actual measured values on either side of the percentage in question.

To make the idea of how the Sampling strategy constructs the workload balance graph a bit more

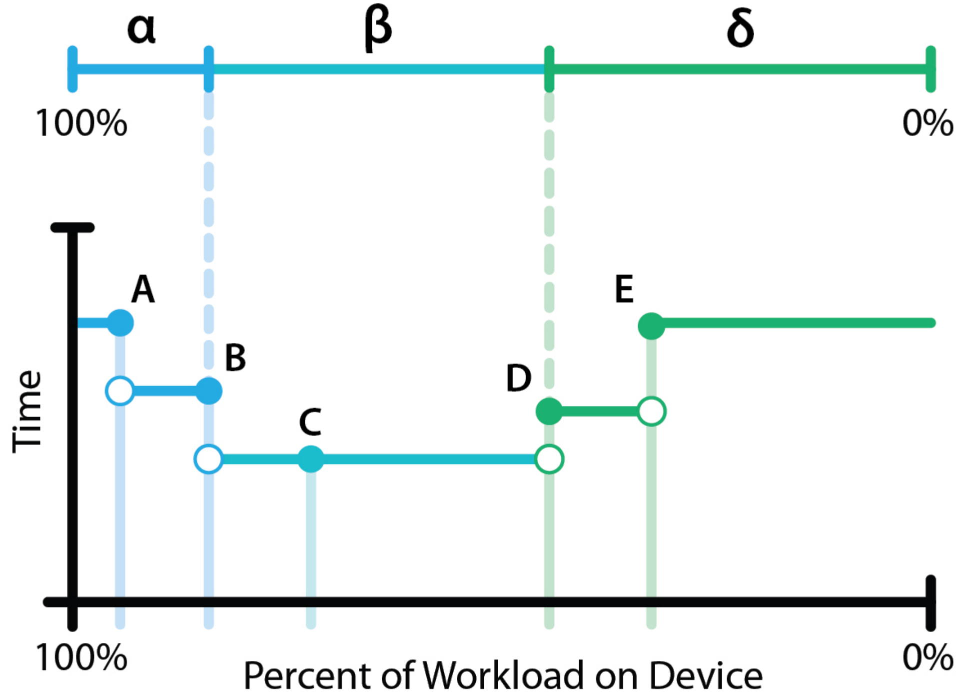


Figure 5.4: An example illustrating the construction of the workload balance graph by the Sampling accelerator load balancing strategy.

concrete, figure 5.4 illustrates how the workload balance graph is constructed. In this figure, it is assumed that the execution of the application is not far along, and as such, the performance of the application has only been sampled for five unique percent device values, A, B, C, D, and E (though, multiple samples have have been taken for each unique percent device value). Further, it is assumed that the default behavior of *takePeriodicSample* is being used, which uses the time intervals between consecutive calls to measure the performance of the application. For percentages that have not been sampled yet, the minimum of the two actual measurements on either side of the given percentage is used as the performance value for the given percentage. For example, in figure 5.4, there are no actual measurements taken for any percentages between B and C, so the performance at every percentage in between these two percentages is assumed to be the best performance level (lowest time) measured at either of these two points (in this case, the performance measured at C). As a whole, the graph can be conceptually broken down into three regions of percentage values, α, β, and δ. The α region includes the percent device values which cause the application to be device bound. In a similar way, the δ region includes the percent device values which cause the application to be host bound. The β region indicates the range of percent device values where the *sweet spot (S)* may be found, or in other words, where the application has achieved the best performance based on measurements taken up to the point in time represented within figure 5.4. Generally speaking, when the application starts,

the entire range of possible percent device values (i.e. 0% to 100%) is included within the β region. As time goes on and more measurements are taken, the β region will continue to narrow down until it only encompasses a small set of percentages located near the *sweet spot (S)* from figure 5.1(a).

Adjusting the Percent Device Value: When deciding how to choose new percent device values, the Sampling strategy will examine the current workload balancing graph and choose the already sampled percentage which resulted in the best performance in the application's execution so far, γ. The strategy will then consult the workload balancing graph to either side of γ at some fixed distance, ε (the exact distance is a configuration parameter at compile time, but is roughly 2-3% within our experiments). The performance levels for these three percentages ($\gamma+\varepsilon$, γ, and $\gamma-\varepsilon$) are then compared to determine what percentages will be sampled next (note that the performance at percentage γ will have been measured, but the performance at percentages $\gamma+\varepsilon$ and $\gamma-\varepsilon$ may be either actual measured values or assumed values as described above). If the three points form a clear slope (e.g. $p(\gamma+\varepsilon) > p(\gamma) >= p(\gamma-\varepsilon)$ or $p(\gamma+\varepsilon) <= p(\gamma) < p(\gamma-\varepsilon)$, where $p(\zeta)$ is the performance, measured or assumed, of the application at percentage ζ), then the Sampling strategy will queue several (3) equally spaced values *downhill* (i.e. in the direction of better performance) of percentage γ, including γ as one of the set. In the case of a flat region or a *valley* centered around percentage γ (i.e. $p(\gamma+\varepsilon) >= p(\gamma) <= p(\gamma-\varepsilon)$), a range of equally spaced percentages centered on γ are queued as the next percent device values that the Sampling strategy will sample.

Whether the newly queued percentages are *downhill* of or centered on γ, their distance from γ is chosen to be relatively small (typically less than 5%). As will show experimentally in section 5.4.2, even small adjustments to the percent device value can result in significant performance degradations. As such, we choose to make smaller adjustments within our implementation. This decision may require more overall adjustments to be made in order to reach the best performance levels. However, the rate at which samples are taken is directly proportional to the performance of the application. Small adjustments that are consistently in the direction of better performance also results in a steady increase in the rate of performance sampling. Recall that each sample represents a period of time that has passed, and that a decrease in performance will result each sample taking a longer period of time. If a large movement is made that greatly decreases the performance of the application, the time spent at that percent device value may also be quite significant since it takes more time to collect each performance sample and there is a minimum number of samples taken between percent device adjustments. For example, consider a case where an accelerator device provides a significant speedup, such as 30x. The time taken to collect the minimum number of performance samples when the percent device value at 0% would be approximately thirty times longer than when the percent device value is 100%.

Measurement Noise: It is also worth noting that, since the Sampling strategy makes use of measurement values, it is inherently exposed to the effects of noise on its measurement values. As already stated, multiple samples are taken in between each percent device adjustment as one method of alleviating the effects of noise within the performance measurements. Additionally, each point in the workload balancing graph accumulates performance samples over time (with old samples eventually being replaced by new samples after a maximum number per percent device value have been accumulated). The value for a given percent device value in the workload balancing graph is calculated by averaging all of the samples, after first removing the minimum and maximum sample values. Noise can arise from many causes, including variances in data transfers (DMAs, network messages, etc.), variances in execution times for entry methods, variances in memory access times (i.e. accessing different levels of a cache hierarchy), resource contentions between asynchronous tasks, and so on.

Beyond the noise that is a natural part of any computing system, the runtime system also introduces a certain amount of noise into the sampling process. Speaking generally, each time the runtime system takes some action, that action takes a certain amount of computation and time, which decreases the performance of the application, at least in an immediate sense, by temporarily stalling application progress as the action is performed. Obviously, the quicker and less invasive the actions taken by the runtime system are, the less noise they will create within the application. Further, the application may go through phases of increased or decreased runtime system activity. Some of these phases may be fairly short but quite intense, such as during the creation of an application checkpoint or during global load balancing.

The Sampling strategy itself creates noise within the application as it attempts to measure the application. This is particularly true in the case when *static assignment* is being used in the presence of *persistent data buffers* (discussed in section 6.3). In the case of the Sampling strategy temporarily stalling the application as it adjusts the percent device value, the pattern of the induced noise is predictable. The noise is introduced during a short time period directly following the percent device value adjustment (as a direct result of the adjustment calculation and during the process of applying the given adjustment). Since the noise introduced by percent device value adjustments is predictable in this way, the Sampling strategy ignores one or more performance values immediately following an adjustment (we ignore two for our experiments, but this amount can be adjust within the runtime system code). This is another method by which our implementation attempts to alleviate the effects of noise. Even though we have introduced this technique within the discussion of the Sampling strategy, it should be noted that other strategies make use of this technique as well.

Modes of Operation: The Sampling strategy can be applied in a centralized or a decentralized way.

In the centralized approach, the periodic sampling and decision process only occurs on host core 0. When each adjustment to the percent device value is made on host core 0, that change is broadcast to all of the host cores, guaranteeing that the same percent device value is being used across all of the host core and accelerator device pairings within the application. The centralized approach introduces a small amount of communication as new percent device values are distributed. A downside to the centralized approach is that all of the host core and accelerator pairings are being forced to use the same percent device value. As such, if a single pairing begins suffering from decreased performance for an extended period of time (e.g. if another application begins sharing the accelerator device, but not the host core), then that pairing's percent device value cannot be adjust independently of the other pairing's percentages. Instead, an across host core load balancer would be needed to prevent the single slow pairing to become a bottleneck by migrating work from the slower pairing to other higher performing pairings. However, on clusters were nodes are dedicated to at most one application at any given moment, this will likely not be an issue.

A decentralized approach can also be enabled, which eliminates the need for the added communication to be introduced into the application's execution. In the decentralized approach, each of the host cores locally take the periodic measurements (requiring the periodic point to occur on all of the host cores if the explicit sampling approach is used). While the decentralized approach removes the added communication, it also suffers from the fact that the host cores' local percent device adjustments are not synchronized. Note that the overall performance of a given parallel application (in particular, tightly coupled parallel applications) is usually defined by the *slowest* processing element. If *any* host core adjusts its local percent device value is such a way that degrades the performance of that host core, the overall performance of the application will degrade as a result of the natural data dependencies within the application. As such, all of the host cores, when measuring the performance of the application within their own local vantage points, will measure a decrease in the application's performance and associate that slowdown with its the percent device value the host core currently has applied. This opens the possibility for a single measurement on a single host core to have rippling effects across all of the host cores. In fact, it can result in an artificially created local minimum forming in each of the local workload balancing graphs for each host core. The issue arises when the percent device values are all either in α or δ regions of the workload balancing graph, referring back to figure 5.4 (i.e. the application is either host bound or device bound), and something causes the percent device values across the host cores to diverge (i.e. one or more host cores temporarily experience a slowdown for any reason). Locally, this core will immediately measure a slowdown, causing its local workload balancing graph to differ slightly from the graphs on other cores. This single host core then may incorrectly decide that it has found a local minimum and revert the percent device value to *uphill* value (i.e. away from S). Meanwhile, the other

host cores correctly deduce that they are on a slope and should continue moving their percent device values *downhill* (i.e. towards S). Since the performance of the host cores are directly related to one another as a result of data dependencies within the application, a slowdown on a single host core will soon slow down the overall application, causing all of the host cores to measure a lower level of performance than they otherwise should have measured had the percent device values not diverged. This can lead to a situation where the percent device values across all of the host cores begin to adjust back and forth in an unsynchronized manner. Since the performance of the application is bound by the slowest pairings, a flat region or even worst a local minimum can artificially form within all of the workload balancing graphs across the host cores as a result of at least one pairing's percent device value moving *uphill* at any given moment. The local minimum may be further reinforced as time goes on if the individual percent device values continue moving in opposite directions relative to one another. As a second order effect, also recall that more time will be spent at the less optimal percent device value since sampling takes longer when performance is diminished, further reducing changes of the host cores working their way out of such a situation. The decentralized approach is susceptible to this behavior and once it occurs, the chances of recovery decreases as the total number of host cores is increased (i.e. it only takes one host core to begin the chain reaction and the slower host cores retain their *bad* percent device values for a longer period of time). As such, even though the centralized approach introduces a small amount of network overhead and requires a single percent device value to be used by all host core and accelerator device pairings, it is the recommended mode of operation when using the Sampling strategy. This behavior was observed during the development of our implementation and testing of the Sampling strategy.

Before moving on, it is work nothing that the AdjustBusy strategy described above is also a decentralized strategy. However, the AdjustBusy strategy will not suffer from the artificially created local minima discussed above. The Sampling *(decentralized)* strategy does not differentiate between busy and idle time when taking its periodic measurement. A single pairing experiencing a slowdown affects the performance measurements on other pairings by inducing idle time between calls to *takePeriodicSample*. While this is an issue for the Sampling *(decentralized)* strategy, the AdjustBusy strategy ignores induced idle time by only attempting to equalize busy times.

Comparison with AdjustBusy: Finally, it is worth making a quick comparison of the AdjustBusy and Sampling strategies. An advantage of the AdjustBusy strategy is that does not require evenly spaced points in the application (e.g. timestep boundaries) at which point performance samples can be taken, making it more generally applicable. However, the AdjustBusy strategy does not actually react to the performance of the application (at least in a very direct manner). Instead, a heuristic is applied that balances the busy

times with the assumption that if the busy times are balanced, then both the host core and the accelerator core are being fully utilized. Conversely, the Sampling strategy reacts to the measured performance of the application. However, to do this, it also requires a periodic point in the application, which not all application may have.

Profilers

Before moving on to the next accelerator strategy, the profilers that have been included within the accelerator manager will be discussed because the next strategy relies on them. On a conceptual level, profilers and accelerator load balancing strategies differ in their goals. The accelerator strategies that we have discussed so far direct the accelerated entry methods to be executed on the host or the accelerator device with the goal of improving the performance of the application. Profilers will also direct the execution of the AEMs. However, profilers move around the workload in order to take various measurements with the goal of determining certain characteristics of the application's workload so that accelerator strategies can then take advantage of that information. On an implementation level, profilers are implemented on top of the same based class, *AccelStrategy*, making use of the same interfaces used by the strategies.

Two profilers have been included with the accelerator manager, the base load profiler and the kernel ratio profiler. Each of these profilers will be discussed below. Profilers are used by placing them in the accelerator manager's strategy queue prior to the strategies that use them. Once an active profiler (or strategy) completes its task, it will indicate to the accelerator manager that it is finished. If the strategy queue has any remaining profilers or strategies to be applied, it will swap out the current one and start the next.

Base Load (*BaseLoad*): The goal of the base load profiler is measure the amount of busy time on the host core when all of the accelerated entry methods are executing on the accelerator. The profiler starts by transferring all of the workload associated with the AEMs to the accelerator device. Once the AEMs have been shifted from the host, they remain there for a specified period of time. During this time period, the runtime system tracks three values on the host core: the busy time, the overlapped busy time, and the idle time. The busy and idle times are straight forward measurements of the time the host core is busy with work or sitting idle, respectively. The overlapped busy time is the subset of the busy time that overlaps with busy time on the accelerator device. Referring back to figure 5.3, in this example the overlapped busy time refers to blocks E, F, H, and the portion of D that occurs after time γ. In the same vein, the busy time is the sum of A, B, C, D, F, and J (note that blocks E and H wouldn't exist since all AEMs are mapped to the accelerator device while the BaseLoad profiler is taking measurements). The idle time is the sum of

G and I (and any additional idle time created from the removal of E and H from the host core). Referring back to figure 5.1(a), the busy time measurement relates to the value H_{empty}. These values are collectively referred to as the *base load of the host core* since they give a basic sense of the minimum workload that will be present on the host core, regardless of the percent device value in use. The actual values generated by the *BaseLoad* profiler are normalized so that $busytime + idletime = 1$. Once generated, these values are passed back to the accelerator manager where they are stored so that future strategies can make use of them.

Kernel Ratio (KernelRatio): The goal of the kernel ratio profiler is to determine if the accelerator device or the host core is better suited for the executing of each type of accelerated entry method. To accomplish this task, the KernelRatio profiler cycles through each of the AEM types, one-by-one. For each AEM, the profiler starts by directing all instances of the given AEM type to the accelerator device. As those instances are executing on the accelerator device, the busy time of the accelerator is tracked. In the meantime, the execution times of the other AEMs executing on the host core are also sampled. Once the profiler has cycled through all of the AEM types, collecting enough samples on each processor type, the profiler calculates an execution time ratio for each of the AEM types. To calculate the ratio for a given AEM type, the profiler starts by calculation the time per AEM invocation on the host core ($H_{time/count} = H_{time}/H_{count}$, where H_{count} is the number of AEM invocations and the H_{time} is the total time taken to execute those H_{count} AEM invocations). This calculation is repeated for the device as well ($D_{time/count} = D_{time}/D_{count}$). The device time value, D_{time}, for a given AEM time is simply the accelerator busy time over the course of the time period that AEM type is active on the accelerator. This has the advantage of also accounting for the secondary effects of streaming data into and out of the device (which will occur if, for example, there are multiple instances of the AEM being streamed through the SPEs on the Cell, multiple batch sets issued to a GPGPU), the AEMs being divided over several cores on the accelerator device, or the overlapping of GPGPU threads within a single batch set on a GPGPU. If the profiler attempted to measure the individual latencies of each AEM invocation executing one-by-one and then sum those values, the various parallelism and streaming effects associated with accelerator devices would not be accounted for, leading to an over inflated value for the D_{time} measurement.

The host-to-kernel ratio is then calculated by dividing the time per AEM invocation on the host by the time per invocation on the device ($R_{host/device} = H_{time/count}/D_{time/count}$). Note that the H_{count} and D_{count} values may differ since the AEM invocations on the host are sampled the entire time the KernelRatio profiler is active, while the invocations for each AEM time are not directed to the accelerator device for the entire time the profiler is active. These ratios are then passed back the accelerator manager so that future strategies can make use of them. Specifically, $R_{host/device}$ and $H_{time/count}$ values for each type of AEM are

stored within the accelerator manager.

Greedy *(Greedy)*

The Greedy accelerator uses the output of the two profilers discussed above, BaseLoad and KernelRatio, to greedily assign the AEMs that are best suited for the accelerator device to the accelerator device, before AEMs that aren't as well suited. Unlike the AdjustBusy and Sampling strategies, the Greedy strategy is not an active strategy. Instead, once it is made active by the accelerator manager, it applies its greedy algorithm once and then remains inactive until it is removed by the accelerator manager.

$$T_{timestep} = B_{host} + \sum_{i \in S} H_{time/count,i}\ C_i \tag{5.1}$$

The Greedy strategy works by first assigning all accelerated entry methods to the host core. Specifically, the percent device value for each type of AEM is set to 0%. The strategy then calculates the total time to execute a single timestep of the application using the output of the BaseLoad profiler, the $H_{time/count}$ values generated by the KernelRatio profiler, and the already existing object counts within the runtime system, using equation 5.1. In equation 5.1, $T_{timestep}$ is time to execute a single timestep, B_{host} is the non-overlapping base load on the host (as measured by the BaseLoad profiler), $H_{time/count,i}$ is the average time to execute an invocation of AEM type i on the host core ($H_{time/count}$, as calculated by the KernelRatio profiler), and C_i is the number of objects local to the host core and accelerator device pairing associated with AEM type i.

For the time being, the Greedy strategy makes two assumptions. The first assumption is that it will be common for chare objects with AEMs to have that AEMs invoked on a frequent basis (e.g. all objects in all timesteps). We believe this is a fairly safe assumption since the AEMs are intended to be used for the calculation intense portions of the application, meaning that they should naturally comprise a significant portion of the application's overall workload. However, this may not always be the case, and as such, we intend to improve upon the Greedy strategy by removing this assumption in future implementations.

The second assumption is that the execution characteristics of the application will be similar before and after the application of the profilers. For example, suppose all of the objects are having their AEMs invoked while the profilers are active, but then sometime after the profilers have completed the application enters a phase where only half of them are being invoked, this approach may produce non-optimal results. This is mitigated to some degree by the fact that the percent device values for each type of AEM are based on the relative execution times of the host core and the accelerator device. In other words, if the workload associated with one type of AEM is halved, for example, then simply keeping the percent device value for

that AEM type constant will result in both the host core's and accelerator device's portions of that workload to be halved (i.e. the ratio of work remains the same). What does change are the ratios of total workload associated with one type of AEM to the total workload associated with another type of AEM and the base load of the host core.

Further, the Greedy strategy does not understand the various data dependencies within the application. In fact, none of the strategies we have included as part of our existing implementation take data dependencies into account. This could cause less than optimal results in situations where a given AEM type is the best suited of all the AEM types and thus the first to have its individual percent device value to be greedily assigned to the device. However, if it just so happens that this AEM type does not overlap with any other work in the application because of data dependencies, it might be better if some small portion of that AEM type is executed on the host core since the host core is otherwise idle. This is a possible short coming of the Greedy strategy.

Crawler *(Crawler)*

The Crawler strategy is similar to the Sampling strategy in that it samples the actual performance of the application in an attempt to determine the best balance of the overall AEM workload between the host and accelerator device. However, unlike the Sampling strategy, the Crawler strategy adjusts the percent device value for each of the various accelerated entry method types individually, instead of using the same value for all AEM types. As such, the Crawler strategy can favor AEMs that are better suited for the accelerator device when assigning AEMs to the device. Also unlike the Sampling strategy, the Crawler strategy does not retain a history of the performance measurements that have been sampled so far for the various percent device values that have already been tried. If there are several types of AEMs, then the amount of data that would need to be retained could grow quickly, and it is not clear that retaining this information is truly required. Another difference is that the Crawler strategy only operates in a centralized mode, compared to the Sampling strategy which can operate in either a centralized or decentralized mode.

The Crawler strategy first takes a series of measurements by going through a *rotation sequence*. There are several steps to the rotation. In the first step, the performance using the current percent device values is measured, referred to as the *center point*. Then the percent device value of each of the AEM types is adjusted by a fixed amount (0.5%) in both directions (increased and decreased by a small amount), by cycling through each AEM type and direction combination one-by-one. As was the case with the Sampling strategy (refer to section 5.1.3), we have decided to use small steps within our implementation to avoid the possibility of drastic performance degradations if a large step in the wrong direction is attempted. Once all

of the AEM types and direction combinations have been cycled through, the best performing combination is selected as the new center point around which the next set of rotations will use as a base. Once the center point has been adjusted, the rotation process starts over. As this rotation sequence repeats over time, the adjustments to center point accumulate, allowing the percent device values to change significantly over time. Since there can be several AEM types per application, the rotation process can be slow, requiring several samples for each AEM type and direction combination, including the center point, before a single center point adjustment is made. Further, it will take several center point adjustments before the percent device value for any of the AEM types will shift by any significant amount. Therefore, a clear disadvantage to the Crawler strategy is that it can take quite some time to converge on a suitable balance point for the workload.

To overcome the downside of taking quite some time to modify the percent device values by any significant amount, the Crawler strategy has been implemented so that it can follow the Greedy strategy. Again, the Greedy strategy is a one time calculation that is applied to the various percent device values for each AEM type, which can result in a significant adjustment to each of those percent device values. A downside to the Greedy strategy is that it only applies the greedy algorithm based on some initial measurements from the BaseLoad and KernelRatio profilers. If conditions under which the application is executing change or those initial measurements are not quite as accurate as they could have been, then the result of the Greedy strategy will be affected. However, if the Greedy strategy is applied and then immediately followed by the Crawler strategy, the combination of strategies can help overcome the shortcomings of either strategy. Further, the implementation of each strategy remains independent, allowing the implementation to remain focused and clear and allowing either strategy to be used in combination with future strategies without requiring the functionality in the existing strategy to be recreated.

Compared to either the AdjustBusy or Sampling strategies, the Crawler strategy is capable of adjust the percent device values of each AEM type independently of one another, similar to the Greedy strategy. Like the Sampling strategy, the Crawler strategy actually measures the performance of the application via the *takePeriodicSample*.

5.2 Across Host Core Load Balancing *(AccelEvenLB)*

In addition to the accelerator load balancing strategies that act locally with a single host core and accelerator device pairing, the across host core load balancing strategy AccelEvenLB has also been created as part of this work. The idea behind the AccelEvenLB strategy is to equally distribute chare objects of the same type across each of the host core and accelerator device pairings. When we say "equally distribute" we mean

in terms of execution time, not the number of objects or even the number of flops performed. During an application's execution, the load balancing framework [97, 9] within the Charm++ runtime system measures the load of each chare objects in terms of the amount of time that object occupied the host core. Periodically, using the load balancing framework's *AtSync* triggering method, the AccelEvenLB load balancer is invoked. The AccelEvenLB load balancer is a centralized load balancer, meaning that the load balancing algorithm is executed on a single core and the results are broadcast back to all the cores in order to carry out the decision.

When accelerated entry methods are used, such a measurement process can lead to an incorrect assessment of which objects take more time than others. For example, consider to chare objects, A and B of the same type and mapped to the same host core and accelerator device pairing. It may be the case that A happens to have its associated AEMs mapped to the accelerator device, while B does not. Assuming the accelerator device provides a speedup for the execution of AEMs, then the time attributed to A will be less than B. However, this is just a matter of circumstance rather that a difference between A and B. If A and B are otherwise identical (apart from their current mappings), then having them swap places would result in B having less time attributed to it relative to A. To account for this timing imbalance as a result of *some host cores appearing to be faster than other host cores*, the AccelEvenLB strategy averages the amount of time per chare object of a given type on a given host core, $\tau_{type,\rho}$. That is to say, the objects within the application are grouped into sets with one set for each object type and host core combination. For each set, the average time per object is calculated. Then, for each individual object, this average time is used in place of the object's individually measured time in order to calculate the overall distribution of objects of the given type across all the host cores.

After computing the average time for each chare object type and processor combination, the algorithm continues by calculating the balance of objects for each type of object. For each type of object, the maximum average time is determined across all of the host cores (equation 5.2a, where P is the set of host cores). The time τ_{type} is then used to invert the average times for each object type and host core combination (equation 5.2b). These inverted averages are then normalized and divided by the their sum to determine the fraction of objects of the given type that will be placed on each of the host cores (equations 5.2c, 5.2d, and 5.2e, where $f_{type,\rho}$ represents the faction of the total objects of the given type to be placed on host core ρ, $\sum_{\rho \in P} f_{type,\rho} = 1$, θ_{type} is the total number of chare objects of the given type, and $\eta_{type,\rho}$ represents the number of chare objects of the given type that should be placed on host core ρ).

$$\tau_{type} = max(\tau_{type,\rho} \; \forall \; \rho \in P) \tag{5.2a}$$

$$[\forall \rho \in P] \; \tau\prime_{type,\rho} = \frac{\tau_{type}}{\tau_{type,\rho}} \tag{5.2b}$$

$$\psi_{type} = \sum_{\rho \in P} \tau\prime_{type,\rho} \tag{5.2c}$$

$$[\forall \rho \in P] \; f_{type,\rho} = \frac{\tau\prime_{type,\rho}}{\psi_{type}} \tag{5.2d}$$

$$[\forall \rho \in P] \; \eta_{type,\rho} = f_{type,\rho}\theta_{type} \tag{5.2e}$$

Once the number of objects per host core, ($\eta_{type,\rho}$), has been determined, AccelEvenLB proceeds to reassign the chare objects to host cores so that the calculated fractions are realized within the application. The reassignment of chare objects occurs in two phases. During the first phase, the number of chare objects of the given type currently assigned to processor ρ are counted. If the number of objects currently assigned to a given host core ρ is greater than $\eta_{type,\rho}$ (the target number of objects), then the *extra chare objects* are placed in a list of objects, Γ, that are available for migration. During the second phase of reassignment, any host core that is found to have less than its target number of chare objects, $\eta_{type,\rho}$, has objects assigned to it from the list of *extra chare objects*, Γ. By making used of the *extra chare objects* list, the AccelEvenLB load balancer only moves objects that are required to be move for load balancing, minimizing the amount of chare objects migrated across host cores to achieve the calculated object distribution. This overall process is repeated for each type of chare object within the application.

As with all general load balancing strategies included within the Charm++ runtime system, the AccelEvenLB strategy is not intended to handle all situations flawlessly. In particular, the AccelEvenLB strategy assumes that there are many objects per processor for each type of object. In situations were there are many objects total, but only a few objects of each type, the approach used in AccelEvenLB will likely not function correctly since it tries to evenly distribute objects across the host cores on a type-by-type basis.

5.3 Description of Hardware

In this section, we describe the hardware that will be used to execute test applications. For the dynamic load balancing strategies, the Forge GPGPU cluster at the National Center for Supercomputing Applications (NCSA) [78] will be used.

5.3.1 Forge (GPGPU Cluster)

Forge is a cluster located at the National Center for Supercomputing Applications (NCSA). The compute nodes within the cluster are Dell PowerEdge C6145 servers. Each compute node includes 16 cores (AMD Opteron Magny-Cours 6136) running at 2.4 GHz. Each compute node also includes some number of GPGPUs (32 nodes include 6 GPGPUs, 12 nodes include 8 GPGPUs, and the specific hardware used is nVidia's M2070 accelerator units).

Unless noted otherwise, the experiments within this chapter will use nodes that include 6 GPGPUs. The GPGPUs will be assigned to the host cores is a round-robin manner. For example, physical GPGPU 0 will be mapped to virtual host core 0, physical GPGPU 1 will be mapped to virtual host core 1, ... physical GPGPU 5 will be mapped to virtual host core 5, physical GPGPU 0 will be mapped to virtual host core 6, physical GPGPU 1 will be mapped to virtual host core 7, ... and so on. The virtual host cores will be mapped to physical host cores based on the virtual host cores' assigned physical GPGPUs' affinity to the physical host cores (the affinity of physical cores to GPGPU processors is located on NCSA's website [78]). Specifically, the option *"+pemap 0,2,8,9,10,11,4,6,12,13,14,15,1,3,5,7, 8,10,0,2,9,11,12,14,4,6,13,15,1,3,5,7, 8,9,10,11,0,2,12,13,14,15,4,6,1,3,5,7"* was used. The *pemap* options indicates how the runtime system should map virtual host cores to physical host cores. Specifically the 'virtual host core number modulus the length of the *pemap* list' is used to index into the *pemap* list in order to lookup the physical host core associated with the given virtual host core. In the case of Forge, the pemap is particularly long since there are 6 GPGPUs and 16 host cores per node, so 48 entries (3 nodes) are required in the *pemap* in order to create a repeating sequence. Otherwise, standard options were used for launching application executions (i.e. "+netpoll", "+p", "++nodelist", and any options specific to the various load balancing strategies being used within each experiment).

5.4 Applications

5.4.1 5-Point Weighted Stencil

One of the applications that we will be using to test our load balancing strategies is a 5-point weighted stencil application. This type of calculation is typically used for simulating heat transfer and as a kernel (step) in iterative solvers. It is a simple (and widely used) stand-in for many real applications with near-neighbor data exchanges.

The 5-point stencel application serves as a fairly good starting point at which to test the included strategies, because there is only a single type of accelerated entry method. This simplifies the job of the

accelerator strategies because there is only a single degree of freedom (i.e. one percent device value) that requires adjusting. In more complex programs, that include various types of AEM, there are added degrees of freedom when balancing the workloads since some of the AEM types may be better suited for the various types of cores than other AEM types. For the same reason, the stencil application serves as a good application in which to introduce the reader to the behavior of the various accelerator strategies (i.e. fewer complexities to reason about when analyzing the results). However, such a simple stencil code has a low flop to byte ratio. That is, the amount of data accessed by each AEM invocation is fairly high relative to the number of flops performed. Typically, such applications aren't the best candidates for making use of accelerator technologies, since the use of an accelerator will typically require additional data transfers to take place between the host core(s) and the accelerator device(s). The overhead of moving the data, while at the same time performing only a small amount of work per byte transferred, usually decreases the impact that accelerator can have in terms of improving the application's performance. As such, the results from the stencil application are included mainly to illustrate various points related to the accelerator load balancing strategies in a situation when the accelerator is unlikely to make significant impact on the application's performance.

Application Description

The structure of the application is fairly straight forward. There is an overall two dimensional (2D) matrix of values. The application's execution is broken down temporally into a set of discrete timesteps. During each timestep, a stencil calculation is performed on each of the elements with in the matrix. The amount that each matrix element changes during the timestep is calculated, referred to as *maxError* within the code itself, and a global reduction is used to calculate the maximum change that occurred across all the elements within the timestep. If the maximum change is below some given threshold, the program halts (i.e. has converged). Otherwise, if the maximum change of any one element is greater than the given threshold, then execution continues on to the next timestep.

In our stencil application, the overall matrix has been broken up into a set of tiles, with each tile being assigned to a given chare array element in a 2D chare array. In particular, the application is being executed with a grid of 32 by 32 tiles (1024 total tiles with each tile being a chare object) and 256 by 254 matrix elements per tile (65024 matrix elements per tile). Note that this code was adapted from a previous version to use our programming model extensions; the non-square shape of each 2D tile is not cause by our extensions, but is rather an artifact of the application code we used. The code holds one matrix element per tile at a constant value of one, while allowing all of the other tile elements to vary as the calculation proceeds. Initially, the elements that can vary are initialized to a value of zero. The weighted stencil pattern that we

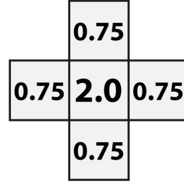


Figure 5.5: 5-Point weighted stencil pattern used in the 5-point stencil application.

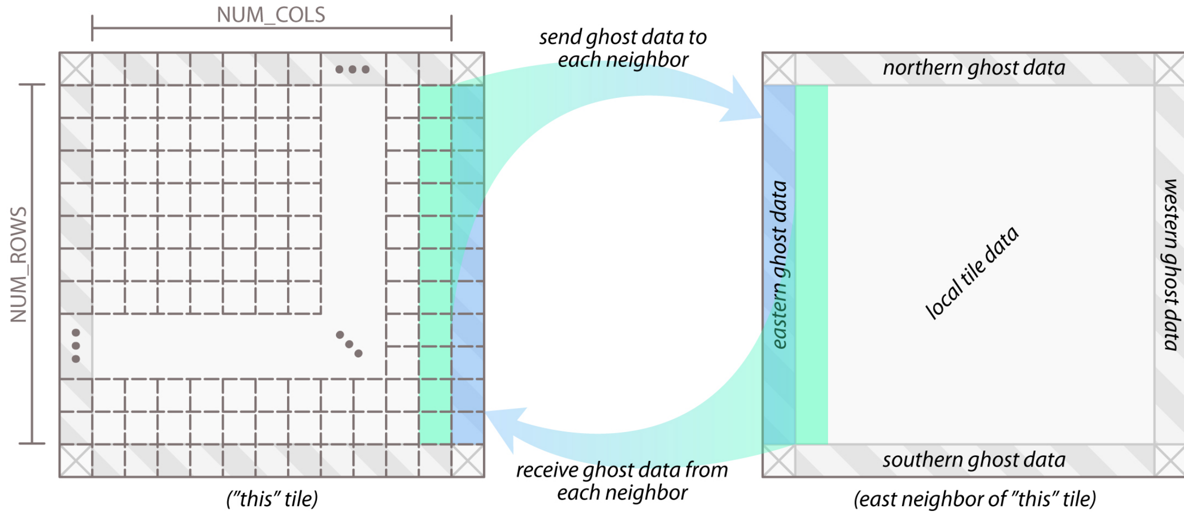


Figure 5.6: Interaction of tiles in the 5-point stencil application.

use is given in figure 5.5. Some stencil applications are more complex, causing the flops per byte ratio to increase. However, as stated previously, part of the reason for choosing the 5-point stencil application is that we wish to consider a case where the accelerator isn't necessarily well-suited for the given computation (i.e. the host core and the accelerator core are more evenly matched in terms of performance).

Within each timestep, the tiles begin by exchanging border information with one another, sometimes referred to as exchanging *ghost* information. For a given chare to perform the stencil calculation on the edges of the tile contained within it, the chare requires some of the element data within the chare objects (or tiles) that neighbor it, and vice versa. Therefore, each timestep begins with the tiles exchanging border information with one another. Once a given tile has all of the ghost information that it requires from it's neighbors, the tile can then proceed to perform the stencil calculation locally. Given the stencil pattern being used, as illustrated in figure 5.5, the ghost regions are a single data element wide. Figure 5.6 illustrates how the ghost data is transferred between neighboring tiles at the beginning of each timestep of the application. Figure 5.6 only shows the interaction between a given tile ("this") and one of its neighbors, but the same process is used for each of the four neighbors surrounding the "this" tile in the figure.

The stencil calculation is written as a splittable accelerated entry method, with the number of splits set

to the number of rows in the local tile. There are actually two copies of the tile data within each chare object. The *matrix* data array contains the actual tile data (input). The *matrixTmp* data array holds calculated values as they are being written in the stencil calculation function (output). Separate input and output buffers are used so that the values generated each timestep only rely on values from the previous iteration, and are unaffected by the calculations of neighboring elements within the same timestep. Because the splits in a splittable entry method are required to be data parallel with one another, similar to the iterations of an OpenMP parallel for loop, an auxiliary array is also included. The length of the auxiliary array is the number of rows (i.e. one element per split) so that each split can write its own locally calculated *maxError* value. The callback function that combines these values, creating the *maxError* value for the entire tile. Each tile will then contribute this tile-wide *maxError* value to a global reduction used to determine the global *maxError* value.

To give the reader a sense of the code that would be required of a programmer writing this stencil application, figure 5.7 includes the accelerated entry method code and callback function used by each of the tile objects. Otherwise, standard Charm++ constructs were used to write the application code. Keep in mind, the point of including the application code in figure 5.7 is to show how a programmer would use accelerated entry methods, and thus has been simplified relative to the actual application code for clarity and brevity. The point of figure 5.7 is not to demonstrate how one should write optimized stencil code.

Application of Accelerated Load Balancing Strategies

Step: As an initial starting point, the Step strategy is applied to the 5-point stencil application to establish a sense of what to expect from the application. As the application proceeds, the portion of the workload implemented using accelerated entry methods is slowly shifted from being executed solely on the accelerator device to having a larger and larger fraction being executed on the host core. Figure 5.8 gives the performance results of the stencil code executing using a single host core and a single GPGPU on the Forge system.

There are two plots included in figure 5.8: the performance of the program over time ("Avg. Time/-Timestep (sec)") and value of percent device as it changes over time ("Percent Device"). In the "Avg. Time/Timestep (sec)" plot, a *V shape* appears, as one would be expected based on the discussions in sections 5.1.1 and 5.1.3. When one considers the stencil calculation for a given tile (i.e. the *doCalculation* accelerated entry method shown in figure 5.7), there is a low flop to byte ratio (10 flops per 8 bytes of data for each matrix element). This low flop to byte ratio makes offloading a given tile calculation to an accelerator core less than ideal, since there is little work to perform relative to the overhead resulting from the transfer of tile data to and from the accelerator device. As a result, the host and accelerator cores provide

```
// Interface File: Accelerated Entry Method Function Body
entry [triggered splittable(NUM_ROWS) accel] void doCalculation()
  [  readonly : float      matrix[DATA_BUFFER_SIZE] <impl_obj->matrix>,
    writeonly : float matrixTmp[DATA_BUFFER_SIZE] <impl_obj->matrixTmp>,
    writeonly : float         tmpMaxError[NUM_ROWS] <impl_obj->tmpMaxError>
  ] {

  // Loop over the rows assigned to this split (if numSplits < NUM_ROWS)
  for (int y = splitIndex + 1; y <= NUM_ROWS; y += numSplits) {

    // Loop over the columns within this row
    float maxError = 0.0f;
    for (int x = 1; x <= NUM_COLS; x++) {
      int index = GET_DATA_I(x, y);

      // Apply the stencil pattern to this 'index'
      matrixTmp[index] = ((2.00f * matrix[index]) +
                          (0.75f * matrix[GET_DATA_I(x - 1, y)]) +
                          (0.75f * matrix[GET_DATA_I(x + 1, y)]) +
                          (0.75f * matrix[GET_DATA_I(x, y - 1)]) +
                          (0.75f * matrix[GET_DATA_I(x, y + 1)])
                         ) * (0.2f);

      // Hold tile element (1,1) constant at 1.0f on every tile and calculate
      //   this element's change and update maxError for the row
      if (y == 1 && x == 1) { matrixTmp[GET_DATA_I(x,y)] = 1.0f; }
      float tmpError = matrixTmp[index] - matrix[index];
      if (tmpError < 0) { tmpError = -1.0f * tmpError; }
      if (tmpError > maxError) { maxError = tmpError; }
    }

    tmpMaxError[y-1] = maxError;
  }
} doCalculation_post;

// Source Code File: Accelerated Entry Method Callback
void Jacobi::doCalculation_post() {

  // Calculate the maximum error amongst all the splits
  float maxError = tmpMaxError[0];
  for (int i = 1; i < NUM_ROWS; i++) {
    if (tmpMaxError[i] > maxError) {
      maxError = tmpMaxError[i];
      tmpMaxError[i] = 0;
    }
  }

  // Contribute this tile's maxError to a global reduction
  contribute(sizeof(float), &maxError, CkReduction::max_float);

  // Housekeeping: Swap the matrix pointers and advance the tile's iteration counter
  float *tmp = matrix; matrix = matrixTmp; matrixTmp = tmp;
  iterCount++;
}
```

Figure 5.7: Stencil code that acts upon a single tile, written as an accelerated entry method.

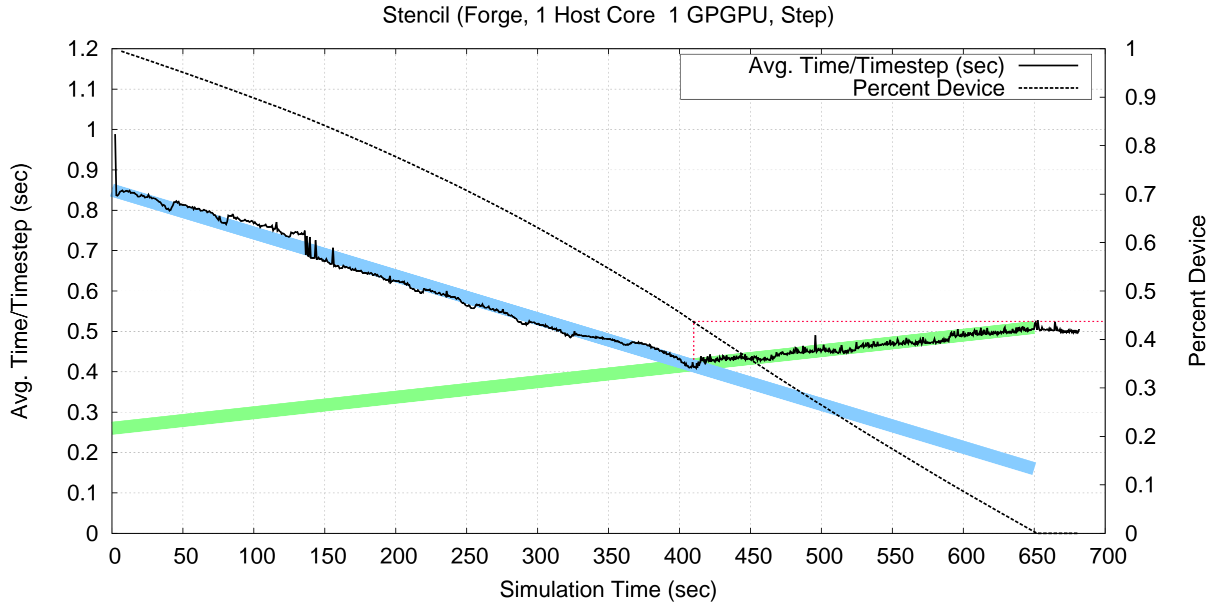


Figure 5.8: The 5-Point stencil application executing on a single host core and accelerator device pairing on Forge using the Step accelerator strategy.

a fairly similar level of performance for the *doCalculation* AEM. In fact, the host core actually has slightly better performance, likely caused by the additional data movement required for AEMs that are invoked on the accelerator device. Later, in section 6.3, we will attempt to decrease the overhead related to data movement between the host core and the accelerator device by through the use of persistent data buffers. As will be shown, eliminating the data movement significantly improves performance when AEMs are mapped to the accelerator device.

AdjustBusy: Next, we consider the case of the stencil application executing on a single host core and a single accelerator device on Forge using the AdjustBusy strategy discussed in section 5.1.3. Over time, the busy times of the host core and the accelerator device are measured. Periodically, these values are used to adjust a single percent device value, which in turn, is used to balance the workload for each type of accelerated entry method. Since the stencil application only has a single type of AEM, the percent device value only effects the one type of AEM, which simplifies the interpretation of the results. The results of applying the AdjustBusy strategy are shown in figure 5.9.

Figure 5.9 includes three plots. The first plot, "Avg. Time/Timestep (sec)," indicates the performance of the application over time. As one can see from the figure, the stencil application initially starts with a relatively bad level of performance, caused by all of the accelerated entry methods being executed on the accelerator device (a GPGPU, in this case). As time goes on, the runtime system, under the guidance of the AdjustBusy accelerator strategy, adjusts the percent device value over time to balance the overall workload

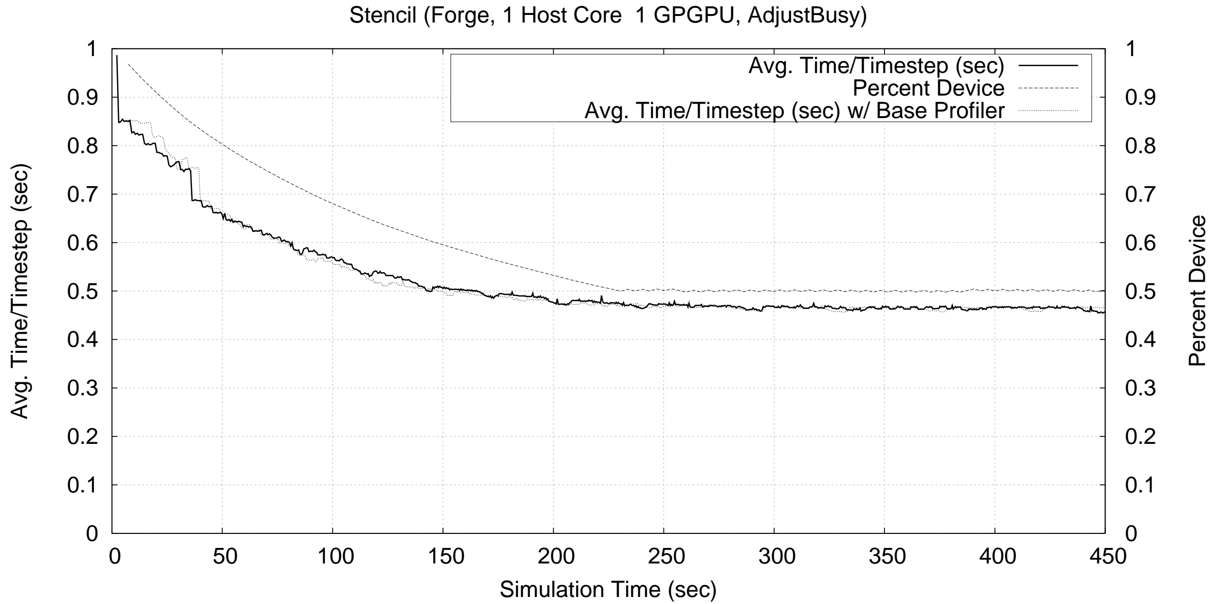


Figure 5.9: The 5-Point stencil application executing on a single host core and GPGPU pair on Forge using the AdjustBusy strategy.

between host and accelerator cores. The second plot, "Percent Device," shows the change of the percent device value over time. Refering back to figure 5.8, we can see that the AdjustBusy strategy converged on a percent device value that is roughly equivalent to the percent device value corresponding to the minima in Step strategy's performance plot. However, remember that the AdjustBusy strategy is only applying a heuristic, without measuring the resulting performance to determine if that heuristic was successful. Next, we move on to the Sampling strategy, which does in fact measure the performance of the application in order to make adjustments to the percent device value.

Sampling *(centralized)*: Figure 5.10 shows the performance of the stencil application executing on a single host core and a single GPGPU with the workload being balanced via the Sampling strategy, as the strategy is described in section 5.1.3. The sampling strategy begins with a percent device value of 100% and begins testing lower and lower values. Eventually, at approximately 100 seconds, a *valley* in the workload balancing graph is detected and the strategy begins to refine its search around a percent device values of 41% to 44% (varrying as time goes on). The sawtooth nature of the horizontal sections of the "Percent Device" plot illustrates the local search behavior of the sampling accelerator strategy as it cycles between values above, at, and below the percent device value that has resulted in the best performance so far. Should the environment in which the application is executing change for any reason, the strategy will be able to detect and react to that change. Note that the resulting performance of the Sampling strategy is roughly equivalent to the minima seen when the Step strategy was applied (refer to figure 5.8). Further, for the

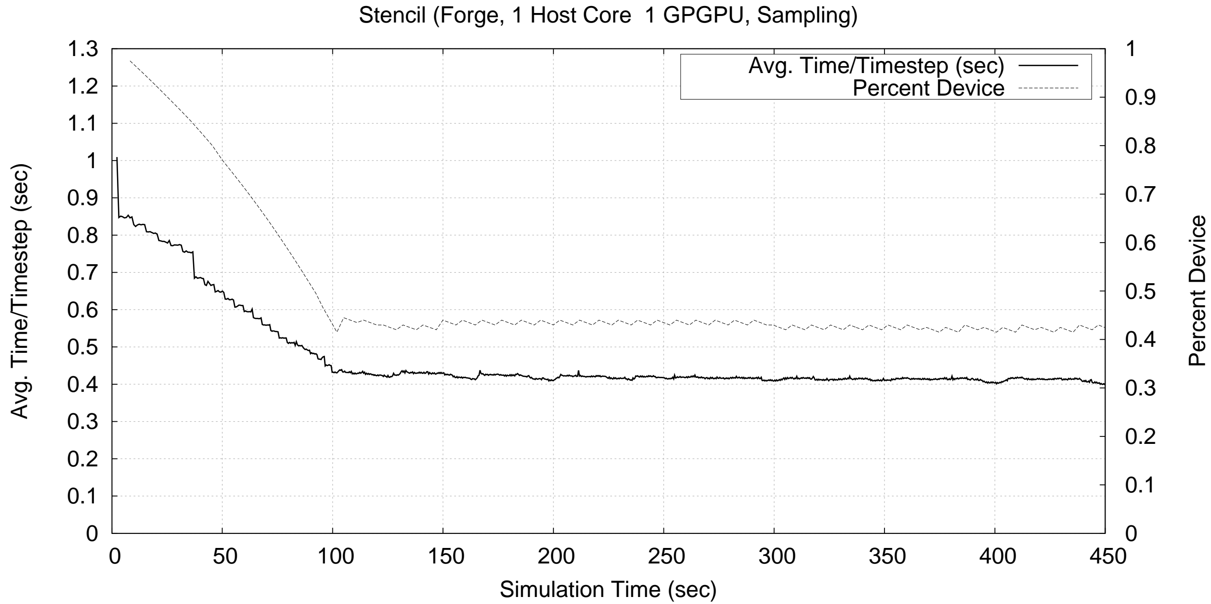


Figure 5.10: The 5-Point stencil application executing on a single host core and GPGPU pair on Forge using the sampling accelerator strategy.

5-point stencil code, the Sampling strategy is achieving slightly better results than the AdjustBusy strategy.

Greedy and Greedy with Crawler: Figure 5.11 moves on to using the greedy accelerator strategy, with and without applying the crawler accelerator strategy afterward. Note that the vertical axis of figure 5.11 does not originate at zero. This was done to allow the reader to more easily see the difference between the two plots. As discussed in section 5.1.3, the greedy strategy uses the information provided by the BaseLoad and KernelRatio profilers to greedily assign work to both the host and the accelerator cores. Initially starting with all of the work mapped to the host core, it selects the accelerated entry method that receives the greatest speedup when executed on an available accelerator and assigns as much of that sub-workload to the accelerator as possible, without causing the accelerator to become overloaded relative to the host core. If moving all of the work associated with a given AEM still leaves the accelerator with some idle time, the process repeats for the AEM with the next best speedup, and so on, until either all of the AEMs have been mapped to the accelerator or a workload balance has been reached.

One of the advantages of the greedy strategy itself is that it is quick, only performing a quick calculation once the two profilers that it relies on have completed. However, it does rely on the profilers, which take an amount of time that grows linearly (in units of timesteps) relative to the number of accelerated entry method types within the application. During this profiling time, the runtime system is constantly adjusting where the work associated with the various accelerated entry methods executes, which can result in dramatic performance decreases during the profiling period. For example, the initial execution of the stencil application

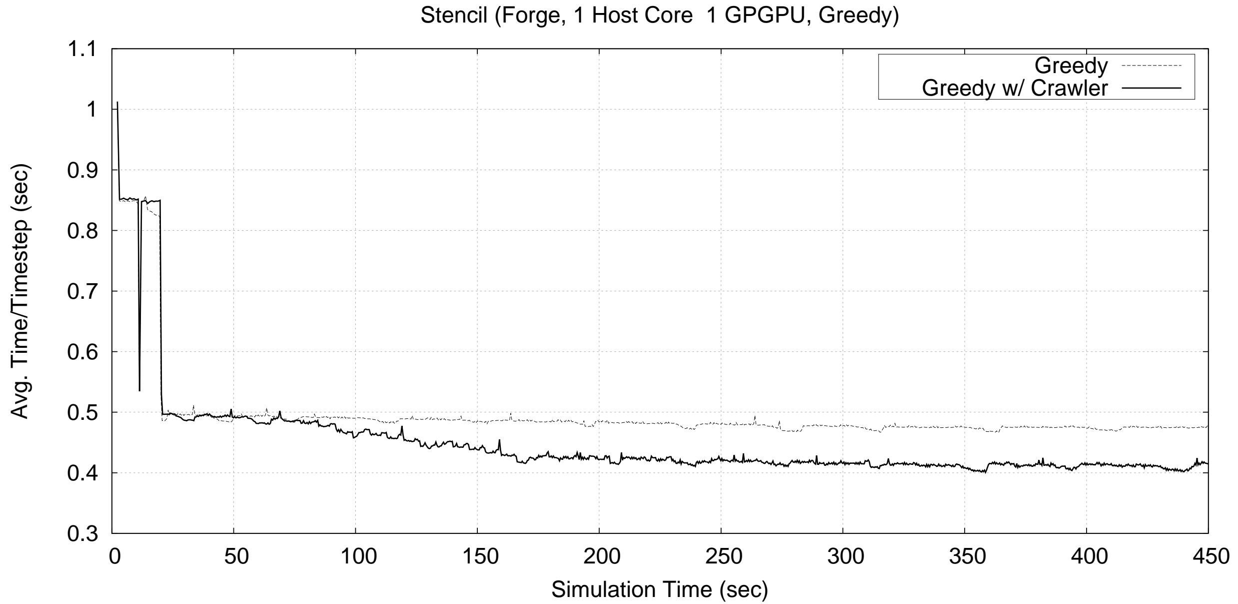


Figure 5.11: The 5-Point stencil application executing on a single host core and GPGPU pair on Forge using the Greedy strategy, with and without the Crawler strategy.

executes at a rate just over 1 timestep per second. While this period only continues for a short time in the stencil application, other applications that have more AEM types will have a longer profiling period. Further, depending on the relative performance for each AEM type on either the host core or the accelerator device, the period of time when the AEMs are largely being executed on the host core could result in a drastic slowdown in performance. Figure 5.11 shows that the stencil application experiences a slowdown of approximately two relative to the portion of the execution when the workload is balanced between the host core and accelerator device. Once the profiling period is complete, the greedy algorithm is applied, marked in the figure with an abrupt increase in performance. From this point on, the performance of the application stays fairly constant as the application proceeds to execute in the "Greedy" plot in figure 5.11, since the Greedy strategy takes no further action. This points out the previously mentioned disadvantage of the Greedy strategy, which is that it does not react beyond the time at which the greed algorithm was applied. This is one of the reasons to include yet another strategy after the greedy strategy in the queue of strategies generated during initialization, such as the crawler strategy.

The second plot in figure 5.11 shows the results of applying the crawler strategy after an initial balancing having been performed by the greedy strategy. The crawler strategy, as described in section 5.1.3, has a similar drawback to the sampling strategy in that it takes time to take measurements (samples) and make balancing adjustments based on those decisions. This process is slow and continuous, as the second, "Greedy w/ Crawler," plot illustrates (between approximately 25 and 170 seconds). However, by allowing accelerator

strategies to be chained, the greedy algorithm can be used to define the initial starting point used by the crawler strategy, greatly speeding up the crawling process by giving the Crawler a better starting point than just some arbitrary fixed point (e.g. all percent device values set to 100%). Figure 5.11 demonstrates that using the two strategies in conjunction with one another combines the advantages of the greedy algorithm with the advantages of the crawling strategy (which are similar to the advantages of the sampling strategy). As time goes on, the crawler strategy uses measurements to adjust the percentage of each type of accelerated entry method that is executed on an available device. As shown in figure 5.11, the crawler strategy is able to slowly improve upon the initial balancing provided by the greedy algorithm. However, the improvement isn't drastic, showing that the initial guess of the Greedy strategy, while not optimal, is a fairly good guess. Like all measurement based schemes, the quality of the output from the BaseLoad and KernelRatio profilers is only as good as the measurements themselves, including the various sources of noise present in computing systems and induced by the runtime system, as discussed in section 5.1.3 under "measurement noise."

Of course, a single strategy could have been created which combines all of the functionality of the BaseLoad profiler, the KernelRatio profiler, the Greedy strategy, and the Crawler strategy (i.e. if the chaining or queuing of strategies was not supported). However, this would create a duplication of functionality if multiple strategies required the same techniques. For example, both the AdjustBusy strategy and the Greedy strategy can make use of the output of the BaseLoad profiler, to the benefit of the stencil application, as demonstrated here. Thus, the ability to chain strategies does indeed demonstrably help ease the burden of introducing new accelerator strategies within the runtime system by allowing the functionality of a given strategy (or profiler) to be reused.

Strategies Compared: Finally, figure 5.12 includes results from the various strategies that have been applied to the stencil application so far, allowing the reader to more easily see their relative performance. Note that the vertical axis does not start at zero, therefor, the best and worst performing strategies are only separated by approximately 0.8 seconds per timestep (or approximately a 13.3% decrease in performance relative to the "Host Only" plot). Notice that the AdjustBusy and Greedy strategies perform similar to one another. This is not entirely unexpected since both approach the problem of load balancing the workload by trying to adjust the amount of busy time on both the host core and the accelerator device. Further, the Sampling (centralized) and Crawler strategies perform similarly to one another. Again, both of these strategies approach the problem of load balancing by measuring the actual effect that a change to the percent device value has on the performance of the application, rather than assuming that the best results will be achieved by balancing the busy times of the host core and the accelerator device.

It should be noted that, when these experiments are repeated, there is a small amount of variance in the

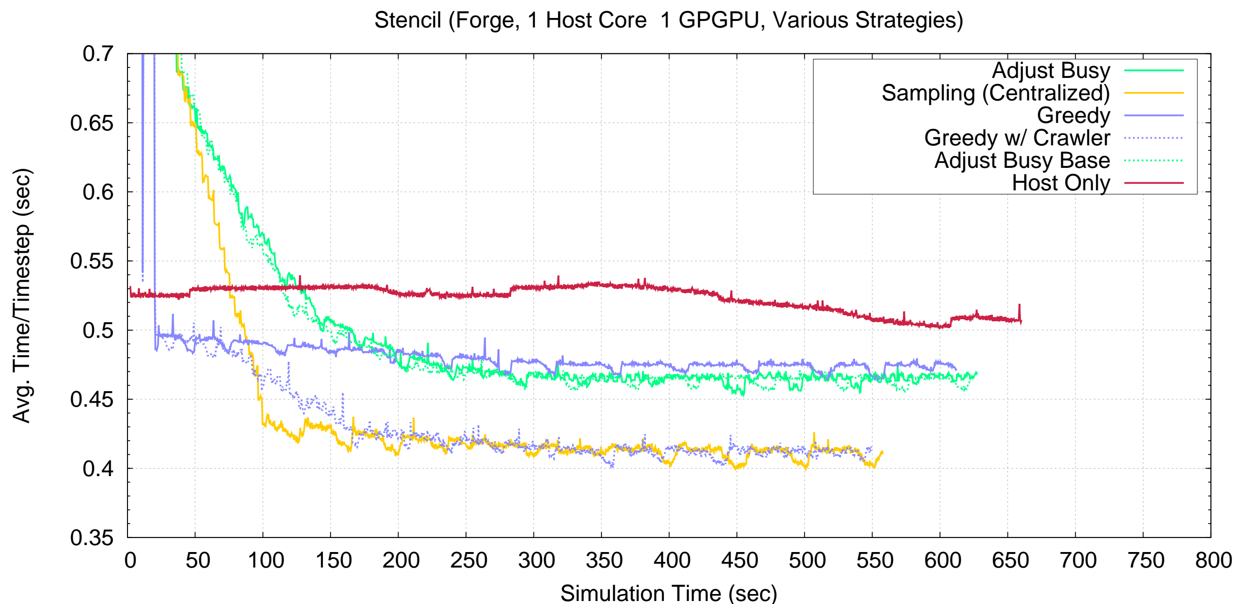


Figure 5.12: The 5-Point stencil application executing on a single host core and GPGPU pair on Forge using the various accelerator strategies available within the Charm++ runtime system. Note that the vertical axis does not start at zero.

relative performance of the various strategies. For example, sometimes the Greedy plot strategy ("Greedy" plot) does slightly better or worst (± 3%) than the AdjustBusy strategy ("AdjustBusy" plot). However, what does stay constant is that (1) the "Host Only" plot always has the lowest performance, (2) the Greedy and AdjustBusy strategies perform better than "Host Only" and relatively close to one another, and (3) the Sampling and Crawler ("Greedy w/ Crawler" plot) exhibit the best performance. As such, figure 5.12 gives a fair representation of the strategies' relative and absolute performances, regardless of the variance between executions.

5.4.2 Molecular Dynamics

We will apply the load balancing techniques discussed in this chapter to the same molecular dynamics (MD) application that was previously discussed in section 4.1 as part of the static load balancing experiments in chapter 4. First, the *triggered* and *splittable* keywords will be applied to the various accelerated entry methods within the application, updating the application code to make use of these newer additions to our approach. Once these modifications have been made, the application will have an increased level of flexibility that will allow the runtime system to make effective use of the GPGPUs, in addition to the Cell processor as previously discussed in section 4.3. Once the application code has been updated to take advantage of these newer extensions, the application of the accelerator load balancing strategies will be discussed.

Making Use of Splittable, Triggered, and Shared

The "splittable" and "triggered" keywords and their usage and effects were previously discussed in sections 3.2.1 and 3.2.1, respectively. The "shared" keyword causes the runtime system to share data buffers between accelerated entry method invocations that are included within a single batch of AEMs, reducing the overall amount of data that needs to be transferred per batch. This is accomplished at runtime using a dynamic pointer lookup mechanism. The usage and effects will be discussed in greater detail in section 6.2.2, as part of the discussion of data management.

Within the molecular dynamics application, there are three accelerated entry methods, including the force computation AEMs in the pair and self compute classes, along with the integrate AEM in the patch class. All of these AEMs have been updated to make use of the "splittable" and "triggered" keywords. Further, the "shared" keyword has been added to the local parameters containing the incoming readonly particle data. Additionally, the static load balancing mechanism within the application code has been removed, since the role of load balancing will be taken over by the accelerator load balancing strategies. Aside from these changes, the previous description of simple MD application's implementation in section 4.1 still applies. Section 6.2.2 includes a figure (figure 6.2) which illustrates the actual use of these keywords within the force calculation AEM in the pair compute class.

The most note worthy effect on the code relates to the use of the "splittable" keyword on the pair compute's force calculation AEM. Since there is no guarantee of the execution order of the individual splits created for a splittable entry method, there should be no data dependencies between the splits, as described in section 3.2.1. The original version of this force calculation AEM used the idea of equal-and-opposite forces to reduce the overall amount of computation required to calculate the forces. However, in the context of a splittable AEM, the use of the equal-and-opposite forces optimization creates data hazard between splits. As such, in the splittable version of this force calculation AEM, the equal-and-opposite force optimization has been removed. The impact of this change will be discussed in further detail in section 6.2.2. For the time being, we simply wish to bring this change to the reader's attention.

Application of Accelerated Load Balancing Strategies

Step Strategy: In this section, the various accelerator load balancing strategies will be applied to the simple MD application. As was the case with the 5-point stencil application, the Step strategy is applied first in an effort to get an idea of what to expect from the application as the workload is shifted between the host and the accelerator device. The performance of the MD application executing on a single host core and a single GPGPU is shown in figure 5.13. Note the maximum and minimum values used for the vertical

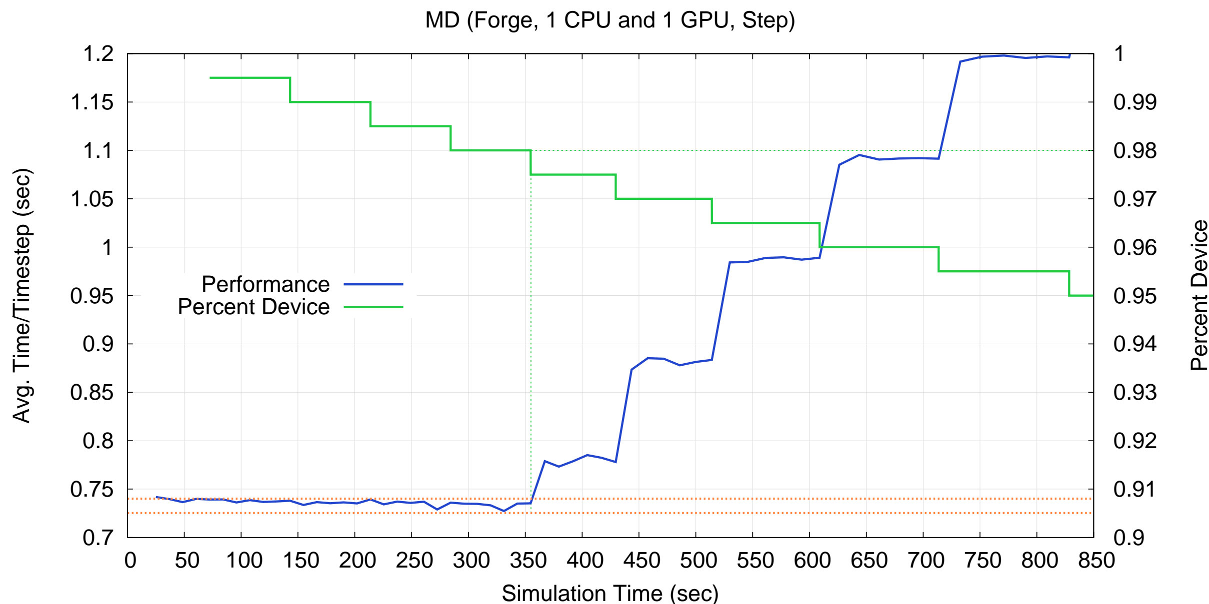


Figure 5.13: The molecular dynamics application scaled to a single node of Forge and using the Step accelerator strategy.

axis in the figure, since the figure is fairly *zoomed in*. As a result, the noise within the performance plot is also being emphasized.

There are a few things worth noting in figure 5.13. First, the percent device value at which the lowest point (highest performance) occurs is at a high percent device value (98%). As it turns out, the accelerated entry methods within the molecular dynamics application are much better suited for executing on an accelerator device. In particular, there is a fair amount of work to be done by each accelerated entry method compared to the number of bytes that need to be transferred per AEM invocation. The amount of data grows linearly with the number of particles per patch, while the work grows as the square of the number of particles within the computes' force calculation AEMs and linearly in the number of particles within the patch's integration AEM.

Second, the portion of the performance plot in figure 5.13 to the left of the *knee* (at approximately 350 seconds) is fairly flat. Recall figure 5.1(b), which shows the expected workload balancing outcome when the accelerator device is capable of out performing the host core by a significant factor. It is expected that the portion of the performance plot in figure 5.13 prior to 350 seconds will be fairly horizontal. Also noticed that the *knee* in the performance plot occurs as the percent device value is approximately 98%. Assuming a D_1 value of zero (which will not actually be the case since a minimal unit of work will not take zero time to execute), that would require the difference between D_{all} and D_1 to be equal to D_{all}, or a value of approximately 0.74 seconds from figure 5.13. Since the time taken by the accelerator is expected to decrease

(roughly) linearly relative to the percent device value, the D_{all} value multiplied by the percentage difference between 100% and the percent device value at the *knee* represents the expected increase in performance at the *knee*. 2% of 0.74 seconds is 0.0148 seconds, or an expected time of 0.725 seconds at the *knee*. However, keep in mind that the minimum of 0.725 seconds shouldn't actually be reached since the value of D_1 isn't actually zero, which was the assumption in this calculation. Two references lines, at performance values of 0.74 seconds and 0.725 seconds, have been included in figure 5.13 to clearly and visually show the relationship between these calculated values and the measured performance of the application. Note that the difference between these reference lines is quite small. In fact, the noise in the plot makes it a bit hard to see, but there actually is a downward slope on the portion of the performance plot prior to 350 seconds (comparing the performance plot to the two reference lines makes the downward slope a bit easier to see).

Third, it should be pointed out that the noise present in the plot may cause issues for the load balancing strategies. The portion of the performance plot prior to 350 seconds is not as smooth as the associated idealized line presented in figure 5.1(a). At the very least, we can say that this portion of the performance plot is nearly horizontal (as expected per the discussion above) and that the noise may cause false judgments within a load balancing strategy since the magnitude of the noise is approximately the same as the expected difference of 0.0148 seconds (between 0 and 350 seconds). As a result, it is unclear how much of a performance gain will be achieved by any load balancing strategy that can correctly 98% as the percent device value to use, versus a strategy that just generally keeps the percent device value somewhere between 100% and 98%.

Fourth, the performance of the MD application drops (i.e. time per timestep increases) fairly quickly as the percent device value drops below 98% (at approximately 350 seconds). This may point out a potential hazard for any measurement based load balancing strategy, since maintaining a percent device value above 98% is clearly not as bad as accidentally incurring the performance degradation caused by decreasing the percent device value below 98%. We will return to this idea as we discuss the results of using the Sampling strategy on the simple MD application.

Figure 5.14 moves on to show the performance of the molecular dynamics strategy on a single node of Forge, using various combinations of host cores and accelerator devices. The performance plot from figure 5.13 is the first plot, "A," in figure 5.14. The plots within figure 5.14 are all shaped as one would expected based on the discussion so far. That is, each plot starts with a "horizontal" region, eventually reaches a *knee* point, and then the further drop in the associated percent device value (not shown) over time results in a steady decrease in performance.

However, there two observations that we would like to point out. First, the *knee* points are staggered in a seemingly odd pattern. Second, the performance within the "horizontal" region of configuration F is lower

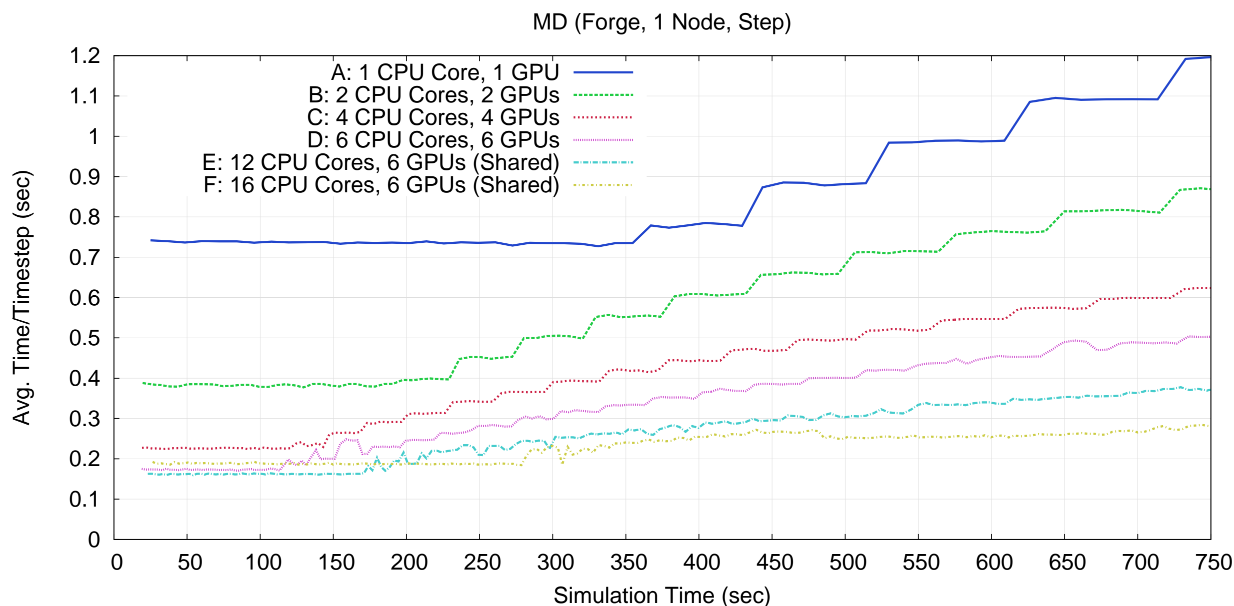


Figure 5.14: The molecular dynamics application scaled to a single node of Forge and using the Step accelerator strategy.

(higher time) when compared to the "horizontal" region of configuration E. To understand these effects requires an understanding of the percent device values at each of the configurations' respective *knee* points, which will be discussed below within the context of the AdjustBusy strategy. For the time being, we simply point out these irregularities.

Before moving on to the AdjustBusy accelerator strategies, we should mention one additional difference between the configurations. The number of timesteps that the molecular dynamics program was executed differs depending on the hardware configuration. Generally speaking, as more hardware resources were used within a configuration, the number of timesteps executed for that configuration was also increased, so there would be plenty of time for the percent device value to stabilize. Unless stated otherwise, this is true for all figures related to the performance of the MD application throughout this section.

AdjustBusy Strategy: Figure 5.15 shows the performance of the MD application with the AdjustBusy strategy being applied to the same hardware configurations that were used for the Step strategy. For the most part, the plots in figure 5.15 are as one might expect. After the AdjustBusy strategy has had some time, the performance levels are similar to the "horizontal" regions of the corresponding plots in figure 5.14. However, configurations D and F exhibit a fair amount of noise in their performance.

To help understand what is going on, figure 5.16 presents the same data for configurations D, E, and F using four *zoomed in* graphs. The first (top) graph simply repeats the performance plots from figure 5.15 so that the reader can more easily compare the performance of these configurations. Notice that from

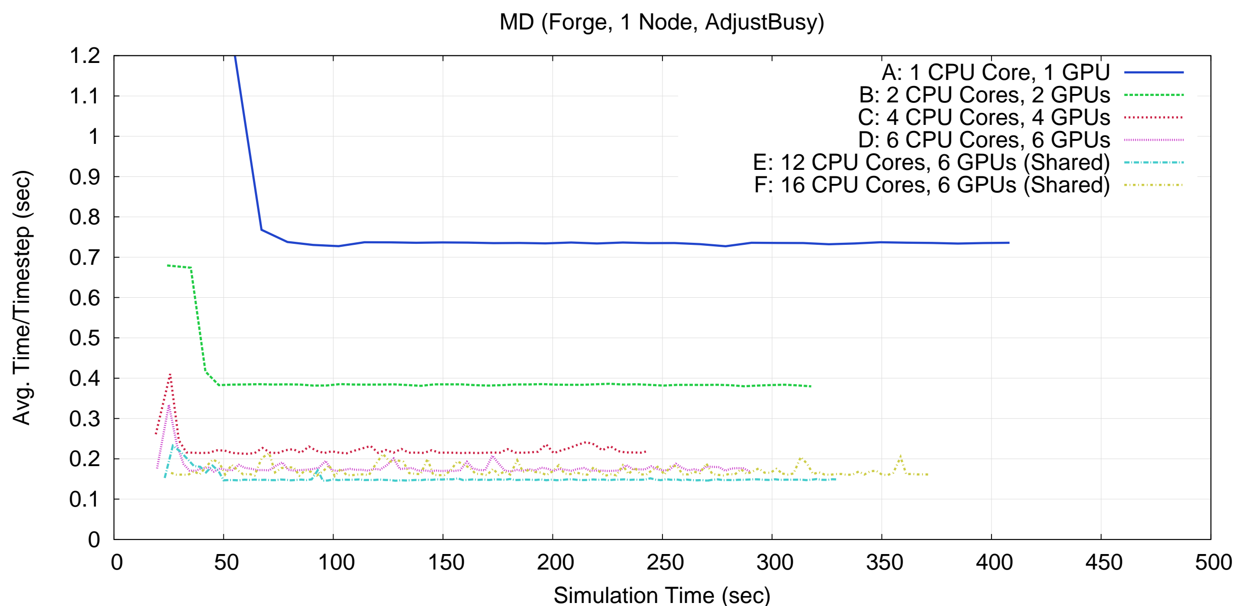


Figure 5.15: The molecular dynamics application scaled to a single node of Forge and using the AdjustBusy accelerator strategy.

approximately 125 seconds to 250 seconds, when configuration D completes, configuration E is generally performing better than configuration D (we are ignoring the noise for a moment, but will discuss it below). The combined peak flop rate of configuration D is 6.295 Tflop/s. The combined peak flop rate of configuration E is 6.410 Tflop/s. The peak flop rate difference between these two configurations is 115 Gflop/s, an increase of 1.83% relative to D's flop rate. Dividing a time of 0.15 seconds (all measured times throughout the simulation for configuration D are greater than 0.15 seconds) by 1.0183 gives a time of 0.1473 seconds (reference lines of $y = 0.15$ and $y = 0.1473$ have been included in the top graph for reference). While it is hard to say conclusively because of the noise levels in the plots, it seems as though configuration E performs at approximately this level of performance during the flat regions of its performance plot, which is a bit better than expected considering D never reaches a performance level of 0.15 seconds per timestep. A likely cause is the reduced pressure on the host cores. In particular, each host core has approximately half the number of compute objects with only a slightly lower percentage of those objects being issued to the accelerator devices (as shown in the lower three graphs, as describe below), allowing the host core to seem as if it were a bit more responsive to the accelerator device in E relative to D. To put it another way, from the point-of-view of an accelerator device in configuration E, since there are two host cores feeding the accelerator work, neither of those host cores have to be as reactive as they would need to be if either were the only host core feeding the accelerator work.

However, the performance of configuration F is clearly lower (higher times) when compared to either D

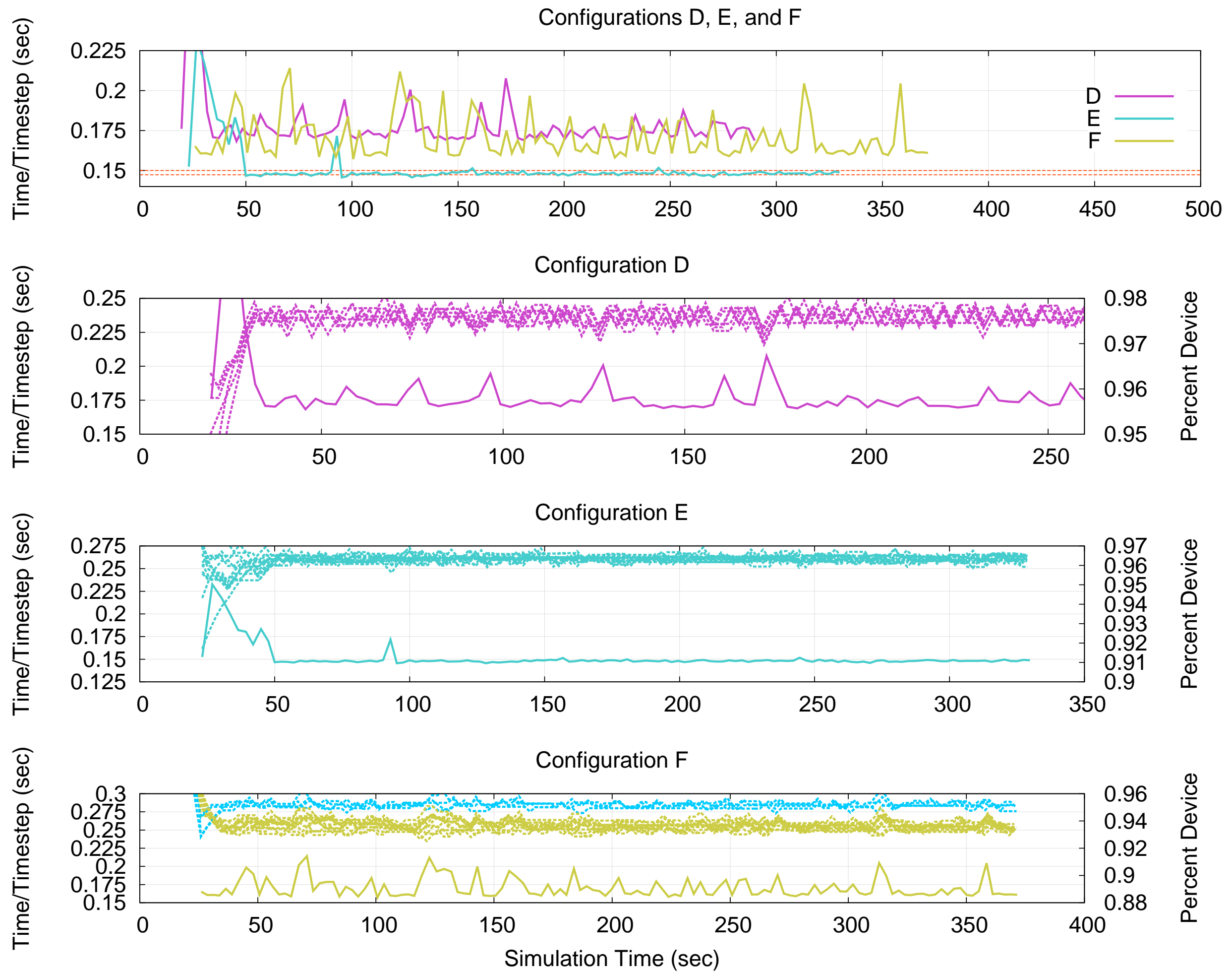


Figure 5.16: Zoomed in views of the data in figure 5.15, along with corresponding percent device values.

or E. This is also to be expected, since the distribution of work in configuration F is uneven per pairing. In configuration F there are 16 host cores and 6 accelerator devices. Note that 16 is not a multiple of 6, and recall that each host core is associated with one and only one accelerator device. As a result, some of the accelerator devices are being shared amongst 3 host cores, while others are only shared amongst 2 host cores. However, since the overall application workload is evenly distributed amongst the host cores and the accelerator load balancing strategies only act locally to a given host core and accelerator device pairing, there is currently no way to overcome the misbalance of work between the pairings without an across host core load balancer (will be addressed later in this section). This misbalance causes the performance of configuration F to suffer. In configurations A through E, the number of host cores is a multiple of the number of accelerator devices, allowing the default distribution of chare objects (i.e. an equal portion of the workload is assigned to all host cores) to result in an even distribution of work to naturally occur across all of the accelerator devices.

The lower three graphs in figure 5.16 show the performance plots of configurations D, E, and F, respectively, along with the percent device values for each of the cores over time. The solid lines are the performance plots, while the dotted and largely overlapping lines are the percent device plots. In the case of configuration F, two colors are used for the percent device plots, one for host cores that share their associated accelerator device with one other host core (cyan) and one for host cores that share their associated accelerator device with two other host cores (yellow). Lower percent device values are being used on host cores associated with accelerated devices shared between three host cores, while slightly higher percent device values are being used by host cores associated with accelerator devices shared by only two host cores. Since all host core and accelerator device pairings are being assigned the same amount of work, it makes sense that a lower percent device value would be used by pairings that are associated with accelerator devices that are shared amongst more host cores.

Finally, the final percent device values across all of the pairings for each configuration is shown in figure 5.17. With the exception of configuration F, which would have a slightly larger range, the last assigned percent device values are roughly the percent device values used through each configuration's execution. We can relate these final percent device values back to figure 5.14 to help explain the location of the *knee* points in the performance plots generated when using Step strategy. The percent device values associated with the *knee* points in figure 5.14 for configurations A through D are relatively the same, as shown in figure 5.17. Since a fixed number of timesteps separate each percent device adjustment when using the Step strategy and the sampling rate increases as the performance increases (between configurations), the percent device value of the *knee* is reached in less time (relative to the performance increase). Configurations E and F

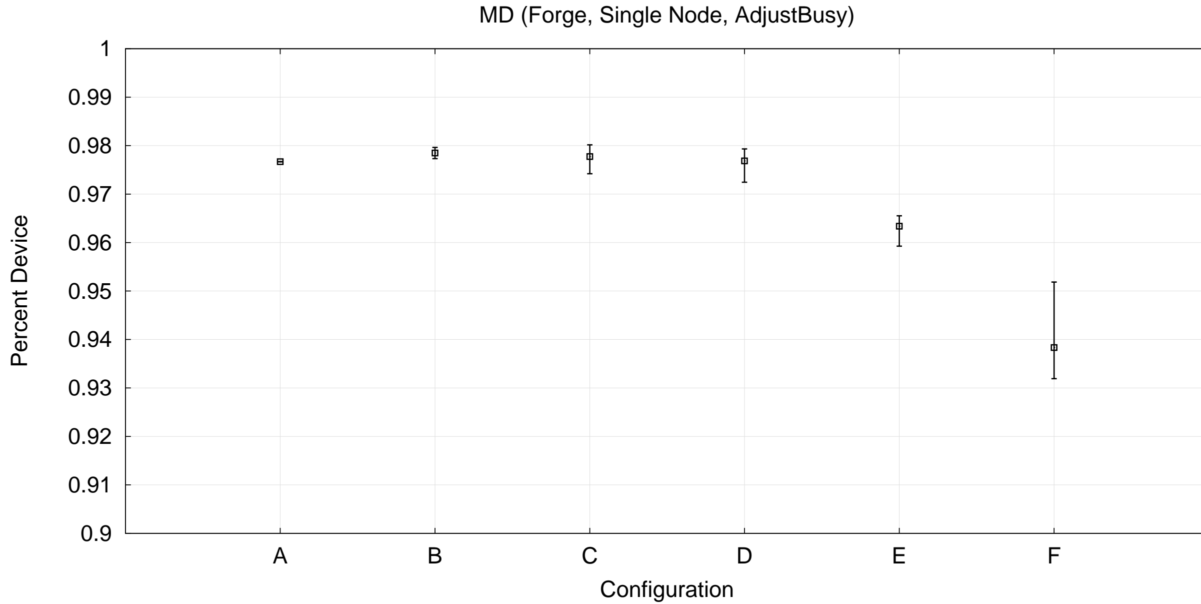


Figure 5.17: The final percent device values for the various configurations used in figure 5.15.

only add a small amount to the available peak flop rate provided by the respective configurations, relative to configuration D. However, all of the added flops come from the addition of more host cores, resulting in a drop in the percent device value associated with the *knee* point. Since the performance doesn't significantly increase, and thus the sampling rate does not significantly increase, from D to E to F, the time taken to reach the *knee* point increases from D to E to F.

The plots in figure 5.18 are the same as the plots in figure 5.15, except that the across host core load balancer, AccelEvenLB, has also been applied to the runs. With the addition of the across host core load balancing, plot F is performing as well as plot E. Recall the differences between these two configurations. In E, each GPGPU is shared by an equal number of host cores (2). In F, some GPGPUs are shared between two host cores, while others are shared between 3 host cores. Because of this imbalance, configuration F performed worst that E when no across host core load balancer was applied (figure 5.15), which has been corrected for in figure 5.18. Its hard to come to any conclusion as to whether or not the performance of F is doing better or worst than E because the noise being seen within the runs is higher than the performance gain one would expect to see from the additional host cores which only increase the combined peak flop rate by 1.2%. Essentially, configuration F is harder to load balance and offers little benefit over E in terms of the peak flop rate. For reference, we have included two horizontal lines in the lower, *zoomed in* graph in figure 5.18 that visually show what an increase of 1.2% would look like based on an initial performance level of 0.16 seconds per timestep.

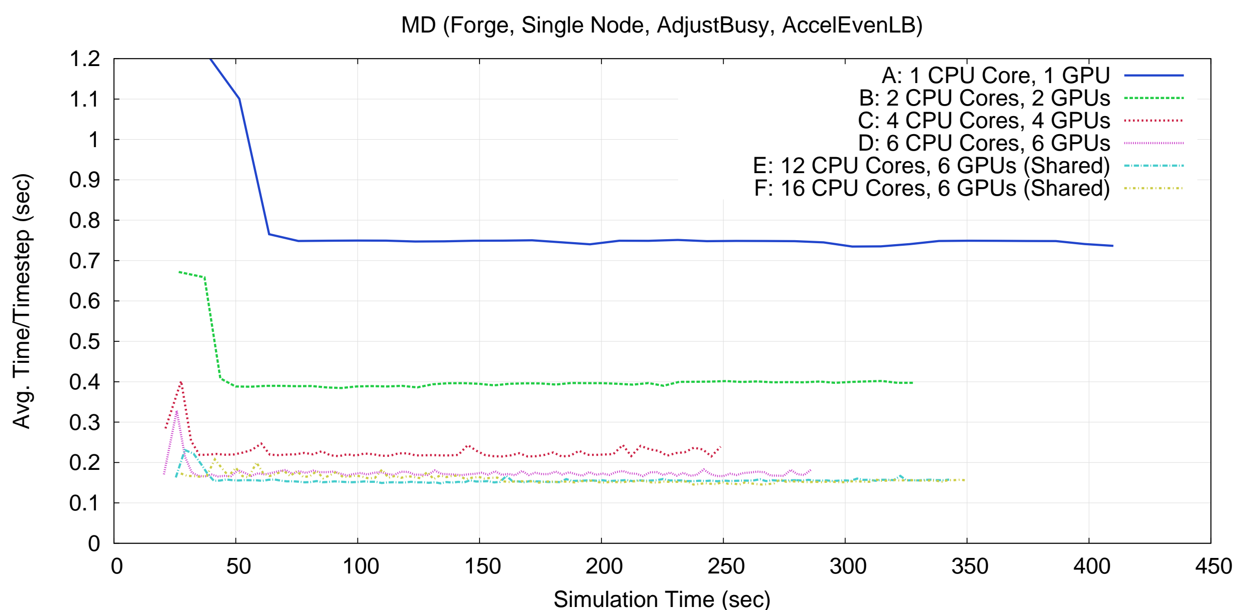


Figure 5.18: The molecular dynamics application scaled to a single node of Forge and using the AdjustBusy accelerator strategy along with the AccelEvenLB load balancing strategy.

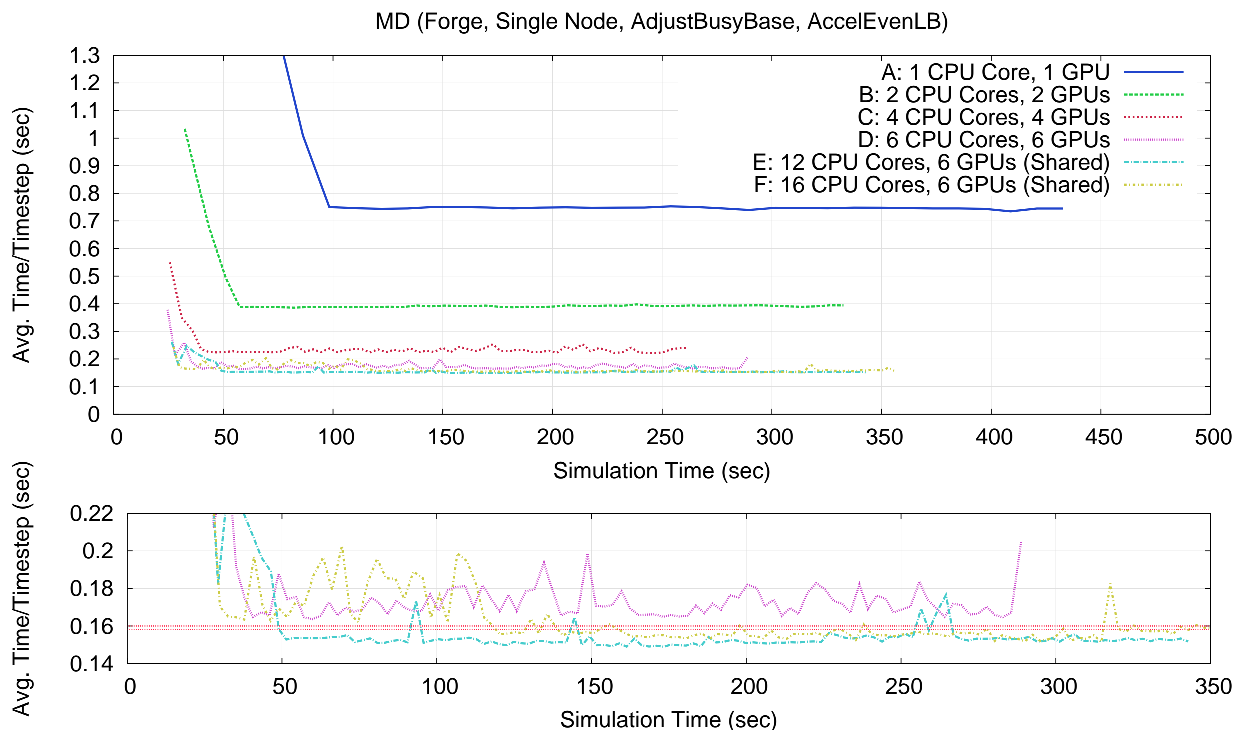


Figure 5.19: The molecular dynamics application scaled to a single node of Forge and using the BaseLoad profiler, AdjustBusy accelerator strategy, and the AccelEvenLB load balancing strategy.

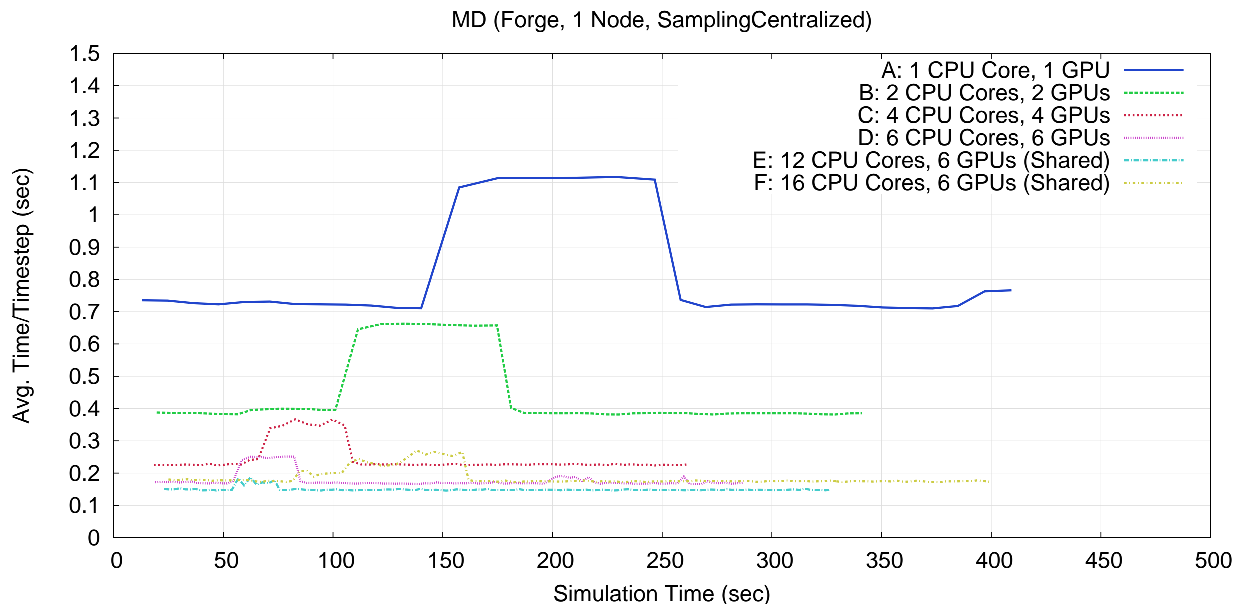


Figure 5.20: The molecular dynamics application scaled to a single node of Forge and using the Sampling-Centralized accelerator strategy.

Figure 5.19, presents the same set of plots that were present in figure 5.18, except that the BaseLoad profiler has also been used for these runs. The combined peak flop rate increases by only 1.2% when moving from configuration E to F. There are two horizontal reference lines in figure 5.19, visually indicating a 1.2% performance increase starting from a base performance level of 1.6 seconds per timestep. As can be seen, the noise within the plots is too large in magnitude to determine if the performance of the application does indeed increase by an additional 1.2% when moving from configuration E to F. One may wonder why the performance of configuration E is clearly better compared to the performance of configuration D. When moving from configuration D to E, the number of host cores associated with each accelerator device increases from 1 to 2 (an equal number per accelerator device in both cases). Configuration E not only provides the additional processing power of the added host cores, but also begins sharing the accelerator devices between host cores (i.e. there are two host cores feed work to each accelerator device, rather than one). As such, switching from configuration D to E represents a more significant change in the hardware configurations compared to the shift from E to F.

Figure 5.20 applies the Sampling (centralized) accelerator load balancing strategy to the same configurations used for testing the Step and AdjustBusy strategies. The most notable feature of the plots is the large *humps* that occur for each configuration. In assist in the explanation of why these humps are occurring, we have included a simplified (idealized) scenario in figure 5.21 to illustrate the cause. Aside from the *humps* and the beginning of the executions, the Sampling and AdjustBusy strategies result in similar performance.

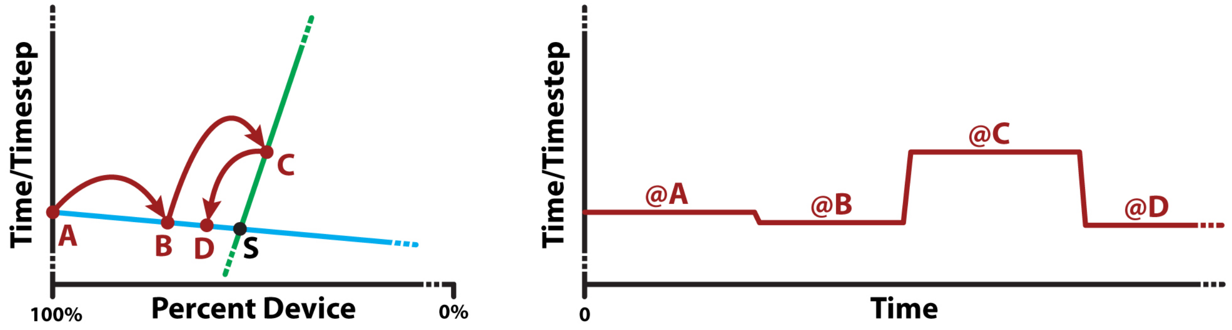


Figure 5.21: Illustration explaining the “humps” in figure 5.20.

To understand how the *hump* occurs, see the idealized scenario in figure 5.21. There are two graphs in figure 5.21, the percent device graph to the left and the time graph to the right. In both cases, the vertical axis represents the performance of the application (i.e. time per timestep), with a lower value equating to better performance. The left graph, is the same as the workload balancing graph (i.e. as shown in figure 5.1(a)). The arrows show how the sampling strategy moves between the points A, B, C, and D as the the strategy takes performance measurements in an attempt to find the sweet spot S. The strategy starts with a percent device value of 100% (A). As time goes on, the strategy starts decreasing the percent device value until, at some point, it passes the point S (point C in figure 5.21). Note that the Sampling strategy (or any strategy like it) must pass the point S in order to detected it (i.e. in order to detect a valley). After sampling at points A, B, and C, the Sampling strategy will believe that it has found a minimum at point B. Keep in mind that the strategy has incomplete information, not the entire graph as shown in figure 5.21. Once it is time to move one from point C, the Sampling strategy (or any strategy that works in a similar way, regardless of the exact implementation) will likely begin picking points around point B (e.g. point D). The graph on the right side of figure 5.21 shows the corresponding time graph (equivalent to the plots in figure 5.20). The four regions in the time graph in figure 5.20 relate directly to the points in the percent device graph. Note that the time spent in each region is not drawn to scale. For example, the region in which the most time will be spend is $@C$ (of just the four shown and assuming the same number of samples are collected in each). If, for example, the time per timestep of C is twice that of B, then the region marked as $@C$ will last twice as long as the region marked as $@B$.

In support of this analysis, figure 5.22 shows the performance of the MD application executing on a single host core and a single GPGPU (i.e. configuration A from figure 5.20). Along side the performance plot, the percent device value is also shown. From 5.13, it is known that the best balance point for this application running on a single host core and a single GPGPU is approximately 97.5%. In particular, there are two

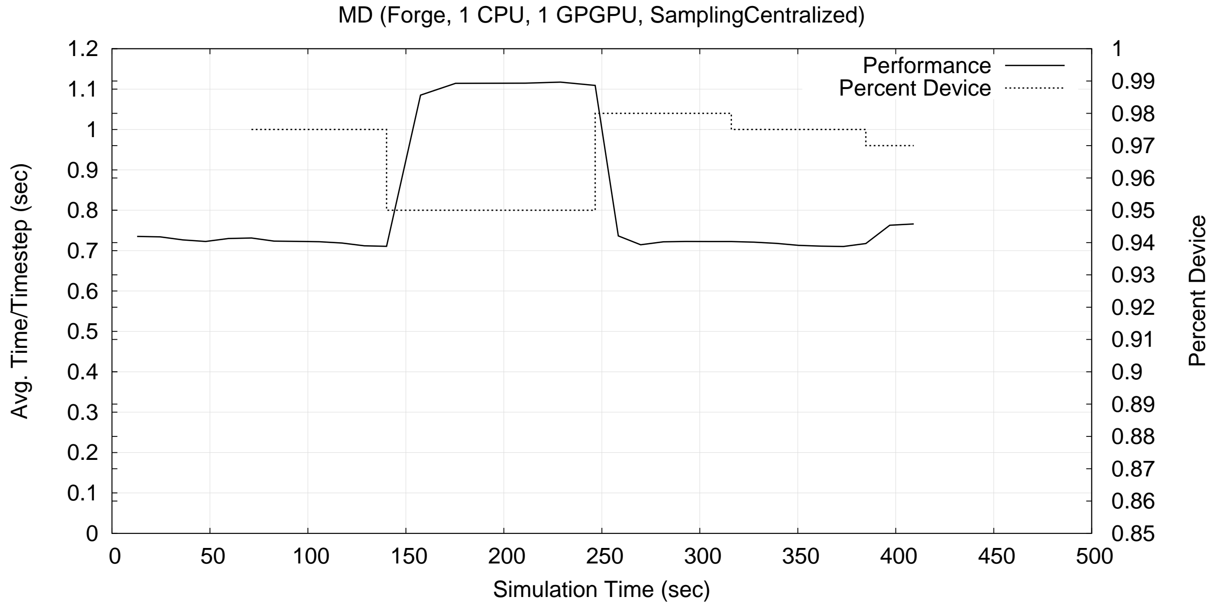


Figure 5.22: The molecular dynamics application on a single host core and a single GPGPU using the Sampling strategy.

points at which the percent device value drops below 97.5% (i.e. the large hump and the small performance drop at the end of the execution). The performance levels seen in figure 5.22 correspond to the performance levels measured in figure 5.13 for corresponding percent device values.

There are a few observations that should be pointed out in relation to figures 5.21, 5.22, and 5.20. First, the height of the hump ($@C$) in the time graph is directly related to the location of the point C in the percent device graph, which is in turn, directly related to the performance difference between the host core and the accelerator device (i.e. the slopes of the lines in the percent device graph from figure 5.21). Recall from the discussion in section 5.1.1 that as the accelerator device becomes more and more effective at executing the workload, the slope of the line representing the host core's time to execute its portion of the workload (i.e. the green line) will become greater and greater in magnitude. Second, any load balancing strategy must sample the percent device graph on both sides of S, regardless of the exact implementation of the strategy, in order to discover S (i.e. detect a *valley* in the workload balance graph). Third, the further to the right (lower percent device value) beyond S a strategy samples, the more of a performance impact there will be in the application and the longer this diminished performance region (i.e. $@C$ from the example in figure 5.21 assuming the number of samples per adjustment remains constant). The more the imbalance between the performance of the host core and the accelerator device, the greater the performance impact of even a small overstep to the right of S.

The observations mentioned above have two implications on the load balancing strategies that would

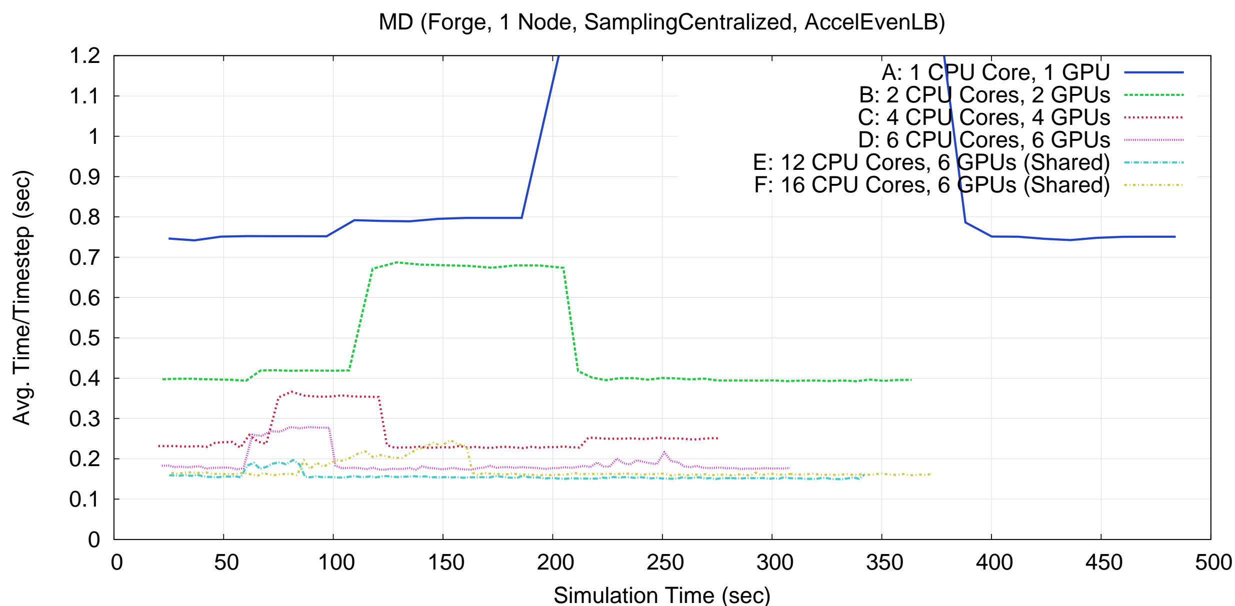


Figure 5.23: The molecular dynamics application scaled to a single node of Forge and using the Sampling-Centralized accelerator strategy along with the AccelEvenLB load balancing strategy.

control the balance of a workload between host cores and accelerator devices. The first directly follows from the observations, that an overstep will be costly in cases where there is a large performance imbalance. For long running applications, the associated time period(s) of lower performance will likely be negligible since their amortized impact will be minor. However, for short running applications, an over step in to the region of the graph where the application is host core bound (i.e. to the right of S) could be quite costly. However, strategies designed for short running applications are likely to be aggressive, so that they can quickly converge on S. This aggression may result in large oversteps. However, perhaps profiling and across execution memory could assist with shorting running applications. This may allow a less aggressive strategy to make smaller steps, because of a good starting point based on data from previous executions of the application. Second, as a strategy approaches the point S, it is likely to bounce back and forth between the left and right sides of S. This is especially true for strategies that do not assume that the point S is stationary and must periodically sample either side of S to detect a shift. If the hardware system the application is running on has sources of interference, such as operating system interference, other applications executing, etc., the location of S may shift as the interference comes and goes. Such strategies will likely test percent device values to either side of S on a regular basis. However, such testing can cause noise within the application as the percent device value moves back and forth to either side of S.

Figure 5.23 shows the performance of the MD application when using both the Sampling (centralized) accelerator load balancing strategy and the AccelEvenLB across host core load balancing strategy.

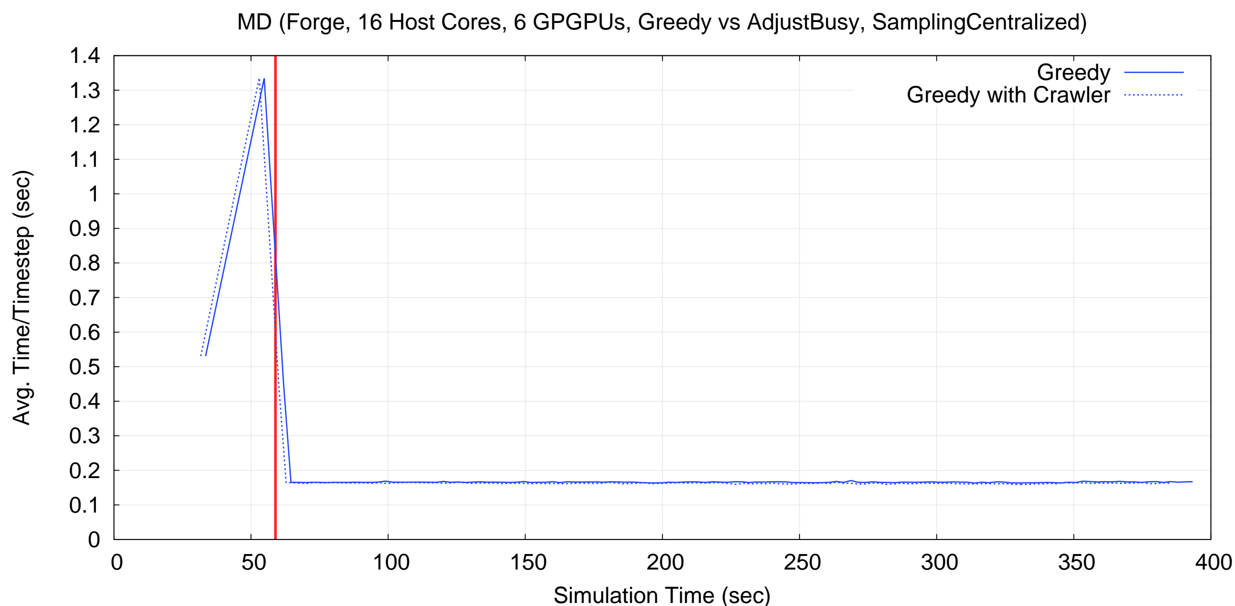


Figure 5.24: The MD application using the Greedy strategy, with and without the Crawler strategy.

Greedy and Greedy With Crawler: Figure 5.24 goes on to apply the Greedy strategy, with and without being followed by the Crawler strategy, to the MD application. Recall that the Greedy strategy relies on the output of both the BaseLoad and KernelRatio profilers. The profilers are applied at the beginning of the application's execution. A vertical reference line has been included in figure 5.24 to mark the time when the Greedy strategy applies its one-time balancing algorithm (at approximately 60 seconds). Note that the MD application only reports its performance after every 16 timesteps (as it was configured for these executions). As such, there is some delay between the Greedy algorithm being applied and the resulting performance increase being reported in the application's output. From figure 5.24, the application of the Crawler strategy seems to have no difference on the application's performance beyond the increased provided by the Greedy strategy itself. However, recall that the Crawler strategy cycles through a rotation sequence in order to test individual percent device value adjustments for each type of accelerated entry method, and that several samples are take for each of those adjustments. The result is that the Crawler strategy can take quite a bit of time to make adjustments.

Figure 5.25 shows the results of the MD application executing using the same configuration as was used in figure 5.24, but for many more timesteps (2048 in figure 5.24, 30720 in figure 5.25). From figure 5.25, we can see that the Crawler strategy can, in fact, provide a performance benefit over the one-time Greedy strategy, if given the time to do so. Note that the vertical axis does not start at zero and is quite a bit more *zoomed in* compared to figure 5.24, emphasizing the noise present in the performance plots. Additionally,

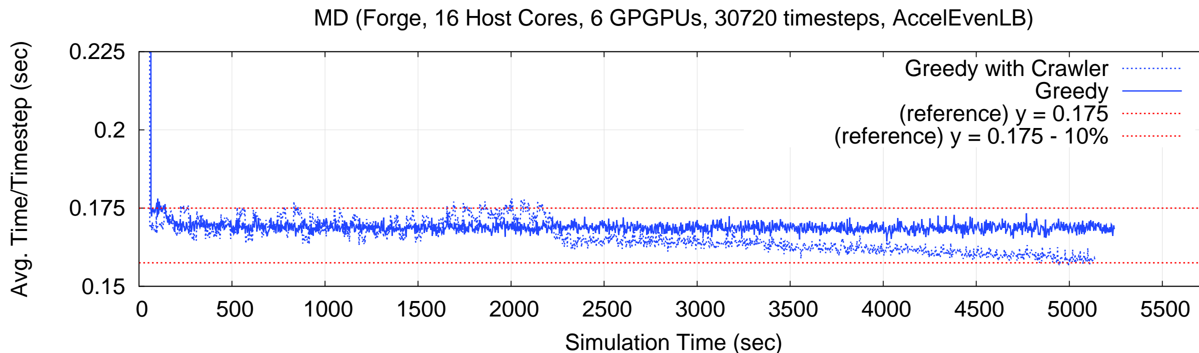


Figure 5.25: The MD application executing for 30720 timesteps using the Greedy strategy, with and without the Crawler strategy.

two reference lines have been included to give the reader a sense of scale. The upper reference line is set to 0.175 seconds per timestep, and the lower line represents a 10% decrease in the time per timestep of the upper reference line.

Finally, figure 5.26 shows the performance of the MD application using the Greedy, Greedy with Crawler, AdjustBusy, Sampling *(centralized)*, and DeviceOnly over the course of 30720 timesteps. For clarity, figure 5.26 has two graphs. The upper graph shows the raw performance output of the MD application while the lower graph shows the same data smoothed so the general trend of each plot is more readily visible. Note that both graphs have the same *zoomed in* view used for figure 5.25, emphasizing the noise in the plots.

There are a few points that we wish to make in reference to figure 5.26. First, the Crawler strategy is capable to increase the performance of the application if it is given enough time to do so. However, it is worth noting that the Crawler strategy can take quite some time to do so, since it must cycle through its rotation sequence several times to have a noticable impact. Second, while it is not clear from the figure, there is some performance variability between runs (plots can shift vertically by approximately 5%, up or down). The same reference lines from figure 5.25 have been included in figure 5.26 to give the reader a sense of scale. What is consistantly true, however, is that the DeviceOnly strategy is not the best performing strategy. Even in a situation where the GPGPUs are providing a large fraction of the combined peak flop rate (approximately 95% of the combined peak flop rate), the host cores are still able to make a noticable contribution to the performance of the application.

Scaling to Multiple Nodes

Table 5.1 shows the results of scaling the application from one to six nodes of the Forge cluster. In all cases, the execution time of the simulation (only simulation time was counted, not startup or shutdown) was

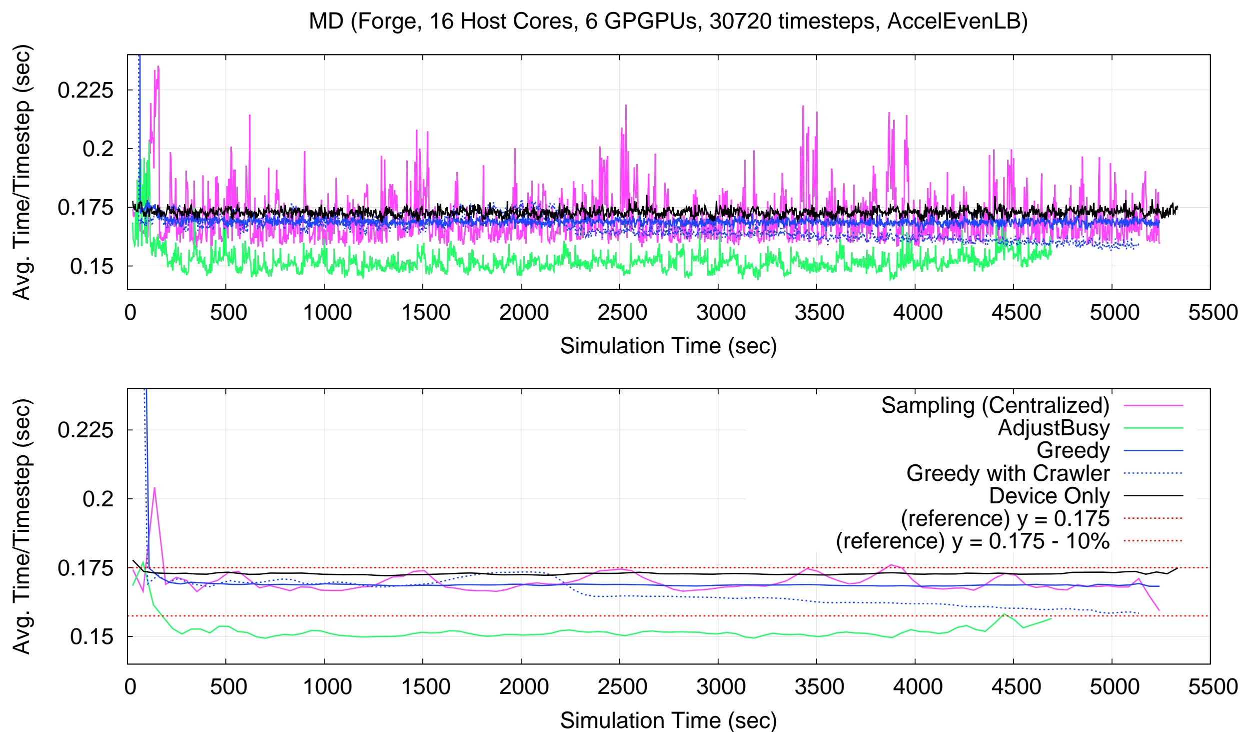


Figure 5.26: The MD application executing for 30720 timesteps using a variety of accelerator load balancing strategies.

Nodes	CPU Cores	GPGPUs	Peak Gflop/s	Time	Measured Gflop/s	% Peak	Efficiency
1	1	1	1049.20	96.55	247.72	23.61%	100.00%
1	16	6	6487.20	19.94	1199.44	18.49%	78.31%
2	32	12	12974.40	11.57	2067.35	15.93%	67.49%
3	48	18	19461.60	9.20	2599.54	13.36%	56.57%
4	64	24	25948.80	7.93	3016.99	11.63%	49.24%
5	80	30	32436.00	7.46	3208.01	9.89%	41.89%
6	96	36	38923.20	6.74	3547.76	9.11%	38.60%

Table 5.1: Scaling performance of the simple MD application from one to six nodes on Forge with GPGPU sharing.

divided by the number of timesteps executed. In the case of table 5.1, the *time* column represents the time to execute 128 timesteps (128 timesteps is an arbitrary choice with no special significance). The values in the *efficiency* column are calculated using 'the measured speedup of the application relative to the first row (1 CPU core and 1 GPGPU)' divided by 'the ratio of the combined peak flop rates of the configuration in question versus the combined peak flop rate of the base configuration.'

The base case of a single host core and a single GPGPU achieves a measured flop rate of 247.72 Gflop/s, or 23.61% of peak. While it is likely that a higher level of performance could be achieved by a programmer programming the force computation kernels by hand, making use of all the hardware features of the GPGPU hardware available to them. Doing so would require quite some effort, and the resulting code would not be portable to other platforms without a fair amount of changes to the code. When considering the performance achieved by the MD application using our programming model extensions and runtime system modifications, recall that the application code does not include any architecture specific code, and as such, the generated GPGPU code is not making used of any hardware specific synchronization mechanisms, constant memory, or texture memory provided by GPGPU hardware (unless the NVCC compiler provided within the CUDA SDK performs these optimizations on the generated code internally). Further recall that the application code does not include any explicit batching of accelerated entry methods into kernels or even kernels themselves. The application programmer programs the MD application using the programming model available to all Charm++ applications (though, using AEMs when appropriate). Further, the SIMDIA abstraction allows SIMD instructions to be used on the host core and scalars to be used within the GPGPU threads. The MD application is capable of achieving 23.61% of peak using a single host core and a single GPGPU, and at the same time, is written in a portable manner using the same programming model and execution model that is presented to any other Charm++ application executing on any other hardware platform. There are certainly additional optimizations that could be made within the runtime system and additional optimizations to be made at compile time that could further improve the performance of such applications executing on platforms that include accelerator technologies. However, we believe that this work, at the very least, shows that applying a message-driven (event-driven) approach to accelerator technologies is not only feasible, but has several advantages over other approaches (discussed further in chapter 8).

As the MD application is scaled from one node to six nodes, the performance continues to increase, however, the efficiency steadily decreases. Note that six nodes on Forge contain a total of 96 host cores and 36 GPGPUs, where 36 GPGPUs = 504 streaming multiprocessors (SMs) = 16128 streaming processors (SPs or SIMD cores). Having given the core counts, it is worth noting that a direct comparison between the host cores and either the SMs or SIMD cores within the GPGPUs is inappropriate (i.e. they are fairly

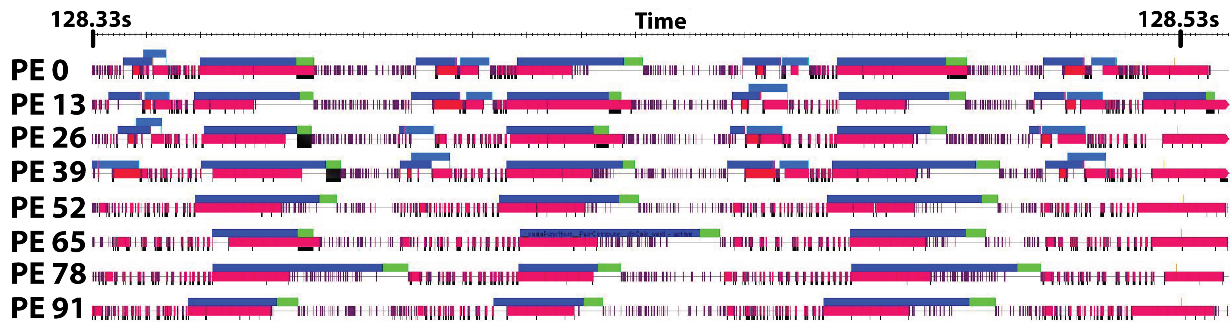


Figure 5.27: Timeline of the molecular dynamics application executing on six nodes of Forge.

different, so trying to convert the number of one type into an equivalent number of the other type in an attempt to give a single value for the "core count" would not be fair). In terms of the peak flop rate, the 96 host cores and 36 GPGPUs have a combined peak flop rate of 38.92 Tflop/s. Assuming an x86 host core with a 256-bit (8 32-bit elements) wide SIMD register and support for fused-multiply-add SIMD instructions executing at 3.0 GHz (i.e. similar to modern host cores), an equivalent host-core-only system would require approximately 811 such cores to have an equivalent peak flop rate. In the six node case, the efficiency has decreased to 38.6%. To understand why this is occurring, a Projections Timeline graph is presented in figure 5.27 for the six node case (last row of table 5.1).

The timeline in figure 5.27 shows the activity on every 13th host core, and associated accelerator device, over the course of 3 timesteps (the start of a fourth on the right-hand side is only partially shown). Not all host cores are shown for brevity, and a prime number is used for the stride between processors to avoid any patterns based on host core counts or GPGPU counts (e.g. some effect that only affects the first host core in each node). Each host core (or "PE") has several rows associated with it, with the bottom most row (thickest) showing entry methods being performed on the host core itself (one "block" per entry method invocation) and one or more (thinner) rows above that showing the activity on that host core's associated accelerator device. The red blocks show force and integration calculations being executed on the host cores, while the blue blocks show force and integration calculations being executed on the accelerator devices. Time increases from left to right.

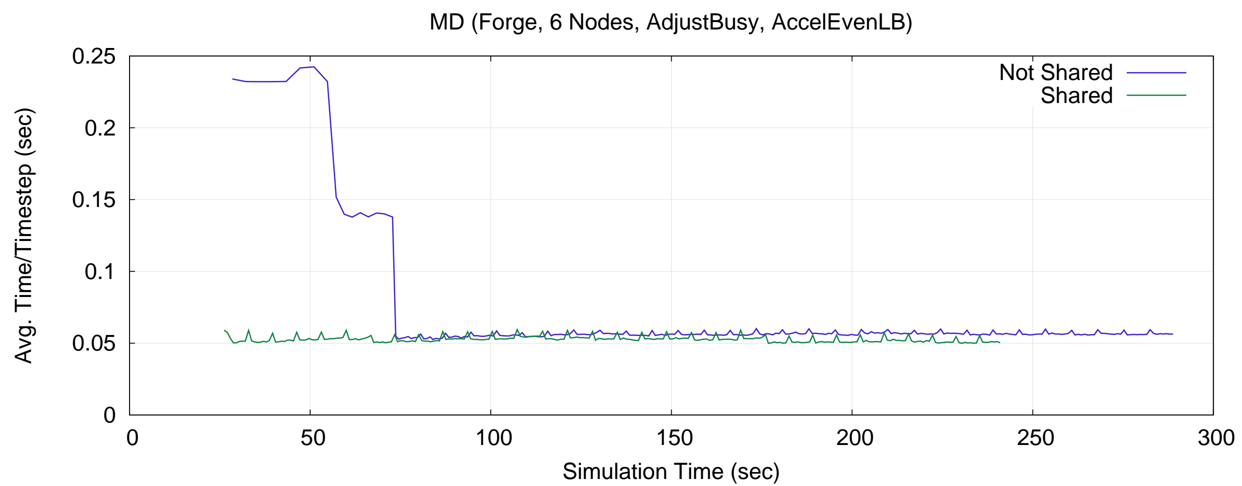
The first thing to take notice of when trying to understand the efficiency drop when using six nodes of Force is the time taken to execute the portion of the workload assigned to the accelerator devices each timestep. In particular, take note of the accelerator work associated with "PE 78." There is a large degree of variability in the time it takes the accelerator device to execute its workload from timestep to timestep (the first timestep shown is relatively long, the second is relatively short, and the third is relatively long). From examining the execution's output, it can be determined that no load balancing occurred (between the

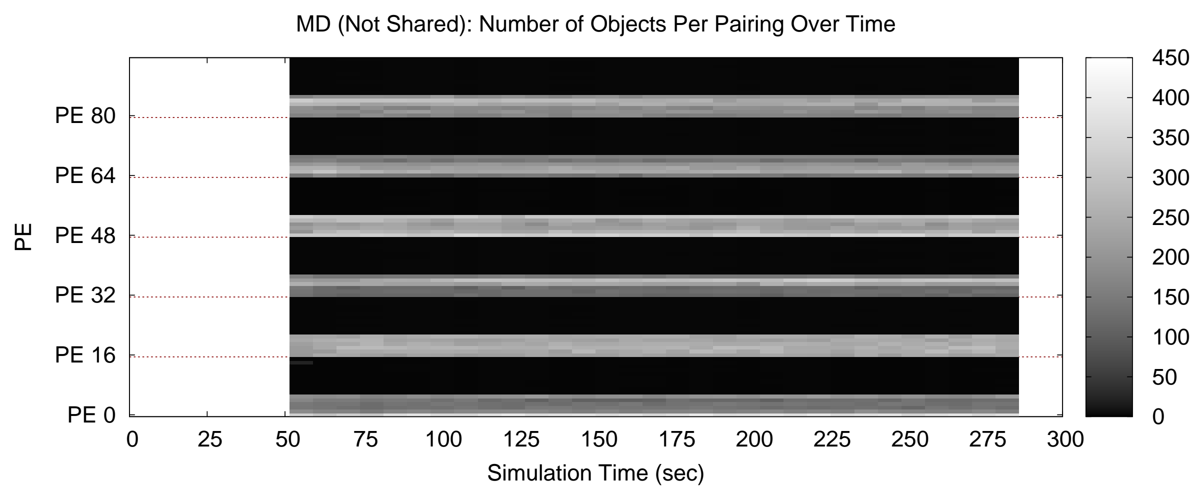
host core and the accelerator device, or between any host cores) during the time period shown. As the MD application scales, this variability has a large impact on scalability since the time per timestep has a lower bound which is determined by the slowest host core and accelerator device pairings. Since the GPGPUs are being shared between host cores, the kernels issued by one host core may be slowed as already executing kernels finish executing (as one possible source of kernel execution variability). Note that the blocks shown in figure 5.27 executing on the accelerator devices include all data transfers associated with the kernel, the kernel execution itself, and any delays (i.e. represents the time from the host core issuing a batch set to the time the callback associated with that batch set is initially triggered on the host core), which is why some of the smaller kernels overlap in time. As such, the interference of kernels being issued by multiple host cores to the same GPGPUs are accounted for in the timeline figure shown.

The second thing to notice is that a significant portion of each host core's time is spend performing communication and any related batching resulting from those messages arriving. Basically, each of the "short" blocks is communication (short in terms of time). During the time periods dominated by these short to execute entry methods associated with data communication, their is a significant amount of idle time on each host core as local execution waits for data to arrive. Also recall that an $O(N^2)$ algorithm is being used for the simple MD code. That not only increases the amount of work (compute objects) but also increases the amount of communication occurring. As this communication, and related idle time occurs, the force computations are not being executed and the overall performance (efficiency) of the application decreases as a result. This is a common effect seen in parallel applications as result of strong scaling. However, the batching process used by the runtime system may be emphasizing the effect since the kernel execution cannot begin until a given number of messages arrive. However, the same is true of hand-coded applications as well, with the difference being that the application code does the batching explicitly, rather than the runtime system implicitly.

Finally, we repeated the six node execution with and without GPGPU sharing. Figure 5.28 shows the results of both runs. The top graph in figure 5.28 shows the performance of both executions, with and without GPGPU sharing, over the course of their executions. Initially, the performance of the "Not Sharing" run is quite bad. This is because the objects within the application are distributed equally to each host cores. As such, the host cores that do not have accelerators are initially a bottleneck for performance. After a couple across host core load balancing steps, however, the performance is greatly increased, as the load balancer migrates the objects away from the *weaker pairings* (i.e. host cores without access to accelerator devices). The "Shared" plot initially does fairly well since all of the host cores have access to GPGPUs.

The center and bottom graphs of figure 5.28 show the object distributions across the pairings for both





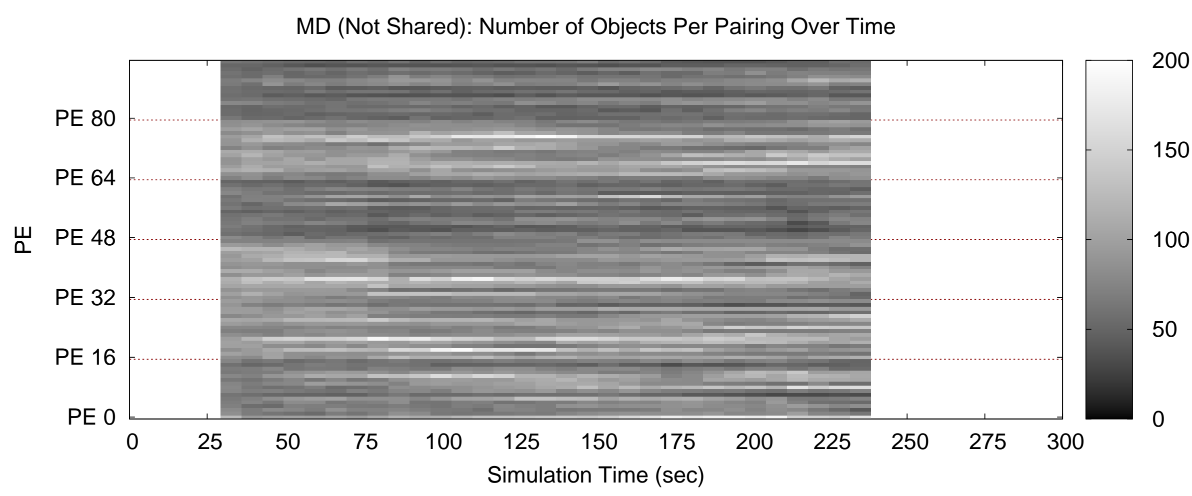


Figure 5.28: The object distribution and performance of the 6 node execution, with and without GPGPU sharing.

the "Not Shared" and "Shared" runs in the top graph, respectively (only pair compute objects are shown). For both graphs, all 96 pairings are shown, but only the first pairing from each node is labeled. There is a single row for each pairing. The gray value indicates the number of objects located on each pairing per the scale on the right side of the graph. Note that the gray value scales for the two object distribution graphs are not the same (i.e. the bottom graph has a smaller range of values and thus noise is emphasized in the gray values more so than it is in the middle graph).

In the "Not Shared" case (center graph in figure 5.28), only the first 6 host cores are paired with one of the 6 GPGPUs (giving them each exclusive access to that GPGPU. The remaining 10 host cores per node do not have access to GPGPUs (though, we still refer to them as *pairings*). This misbalance is clearly evident in the "Not Shared" execution's object distribution (middle graph of figure 5.28). The AccelEvenLB across host core load balancer quickly moves migrates objects away from the host cores without GPGPUs, indicated by the six bands of lighter gray values (each containing six pairings). Note that the initial object balance is not shown since the execution's output only reports object distributions after a load balancing step (that is why the middle graph seems to start just after 50 seconds). Note that the 60 host cores without access to GPGPUs combined only contribute approximately 3% of the total peak flop rate provided by all six nodes, so very AccelEvenLB is mapping very few chare objects to them. While it is hard to tell from the figure, there are approximately 10 to 13 pair compute objects mapped to each host core without GPGPU access at any given moment.

In the "Shared" case (bottom graph in figure 5.28), all of the host cores have access to the GPGPUs. Some of the GPGPUs (on a per node basis) are shared by two host cores, while others are shared by three host cores, a nutural result of 16 (number of host cores) not being evenly divisible by 6 (number of GPGPUs). This makes the job of the across host core load balancer, AccelEvenLB, more difficult in that it introduces *side effects* to object movement. In particular, we mean that moving chare objects from pairing i to pairing j not only affects the performance of pairing j, but also affects the performance of any pairing that is also making use of the same physical GPGPU. This may account for some of the noise seen in the bottom graph of figure 5.28.

5.5 Adapting to Interference

In addition to load balancing a given workload from an initially imbalanced state, there may be situations in which an application may need to react to external influences. For example, consider a situation where an application, A, is executing on a shared workstation and another application, B, begins executing. If

some subset of the hardware resources that where originally being exclusively used by A suddenly become shared between both A and B, A is likely to see a performance degradation. Consider the case where one or more accelerator devices become shared, but the host cores remain exclusive. In such a case, the performance of the shared accelerator devices will decrease from A's point-of-view, and thus, the ratio of measured flops between any exclusive resources (host or accelerator) will increase relative to the shared accelerator resources. This degradation occurs for two reasons. First, the shared resource must divide its time between multiple applications, which will likely result in any tasks associated with that shared device to increase in latency. Second, if a proper subset of the hardware resources transition from exclusive to shared, an imbalance will form, causing the application to be limited by the slowest component. In such situations, it may be beneficial to shift some of the workload associated with the shared resources to the exclusive resources in order to rebalance the workload such that all resources complete their portion of the workload in the same amount of time.

To show how our approach handles such cases, two situations are simulated using the MD application. The first situation will be a case of *overlapping* application instances, where each application instance is completely sharing the accelerator devices available to them. In the overlapping situation, the relative performance of the accelerator devices compared to the host cores is shifted during execution, which the application instances must adapt to in order to minimize the impact on their performance. The second situation will be a case of *interference*, where one of the application instances will make use of only a subset of the other application instance's accelerator devices, causing an imbalance between the accelerator devices themselves, in addition to the host cores.

5.5.1 Overlapping Execution

The first example of an external influence considered is the case of *overlapping* interference. By "overlapping" we simply mean that the set of accelerator devices available to all of the applications sharing them is completely overlapping.

In the case of GPGPUs in particular, multiple applications can make use of the accelerator at the same time. As the applications are sharing the accelerator, the accelerator's performance may decrease from the point-of-view of either application, since only a fraction of the accelerator's time can be given to either application. As such, it may be advantageous for the applications to react accordingly and migrate some of the workload assigned to the shared accelerator devices to the host cores to mitigate the perceived decrease in performance. We use the phrase "perceived decrease in performance" since, from the point-of-view of the accelerator itself, the accelerator may actually be doing more work per unit time, even though each

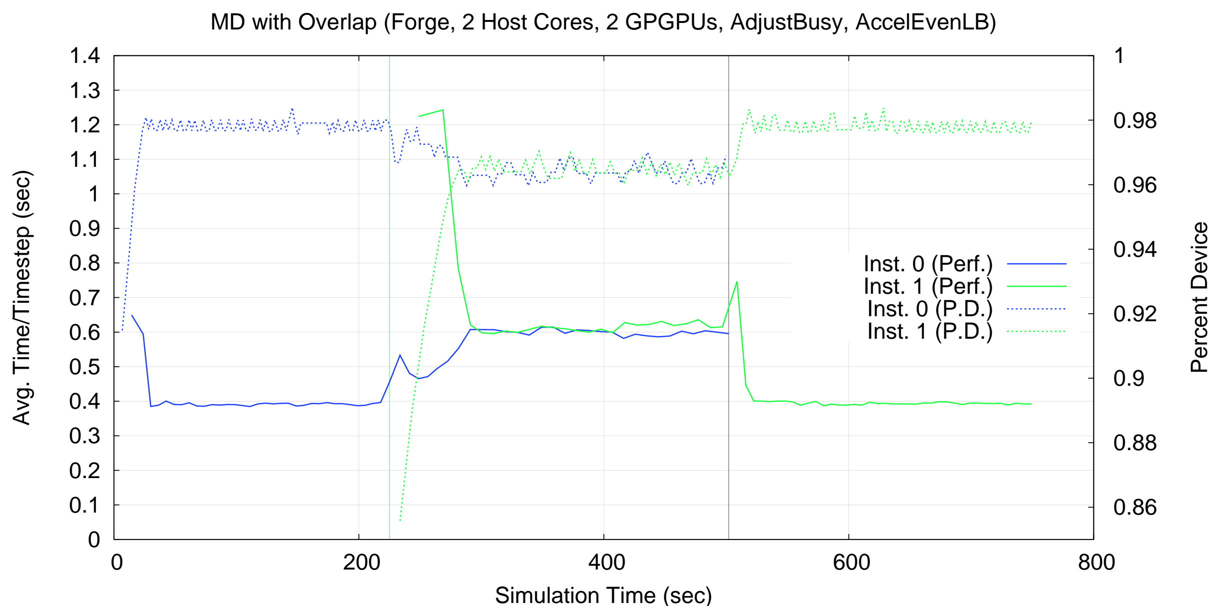


Figure 5.29: Two instances of the MD application with the same configuration overlapping in time.

application individually is submitting a lower portion of it's overall workload.

To demonstrate that our approach can assist applications in this regard, multiple instances of the molecular dynamics (MD) application will be executed on the same node of Forge such that their executions overlap in time. Both instances of the application are using the exact same configuration (number of host cores, number of accelerator cores, number of timesteps, number of particle counts, load balancing strategies, and so on). However, each instance makes use of a mutually exclusive set of physical host cores, while sharing the same physical accelerator resources. For these executions, 2 host cores and 2 GPGPUs were used for each application instance.

Figure 5.29 shows the result of the overlap experiment. First, notice that the executions of the application instances do not completely overlap in time. This was done so that the reaction of each application instance could be seen, showing that the application and runtime system are in fact adapting to the environment in which they are executing. The first application instance, "Inst. 0" (blue), begins execution at time 0. The second application instance, "Inst. 1" (green), is started sometime after the first. As such, "Inst. 0" begins execution with exclusive access to the accelerator devices and must later adapt to sharing them with "Inst. 1." "Inst. 1" begins execution by sharing the accelerator devices with "Inst. 0," but later obtains exclusive access when "Inst. 0" completes its execution.

The first two plots, "Inst. 0 (Perf.)" and "Inst. 1 (Perf.)," show the performance of each instance of the MD application. The second instance (*1*) was started approximately half way through the execution of the

first instance (*0*). The green vertical line, at approximately 225 seconds, shows the time at which the second instance began executing relative to the execution time of the first instance (this is estimated based on the output of the first instance, since there are no shared clocks between the executions). It should be noted that the performance plot of the second instance, "Inst. 1 (Perf.)," begins slightly to the right of the green vertical line. This delay occurs for two reasons. First, there is some application initialization which delays the beginning of the first timestep. Second, the MD code reports performance after each set of 16 timesteps which causes the first data point to appear only after the first 16 timesteps have completed, making the delay at the beginning of the application's execution seem longer than it is in reality. In addition to the performance levels of each instance of the application, the percent device values for the first host core and GPGPU pairing are also shown for each application instance, "Inst. 0 (P.D.)" and "Inst. 1 (P.D.)." In the same vein as the performance plots, the data points that make up the percent device plots only occur when a change in the percent device value occurs. Each instance of the application begins with all percent device values (i.e. for all host core and GPGPU pairings) set to a value of 100%. A second vertical line, blue in color and at approximately 502 seconds, marks the time at which the first instance of the application (*0*) finishes executing. Thus, the two vertical lines mark the region of time where the two instances of the application are overlapping in time and sharing the GPGPUs.

The first thing worth noting in figure 5.29 is the performance of each application during the time period of overlapping execution once both instances have reached a fairly stable level of performance (i.e. from approximately 300 seconds to approximately 500 seconds). In particular, notice that the level of performance for both instances is approximately 0.6 seconds per timestep. Compare that to the performance of each instance of the application execution in isolation, which is approximately 0.4 seconds per timestep. Since neither application is making full use of the physical accelerator devices (i.e. is not actively executing work 100% of the time on the accelerator), the net effect is that the performance of the application instances during the overlapping time period does not drop to half (i.e. 0.8 seconds per timestep) the performance level measured when either is executing in isolation. Instead, each instance of the application is able to share the accelerator devices, filling in what would have otherwise been idle time in the other instances execution. Also notice that the percent device plots of each instance of the application decrease during the period of time when both instances are active. From the point-of-view of either application instance, the other instance is seen as interference causing a perceived decrease in the performance of the accelerator devices relative to the host cores. The accelerator load balancing mechanism reacts to this perceived decrease by shifting a portion of the workload from the accelerator devices to the host cores. However, it should be noted that amount by which the percent device value changed is fairly minor, moving from just under 98%

to just over 96% (i.e doubling the workload on the host cores associated with AEMs), so the accelerator devices are still handling the majority of the application's workload associated with AEMs. These shifts are consistent for both applications (i.e. same levels during the overlapped time period and the isolated periods, respectively).

Another (minor) point to take note of is the initial behavior of the AdjustBusy accelerator load balancing strategy. Specifically, the AdjustBusy strategy begins with a percent device value of 100%. If there is any idle time on the host, the AdjustBusy strategy will migrate work from the device to the host. Notice that the percent device value of the first host core and GPGPU pairing in the first application instance (*0* in figure 5.29) drops to approximately 91.5%. However, for the second application instance, there is "interference" from the very beginning of the instance's execution. As such, the AdjustBusy strategy initially drops the percent device value of the first host core and GPGPU pairing to an even lower percentage (approximately 85.6%), indicating that it is in fact detecting and reacting to the relative performance between the host core and the accelerator device for the first pairing.

5.5.2 Partial Interference

In the case of figure 5.29, the two instances of the MD application were executed using exactly the same physical GPGPUs. As such, all of the GPGPUs were experiencing an equal amount of interference by the executed of the second instance of the application. However, what if only a subset of the GPGPUs being used by an application were experiencing interference? In this section, we explore the *interference* case, when only a subset of the accelerator devices are shared with other applications.

Figure 5.30 shows three instances of the same MD application. However, in figure 5.30, the three instances of the application are being executed on different hardware configurations. In the case of instance A, 12 host cores and 6 GPGPUs are being used to execute the MD code for 6144 timesteps. Instances B and C are executed using 2 host cores and 2 GPGPUs, and only execute 768 timesteps (each).

Initially, in figure 5.30, instance A of the MD application begins execution. At approximately 221 seconds and 229 seconds, instances B and C, respectively, begin execution. As was the case in figure 5.29, each of the instances of the MD application are executing using a mutually exclusive set of host cores. In this case, instance A is using physical cores 0, 2, 8, 9, 10, 11, 4, 6, 12, 13, 14, and 15 (which are equivalent to virtual cores 0 through 11). The irregular mapping is caused by the affinity of the various GPGPUs to the various host cores, as defined by the Forge architecture. Instance B is using physical cores 1 and 3. Instance C is using physical cores 5 and 7. However, the applications are share the physical GPGPU resources. In particular, instance A is using all 6 GPGPUs (0-5), which are paired with the host cores in a round-robin

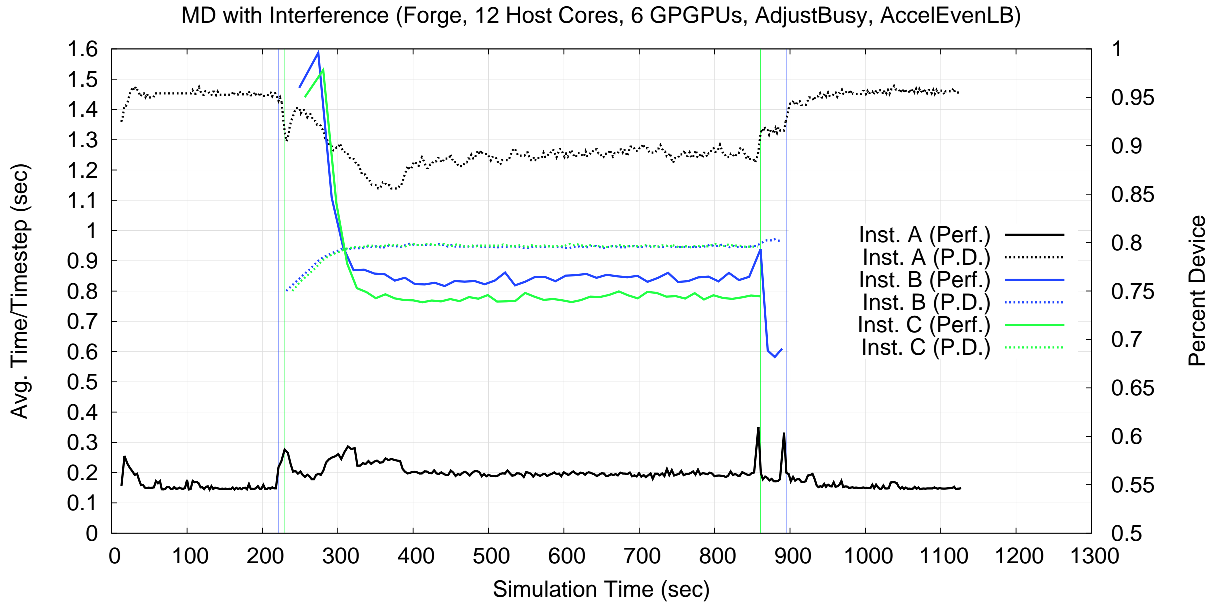


Figure 5.30: An instance of the MD application executing with interference.

manner (i.e. host core 0 uses GPGPU 0, host core 2 uses GPGPU 1, 8 uses 2, 9 uses 3, and so on in round-robin order). Instance B is using GPGPUs 0 and 1. Instance C is using GPGPUs 0 and 1. Since all of the instances share GPGPUs 0 and 1, once the two extra instances of the MD application, B and C, these two GPGPUs will become overloaded compared to the other four GPGPUs. Otherwise, the plots within figure 5.30 have been displayed in a manner that is very similar to figure 5.29 above. The vertical lines indicate the starting and ending times of application instances A and B. The percent device value of the first host core and accelerator device pairing over time is shown for each configuration. The percent device values for the second pairing in both B and C are nearly identical to the first pairing's percent device plots. The other percent device value plots for instance A are displayed in figure 5.31, as described below.

Notice that the performance of instance A initially suffers as instances B and C begin their execution. However, between 250 and 400 seconds, the all of the applications once again reach a stable point as both the across host core load balancing and the accelerator load balancing strategies correct for the interference. Note that the performance of instance A moves from approximately 0.15 seconds per timestep before the interference to approximately 0.2 seconds per timestep, which is a decrease in performance of 33.3% relative to the pre-interference performance level. Note that instance A is configuration E from figures 5.15 and 5.16 in section 5.4.2. Instances B and C are configuration B from the same figures, which when executed in isolation, achieved a performance level of 0.375 seconds per timestep. However, when the executing in an environment where the accelerator devices are being shared between three separate application instances,

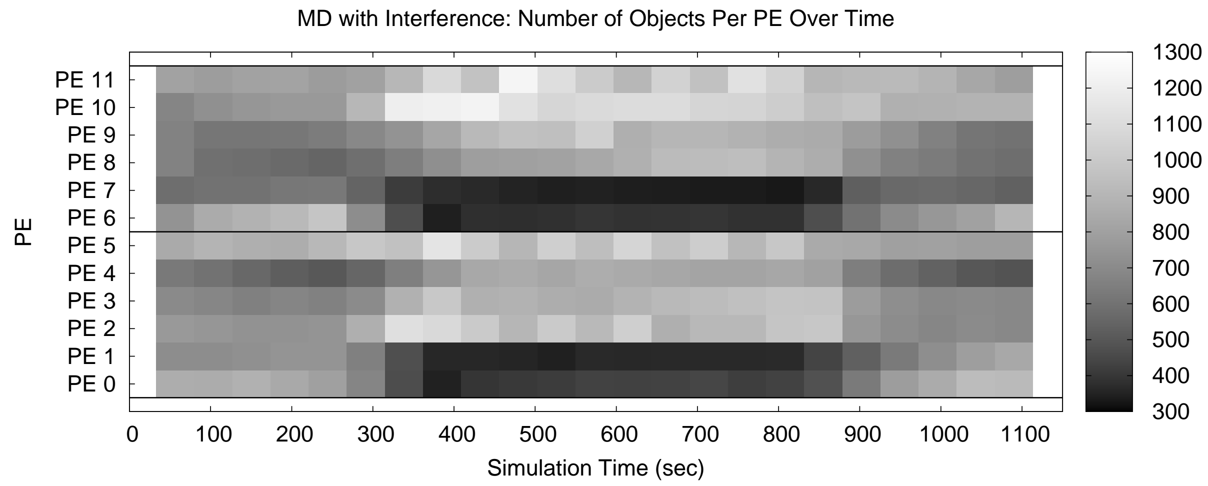
the performance for instances B and C is closer to 0.8 seconds per step (roughly speaking), which is a slowdown of approximately 113% relative to executing in isolation.

The two graphs in figure 5.31 illustrate how the load balancing mechanisms reacted to the induced interference on GPGPUs 0 and 1 within instance A of the application. Figure 5.31(a) shows the number of chare objects that are being mapped to the various processors over the course of instance A's execution. The horizontal axis represents time, while the vertical axis lists each of the host cores (PEs) in use by the application. There are 12 rows made up of varying gray values, one per PE, where the gray value reflects the number of chare objects that are located on that PE at the corresponding time in the simulation. The bar to the right of figure 5.31(a) maps the color to the number of chare objects. A darker gray value represents fewer objects, while a brighter gray value represents more objects. Across host core load balancing is triggered every 256 timesteps. Since there are 12 host cores and 6 GPGPUs paired with those host cores in a round-robin manner, each GPGPU is shared by two host cores within instance A.

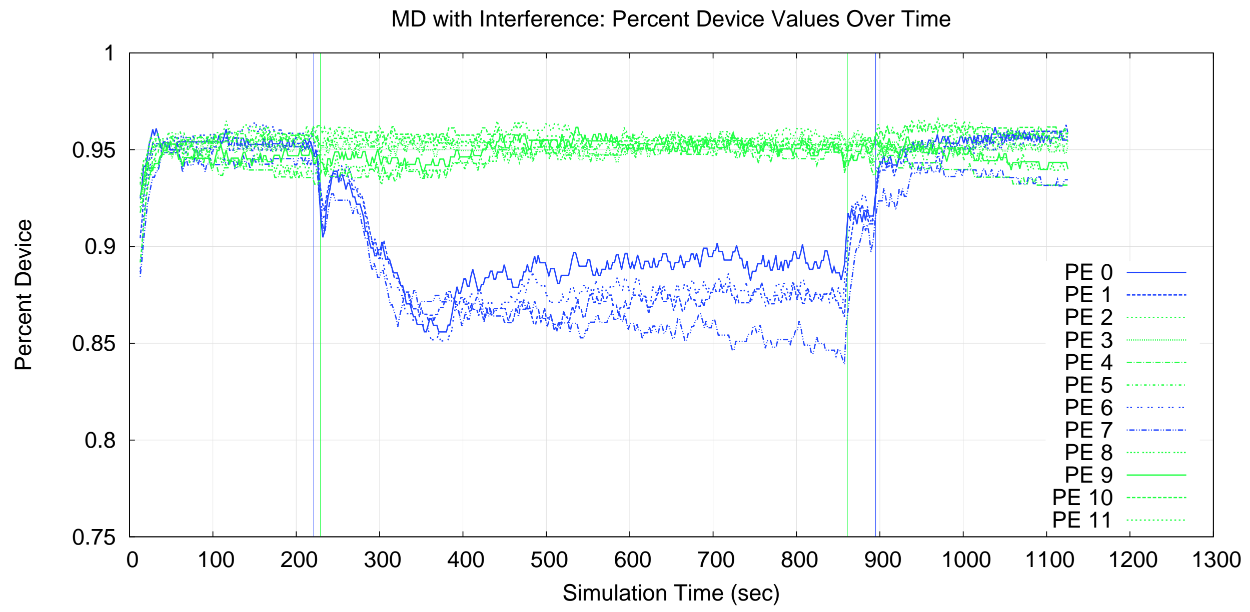
At this high level, the overloading of GPGPUs 0 and 1 is very apparent in figure 5.31(a). For host cores 0, 1, 6, and 7 (using virtual numbering, not physical numbering), which are the host cores using the shared GPGPUs 0 and 1, a significant fraction of the objects originally mapped to these host cores are moved to other host cores during the interference period (these host cores become a darker gray value while the other cores become a lighter gray). Since there are twice as many exclusive accelerator devices compared to the number of shared accelerator devices, the number of chare objects on these host cores should only increase by half the decrease seen host cores associated with shared accelerator devices. Once instances A and B have completed, the number of objects across all of the host cores once again moves to levels that are similar to the levels seen prior to the interference period.

Figure 5.31(b) shows the percent device values for application instance A over time for each host core and accelerator device pairing. The percent device plots for host cores associated with shared accelerator devices are colored blue, while the plots for host cores associated with exclusive accelerator devices are colored green. As figure 5.31(b) shows, the percent device values associated with the shared accelerator devices are decreased during the time period that the interference is occurring. Once the interference period has ended, the percent device values associated with the shared accelerator devices return to levels seen prior to the interference occurring. Further, during the same time period, the percent device values associated with the non-overloaded GPGPUs remain fairly constant, as these exclusive accelerator devices are largely unaffected by the interference caused by the other two instances of the MD application.

When viewed together, the graphs in figure 5.31 illustrate the overall effect of the interference as seen from the point-of-view of instance A. In summary, during the interference period, many chare objects are



(a)



(b)

Figure 5.31: An instance of the MD application executing with interference.

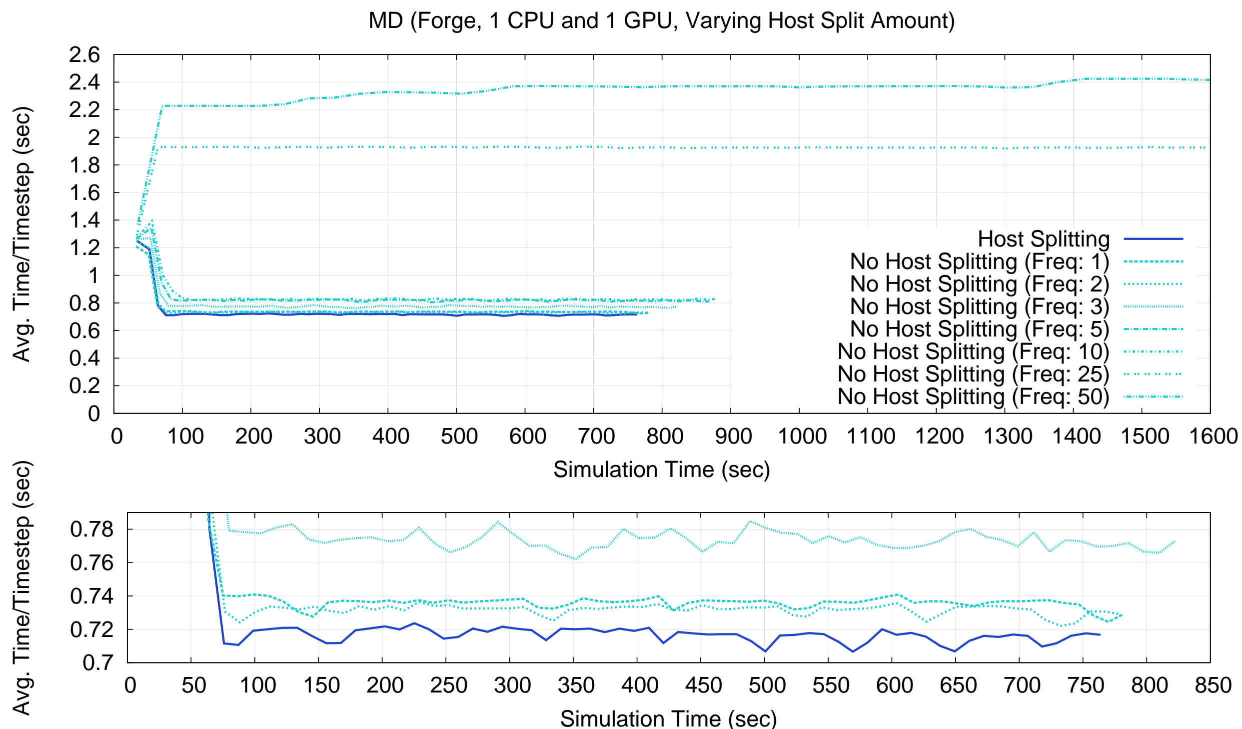


Figure 5.32: Performance of the MD code using various progress call frequencies, with and without splitting accelerated entry methods on the host core.

migrated away from the host cores associated with the shared accelerator devices. Of the remaining objects, an increased percentage of those objects are having their accelerated entry methods invocations directed to the host cores rather than the shared accelerator devices.

5.6 Effects of Host Splitting

Figure 5.32 shows the effects of using various progress call frequencies to the GPU Manager from within the Charm++ scheduler, along with and without splitting the accelerated entry methods assigned to the host core. In each run, a single host core and a single GPGPU is used to execute the MD application. A single host core and accelerator device pairing was used to minimize the amount of noise within the application's execution. There are two main types of plots shown in figure 5.32, "Host Splitting" and "No Host Splitting." In the case of the "No Host Splitting" plots, any splittable AEMs executing on the host core are executed using a single split. In the case of the "Host Splitting" plot, any splittable AEMs executed on the host core were executed using two splits. A fixed number of splits was used for this experiment, but in the future, we may make this a variable amount that the runtime system adjust dynamically. Additionally, for each plot, a frequency ("Freq.") value is given (actually, the value actually relates to the period rather than the frequency,

but "freq" is used to identify the configuration parameter within the code, so we retain the relationship to the name here). The frequency value indicates the frequency with which the Charm++ runtime system scheduler calls the GPU Manager's progress function. This progress function essentially checks the state of the accelerator device, and has the host core perform any actions required by the accelerator device (issuing pending AEM batches (kernels), triggering data transfers, triggering callback functions, and so on). By calling the progress function with a greater frequency (lower value), the host core will seem more responsive to the needs of the accelerator device. Specifically, the frequency number indicates the number of consecutive entry methods (of any type) that the scheduler will be execute between calls to the progress function (i.e. the higher the number, the lower the frequency). Note that, the progress function will also be called if the host core goes idle.

As can be seen from figure 5.32, as the frequency of calls to the progress call decreases (a higher number), so does the performance of the application. The highest frequency at which the Charm++ scheduler can call the progress function is between every entry method (a value of 1). However, recall that entry method are non-interruptible. That is, once the runtime system has passed control of the host core to application code within an entry method, control is not returned to the runtime system, and thus the scheduler, until the entry method has completed (or yielded in the case of a threaded entry method). As such, the actual time between calls to the progress function is dependent on the application code (amount of time spent in each entry method). By splitting the AEM entry methods assigned to the host core and executing those splits one-by-one, the runtime system is effectively decreasing the average amount of time per task, giving the runtime system a chance to make progress calls between splits. As shown in figure 5.32, the MD application performs best when the AEMs assigned to the host core are split.

Beyond decreasing the granularity of tasks in the case of mapping accelerated entry methods to GPGPUs, this result shows an additional use for splittable AEMs. The original intent of AEMs was to decouple the number of messages from the number of entry methods (tasks) executed by the application. By creating many smaller tasks out of what would have otherwise been a single larger tasks, the runtime system is able to increase the amount of parallelism within the application while, at the same time, decreasing the granularity of individual tasks. This could also be achieved by creating smaller chares (and thus more shorter-executing entry methods associated with those chares). However, in doing so, one would also increase the number of messages while also decreasing the amount of data per message (i.e. decreasing the effectiveness of the interconnect and increasing scheduler overhead). Instead, splittable AEMs allow for both larger messages and smaller tasks. This second use of AEMs allows the runtime system decrease the average time per entry method by splitting AEMs on the host and then executing them sequentially. The cost is that the overall

amount of time to do the same amount of work that was encapsulated within the original AEM (before splitting was applied) is increased because of the added overhead of the splits. However, a potential gain (i.e. allowing the host core to be more reactive to the needs of an accelerator device) is demonstrated in figure 5.32.

Something that may seem a bit odd about figure 5.32 is that when host splitting is not used and the frequency of progress calls is low (values 25 and 50), the performance of the application actually decreases. Since the scheduler is calling the progress function rather infrequently, there may be some delay between the time an AEM batch (kernel) actually finishing on the accelerator device and the time that the host core actually notices the AEM batch has completed (i.e. the start of the associated callback function). Since the accelerator busy time is actually measured on the host core by the accelerator manager, which makes use of the GPU Manager but is not apart of the GPU Manager, any delay will be measured by the accelerator manager as an increase in the busy time of the accelerator device. This artificial increase in the accelerator's busy time, from the point-of-view of the accelerator device, causes the AdjustBusy strategy to shift some work from the accelerator device to the host core in an attempt to even out their respective busy times. The fundamental issue is that any runtime system that must periodically poll for the completion of one or more tasks on an accelerator device and that cannot collect timing data directly on the accelerator device itself will have this issue. Specifically, the effectiveness of any dynamic, measurement-based load balancing strategy will rely on the accuracy of the timing measurements, which in turn, will rely on the rate at which the runtime system polls the accelerator device. Notice that the artificial increase relies on two causes that must both be present, not being able to measure timing on the device itself (i.e. avoiding the inclusion of the artificial delays) *and* the timing data collected on the host relies on polling to detect the events to be timed (i.e. only takes a time sample when an event is noticed rather than when it actually occurs). While this will be corrected for in our approach, we point out this issue since it is something that other implementations may encounter.

5.7 Revisiting the Cell Processor

In this section, we will revisit the Cell processor, making use of the Cell nodes within the heterogeneous cluster that were used for the static load balancing experiments presented in chapter 4. In particular, we will examine the performance of the MD application executing on the Cell-based nodes (i.e. QS20s and PS3s) within the heterogeneous cluster using both the Sampling (centralized) and AdjustBusy accelerator load balancing strategies.

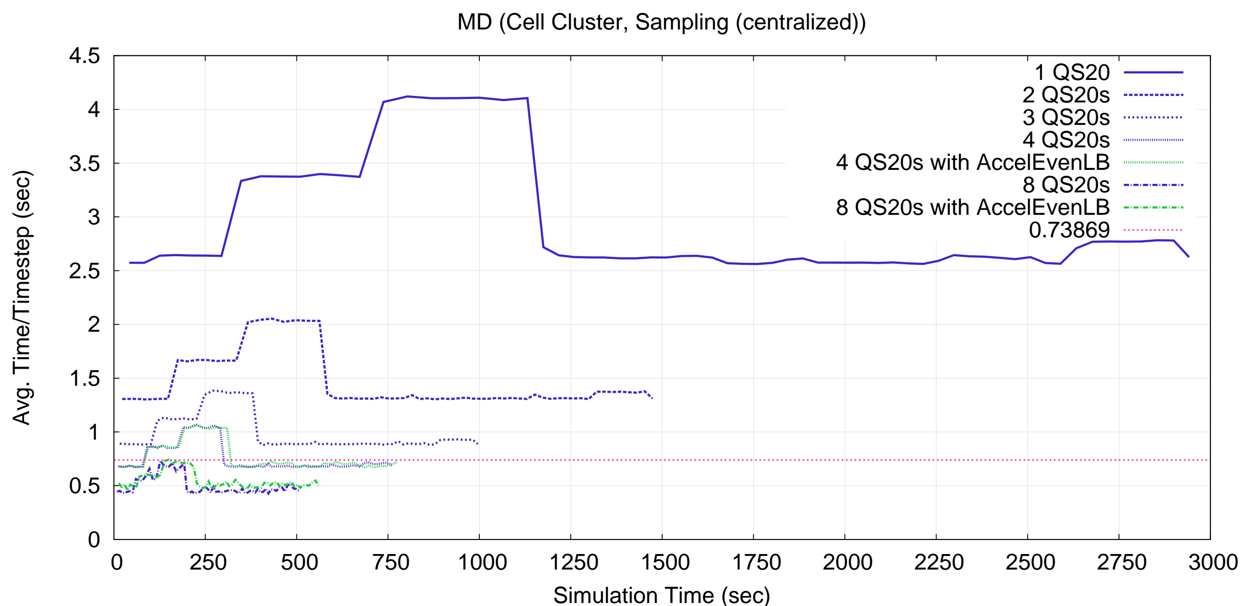


Figure 5.33: The MD application executing on QS20 blades in the cell cluster, using the Sampling (centralized) accelerator load balancing strategy.

5.7.1 Sampling *(centralized)*

Figure 5.33 shows the performance of the MD application executing on the QS20 blades (one Cell processor each) using the Sampling (centralized) accelerator load balancing strategy. For the 4 QS20 blades configuration, the application was executed with and without the AccelEvenLB host core load balancer. As the figure shows, the addition of the AccelEvenLB load balancer does not improve performance, which is expected since each of the Cell processors (and the nodes they are located in) are identical to one another and the default mapping of chare objects is to equally distribute the objects across the Cells. Each of the plots in figure 5.33 have the hump that was seen when GPGPUs were being used to execute the MD application. However, note that the hump has two levels in it. This is caused by our implementation, which always starts by sampling percent device values of 100%, 97.5%, and 95% at the beginning of an application's execution. Since the percent device value that gives the greatest performance is 100% (will be discussed further below), sampling at 97.5% and 95% results in decreased performance. Plot 5.34 illustrates this point for the 1 QS20 blade Cell configuration. When viewing figure 5.34, recall that the percent device value is set to 100% at the beginning of an application's execution and the runtime system only reports the percent device value when the value changes. As such, the initial portion of the percent device plot (i.e. before 294 seconds) is not shown, but the percent device value is in fact set to 100%. For completeness, the MD application was also executed on the QS20s using both Cell processors (i.e. 4 nodes, 2 processors per node). In the previous

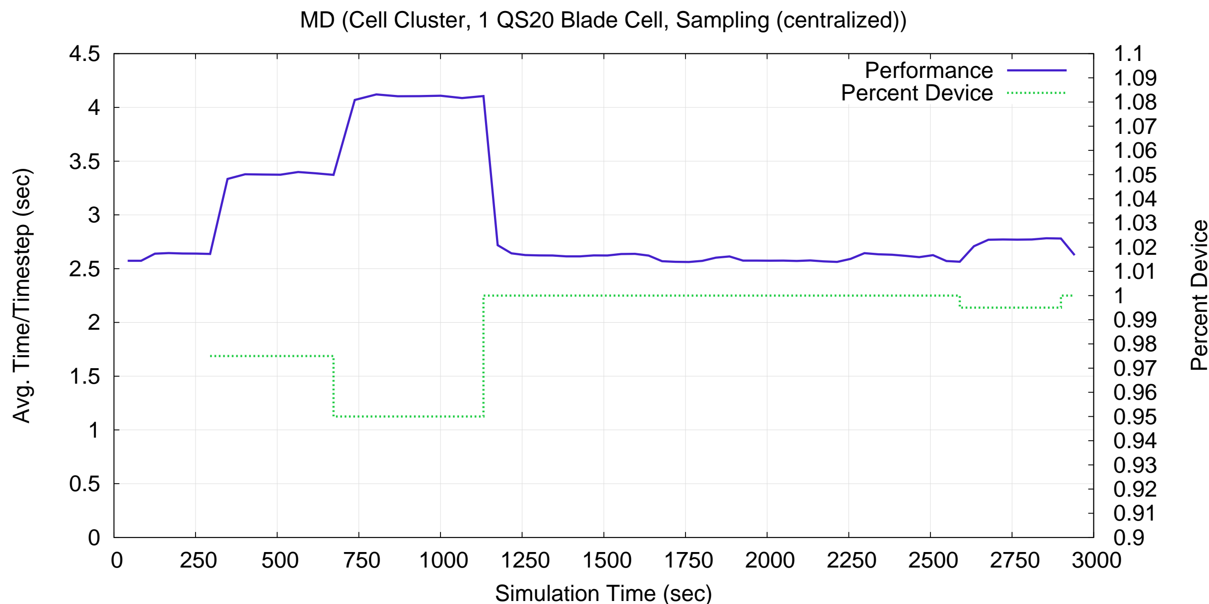


Figure 5.34: The MD application executing on 1 QS20 blade in the cell cluster, using the Sampling (centralized) accelerator load balancing strategy.

static load balancing work performed on this cluster [64], it was shown that the message send times on the QS20 Cell blades were fairly long relative to the other nodes when only a single Cell processors was being used per QS20 blade. When using two Cells per QS20 blade, these communication overheads hinder the MD application scaling from 4 QS20 Cell processors (1 per node) to 8 QS20 Cell processors (2 per node).

As a reference, figure 5.33 also includes a red horizontal line that represents the time per timestep achieved with the measured flop rate of 178.22 Gflop/s (the 4 QS20 Cell blades case presented in table 4.2 using static load balancing). Since the best percent device value happens to be 100%, all this comparison really serves in showing is that the added overhead of using dynamic load balancing is not that costly (except for the *hump*). There will be a periodic decrease in performance as the dynamic load balancer periodically samples the performance below 100%, but the performance is similar otherwise. The slight performance increase for the 4 QS20 Cell case is actually a result of a slight configuration difference between the previous static load balancing results presented in table 4.2 and the dynamic load balancing results presented in figure 5.33. In order to periodically output coherent timing information every X timesteps, a *barrier* operation is performed every X timesteps that would not otherwise be required. A larger value of X was used for the dynamic load balancing experiments, which largely accounts for the small performance increase.

An important, but non-obvious, point to be made here is that the MD code being used on the Cell cluster is the same code being used on the GPGPUs cluster except for a single difference. In particular, the pair compute object's force calculation accelerated entry method is not splittable when executing on

the Cell cluster, but is splittable when execution on the GPGPU cluster. To be clear and direct, we do not mean *splittable and executed with a single split*, rather we mean *not using the "splittable" keyword*. In the case of the GPGPU cluster, the pair compute objects' force calculation code was made splittable so that the runtime system had the option of creating tasks with a finer granularity (i.e. splitting the AEMs). However, there is a cost related to splitting the pair compute's force calculation AEMs. Specifically, the equal-and-opposite forces optimization cannot be applied to the splittable AEMs (refer to section 6.2.2), which increases the number of flops required per timestep. In the case of GPGPUs, this tradeoff is fine since the increased flop rate provided by the splitting results in better performance (time per timestep, and thus time to solution). However, in the case of the Cell processor, the equal-and-opposite optimization can be applied, which results in better performance overall. This difference illustrates a weakness in our unified programming model. To be clear, it is the case that both versions of the code (with and without the equal-and-opposite force optimization applied) are portable between both platforms. Both the Cell cluster and the GPGPU cluster can execute either version of the MD code (with and without splits). In this sense, our unified programming model represents a minimal set of abstractions/features required to provide portability. However, our approach is minimal in that it only supports this portability in a *mechanical* sense (i.e. allowing it to occur). Our approach does not optimize an AEM's function body in a way that specific to each type of processing element as the AEM is compiled for each type of processing element (host core, SPE, or GPGPU). Given a sufficiently advanced compiler, the data dependencies of the AEMs function body could be analyzed for both data dependencies and loop-level parallelism (e.g. data parallel iterations) such that the programmer could write a straight forward inner/outer loop combination used for particle-to-particle interactions with the equal-and-opposite optimization. Knowing that the AEM's function body is going to be compiled for each type of core (host and accelerator) and knowing the hardware characteristics of each core type, the compiler could make different tradeoffs for each version of the AEM code. In the case of a host core or a SPE core, the compiler could keep the general nature of the code the same. However, in the case of of a GPGPU, the compiler could duplicate work as a tradeoff for splitting the loop iterations (i.e. create a data parallel loop by duplicating calculations in order to remove race conditions), knowing that the duplication of work will likely lead to better performance (i.e. not better performance in terms of achieved flops per second, but rather better performance in terms of time to solution). The creation of such a compiler, and the associated data dependency analysis and loop-level transformations, is left as future work from the point-of-view of this work.

Figure 5.35 shows the performance of the MD application executing on the Cell cluster using 4 QS20 blade Cells and 2 PS3 Cells and making use of the Sampling (centralized) accelerator load balancing strategy.

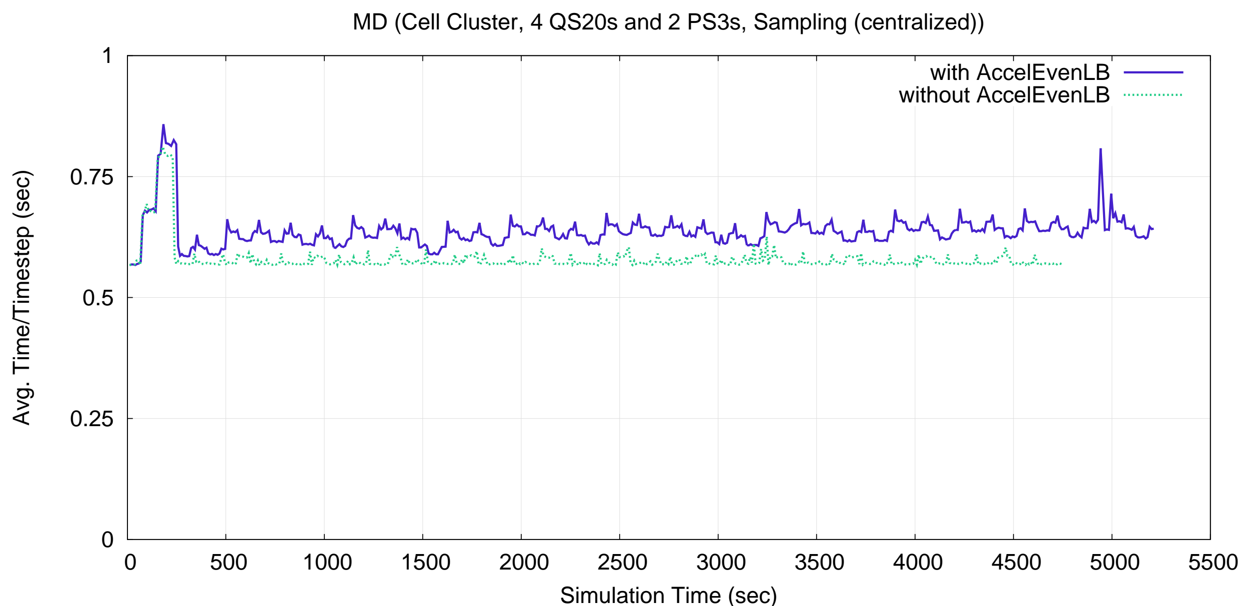


Figure 5.35: The MD application executing on 4 QS20 blades and 2 PS3s in the cell cluster, using the Sampling (centralized) accelerator load balancing strategy.

Only 2 of the 4 PS3 nodes were used because, at the time these executions were performed, 2 of the PS3 nodes were not operational. As the figure shows, the application is able to make use of both types of nodes simultaneously. From the figure, we can see that the use of AccelEvenLB is actually decreasing the performance of the application. Figure 5.36 shows the number of pair compute objects assigned to each pairing (PE) by the AccelEvenLB balancer over time.

Figure 5.36 shows that the AccelEvenLB balancer is mistakenly placing more objects on the PS3 Cells (PEs 4 and 5) compared to the QS20 blades Cells (PEs 0 through 3). The reason for this is that the AccelEvenLB does not directly read the performance (absolute busy time, ratio of busy time to idle time, or

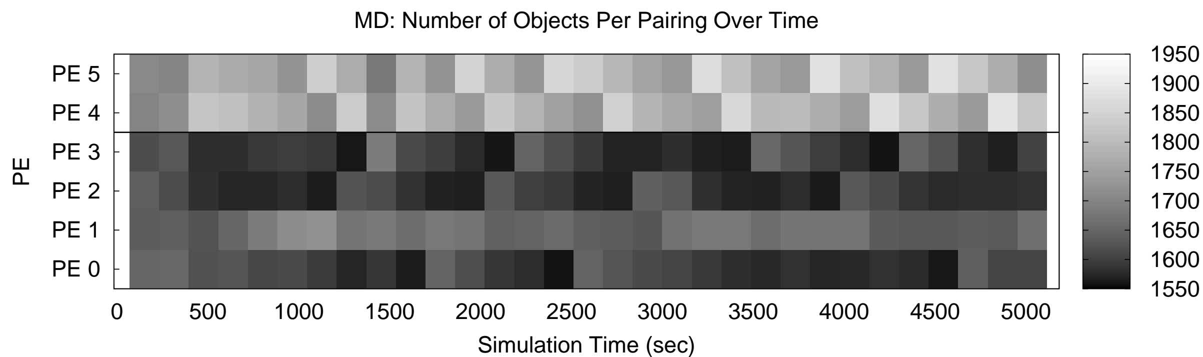


Figure 5.36: The balance of objects as set by AccelEvenLB for the MD application executing on 4 QS20 blades and 2 PS3s in the cell cluster, using Sampling (centralized).

any other direct measurements) of the accelerator devices. Instead, AccelEvenLB was originally designed to be oblivious to the presence of an accelerator. From the point-of-view of AccelEvenLB, a host core should only *seem* more powerful if it has an accelerator attached since it will get through its portion of the workload faster, relative to a host core without an attached accelerator device. When executing on the GPGPU cluster (Forge), this was in fact the case. However, it was only the case because *at least some* accelerated entry methods were being executed on the host cores. When executing on the heterogeneous cluster using the Cell processors, this is no longer true. For the Cells, the ideal balance of the workload is to push any and all AEMs to the accelerator device (the SPEs), leaving the host core (PPE) as free as possible. In other words, the PS3s Cells are taking less time to dispatch their portion of the AEMs to their SPEs compared to the QS20 blade Cells and only the host core (PPE) time is being taken into account since AccelEvenLB is oblivious to accelerator measurements. As a result, AccelEvenLB is being tricked into assigning more chare objects to them.

This identifies another fundamental difference between the Cell processor and GPGPUs. There is a tendency that *one unit of work* for an SPE is smaller in size than one unit of work for a GPGPU. By "unit of work" we mean some amount of calculation such that an interaction with other components of the system, such as other cores, system memory, or the network, is not required for continuation. Put another way, SPEs, which stream data through their local stores, tend to require interactions with other components of the system on a more regular basis when compared to GPGPUs, which tend to operate on larger kernels (relatively speaking). Note that this is not true for every approach, especially approaches that limit the computation to operations on very regular data structures, such as matrices, and thus allow the SPEs to be more autonomous. However, it tends to be true in our approach since the tasks being streamed through the SPEs (i.e. AEM invocations) are triggered by messages, which are processed on the host core and often triggered by messages originating from the network. As such, the PPE needs to remain highly active in order to constantly *feed* work to the SPEs, leading to a percent device value of 100%.

On the other hand, the larger granularity of the kernels executing on the GPGPUs allows the host core to perform more work unrelated to the accelerator device for longer periods of time. As such, the balance point was almost always less than 100%, meaning that there are AEMs executing on both the host core and the GPGPU, allowing AccelEvenLB to pick up on differences between the accelerator devices in an indirect manner (as we originally intended). As an example, recall the percent device values from figure 5.16 for configuration F. The percent device values for host cores associated with physical GPGPUs shared by two host cores were greater than the percent device values for host cores associated with physical GPGPUs shared by three host cores. The increased ratio of AEMs executing on host cores versus AEMs executing

on the accelerator device leads to an increase in the average amount of time to execute each AEM. The increase in this average makes the host core seem less powerful from the point-of-view of AccelEvenLB, which balances objects across host cores based on this average time to execute (i.e. in a way that is oblivious to accelerators, but indirectly affected by their presence). However, this is not possible on the Cell processor since 100% of the AEMs are being mapped to the SPEs (i.e. the ideal balance point is to keep the PPE unburdened and thus reactive to the SPE's needs). Instead, since the PPEs are the same regardless of the node type and the same number of objects are initial mapped to each Cell processor, the workload of each PPE seems the same (i.e. the amount of work it takes to offload to an SPEs, whether there are 6 SPEs or 8 SPEs, is the same). In fact, it is even worst. In our previous static load balancing experiments, we were able to show that the PS3s took less time to perform a network send compared to the QS20 blades when Ethernet was being used as the interconnect [64]. This can result in the PS3s taking less time per AEM invocation, on average, which helps to explain the greater number of pair compute objects being mapped to the PS3 Cells (PEs 4 and 5), rather than the QS20 Cells (PEs 0 through 3) as shown in figure 5.36, along with the decrease in performance when using AccelEvenLB in figure 5.35. Indirectly measuring the influence of accelerators by across host core (or across node) load balancers is not sufficient in all situations. Through the use of the accelerator manager and the PercentDevice accelerator load balancing strategy, a single high-level load balancing mechanism that includes all host cores and accelerator devices alike could be created (i.e. components external to the accelerator manager can set the percent device values). However, the creation of such a load balancer is left as future work. Here, we simply wish to point out one way in which indirect measurement can be insufficient for one accelerator architecture, even though the same method is sufficient for another architecture.

5.7.2 AdjustBusy

Figure 5.37 shows the MD application executing using the QS20 blades within the Cell cluster and with the AdjustBusy accelerator load balancing strategy being applied. The MD application was run with and without the AccelEvenLB strategy. The difference between using and not using the AccelEvenLB strategy is negligible. Additionally, the performance of the MD code executing on 4 QS20 blade Cells has also been included for reference.

What is quite noticeable about figure 5.37 is that the performance of the MD application while using the AdjustBusy strategy steadily decreases until the application is executing quite a bit slower (per timestep) compared to executing the same configuration using the Sampling (centralized) strategy. While the performance differences between the AdjustBusy and Sampling strategies were generally much more comparable

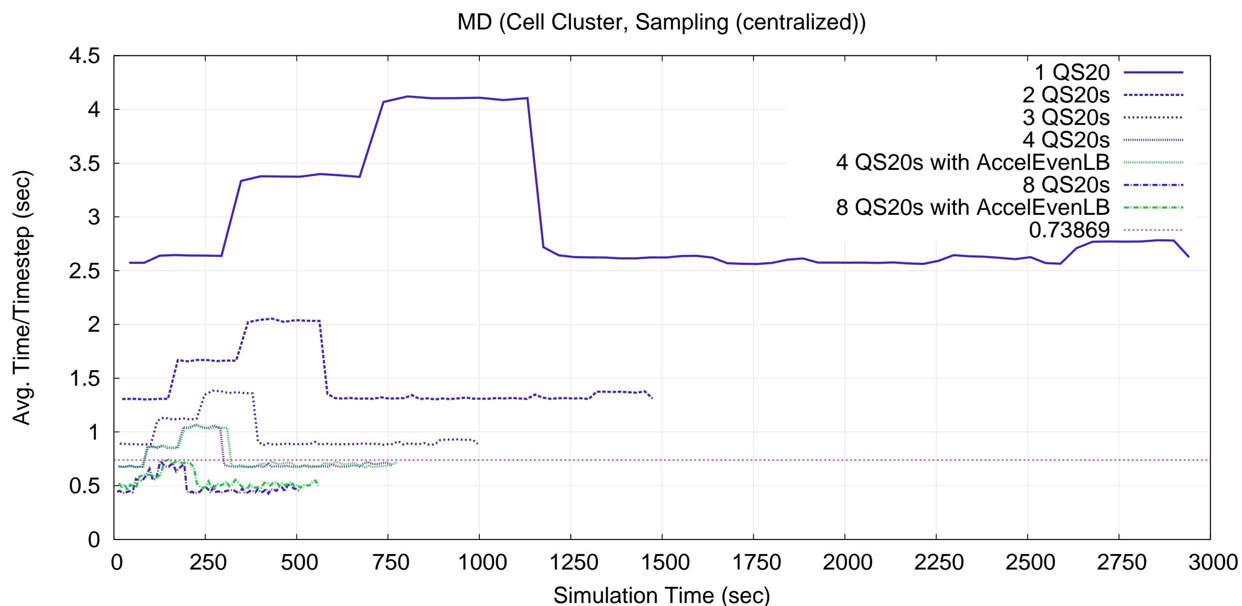


Figure 5.37: The MD application executing on QS20 blades in the cell cluster, using the AdjustBusy accelerator load balancing strategy.

when executing the MD application on the GPGPU cluster (Forge), the difference in performance is rather drastic on the heterogeneous cluster.

To understand the steady performance degradation within the MD application caused by the AdjustBusy strategy, recall that the AdjustBusy strategy does not actually measure the performance of the application in order to determine how to balance the workload between host cores and accelerator devices. Instead, the AdjustBusy strategy applies a *blind* heuristic that simply tries to even out the busy times of both the host core and the accelerator device in order to ensure both compute resources are being fully utilized. As discussed above within the context of the Sampling strategy, one characteristic of the Cell processor is that the PPE must remain responsive to the needs of the SPEs in order for the SPEs to remain busy. As a consequence, having idle time on the PPE is actually beneficial to performance. More to the point, any long running block of code (i.e. entry method that takes a while to execute), such as the compute objects' force computation accelerated entry methods, cause the PPE to become unresponsive and the SPEs go idle waiting for more work.

To make matters worst, if the accelerator device's busy time measurement takes place solely from the point-of-view of the host core, then the cycle of moving work from the accelerator device to the host core becomes a self-reinforcing endeavor. First, by "busy time measurement ... from the point-of-view of the host core," we mean to say that the busy time of the accelerator device is defined as the total amount of time that there is at least one AEM dispatched to the accelerator devices. This definition is rather intuitive and

useful since it measures to the actual amount of time it costs to perform a set of AEMs on an accelerator device, regardless of the device's architecture, in a way that has low amount of overhead and naturally includes many secondary effects, such as overlap of computation with data movement, parallelism within the accelerator device itself, and other factors as well. Using this definition, it is natural to make the timing measurements on host core itself as AEMs (or batches of AEMs) are dispatched and completed. Doing so includes all of the data transfer times and notification delays as part of the *cost of offloading* the AEM(s) to the accelerator device from the point-of-view of the host core. However, what this does not do is take into account the root cause of the delays. In particular, if there is a long running entry method (task) executing on the host core, the host core is not paying attention to the accelerator device. As such, the timing measurements themselves are delayed since the host core does not notice the completion of one or more AEMs in a timely manner. The delay on the host not only causes the SPEs to become idle, but it also delays the time measurements associated with the AEMs executing on the SPEs. As such, the more work that is placed on the PPE, the slower the SPEs seem to be from the point-of-view of the host core. As the SPEs seem slower, even more work is shifted from the SPEs to the host core by the AdjustBusy strategy, causing a self-reinforcing cycle of moving data from the accelerator device to the host core. This continues until enough work has been removed from the SPEs so that, even with the long, artificial delays, the PPE's busy time is equal to the SPEs' "busy time."

From this we learn that the way in which measurements are made by the runtime system can have a large impact on resulting decisions that are made by that runtime system. While a method of measuring may seem fairly straight forward and intuitive, it may in fact lead rather bad results in certain conditions. When measuring an accelerator device's busy time, there are two key factors that lead to issue discussed above. First, the timing measurements are being taken from the host core's point-of-view. Second, the host core only performs those timing measurements as it polls the state of the device. In the case of the Charm++ runtime system, the "polling" effect is a natural result of the semantics of the programming model itself, which Charm++ has in common with many other parallel programming models. In particular, the runtime system can only take action at particular points within the application's execution. In Charm++ specifically, since the tasks (entry methods) are non-interruptible, the runtime system only regains control of the host core in-between entry method executions or at any other time the application code explicitly calls an API function provided by the runtime system, creating the "polling" effect. This is also true of using CUDA directly, since the programmer is required to either block until a kernel completes or have the host core test (poll) for the completion of a kernel directly. Anything that delays the runtime system from polling the accelerator device will cause an incorrect measurement, as long as the measurements are taken during those

polling points within the runtime system.

5.8 Summary of Dynamic Load Balancing

In this chapter, we have demonstrated the application of our approach on both a 5-point stencil application and a simple MD application. Using the accelerator load balancing strategies within the accelerator manager, the workload of these applications were automatically and dynamically balanced between the host cores and the accelerator devices available to the applications. In the discussing the performance of the applications when applying the various strategies, we pointed out some of the pitfalls that could be encountered when implementing strategies.

Interference between multiple applications sharing the same physical accelerator devices was also discussed. It was shown that our approach is capable of reacting to interference created by other applications making use of the same physical hardware, by shifting some of the application's workload from the shared hardware resources to the exclusive hardware resources. The net effect is that all of the applications are able to make use of the shared hardware resources. In some cases, the overall throughput of the device is increased, even if the throughput of the device from any one application's point-of-view is diminished.

We pointed out at least one other use of splittable accelerated entry methods. In particular, that splittable AEMs can be used to decrease the average time per entry method, and thus increase the frequency at which the runtime system regains control of the host core(s) from application code. This turns out to be important in the case of accelerators in cases where the host core is required to perform tasks on the accelerator's behalf, including issuing work and polling for the complete of that work. Further, depending on the method used by the runtime system to take time measurements related to the accelerator device, the polling frequency can have adverse effects on the quality of those time measurements.

Finally, we demonstrated that our approach does indeed allow for the creation of portable code by applying the runtime system modifications and dynamic load balancing techniques developed for the GPGPU to the MD application executing on a set of Cell processors. We also discussed why the AdjustBusy accelerator load balancing strategy is inappropriate for the Cell processor, at least as we have implemented it so far.

Chapter 6

Data Management

This chapter will discuss the various ways in which the runtime system moves and/or modifies the application data in an attempt to make programming heterogeneous systems less cumbersome for programmers. In some cases, the host cores of a system are heterogeneous and require the programmer to perform various mundane tasks, such as correcting for endianness differences between cores. From the point-of-view of the programmer, such tasks are tedious and mainly serve as a potential source of programming errors since their functionality has little (or nothing) to do with the overall goals of their application. In other cases, the underlying hardware may have some performance considerations to take into account when executing chare objects on them, such as the shared memory area of the GPGPU which is banked. Since GPGPUs have a SIMD execution model, it would be nice if the scalars from the various chare objects didn't all map to the same physical banks, forcing the accesses to a single bank to be serialized. In yet other cases, it might be advantageous to keep some of the application data within the accelerator device's memory, as the accelerator device will be the main or only processing element that accesses that data. As part of this work, we apply some of these techniques to the Charm++ runtime system to illustrate how these techniques can be applied to applications.

6.1 Automatic Modification of Data Crossing Processor Boundaries

One of the most straight forward cases where an underlying runtime system can assist a programmer is to automatically modify application data for host core differences. Many of these differences require in fairly straight forward adjustments within the application code, though such adjustments may be numerous. For example, if an application is making use of multiple host core architectures, including architectures with big endian and little endian data encodings, the programmer will be required modify the byte orders of the data within their application. At each point in their application where data is either being sent or received, such as entry method invocations in Charm++ or MPI send/receive calls, the programmer would have to modify

the data to correct for endianness differences. At a minimum, this must occur at points in the application where data is received, in the case of lazy data modification.

In the specific case of implementing an automatic data modification technique within the context of Charm++, make make the observation that there are clear communication boundaries within Charm++ applications. In particular, a given piece of application data is contained within one of the chare objects in the application (or perhaps multiple objects if the data is duplicated). For this given piece of data to transferred from one chare object to another chare object, it must be passed as a parameter to a entry method. At that point, the Charm++ runtime system and charmxi tool have access to typing information for the data, at least in principle, via parameter types and PUP routines for those types. Languages and models that have such clear communication boundaries have a natural place for such data modification techniques to occur. However, to automate the application of those techniques, type information is required.

The automatic modification of application data as it passes between host cores with differing architectures was previously discussed [64].

6.1.1 Programmer Impact

Regardless of the mundane nature of such data modifications (e.g. handling endianness or pointer sizes), there are a few things to note about how such requirements impact the programmer. First, the code must be explicitly included by the programmer in many programming models, since the programming models do not implicitly support such automatic data modifications. Because of this, the programmer is required to include, and potentially clutter, their application code with data correction code that has little or nothing to do with the overall goal of their application. The overall impact of this requirement is dependent on the nature of the application code. In cases where the communication points are limited, then the required amount of data modification code would also be limited. However, in cases where communication is happening at several locations, such data modification code would be littered throughout the application code. Second, as long as there is typing information, such data modification should be fairly straight forward to automate through compiler and/or runtime system support. In our case, we make use of a runtime system to automate such modifications in a lazy manner. This not only removes the need to have the data modification code explicitly included within the application code, but removes the requirement that programmers spend time thinking about and implementing techniques to handle such architecture differences, allowing programmers to focus on the goal of their application code. Further, such efforts aren't duplicated by multiple programmers working on multiple applications. Third, the inclusion of such functionality, while required, also servers as a point for programming errors to be introduced within an application. Removing the requirement for programmers to

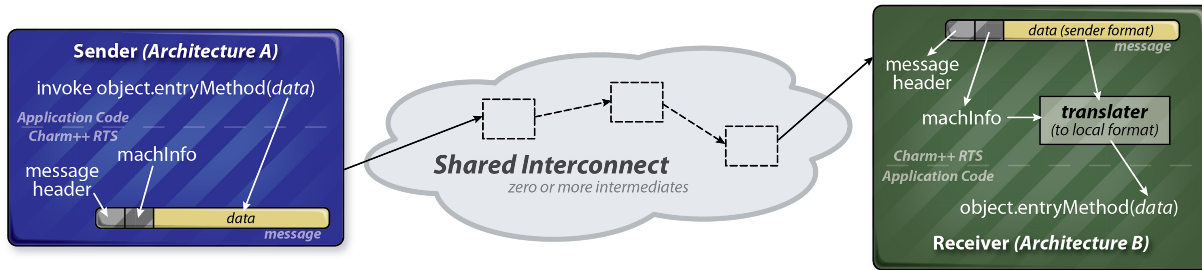


Figure 6.1: Modifications to the send/receive process in a heterogeneous builds of the Charm++ runtime system.

explicitly write such data modification code would also remove the potential for programming errors being introduced in relation to that code. There are clear benefits for automation such data modifications within the underlying runtime system.

6.1.2 Method

Figure 6.1 illustrates the overall method used to implement automatic application data modification within the Charm++ runtime system. There are two main modifications to the message send/receive process within Charm++. The first modification is on the sending side of the communication process. Charm++ messages can conceptually be broken down into two components, the header and the payload. In actuality, the header has several sub-components, but those details can be ignored for this discussion. The pertinent piece of information for the reader is that additional formatting information has also been included as part of the header as part of this work. This formatting information is simply included. No modification of application data occurs as a message is sent from the source processing element. In figure 6.1, this additional formatting data is marked as “machInfo.”

On the receiving side, this formatting information is compared to the receiving processor’s local formatting information. If the incoming message has the same format of the target processor, then no translation of application data takes place. However, if the formatting of the message differs from the format of the target processors, the payload of the message is passed through a translator to account for endianness and data type differences between the source and target processor architectures.

The actual implementation of the data translation process is *lazy*. That is, any data translation only occurs as needed and just prior to that need. In the case of the message headers, a network format is used when the heterogeneous functionality of the runtime system is enabled. This network format defines the endianness of the network messages, resulting in a message format that is independent of the sending and

receiving processors' architectures. However, the payload of the message remains in the format of the sending processor until the moment just prior to the entry method being executed. An advantage of this approach is that the payload doesn't have to be translated multiple times if an intermediate node in the network does not share the same architecture as the sending and receiving processors. For example, consider the case of the runtime system making use of a spanning tree optimization to implement a broadcast operation. In such cases, an intermediate node can examine the contents of the header and *realize* that the message needs to be forwarded on to the next node in the spanning tree without having to apply any translation to the payload. Furthermore, if the sending processor and a given receiving processor of the broadcast message have the same architecture but an intermediate node has a different architecture, then no translation of the payload would occur. As result, translation only occurs as needed.

At this point, it should be noted that the *machInfo* (a data structure describing a processor's encoding) and the *translator* (a *PUP::er* that does the grunt work of swapping endianness) entities existed in Charm++ prior to this work. The contributions made by this work revolve around the modification of a machine layer to support messages between different architectures (i.e. implementing the network format with message headers), the definition and implementation of the lazy mechanism that actually applies the translator to message payloads, and the related code generation modifications/additions within charmxi that assist enable this implementation.

6.2 Data Within GPGPU Batch Sets

When it comes to the data associated with accelerated entry methods, there are two types: streamed and persistent. Every parameter is assumed to be streamed data, as the values embodied within those parameters are only valid for the given invocation of the AEM. Local parameters, on the other hand can be marked with the "persist" keyword, indicating to the runtime system that the values associated with these parameters can persist within device memory between invocations of the AEM on that chare object. Persistent data will be discussed later in section 6.3. All values associated with passed parameters and local parameters not marked with the "persist" keyword are streamed through the GPGPU at the same time the AEM invocations are streamed through.

The streamed data associated with a batched set of accelerated entry method invocations (i.e. a single CUDA kernel invocation) is placed into a single *combined data buffer*. This combined data buffer contains various pieces of control information associated with the kernel's execution, the data values contained within the streamed parameters, and pointers used to locate the persistent data that will be accessed within the

kernel. The combined data buffer has three major regions: the control region, the scalar region, and the array region. The control region contains information related to the overall kernel execution, such as fields used for kernel execution verification and an overall error status. The control region also contains information specific to each AEM invocation, such as the location of data associated with each AEM invocation so that it can be located in constant time on the device itself and the number of splits associated with each splittable AEM invocation. The remaining two regions contain the scalar and array values associated with the scalar and array AEM parameters, respectively. The reason for using two sections will be discussed further in section 6.2.1.

Each AEM invocation with a batched set of invocations is given an index value which is used to uniquely identify the AEM invocation with the set and locate the data associated with that specific invocation. As each AEM invocation is added to the batched set, this index value is incremented from a starting point of zero. The GPGPU threads are also assigned to the AEM invocations on the device in a similar manner, with the lowest indexed threads being assigned to the lowest indexed invocation. If the AEM type being batched is not splittable, then there is a one-to-one correspondence between the AEM invocation index and the GPGPU thread index. In cases where the AEM type being batched is splittable, a consecutive number of GPGPU thread indexes are assigned to the AEM invocation index, based on the number of splits created by the runtime system for that particular AEM invocation.

6.2.1 Data Alignment of Streamed Parameters on GPGPUs

In the case of CUDA-based GPGPUs, the runtime system is manipulating the layout of data associated with a batched set of accelerated entry methods. To understand why manipulating the layout of the data within a batched set of AEMs could be advantageous, one first needs to understand the architecture of the underlying hardware itself.

In modern Fermi-based GPGPUs, the shared memory associated with each SM is divided across 32 banks [80]. The reason for doing this should be fairly clear. There are 32 GPGPU threads executing in lockstep (i.e. SIMD) per SM at any given moment. Each time a load or store instruction is reached in the instruction stream being executed by a warp, all of the active GPGPU threads will attempt to access memory at the same time. As long as each thread is accessing data in a separate bank of shared memory, each of the 32 banks can then service one of the 32 GPGPU threads' memory requests, allowing all 32 memory access to occur in parallel. However, when using CUDA directly, a programmer must mindful of the memory access patterns within their application as the GPGPU threads access shared memory. Otherwise, if the programmer unknowingly causes all of the threads to access the same bank or a small fraction of the banks,

there may be a large impact on the performance of the GPGPU code, and thus the application, since only a fraction of the bandwidth between the GPGPU's processing cores and shared memory banks will be utilized. This is just one of many examples of how the architecture details associated with accelerator architectures can have drastic effects on the performance of application code.

A second important architecture detail is that, in Fermi-based GPGPU architectures, the shared memory of a given SM can be used in one of two ways, as shared memory or as an L1 cache that sits between device memory and the processing cores of the SM. In the case of accelerator entry methods being executed on CUDA-based GPGPUs, the runtime system automatically places the shared memory into the L1 cache mode (as much of the memory as possible) and treats all memory accesses as accesses to device memory.

With these two architecture specific details in mind, the batching process creates an array for each scalar parameter for an AEM within the scalar region of the combined data buffer. The values of the scalar values are then packed into consecutive indexes of these arrays each time an AEM invocation is added to the batch set (i.e. the scalars for AEM invocation index i can be found in array element i of each scalar array). There are two cases where this data layout will be beneficial to the program, non-splittable AEMs and splittable AEMs with low (less than 32) splits. In both of these cases, the non-overlapping subsets of the GPGPU threads within a single warp will be accessing different memory addresses each time a scalar parameter is accessed within the function body of the AEM, since the non-overlapping subsets of threads will be assigned to different AEM invocations. However, since the scalars are being packed into consecutive array elements and the GPGPU threads are assigned in the order of AEM invocation indexes (i.e. lower GPU thread indexes to lower AEM invocation indexes), the threads within a single warp will either access the same memory address (i.e. the same hardware bank for threads assigned to the same AEM invocation) or consecutive memory addresses (i.e. different hardware banks for threads assigned to different AEM invocations). It should be further noted that each scalar array is 128 byte aligned. As a result, if each thread within a warp is assigned to a separate AEM invocation, a highly efficient data access pattern from the point-of-view of the hardware is used (i.e. one thread per bank), making maximal use of the shared memory bandwidth. Further, this is accomplished without requiring the programmer to do anything special within the application code.

A downside to this approach is that the runtime system is arbitrarily moving around data within a single chare object. The data layout guarantees related to member variables of chare classes that would otherwise be provided by C++, will not be maintained within the function body of an accelerated entry method. Examples of such guarantees may include the ordering of member variables with memory, the differences in the memory addresses of those member variables, and so on. However, we view this as a minor inconvenience

for programmers.

6.2.2 Shared Buffers

In some cases, the data being streamed through the accelerator device may be shared between two or more of the accelerated entry method invocations. For example, in the case of the molecular dynamics program, the particle data for any given patch object is required by all of the compute objects associated with that patch object. If multiple compute objects associated with a given patch are located on the same host core, then it would be a waste to transfer two copies of the particle data, one for each invocation, to and from the device. Through the use of shared parameters, only a single copy will be transferred per batch set.

Within a single batch set, for each array parameter that is marked with the shared keyword, a realtime lookup is done on the pointer value and length associated with the array. The lookup is done against a hash table, using the pointer value and the length as the key value. If the data has not already been included within the combined buffer for the batch set, the data is included and the offset of the data within the batch set's combined buffer is included with the hash table entry for the array. Any other AEM invocation that is included as part of the same batch set and that accesses the same data buffer using one of its parameters (it does not have to be the same parameter, just the same data) will have that parameter directed to point to the same memory address as the original copy. The overall result is that only a single copy of the data is streamed through the accelerator device per batch set that requires it, which can significantly reduce the size of the combined data buffer. Of course, the extend to which shared buffers can be used is entirely dependent on the nature of the application code.

As a concrete example of how this mechanism operates, figure 6.2 gives the general structure of the PairCompute objects' force calculation, "doCalc," which is embodied within the accelerated entry method in figure 6.2. The function body of this AEM is made up of three loops, referred to as the *outer-loop*, *inner-loop 0*, and *inner-loop 1*. Two list of particles are being passed into the accelerated entry method, "0" and "1," the contents of which are divided amongst the local parameters with the associated numerals. The local parameters staring with the letter "p" represent input data coming from the associated Patch objects. Of these local parameters, the last letter in the name represents the attribute represented by the list (i.e. "x" is the x-coordinate, "y" is the y-coordinate, "z" is the z-coordinate, and "q" is the charge). In the same vein, the local parameters starting with the letter "f" represent the force vectors being generated within this function, each of which has an x, y, and z component.

There are a couple things worth noting about the inner-loops in figure 6.2. First, there are two inner-loops because the AEM is splittable, and thus separate splits may be executing each iteration of the outer-loop. If

```
entry [ triggered splittable(numParticles) accel ] void doCalc()[
        readonly : int numParticles <impl_obj->numParticles>,
 readonly shared : MD_FLOAT p0_x[numParticles] <impl_obj->particleX[0]>,
 readonly shared : MD_FLOAT p1_x[numParticles] <impl_obj->particleX[1]>,
 readonly shared : MD_FLOAT p0_y[numParticles] <impl_obj->particleY[0]>,
 readonly shared : MD_FLOAT p1_y[numParticles] <impl_obj->particleY[1]>,
 readonly shared : MD_FLOAT p0_z[numParticles] <impl_obj->particleZ[0]>,
 readonly shared : MD_FLOAT p1_z[numParticles] <impl_obj->particleZ[1]>,
 readonly shared : MD_FLOAT p0_q[numParticles] <impl_obj->particleQ[0]>,
 readonly shared : MD_FLOAT p1_q[numParticles] <impl_obj->particleQ[1]>,
       writeonly : MD_FLOAT f0_x[numParticles] <impl_obj->forceX[0]>,
       writeonly : MD_FLOAT f1_x[numParticles] <impl_obj->forceX[1]>,
       writeonly : MD_FLOAT f0_y[numParticles] <impl_obj->forceY[0]>,
       writeonly : MD_FLOAT f1_y[numParticles] <impl_obj->forceY[1]>,
       writeonly : MD_FLOAT f0_z[numParticles] <impl_obj->forceZ[0]>,
       writeonly : MD_FLOAT f1_z[numParticles] <impl_obj->forceZ[1]>
 ] {
 ...
 // The outer-loop
 for (int j = splitIndex; j < numParticles; j += numSplits) {
  ...
  // The inner-loop 0 (interact particle in list 0 at index j with list 1)
  for (int i = 0; i < numParticles; i++) { ... f0_x = ...; f0_y = ...; f0_z = ...; }
  ...
  // The inner-loop 1 (interact particle in list 1 at index j with list 0)
  for (int i = 0; i < numParticles; i++) { ... f1_x = ...; f1_y = ...; f1_z = ...; }
  ...
 }
 ...
} doCalc_callback;
```

Figure 6.2: Example of using shared local parameters (PairCompute from MD code).

the runtime system does use separate splits at runtime, making use of *equal-and-opposite force calculations* would result in a data race hazard. By creating the two separate inner-loops, this data race is avoided. If the AEM were not splittable, forcing the runtime system to use a single thread of execution for all iterations of the outer-loop, then an *equal-and-opposite for calculation* could be used. Second, the inner-loops in the actual code are written to make use of SIMD instructions, even though the code in figure 6.2 suggests that they are not. Generally speaking, the code presented in figure 6.2 has been simplified for the purposes of clearly presenting the ideas to the reader and for brevity, but not in such a way as to miscommunicate its general nature.

In figure 6.2, the local parameters containing the particle data (i.e. starting with the letter "p") are all marked as *shared*. Further, the actual pointer values that are passed into the AEM via these parameter are shared by multiple invocations of this same AEM. For example, consider the case (which does occur in the execution of the MD code) where PairCompute α uses Patch γ's particle data for its first list of particles and PairCompute β uses Patch γ's particle data as its second list of particles. In this case, the pointer value passed into PairCompute α's *p0_x* local parameter will match the pointer value passed into PairCompute β's *p1_x* local parameter. This is also true for all of the particles' attributes (i.e. α's *p0_y* also matches β's *p1_y*, and so on). Since the local parameter is marked as "shared," the otherwise separate invocations of the *doCalc* AEM on PairCompute objects α and β will operate on the same physical copy of the particle data in memory (as long as they are batched together into the same batch set of AEMs, otherwise, the "shared" keyword has no effect). This occurs because of the live lookup of the pointer values, even though the single pointer value is being passed to the two AEMs through different local parameters. The final result is that only a single copy of the data is transferred per batch set between the host and the accelerator device, reducing the overall amount of data required to be transferred for that batch set.

6.3 Persistent Data

In some cases, it may be advantageous to keep some portion of the application data located in the accelerator device's memory, when possible. In particular, any data that is only accessed by accelerated entry methods has no need to be transferred to and from the host memory in between calls to the accelerated entry methods that make use of it. With the exception of initialization or cleanup at the beginning or end of the application's execution, the movement of this subset of the application data may only take time without providing any benefit as it is not actually accessed by the host core. As such, we have included the ability to mark some of the local parameters associated with an accelerated entry method as persistent.

6.3.1 Description

In the local parameter list associated with an accelerated entry method, the programmer can mark a parameter as being persistent. To do so, the programmer uses the keyword "persist" as the modifier to the local parameter. When the "persist" keyword is used on a local parameter, the intent is that the actual buffer being passed into the accelerated entry method will not be modified on the host core outside of any accelerated entry methods. The exception to this is during the initialization of the application, before any accelerated entry methods are triggered.

The first time a given host buffer is passed as a persistent local parameter, the runtime system allocates a buffer on the accelerator device, creates an association between the host and device buffers, and copies the contents of the host buffer into the associated device buffer (assuming this process is appropriate for the given accelerator device). As accelerated entry method invocations occur, rather than including the content of the host buffer directly into the kernel's data buffer as described in section 6.2, a pointer to the device buffer is included instead. Upon completion of the accelerated entry method, the device buffer's contents remains in the device's memory, rather than being transferred back to the host core. As such, any data generated by an accelerated entry method and written into a persistent local parameter will not be available to code executing on the host core. However, it will be available to another invocation of an accelerated entry method, whether its the same accelerated entry method or not, if that invocation occurs on the accelerator device.

Since the host buffers are looked up each time they are used in an accelerated entry method invocation. This allows techniques, such as double buffering, to still be used by simply swapping pointers of the persistent buffers on the host core. In section 6.3.2, the 5-point stencil application will be modified to use persistent buffers, giving the reader an example of how to use persistent buffers. The 5-point stencil application makes use of a double buffering technique, which will illustrate the dynamic associate of local parameters within accelerated entry method invocations. If the buffers were associated with particular local parameters, rather than host buffers, then any form of dynamic selection would not be possible since a given local parameter would have to access the same device buffer for every invocation of that accelerated entry method. Further, local parameters to different accelerated entry methods would not be able to access the same device buffer.

It should be noted that mechanisms to *push* and *pull* the contents of a persistent buffer are available to the programmer. Should the host core require access to the contents of a data buffer, the code executing on the host core can explicitly *pull* the content of the device buffer back into the associated host core buffer. If the host core modifies the data, the updated contents can be explicitly *pushed* back to the device's memory. However, this push is only required if the contents have been updated, as the contents of the device buffer

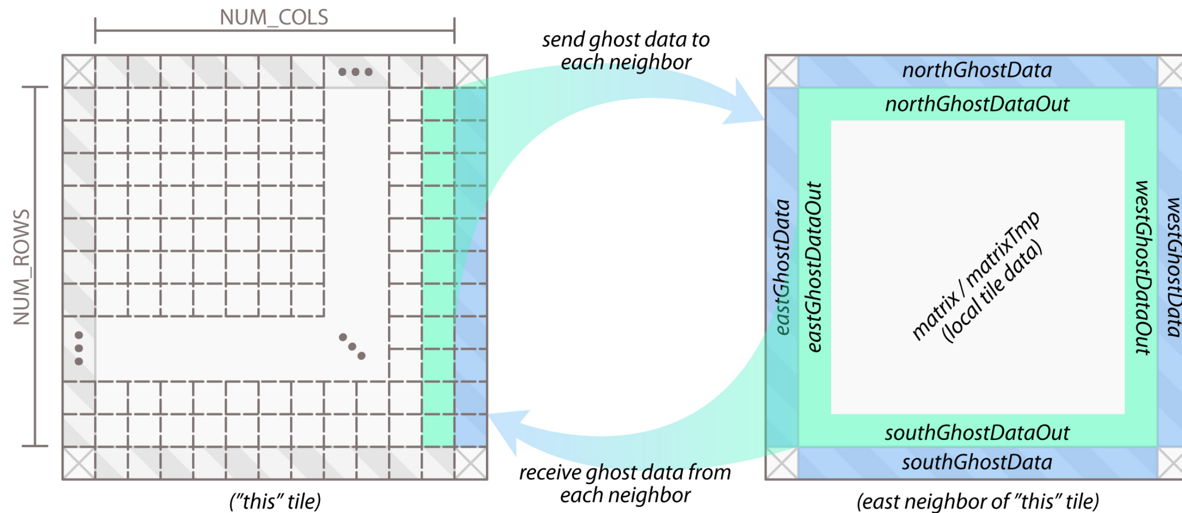


Figure 6.3: The data patterns of the 5-point stencil application.

remain intact. These push and pull operations are blocking, causing giving the host code immediate access to the data without having to poll for completion. The use of these operations is discouraged since their blocking nature could potentially impact performance in a negative way if used too often. These same push and pull mechanisms are used within the runtime system as chare objects are load balanced, checkpointed, or at any other time where the values within the device buffer must be accessed on the host core.

6.3.2 Application Code Example (5-Point Stencil Modification)

To illustrate the use of persistent buffers, the 5-point stencil code has been modified to make use of the "persist" keyword. Before describing the code changes required by the use of the "persist" keyword, we first revisit the data pattern of the 5-point stencil application since the changes revolve around those data patterns.

Figure 6.3 illustrates the data access pattern of the 5-point stencil application from the point-of-view of a single tile interacting with its east neighbor. The same data transfers occur with each of the tiles other neighbors, even though those interactions are not reflected in figure 6.3. Before executing the local tile calculation, a given tile must exchange ghost information with each of its neighbors. Each tile will pass data elements along each edge of its local tile data to the corresponding neighbor tile (north edge passed to northern neighbor, east edge passed to eastern neighbor, and so on). In figure 6.3, this is shown by the green region of the "this" tile being copied to the blue region of "this" tile's eastern neighbor. The inverse data transfer also occurs.

In making use of persistent buffers, the pattern of exchanging data between tiles has not changed.

However, because the tile data will remain on the accelerator device itself (the effect of using persistent buffers), additional buffers are required to pass the incoming and outgoing data for each tile between the host and accelerator. For each neighbor there is a buffer for holding the input ghost data (*xxxxGhostData*, where *xxxx* is replaced by the direction of the neighbor). As the ghost data arrives, it is placed into this input buffer, prior to the execution of the *doCalculation* accelerated entry method. For each neighbor, there is also an output buffer hold the associated edge of the tile data (*xxxxGhostDataOut*). Once this data has been exchanged, the updated *doCalculation* accelerated entry method, shown in figure 6.4, is triggered.

Notice that the code in figure 6.4 is very similar to the code presented in figure 5.7. The modifications to the code have been marked with comments. The main difference is that the contents of the input and output ghost buffers are now contained separate buffers, rather then being included directly within the local tile matrices, *matrix* and *matrixTmp*. Again, this is because of the effect of the "persist" keyword, which causes these matrices to remain within device memory. The other entry methods within the application that take care of sending and receiving the ghost information to and from the neighboring tiles will access the ghost buffers, rather than accessing the *matrix* and *matrixTmp* buffers directly (this change is not reflected in figure 6.4).

6.3.3 Results

Host or Device Only

Figure 6.5 shows the relative performance of the 5-point stencil application executing with and without using persistent buffers. The first plot ("Non-Persist w/ DeviceOnly") shows the performance of the stencil code executing without making use of persistent buffers. The second plot ("Persist w/ DeviceOnly") shows the performance of the stencil code making use of persistent buffers. Clearly, the performance of the stencil application is increased through the use of the persistent buffers. This plot shows that the overhead associated with streaming each tile's matrix elements through the accelerator's memory is significant. Since the values are only accessed and updated by the tile's calculation accelerated entry method, the "persist" keyword can be used to indicate that the runtime system can simply leave the values associated with the persistent buffers on the device where they were generated.

Two additional reference plots have also been included in figure 6.5, showing the relative performance of the various approaches attempted so far. The first reference plot shows the performance of the 5-point stencil application executing without persistent buffers and with the Sampling *(centralized)* accelerator strategy (see section 5.4.1). The second reference plot shows the performance when executing using only the host core (HostOnly strategy). The performance achieved when executing the AEMs on the host is

```
entry [triggered splittable(NUM_ROWS) accel] void doCalculation()
  [  readonly persist : float      matrix[DATA_BUFFER_SIZE] <impl_obj->matrix>,
    writeonly persist : float matrixTmp[DATA_BUFFER_SIZE] <impl_obj->matrixTmp>,
    writeonly : float         tmpMaxError[NUM_ROWS] <impl_obj->tmpMaxError>,
     readonly : float       northGhostData[NUM_COLS] <impl_obj->northGhostData>,
     readonly : float       southGhostData[NUM_COLS] <impl_obj->southGhostData>,
     readonly : float        eastGhostData[NUM_ROWS] <impl_obj->eastGhostData>,
     readonly : float        westGhostData[NUM_ROWS] <impl_obj->westGhostData>,
    writeonly : float  northGhostDataOut[NUM_COLS] <impl_obj->northGhostDataOut>,
    writeonly : float  southGhostDataOut[NUM_COLS] <impl_obj->southGhostDataOut>,
    writeonly : float   eastGhostDataOut[NUM_ROWS] <impl_obj->eastGhostDataOut>,
    writeonly : float   westGhostDataOut[NUM_ROWS] <impl_obj->westGhostDataOut>
  ] {

  for (int y = splitIndex + 1; y <= NUM_ROWS; y += numSplits) {

    // Copy this row's east and west ghost data
    matrix[GET_DATA_I(0, y)] = westGhostData[y - 1];
    matrix[GET_DATA_I(NUM_COLS + 1, y)] = eastGhostData[y - 1];

    float maxError = 0.0f;
    for (int x = 1; x <= NUM_COLS; x++) {

      // Copy in north and south ghosts as we move down the row
      if (y == 1) { matrix[GET_DATA_I(x, y - 1)] = northGhostData[x - 1]; }
      if (y == NUM_ROWS) { matrix[GET_DATA_I(x, y + 1)] = southGhostData[x - 1]; }

      int index = GET_DATA_I(x, y);
      matrixTmp[index] = ((2.00f * matrix[index]) +
                          (0.75f * matrix[GET_DATA_I(x - 1, y)]) +
                          (0.75f * matrix[GET_DATA_I(x + 1, y)]) +
                          (0.75f * matrix[GET_DATA_I(x, y - 1)]) +
                          (0.75f * matrix[GET_DATA_I(x, y + 1)])
                         ) * (0.2f);  // Divide by 5.0f

      if (y == 1 && x == 1) { matrixTmp[index] = 1.0f; }
      float tmpError = matrixTmp[index] - matrix[index];
      if (tmpError < 0) { tmpError = -1.0f * tmpError; }
      if (tmpError > maxError) { maxError = tmpError; }

      // Copy out north and south output ghosts as we move down the row
      if (y == 1) { northGhostDataOut[x - 1] = matrixTmp[index]; }
      if (y == NUM_ROWS) { southGhostDataOut[x - 1] = matrixTmp[index]; }
    }
    tmpMaxError[y-1] = maxError;

    // Copy this row's east and west output ghost data
    westGhostDataOut[y - 1] = matrixTmp[GET_DATA_I(1, y)];
    eastGhostDataOut[y - 1] = matrixTmp[GET_DATA_I(NUM_COLS, y)];
  }
} doCalculation_post;
```

Figure 6.4: The calculation accelerated entry method in the 5-point stencil application modified to use persistent buffers.

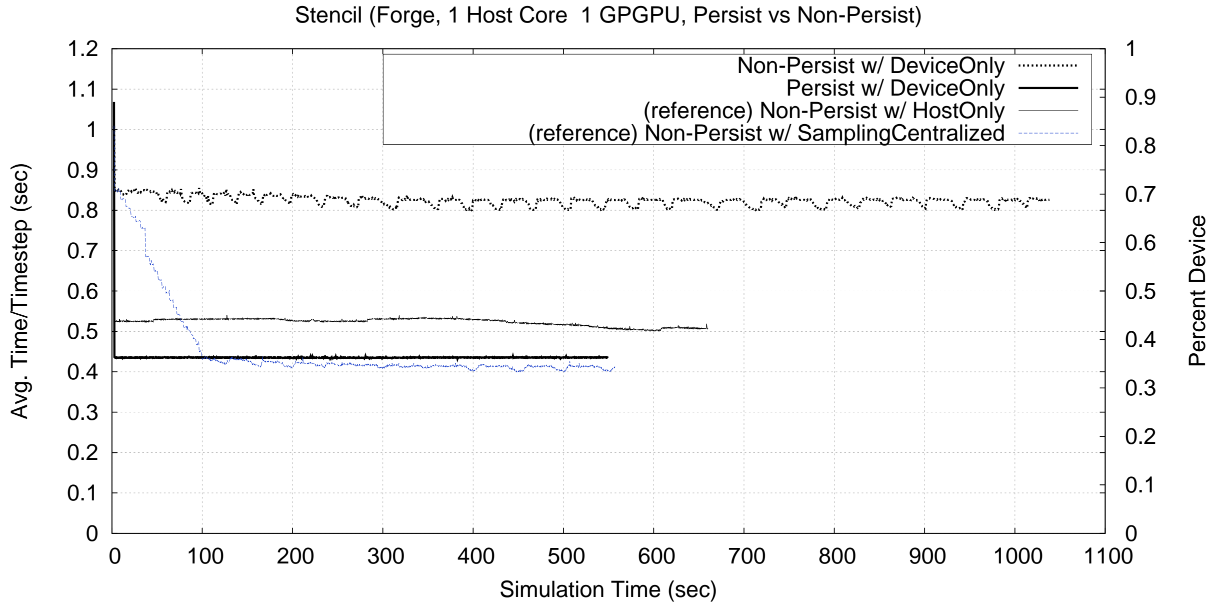


Figure 6.5: The 5-point stencil application executing on a single host core and GPGPU pair on Forge, with and without the use of the persistent data.

particularly important to keep in mind for the stencil application since the stencil application isn't ideal for execution on the accelerator device (i.e. has a low flop rate per byte transferred each timestep). With the inclusion persistent buffer support, the amount of data being transferred to and from the device is reduced to the ghost and the *maxError* data. When comparing the plots in figure 6.5 it is clear that both the use of persistent buffers and the use of load balancing are important to improving the performance of the 5-point stencil application. As such, we will apply the previously discussed accelerator load balancing strategies to the version of the 5-point stencil application that includes persistent buffers.

Load Balancing with Dynamic Assignment

Figure 6.6 goes on to apply some of the load balancing approaches that were discussed in chapter 5 using a dynamic assignment scheme. The term "dynamic assignment" refers to the fact that AEMs that are invoked first are pushed to the accelerator hardware while the those that are invoked last remain on the host. As such, the location where any given object's AEMs are assigned to execute is determined dynamically as the AEM is invoked its associated chare object. From figure 6.6, we can see that the load balancing strategies used do not improve the performance of the 5-point stencil application beyond that of the DeviceOnly strategy. Essentially, the balance point is being correctly determined by strategies as *run all of the AEMs on the device*. The reason running everything on the accelerator device makes sense in these cases is that the accelerator is faster than the host when the amount of data being transferred is reduced, combined with the

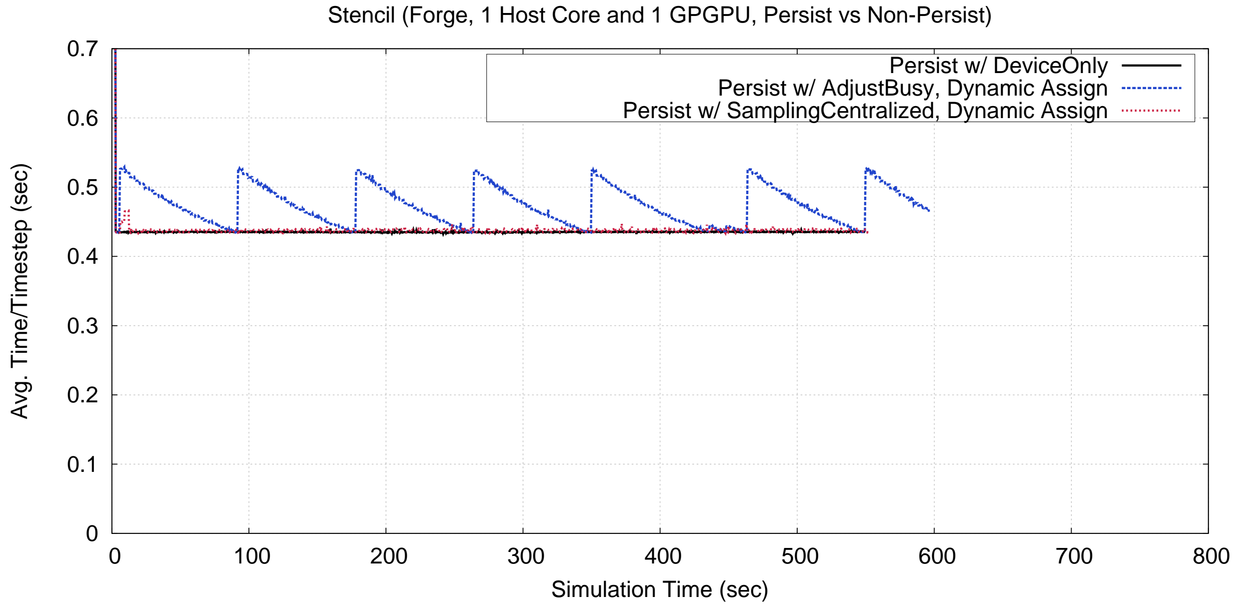


Figure 6.6: The 5-point stencil application using persistent buffers and dynamic assignment.

needs to grab that data from the device whenever an AEM is dynamically invoked on the host core. Since the accelerator is now in a position of out performing the host core, it makes sense to choose the accelerator's memory as the location to keep the data since it is likely that the majority of the AEMs will be executed on the accelerator. Further, the dynamic assignment prevents the runtime system from predicting which AEM invocations will be on the host core, and this preemptively move the data before it is needed. As such, the data transfer is forced to be a synchronous transfer.

The "Persist w/ SamplingCentralized, Dynamic Assign" plot is hard to see since it overlaps completely with the "Persist w/ DeviceOnly" plot. The only significant difference is near the beginning of the execution (at approximately ten seconds) when the SamplingCentralized strategy first attempts to adjust the balance so that some of the AEMs will be executed on the host core. However, based on the actual measurements of the application's performance, the strategy quickly returns to completely executing all of the AEMs on the accelerator device.

The "Persist w/ AdjustBusy, Dynamic Assign" plot has a sawtooth shape to it because the AdjustBusy strategy has is constantly try to balance the busy times of the host and accelerator device. Since the performance of the application is at its best when the accelerator device is executing all of the AEMs, the busy value of the host is lower than the busy time of the device. As such, the AdjustBusy strategy immediately moves some of the work over to the host core. In particular, the AdjustBusy strategy is implemented in such a way that, if it finds the work pushed all the way towards either of the cores, it

takes a larger than normal step off that balance point extreme. In other words, the AdjustBusy strategy is implemented in such a way that it assumes both the host and accelerator will be useful to some degree, so either extreme is bad and will be moved away from quickly. If this is not the case, then one should simply use HostOnly or DeviceOnly strategies instead. We point this out simply because the larger than normal step off the either extreme point (*all on device* or *all on host*) is a detail of our implementation, but certainly not a requirement when implementing a load balancing strategy that attempts to equalize the busy values of the two processing elements. However, this does emphasize a problem with the general approach of balancing busy time values of the host and the accelerator. If the optimal balance point truly is one extreme or the other (i.e. a percent device value of 100% or 0%), a pattern such as the one seen for AdjustBusy can easily occur. When at an extreme, the slower processing element will have 0% of the tasks (AEMs) and the faster processing element will have 100% of the tasks. This extreme balance of tasks may lead to the busy time of the slower processing element being (significantly) smaller than the busy time of the faster processing element, and thus any heuristic that attempts to balance busy times will attempt to move work from the faster processing element to the slower processing element. However, since the balance point truly should be the extreme, the result will be that any amount of work moved to the slower element will result in the slower processing element suddenly having a greater busy time than the faster processing element, and thus the application experiencing diminished performance.

Figure 6.6 shows that the use of persistent buffers also comes with a drawback when used with dynamics assignment techniques. In particular, the chare objects that make use of persistent buffers will have data that is located on the accelerator device. Further, when the workload of a given set of local chare array elements is divided between the local host core and accelerator device, the chare objects that are last (in time) to have their accelerated entry methods invoked are the objects where the AEM will execute on the host core. When taking this approach to AEM assignment, the set of chare objects that execute their AEMs on the host core can change from timestep to timestep. Each time an AEM with persistent buffers executes on the host core, the contents of those buffers must be shifted from the accelerator device's memory to the host's memory, and back again after the AEM completes (updated copies on both cores). As a small optimization, some of these transfers are skipped by the runtime system based on the read/write access modifiers used by the programmer when specifying the local parameters. While this negatively impacts performance when balancing AEMs with persistent buffers between the host and accelerator cores, it does allow for the correct execution of the application.

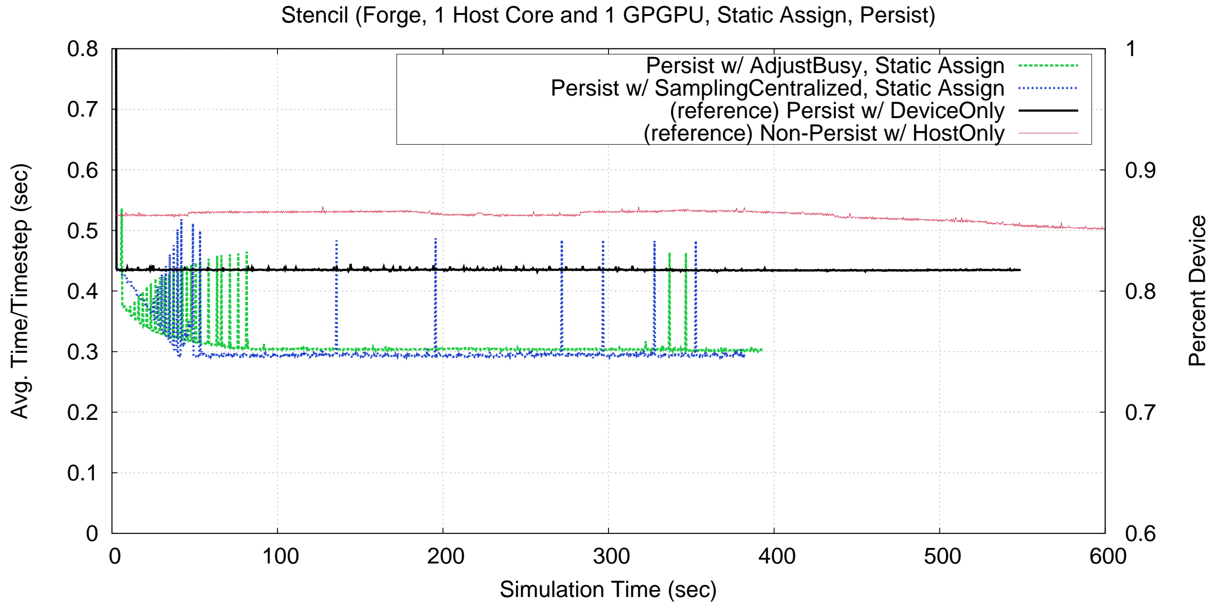


Figure 6.7: The 5-point stencil application using persistent buffers and static assignment.

Load Balancing with Static Assignment

Since the persistent buffers don't seem to be working well with the dynamic assignment schemes used by the accelerator manager and related strategies, we also implemented a static assignment scheme. The term "static assignment" refers to a scheme where the the objects associated with the accelerated entry methods being invoked are assigned to have their AEMs execute on either the host or the device until another rebalancing occurs. Unlike the dynamic scheme, if a remote message associated with an object assigned to execute AEMs on the accelerator device sees an unusually long latency, because of network interference, OS interference, or some other reason, the execution of the batch that AEM will be placed in will also be delayed. Note that this effect is mitigated since the batching process is dynamic (i.e. objects assigned to the device will be batched, but are not assigned to particular batches). In the dynamic scheme, this issue is avoided by simply dynamically assigning the last-to-be-invoked AEMs to the host, rather than to a fixed location, allowing the AEM batch to full up with invocations as quickly as possible. The advantage of using a static assignment scheme is particularly strong when using the scheme with persistent buffers. Since the runtime system is aware that the object's AEMs will be executing in a fixed location, then the persistent buffers associated with that object can also be placed in that location and remain there for an extended period of time (i.e. until the load balancing strategy reassigns the object to a different processing element).

Figure 6.7 show the performance of the 5-point stencil code executing with the AdjustBusy and Sampling (centralized) accelerator load balancing strategies, using static assignment of chare objects. Since the point

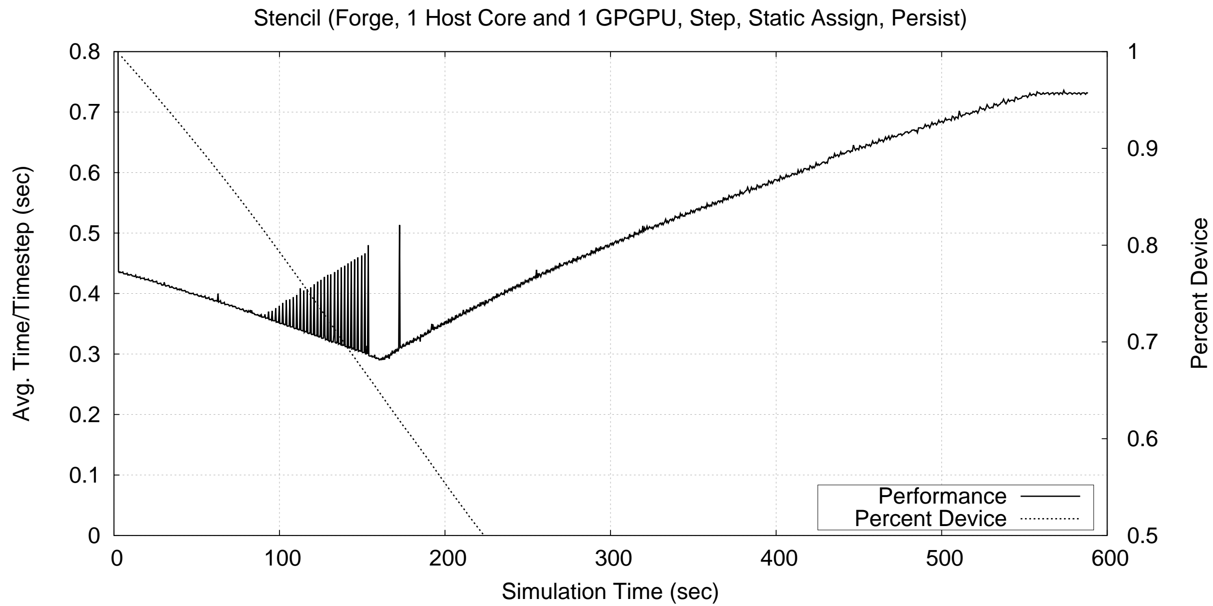


Figure 6.8: The 5-point stencil application using persistent buffers, static assignment, and the Step strategy.

of including this figure is to illustrate the effect of using static assignment rather than dynamic assignment in the context of balancing a workload across host cores and accelerator devices, we only include results from two accelerator load balancing strategies to illustrate the associated points, rather than exhaustively including all strategy combinations. For comparison purposes, figure 6.7 also includes the "Persist w/ DeviceOnly" plot from figure 6.6 and the original plot showing the performance of the 5-point stencil application executing on only the host core and without using persistent buffers from figure 5.12.

One thing that is clear from figure 6.7 is that the use of static assignment is beneficial compared to the use of dynamic assignment when persistent buffers are in use. The benefit of static assignment is that the runtime system does not have to reactively move persistent data associated with an AEM invocation from its current location to location where the AEM invocation has been dynamically assigned to (i.e. the host core or the accelerator device). However, the unpredictable nature of message arrivals (i.e. timing, ordering, etc.) is part of the reason that dynamic assignment was first considered as part of this work. In cases where persistent data is not being used, dynamic assignment allows the runtime system to fill AEM batches as quickly as possible.

To get an idea of how the load balancing strategies in figure 6.7 are performing, the Step strategy has also been applied to the 5-point stencil code in conjunction with persistent buffers and static assignment of chare objects. Figure 6.8 shows the performance of the 5-point stencil code as the percent device value is slowly decreased over the course of the execution from a starting point of 100%. A maximum performance

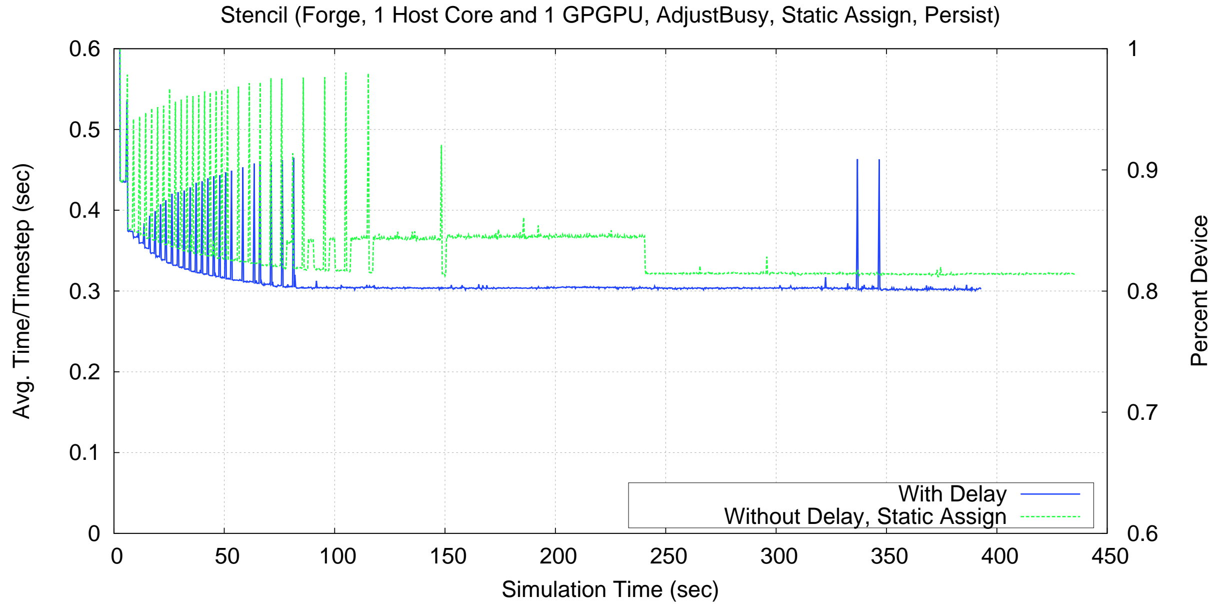


Figure 6.9: The 5-point stencil application using persistent buffers, static assignment, and the Sampling strategy.

level (i.e. minimum time per timestep value) is reached at approximately 160 seconds with a percent device value of approximately 65%.

Delaying AEMs Assigned to the Host: It should be noted that the accelerator load balancing strategies have the option of either issuing AEMs to the host immediately or in a delayed manner. In the cases of static assignment, the accelerator strategies who's implementations include static assignment are currently implemented to delay the execution of AEMs on the host until all of the AEM instances that are assigned to the accelerator device have been batched for triggered accelerated entry methods. The delayed execution of AEMs on the host core allows static assignment to have some of the benefits of dynamic assignment (i.e. overlapping execution of AEMs on the host with the execution of AEMs on the device by ensuring the accelerator is serviced before the host core enters into application code) along with the benefits of static assignment (i.e. not requiring data associated with the AEMs to be moved in a reactive manner as messages arrive and the runtime system makes realtime decisions about where to direct the AEMs). Another benefit of issuing the AEMs on the device before AEMs on the host is that once the batch set is created, the data can start transferring to the device before the execution of the previous batch has finished. In other words, the batches can be streamed through the accelerator device, allowing the execution of one batch to overlap with data transfers, input or output, related to other batches.

Experiments were also attempted with and without the immediate execution of the AEMs assigned to the host. Figure 6.9 shows that the immediate execution of AEMs assigned to the host core results in a small

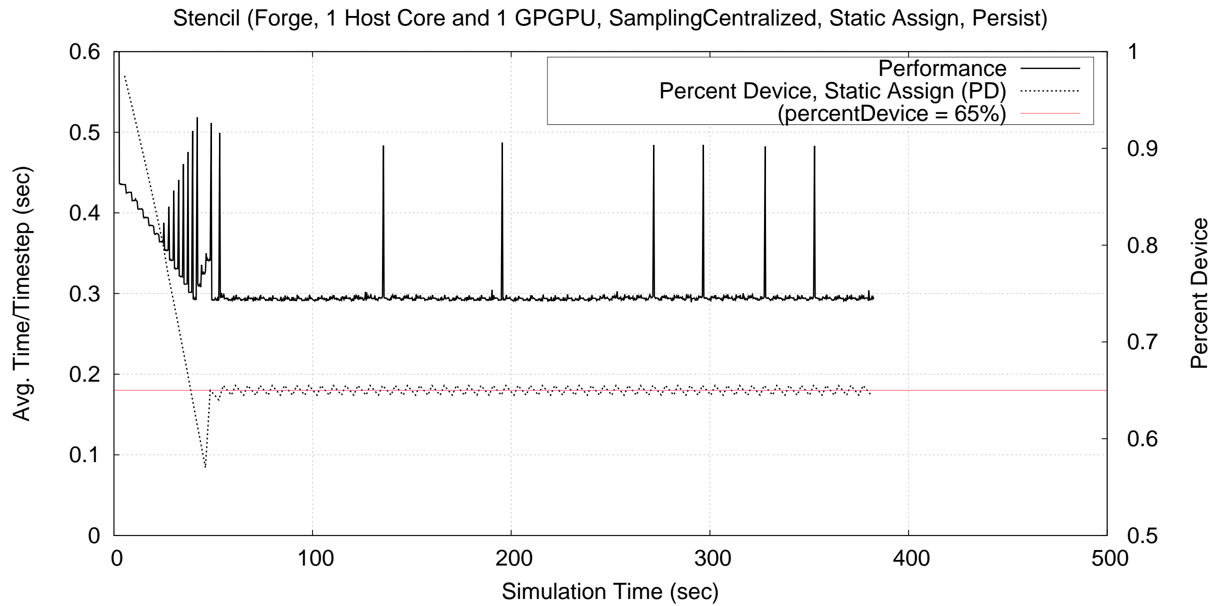


Figure 6.10: The 5-point stencil application using persistent buffers, static assignment, and the Sampling strategy.

decrease in performance on average. In case of immediate execution, if the host core is busy execution an AEM after starting, but before finishing, a batch of AEMs destined for the accelerator device, the execution of that batch on the accelerator device is delayed. Even worst, if no batches have been dispatched to the accelerator device, the AEM executing on the host core is not overlapped with useful work on the accelerator device, resulting in only one processing element is active. However, if the AEMs assigned to the host core are delayed, the batching process can proceed and work can be issued to the accelerator device as soon as possible. Once the accelerator is busy executing its portion of the AEM workload, the host core can move on to execute its portion of the AEM workload in parallel with the accelerator device.

Finally, the exact cause of the short, tall spikes seen in the various plots in figures 6.7, 6.8, and 6.9 remains unknown. Since the spikes only seem to occur when persistent buffers are used, a likely cause could be situations were the runtime system is forced to wait for a blocking copy before forward progress in the application can be made. For the reader's reference, figure 6.10 shows the performance of the 5-point stencil application along with the percent device value being used to balance the workload. This shows both the balance point that was chosen and that the load balancing strategy itself is not the cause of the performance spikes.

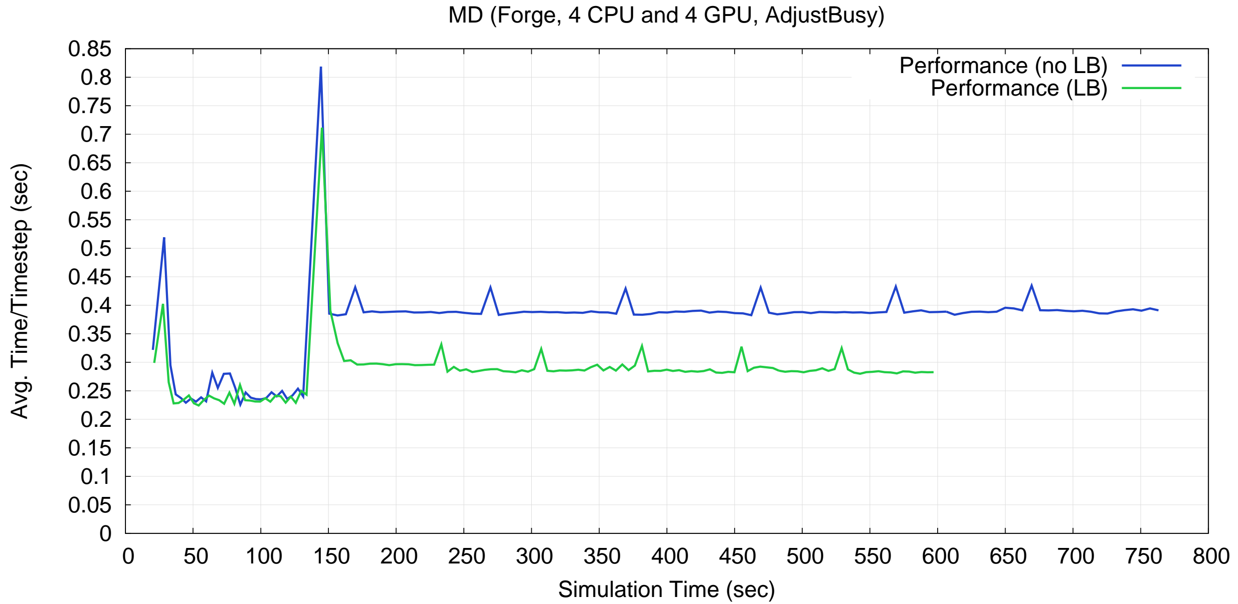


Figure 6.11: Performance of the MD application as it recovers from a host core fault.

6.4 Integration with Fault Tolerance Support

Within the Charm++ runtime system, there exists several methods for recovering from hardware faults [75, 98]. One such mechanism is referred to as automatic memory checkpointing. In this scheme, a synchronization point within the application is used to trigger a memory checkpoint. Upon triggering this memory checkpoint, the runtime system will iterate through all of the chare objects within the applications and use the provided PUP routines to checkpoint the object's state (collectively referred to as a single global application checkpoint). This checkpointing process is typically triggered at periodic points within the application (e.g. every N timesteps). In the event of a hardware fault, the runtime system will rollback the state of the application by recreating all of the chare objects using the state information that was embodied within the most recent checkpoint.

Figure 6.11 shows the performance of the MD application as it recovers from a (simulated) hardware fault using the Charm++ memory checkpoint scheme. Initially, there are four host core and accelerator device pairings in use by the application. At some point within the application's execution (just before 150 seconds), one of the processes associated with the application's execution is *killed* via a signal. At that moment, from the point-of-view of the remaining processes, the host core and accelerator device pairing becomes unresponsive, as would be the case if a hardware fault occurred. Thus, after the fault occurs, there are only three pairings available to the application. The objects associated with the lost pairing are simply recreated on one of the other pairings (only a single pairing) creating a load imbalance.

There are two plots within figure 6.11, one that includes across host core load balancing ("LB" in green) and one that does not ("no LB" in blue). In both cases, the lost of a host core and accelerator device pairing results in a decrease in performance after the fault. Without load balancing, the performance of the application is limited by the pairing that receives all of the work related to the lost pairing. In particular, the performance of the MD application is decreased to half its performance prior to the fault (twice the time per timestep) since one pairing now has twice the work and becomes a performance bottleneck. However, in the case of the "LB" plot in figure 6.11, load balancing is being applied periodically throughout the application's execution. Clearly the use of load balancing after a fault occurs allows the application to readjust to the new hardware configuration (i.e. short one pairing) by redistributing the workload from the faulted pairing so that it is distributed over all of the remaining pairings. In the load balancing case, the resulting performance is approximately 33% slower than the performance seen prior to the fault, which is what one would expect when moving from four pairings to three pairings. Note that four pairings were used because the pre-existing memory checkpointing mechanism that we used required a minimum of four host cores. We use the minimum number of cores to emphasize the performance lost from a single core (i.e. is clearly noticeable in figure 6.11).

In terms of the memory checkpoint mechanism itself, little was done to the checkpointing process itself. Essentially a hook was added into the Pack-UnPack (PUP) process to ensure that whenever a chare object is having its state serialized into a buffer, that any persistent data on the device is pulled and the host has an up-to-date copy of the data. The PUP routines are used to create a checkpoint of the application's state at runtime, which can be used to rollback the application's state in the case of a fault. The memory checkpoint mechanism already requires that an application to be in an idle state (objects are not actively updating), so this already removes the need to account for any accelerated entry method invocations that are currently pending on an accelerator device.

In terms of the recovery process, again, little was changed. Essentially, a single hook was added early in the recovery process that notifies the accelerator manager that a fault has occurred and that appropriate action should be taken. In the event of a fault, the accelerator manager simply flushes its state associated with any objects. If static assignment is being used, then the assignments of objects to particular locations (host core versus accelerator device) is dropped. Any pending accelerated entry method batches are also marked as abandoned. If an AEM batch is still in the process of being formed, but has not been issued to the accelerator device, then the batch is simply destroyed. If an AEM batch set has already been issued to the accelerator device, then the batch is marked as abandoned and a reference to the batch is set aside until the accelerator device completes the batch. Once completed, the generated callback code verifies whether

or not a batch has been abandoned, and if it has, disregards any output data that has been passed back from the device. If persistent data buffers are being used, all associations between host core buffers and accelerator device buffers are broken (which will cause new buffers to be created once the application restarts after the recovery process). Any existing accelerator buffers are retained until all abandoned AEM batches have completed, since any currently executing batches may refer to those device buffers. Once all of the abandoned batches have completed, it is known that no other current or future batches will refer to the abandoned device buffers, so the abandoned device buffers are finally discarded.

In the memory checkpointing mechanism, all objects are discarded in the case of a fault. However, more advanced fault tolerance mechanisms within Charm++ only require a subset of the objects to rollback, reducing the cost of recovery. In these more advanced fault tolerance mechanisms, the states of individual objects are checkpointed, along with the messages associated with entry methods that have been invoked on them since their last checkpoints. In order to accommodate these fault tolerance mechanisms, the checkpoints of the objects themselves would have to be delayed until the completion of any pending accelerated entry methods on the given chare object. With no pending AEMs, there is no risk of an accelerator device update the state of the chare object during the checkpointing process itself. Note that this does not change the expense of checkpoint, it simply changes when checkpoint can occur. If a fault tolerance mechanism were to require the deletion of objects on pairings that have not failed as a result of a fault on one of the pairings, like the memory checkpoint mechanism does, then the abandonment mechanism currently used by the accelerator manager would have be updated to act on an object-by-object basis rather than batches. Essentially, the requires a single abandonment flag to become an array of abandonment flags (as part of the batch's data structure) and a check be added within the generated callback that checked each of these flags individually as it processes the output of the batch. Persistent data would be handled in the same manner, being disregarded after all abandoned objects have completed, rather than all abandoned batches have been completed.

Note, at the time of this writing, the support for the memory checkpointing mechanism had not been back ported to the generated code for the Cell processor (i.e. is only supported for GPGPUs). Since the popularity of the Cell continues to decline, such support may or may not be implemented in the future. However, if such support were to be implemented, it would be a straight forward extension of the current implementation for GPGPUs (i.e. batches of size one and/or on an object-by-object basis).

6.5 Summary of Data Management

In this chapter, we have discussed the various ways in which the runtime system can assist the programming model and the application programmer by handling various issues related to managing an application's data and/or state. First, the automatic manipulation of parameters passed through an entry method's passed parameter list was demonstrated (the discussion of the mechanism was in section 6.1, while the demonstration was in section 4.3.3). While such corrections are mundane, they are a source of programming errors if the programmer is forced to explicitly handle architecture differences. Since automating these correction via a runtime system is fairly straight forward, such automation can benefit the programmer by simplifying the code (removing explicit correction code) and removing a source of programming errors. Second, the data portion of the accelerated entry method batching process was discussed. By coalescing scalar values during the batching process, the runtime system attempts to avoid shared memory bank conflicts when targeting GPGPUs. Additionally, separate AEM invocations that reference the same data via their shared local parameters can utilize a single buffer on the accelerator device to reduce the overall footprint of an AEM batch within the device's memory. The runtime system automates this sharing of data across AEM invocations via pointer lookups, leaving the programmer to simply mark the appropriate local parameters with the "shared" keyword. Third, the use of persistent local parameters was also discussed. Again, local parameters marked with the "persist" keyword, which causes the runtime system to retain the local parameter's data on the device when appropriate, eliminating data transfers to and from the device between invocations. Within this context, static versus dynamic assignment of chare objects to either the host core or the accelerator device as also discussed. Finally, support for the memory checkpoint fault tolerance mechanism was also discussed in this chapter. Support for memory checkpointing within the accelerator manager enables the runtime system to extend its preexisting memory checkpoint mechanism to include applications that make use of accelerator devices.

Chapter 7

Related Work

In this chapter, we will discuss how other approaches have addressed the difficulties of heterogeneous systems. Along the way, we will compare and contrast our approach with these other approaches.

7.1 *Native* Programming Models

A natural starting place to begin is to mention the programming models that were/are co-developed by the creators of the accelerator technologies that have been discussed within the context of our work. When the Cell processor was introduced, the Cell SDK was also introduced [23], which includes various libraries, intrinsics, and interfaces that allow programmers to write application for the Cell processor. When using the Cell SDK directly, the programmer is required to explicitly perform many of the tasks associated with executing computation on the SPEs, such as initiating direct memory accesses (DMAs) and managing each SPE's local store memory.

In the context of the CUDA-based GPGPUs, the most notable programming model in use today is the compute unified device architecture (CUDA) programming model [80]. We have already eluded to how the CUDA programming model works throughout this document, particularly in section 2.2 (i.e. using kernels, warps, many threads, the SIMD execution model, and so on). As is the case with the Cell SDK, when using CUDA directly, the programmer is responsible for explicitly moving data, determining how the resources will be shared (i.e. determining and optimizing their code to maximize occupancy), and so on.

When comparing our approach to both the Cell SDK and the CUDA approaches, there are several differences. First, our approach has been used to program both the Cell processor and GPGPUs, which cannot be said for either the Cell SDK or CUDA. Though, it is certainly not difficult to imagine mapping CUDA to the Cell processor. Further, our approach creates a unified programming model, allowing a single expression of the application code (i.e. the accelerated entry methods) to be portable between core types, enabling features such as automatic load balancing between host cores and accelerator devices. We also address concerns such as fault tolerance.

Because of it is similar to CUDA in many ways, we will include the OpenCL programming model in this section as well. In the OpenCL programming model [40], a programmer issues kernels (collected into "program" constructs) made up of work-items which are collected into sets or *workgroups*. Synchronization is only possible between work-items within the same workgroup. Kernels have to complete in order achieve synchronization across the global set of work-items. This is quite similar to the CUDA programming model's use of streams, kernels, warps, and threads. In particular, the programmer creates many relatively small work-items that can be executed in parallel during any given kernel execution. Additionally, OpenCL supports separate types of memory that can be used explicitly by programmers (private to work-items, shared between work-items in the same workgroup, global, constant, etc.). Unlike CUDA, the OpenCL programming model has been generalized so that is applicable to different types of processing devices, such as GPGPUs, Cell processors, and commodity host cores. However, in the presence of multiple devices, a separate *command queue* is used explicitly by the programmer to issue work to each of the devices, leaving the programmer to decide where portions of the application should be executed. There are several other features included within OpenCL, but we will limit our discussion of OpenCL to these features for brevity. In many ways, the difference between CUDA and our approach are the same as the differences between OpenCL and our approach (as listed above).

7.2 Extensions to OpenMP

While our approach focuses on runtime system support for heterogeneous systems, there are several other approaches based on compile-time techniques extending OpenMP [32]. There are obvious differences between compiler-based approaches and our work, which focuses on the role of the runtime system. While we do make use of a source-to-source compiler in our work for code generation purposes (i.e. charmxi), we do not implement many of the more sophisticated compile-time techniques discussed in this chapter.

The XLC single source compiler, or "Octopiler," [35, 34] optimizes code written using OpenMP *pragma*s by performing several compile-time optimizations to the application code, including automatic SIMDization of application code, branch optimizations, and others. Additionally, a shared memory model is presented to the programmer through the use of a software cache. DMA transactions generated by the compiler assist in moving data to and from the SPEs' local stores (i.e. cache misses). As stated above (in a general way), this approach has the benefit of being able to apply sophisticated compile-time techniques to assist programmers in making used of the Cell processor. However, our approach differs in that Charm++ has a message queue which is essentially a list of ready-to-execute tasks. This, combined with the encapsulated nature of entry

methods, allows the runtime system to pro-actively move all of the data that will be accessed by the task executing on the SPE. Software caches, however, are reactive, only starting a DMA transaction after the data is already required. Certainly, the Octopiler can included software prefetches as easily as it inserts the DMA transactions themselves (of course, with the same limitations that all prefetch mechanisms encounter).

OpenMP to GPGPU [67] is another technique that allows programmers to write their application code using OpenMP *pragma*s. At compile time, the compiler analyzes the OpenMP portions of the application, mapping the loop iterations to GPGPU threads. This is done by analyzing the code to detect implicit or explicit synchronization points within the OpenMP constructs (e.g. parallel loops). Using the synchronization points as dividing points, the OpenMP portions of the application are broken down into kernel regions, with each kernel region mapping to a single kernel execution to enforce global synchronization at the required points.

Our approach could certainly benefit from several of the techniques used by the compiler techniques mentioned above, particularly in the case of splittable accelerated entry methods and removing the need to use SIMDIA explicitly within application code (see section 8.3). However, our approach goes beyond the compiler-based methods mentioned in this section in that our approach targets clusters rather than single nodes (i.e. we use the same programming model for all cores, not requiring a second programming model for across node communication), apply dynamic load balancing techniques, integrate with fault tolerance support, and so on.

Of course, all OpenMP-based approaches influence programmers in the same way that OpenMP influences programmers. Another high-level point is that the use of concurrent objects within the Charm++ programming model essentially forces programmers to think about their applications in a parallel manner. On the other hand, OpenMP encourages programmers to start with a serial implementation of their application and to parallelize portions of it, piece-by-piece. In this sense, one should also consider the influence a programming model has on a programmer in terms of how the model forces the programmer to envision their application code. While is is quite difficult to quantify such influences, OpenMP in particular almost begs programmers to fall victim to Amdahl's Law [1] by having programmers structure their applications based around a single sequential *master* thread. Of course, the same parallel structure possible in any programming model, but the question one should ask themselves is 'how far out of their way does a programmer have to go to implement such a structure within a given parallel programming model?'

7.3 Extensions to MPI

Having discussed approaches based on OpenMP in the previous section, we now move on to talk about approaches based on MPI. The message passing interface (MPI) [37] is well-known and commonly used parallel programming model. As such, there have been several approaches which build upon MPI as the basis of their work.

Open MPI [39] is one such approach. Like our approach, Open MPI will automatically modify application data passing between processing elements over the network. The main advantage of Open MPI over our approach is that Open MPI provides support for network heterogeneity, allowing multiple types of interconnects to be used within a single parallel execution. However, Open MPI does not address heterogeneity between nodes (beyond data encoding differences such as endianness, size of data primitives, and so on), whereas our approach can make use of a mixture of processing element types within a single execution.

The HeteroMPI [66] approach is also based on MPI. Like Open MPI, HeteroMPI goes beyond our approach in that it supports heterogeneous networks, allowing communication within an application to pass over multiple types of interconnects. HeteroMPI also supports a heterogeneous collection of processing elements. However, a benchmark code must be created by the programmer, which is used to predict the relative performances of the application for each phase of the application's execution (HMPI_Recon). Further, the use of HMPI_Recon is done explicitly within the application code. In our approach, we directly measure the performance of the application as it executes, without requiring the programmer to include any code. HeteroMPI, to the best of our knowledge, has only been demonstrated to execute on heterogeneous clusters with varying host cores. Our approach goes a step further in that we also support heterogeneous clusters that include accelerator devices.

MPI microtasks [81] extends MPI to allow for the execution of MPI microtasks on the SPEs of a Cell processor. By using the same (or at least very similar) API calls, the microtasks are able to communicate with one another and the PPE. On the host core itself, the programmer uses standard MPI calls for communicating between PPEs (which is to say, the MPI microtask does not address it directly, but so closely resembles MPI that it all but eliminates the mental effort required by a programmer to 'use another API' for across host core communication).

Our approach further differs from the above MPI-based approaches in the same ways MPI and Charm++ differ from one another. For example, MPI is centered around the used of long-lived MPI threads, while Charm++ makes use of chare objects (long or short lived) and entry methods (tasks) that execute on those chare objects. Further, Charm++ naturally allows for dynamic load balancing, fault tolerance support, and other features due to the migratable nature of chare objects and through the assistance of an active runtime

system. Though, it is worth noting that AMPI [46, 54] has shown that MPI applications can also make use of the techniques commonly available to Charm++ applications.

7.4 Other Approaches

Several abstractions that have been created to ease the burden of programming accelerator technologies do so by creating libraries and frameworks that are focused on one or more problems types or computation patterns. One such approach is Mercury's Multicore Framework (MFC) [16], which allows a programmer to initiate a N-dimensional matrix computation on the accelerator device from the host core. Another framework is RapidMind's Development Platform [72], which allows a programmer to create collections of values which are then passed into and out of program constructs. To be clear, "program constructs" are not programs in the sense of applications, but rather well-defined task definitions that are applied to collections of values. Code examples that we have seen are limited to dense data structures such as arrays and matrices, though there is no reason to believe that the approach is limited in this way (conceptually speaking).

Sequoia [36] is another interesting approach in that it focuses on the idea of a code expression that allows for the granularity of the tasks to be specified separately from the code itself. In Sequoia, a programmer expresses a calculation as a recursively divisible operation. This is done by creating both "inner" and "leaf" steps to the calculation, which one can conceptually think of as forming a tree structure representing portions of the overall calculation. At each "inner" node in the tree, one or more *tunable* variables are used to recursively divide the work based on a specification (i.e. a representation the specific hardware being targeted). Using the specification and the tunables, Sequoia breaks the overall workload into portions that map to the various memory levels within the hardware. For example, the levels may correspond to L1 and L2 caches on a commodity core, the local stores within SPEs on a Cell processor, shared memory nodes within a cluster, and so on. The overall idea being to allow the granularity of sub-tasks within a given computation to not be defined by the expression of the computation, but rather to express the computation in a manner that allows the granularity to be adjust based on the circumstances under which an application is executing. Our approach addresses granularity adjustments at runtime through the use of splittable accelerated entry methods. Additionally, in this work we have applied our approach to dynamically load balance a given workload across a heterogeneous set of cores (versus a static specification) allowing it to adapt to interference, included support for SIMD instructions, and included integration for fault tolerance support.

There have also been several programming models explicitly created with the goal of targeting hetero-

geneous systems using task-based approaches, similar to the entry method (task) approach that we used in our work. StarSs [85] is a particularly flexible approach in that it supports a variety of hardware platforms by creating several individual runtime libraries, including the Cell processor [11, 10] (CellSs), GPGPUs [5] (GPUSs), SMP processors [83] (SMPSs), OpenMP [2] (OmpSs), FPGAs [6], and others. Our approach does not currently support FPGAs or allow for an extended version of OpenMP to be used to express application code. In the StarSs approach, the programmer marks functions within their application code as tasks. These tasks are arranged a hierarchy, requiring children tasks to complete before parent tasks are able to complete. This is unlike our approach where there is implicit or explicit hierarchy associated with the entry methods (tasks). As such, our approach does not introduce any implicit synchronization between tasks. StarPU [3, 4] and the Hybrid Multi-core Parallel Programming (HMPP) environment [33] are additional task-based approaches, which both make use of tasks which they refer to as *codelets*. In StarPU, the programmer is required to write a version of the codelets for each type of processing element, while in HMPP, the codelets are portable. In HMPP, however, there is no direct support for across host core communication, requiring programmers to make use of another programming model, such as MPI, for that purpose. In our approach, the programmer also effectively *marks* which functions (entry methods) can be offloaded to an available accelerator devices by making used of accelerated entry methods rather than standard entry methods. Our approach goes beyond all of these approaches in that we further provide integration with the Charm++ automatic memory checkpoint scheme so that it can be used on systems that include accelerators as well. Finally, we introduce the SIMDIA abstraction, allowing programmers to make used of SIMD instruction extensions within the AEMs (tasks) in a portable manner.

7.5 Previous Work Within Charm++

In addition to comparing and contrasting our approach with that of other approaches that are not based on Charm++, we also would like to point out previous work that has been based on Charm++ also. First, within Charm++ it is possible to make use of both the Offload API [61] for the Cell processor and the GPU Manager [93, 55] (previously known as the Hybrid API) directly from within Charm++ application code. In both cases, the programmer is required to write one or more device-specific functions they wish to offload to the devices and explicitly *invoke* those offloaded functions via work requests. The work presented here draws upon this initial work (see chapter 2), extending it to the approach we have presented in this document.

Additionally, there has been previous work within our group that explores agglomeration techniques for

GPGPU devices [68]. Again, the programmer is required to explicitly include code for each type of device. However, an accelerator FIFO is used to agglomerate units of work together into larger sets, which are then issued to an available GPGPU as a single kernel. A scheduler is used to select work units from a pool of work units based on each work unit's priority and then directs work units to either the host cores or the accelerator FIFO for later execution on a GPGPU. Unlike the agglomeration work, the approach presented in this document makes use of a unified programming model (i.e. no explicit kernel code is written), integrates with other mechanisms within Charm++ (e.g. the load balancing framework and automatic memory checkpointing mechanism), includes support for the Cell processor, and so on.

Saletore et al. extended CHARM (a predecessor to Charm++) in order to support heterogeneous clusters [89]. In particular, they demonstrated the execution of CHARM using a mixture of Sun Sparc, HP-PA, and IBM RS/6000 workstations using dynamic load balancing techniques. However, it is worth noting that the extent to which these workstations differ is mainly limited to processing power. We have demonstrated our approach on a heterogeneous cluster that includes peak flop rate and architecture differences between host cores, along with support for accelerator devices. Further, our approach includes support for SIMD instructions through SIMDIA, integration with at least one fault tolerance mechanism, and so on.

Chapter 8

Conclusions

In this work, we have extended the Charm++ programming model and runtime system to support accelerator devices and heterogeneous clusters. This chapter presents a summary of the document, some concluding remarks, a discussion on future work, and some thoughts on the future of heterogeneity in computing.

8.1 Summary of Content

Chapter 1 introduces the reader to the context and motivations surrounding this work. Chapter 2 moves on to provide background information to familiarize the reader with the accelerator devices used in this work, along with various aspects of the Charm++ programming model and runtime system that are relevant to the work presented here.

Chapter 3 presents the general approach used in this work. The Charm++ programming model and runtime system were used as the basis of this work. The Charm++ programming model has been extended to include accelerated entry methods (AEMs) and the SIMD instruction abstraction (SIMDIA). With these extensions, we create a unified programming model that can be used to write application code that targets both host cores and accelerator devices alike. Additionally, the runtime system has been modified to include the accelerator manager, which assists in executing the accelerated entry methods on either the host core or an available accelerator device. The accelerator manager helps to facilitate the execution of Charm++ applications making use of AEMs by providing several services to the application, such as dynamic load balancing between the host core and accelerator device and integration with fault tolerance mechanisms.

Chapter 4 presents previous experiments involving a simple molecular dynamics (MD) application that uses our programming model extensions to execute on a heterogeneous cluster. The heterogeneous cluster is composed of four Playstation 3s (PS3s) nodes, four QS20 Cell blades (blades) nodes, and an x86-based node. We demonstrate the MD application executing on the heterogeneous cluster and making use of a static load balancer to divide the workload across the various nodes. When executing using all the nodes, the application is making use of three types of cores (PPEs, SPEs, and x86s), three types of SIMD instruction

extensions, and two types of memory structures (cache hierarchies and scratchpad memories requiring DMA transactions). Despite all of the architectural differences between the cores, the application does not include any architecture specific code within the application code itself. Further, the application code remains portable to traditional (homogeneous) clusters that do not contain any accelerator cores (i.e. the SPEs, in this case).

Chapter 5 moves on to discuss our recent efforts to support general purpose graphical processing units (GPGPUs) in addition to the Cell processor using the same programming model extensions presented in chapter 3. Both a 5-point stencil application and an updated version of the MD application used within chapter 4 are used to demonstrate Charm++ applications written using our programming model extensions executing on a combination of x86-based host cores and GPGPU accelerator devices. Dynamic load balancing strategies are used to balance the overall workload between the host cores and the accelerator devices. Further, the same load balancing strategies are also used by the MD application to demonstrate adaptability to the *interference* of the other applications sharing the same accelerator resources. Additionally, we briefly discuss the updated MD application executing on the heterogeneous cluster introduced in chapter 4 to demonstrate the use of our dynamic load balancing strategies on a cluster including Cell processors.

Chapter 6 discusses various data management services provided by the runtime system, including automatic manipulation of application data, shared and persistent local parameters, and the integration of the accelerator manager with the memory checkpointing fault tolerance scheme within the Charm++ runtime system. The automatic manipulation of application handles some of the mundane tasks related to applications executing on heterogeneous hardware, such as correcting for endianness differences between cores with different architectures. While such corrections are indeed mundane in nature, requiring the programmer to handle these differences, the correction of which can be automated by an underlying runtime system, serves as a possible source of programming errors. Further, requiring the programmer to explicitly handle such differences does not serve the general goal of the application beyond simply allowing the application to execute on a heterogeneous system. The use of various modifiers to the local parameters associated with accelerated entry methods allows the runtime system to dynamically reduce the total memory associated with a batch of AEMs executing on a GPGPU device (i.e. shared local parameters), along with determining where data should reside, in system memory or on the accelerator device (i.e. persistent local parameters). Finally, some of the automated mechanisms within the Charm++ runtime system, such as fault tolerance schemes, require additional support from the accelerator manager in order to correctly operate within the context of heterogeneous systems. We demonstrated one such approach by allowing the memory checkpointing mechanism within the Charm++ runtime system to interact with the accelerator manager.

Finally, chapter 7 compares and contrasts our approach with various other approaches. Refer to chapter 7 for specific comparisons.

8.2 Concluding Remarks

At the highest level, this work explores the application of a message-driven object-based parallel programming model to heterogeneous systems including accelerator technologies. In this work, the Charm++ programming model has been applied to clusters including Cell processors or CUDA-based GPGPUs. Typically, programming models that are highly data parallel in nature are used for GPGPU programming, while streaming programming models are used for the Cell processor. In this work, we have demonstrated that it is in fact possible to target a variety of fairly different hardware architectures (i.e. commodity host cores, Cell processors, and GPGPUs) using a single, flexible programming model (i.e. Charm++ with the extensions presented in this work). Through the use of this *unified programming model*, we enable programmers to focus on the overall goals and purposes of their applications, rather than becoming over-burdenned by architecture-specific details presented by the hardware their applications will execute upon.

At the core of our approach is an intelligent, adaptive, and active runtime system. The runtime system is *intelligent* in that it uses information collected at runtime make decisions and apply heuristics. The runtime system is *adaptive* in that it is able to react to changes within the environment in which an application is executing. The runtime system is *active* in that it does not simply fulfill requests made by the programmer via API calls, but rather proactively performs various actions when given permission to do so, such as improving the performance of applications via load balancing strategies, protecting applications against hardware faults, maps appropriate portions of an application's workload to available accelerator devices, modifies application data to correct for architecture differences in heterogeneous systems, reduces communication demands through shared and persistent buffers, and so on. Generally speaking, the runtime system maps the message-driven programming model to the specific hardware on which the message-driven application is executing. The overall runtime system can be broken down into a set of sub-components, including the accelerator manager which was introduced as part of this work. The accelerator manager is responsible for a variety of tasks related to the execution of accelerated entry methods, including dynamic load balancing between host cores and accelerator devices, batching AEMs, coalescing of shared data related to AEMs, management of persistent data related to AEMs, splitting splittable AEMs into smaller subtasks when appropriate, fault tolerance support as it relates to accelerators, and so on. With the runtime system actively performing a variety of tasks for the application, we begin to see in what ways a runtime system

can assist programmers in creating applications that target heterogeneous systems. In particular, when we say "assist programmers" we mainly mean allowing programmers to focus on the goal or intent of their applications rather than the mechanics of their application executing on specific hardware.

One aspect of the mechanics of an application executing is the process of balancing the application's workload across the available processing cores, host and accelerator alike. Dynamic load balancing between the host cores and accelerator cores is achieved by taking various runtime measurements of the application and rebalancing the workload in response to those measurements via the various load balancing strategies discussed in chapter 5. However, there are challenges and pitfalls related to automating the load balancing process as it relates to heterogeneous hardware. As part of our static load balancing experiments, we showed that the communication characteristics of the various nodes is an important consideration when balancing a workload within the context of a heterogeneous cluster. In fact, the presence of nodes with better communication characteristics can help applications scale better if the communication heavy portions of the workload can be focused on hardware with good communication characteristics, even if that same hardware offers very little additional benefit in terms of the combined peak flop rate. Other pitfalls arise out of having host cores and accelerator devices with significantly different peak flop rates. In such a situation, automatic load balancing strategy runs the risk of moving too much work in the direction of the lower performing core, which may have drastic performance implications if even a small amount of work is shifted. Dynamic and automatic load balancing strategies must be cautions not to accidentally degrade the performance of applications for a significant period of time when seeking an optimal balance point for the application's workload. Yet other pitfalls arise out of the actions of the runtime system itself. As the runtime system performs tasks on an application's behalf, it inevitably affects the behavior of the application itself. These effects can, in turn, affect the measurements take by the runtime system, resulting in cascading effects and/or feedback cycles that reinforce themselves if proper care is not taken in the implementation of the runtime system.

Further, we have shown that our approach is capable of integrating with other features provided by runtime systems in order to assist applications and programmers in coping with a changing and imperfect computing environment. We have integrated our approach with at least one fault tolerance mechanism (automatic memory checkpointing). If a host core fails, the runtime system is able to use the memory checkpointing mechanism to automatically role the state of the application back to a previous *known-to-be-good* state and restart execution from that state without requiring user intervention. Future integration with other fault tolerance mechanisms will provide support for more than a single host core failure. Additionally, we have shown that our approach is capable of adapting to interference caused by shared accelerator resources.

This support is a natural result of the dynamic load balancing mechanisms included as part of this work.

8.3 Future Work

Our attempt to create a unified programming model is somewhat limited in its current form. It is true that our approach enables applications to be portable between and concurrently make use of various types of processing elements. However, our approach also presents the programmer with a minimal set of features for creating such applications. For example, our approach does not include a compiler to perform complex architecture-specific optimizations such as automatically analyzing and transforming standard entry method into accelerated entry methods, including triggered and/or splittable AEMs. While we consider such optimizations as future work, it should also be noted there is an ongoing effort to create a Java-like programming language called Charj that embodies the Charm++ programming model [8]. Charj represents a possible future approach to creating a "front-end" for the various abstractions and runtime system features presented in this work, which would enable a compiler to provide additional analysis and optimizations during the compliation process. Without such "front-end" analysis and optimizations, the programmer is still limited in that they must choose a single representation of their application code. Such as a particular AEM being splittable or not splittable, as discussed in section 5.7.1. The creation of such a "front-end" is left as future work.

Additionally, there remains several opportunities to improve upon our existing runtime system support. One such improvement is the reduction of overhead. While we have strived to create a lean implementation, we are continually thinking of ways to improve the implementation. As an example, we are revisiting some of the dynamic data structures and implementing pooling techniques to minimize memory allocations and deallocations at runtime, finding ways to use asynchronous data transfers for persistent data whenever possible, and so on. Since Charm++ is used in production-level scientific codes, a long term goal of this work is to continually improve our implementation so that future applications will receive as much benefit from this work as possible. This also includes extending support to additional accelerator technologies, such as MIC.

There is also additional future work in providing greater integration with other components within the Charm++ runtime system. As part of this work, the accelerator manager has been integrated with the existing load balancing framework and the automatic memory checkpointing mechanism. However, work remains to the increase support for other components. For example, other fault tolerance mechanisms such as memory logging [75] or the dynamic resizing feature [60] (allowing Charm++ applications to dynamically

	N11	J11	N10	J10	N09	J09	N08	J08
Cell		3	3	6	6	4	7	3
GPGPU (nVidia)	4	2	4	2				
GPGPU (ATI)	1	2	1		1			
GRAPE-DR			1		1	1		
BlueGene/Q	5	2	1					
BlueGene/P					6	8	7	8
Accelerator	5	7	9	8	8	5	7	3
BlueGene	5	2	1		6	8	7	8
other		1	1	2				
Total Systems	10	10	11	10	14	13	14	11

Table 8.1: The top 10 systems on the Green500 list by year [21].

change the number of processors they are using as other applications start and finish in order to maximize cluster utilization) will require additional support within the accelerator manager to provide full support for these features in the context of heterogeneous systems.

8.4 The Future of Heterogeneity

Finally, it is worth taking a step back and discussing the future of heterogeneous systems in general, not just in the context of our work. Table 8.1 is an expanded version of table 1.1 originally presented in section 1.2.1. In addition to table 8.1, figure 8.1 presents four graphs showing data for the top 50 systems in the two most recent lists. Since energy consumption is quickly becoming a main concern in the design of large supercomputers and data centers, we place our focus on the Green500 list, as opposed to the Top500 list.

From table 8.1, we can see that many of the top 10 systems in the world in terms of Mflop/s per Watt are heterogeneous systems. However, the table has been expanded to include the BlueGene [38] systems. BlueGene systems use a greater number of weaker processors, along with a reduced software stack on the compute nodes themselves. BlueGene systems have certainly had an significant presence on the list of green supercomputing systems over the last four years. Further, the 5 BlueGene\Q systems in the November 2011 Green500 list are *the top 5 systems*. Further, these systems hold ranks 17, 29, 64, 65, and 284 on the Top500 list, showing that they are also amongst some of the most powerful systems in use today. From the upper right graph in figure 8.1, we can see that scaling these systems to higher core counts does not seem to have a large impact on their overall power efficiency, meaning that it is likely that even larger BlueGene\Q systems will appear amongst the highest ranked systems on the Top500 in the very near future. Further, figure 8.1 show how the heterogeneous systems compare to non-heterogeneous systems for the top 50 systems on the Green500 list. These figures call into question the idea that heterogeneity will increase the power efficiency of large clusters. Additionally, from the lower graphs in figure 8.1 there is also a trend of heterogeneous clusters

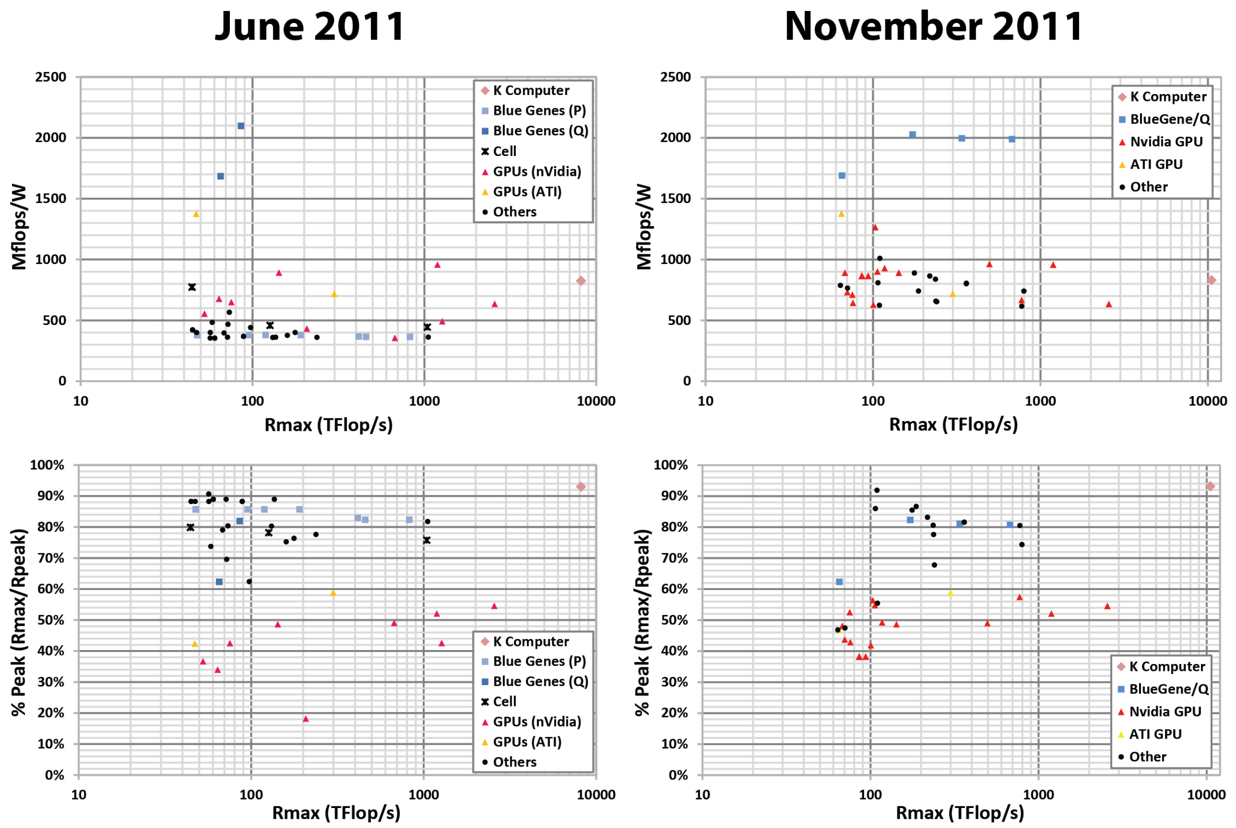


Figure 8.1: The top 50 systems from the June 2011 and November 2011 Green500 lists.

achieving a lower percentage of their peak flop rates, specifically amongst clusters that include GPGPUs. This fact could be a hint that clusters including GPGPUs are more difficult to program relatively to other types of clusters. It could also suggest the presence of design imbalances within the clusters, such as a cluster's ability to perform communication as fast as it can perform computation. Considering all of these ideas together, it is not unlikely that system designs similar to BlueGene systems may increase in popularity in the coming years. On the other hand, it is also entirely likely that heterogeneous systems may increase in popularity, especially if the hurdles related to programming such systems can be overcome. As it stands now (November 2011 list), there are 39 systems that include accelerators and 16 BlueGene systems (5 'Q's, 4 'L's, 7 'P's), meaning that heterogeneous systems currently outnumber BlueGene systems by a factor of almost 2.5.

However, the above discussion centers around the largest clusters on Earth. There is also the question of smaller, locally operated clusters that include accelerators. For smaller systems, power concerns are not as critical. Rather, performance levels tend to be a prime concern. In these situations, accelerators provide great benefit.

With these points in mind, the future of heterogeneous clusters is unclear, particularly for large clusters. At the very least, we can say that heterogeneous clusters are likely to be around for several years to come.

References

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] Michael Andersch, Chi Ching Chi, and Ben Juurlink. Programming parallel embedded and consumer applications in openmp superscalar. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 281–282, New York, NY, USA, 2012. ACM.

[3] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Maik Nijhuis. Exploiting the Cell/BE architecture with the StarPU unified runtime system. In *SAMOS Workshop - International Workshop on Systems, Architectures, Modeling, and Simulation*, Lecture Notes in Computer Science, Samos, Greece, July 2009. To appear.

[4] Cdric Augonnet and Raymond Namyst. A unified runtime system for heterogeneous multi-core architectures. In Eduardo Csar, Michael Alexander, Achim Streit, Jesper Trff, Christophe Crin, Andreas Knpfer, Dieter Kranzlmller, and Shantenu Jha, editors, *Euro-Par 2008 Workshops - Parallel Processing*, volume 5415 of *Lecture Notes in Computer Science*, pages 174–183. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-00955-6_22.

[5] Eduard Ayguade, Rosa M. Badia, Francisco D. Igual, Jesus Labarta, Rafael Mayo, and Enrique S. Quintana-Orti. An extension of the StarSs programming model for platforms with multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference, Lecture Notes in Computer Science*, Lecture Notes in Computer Science, Delft, The Netherlands, Aug. 2009. To appear.

[6] Eduard Ayguad, Rosa Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc Gonzlez, Francisco Igual, Daniel Jimnez-Gonzlez, Jess Labarta, Luis Martinell, Xavier Martorell, Rafael Mayo, Josep Prez, Judit Planas, and Enrique Quintana-Ort. Extending openmp to survive the heterogeneous multi-core era. *International Journal of Parallel Programming*, 38:440–459, 2010. 10.1007/s10766-010-0135-4.

[7] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[8] Aaron Becker. *Compiler Support for Productive Message-Driven Parallel Programming.* PhD thesis, Dept. of Computer Science, University of Illinois, 2012. (to appear).

[9] Aaron Becker, Gengbin Zheng, and Laxmikant Kale. Distributed Memory Load Balancing. In D. Padua, editor, *Encyclopedia of Parallel Computing (to appear)*. Springer Verlag, 2011.

[10] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[11] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Exploiting locality on the cell/b.e. through bypassing. In *SAMOS '09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 318–328, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] Abhinav Bhatele. Topology Aware Task Mapping. In D. Padua, editor, *Encyclopedia of Parallel Computing (to appear)*. Springer Verlag, 2011.

[13] Eric Bohm, Abhinav Bhatele, Laxmikant V. Kale, Mark E. Tuckerman, Sameer Kumar, John A. Gunnels, and Glenn J. Martyna. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems*, 52(1/2):159–174, 2008.

[14] Eric Bohm, Sayantan Chakravorty, Pritish Jetley, Abhinav Bhatele, and Laxmikant V. Kale. CkDirect: Unsynchronized One-Sided Communication in a Message-Driven Paradigm . In *Proceedings of International Workshop on Parallel Programming Models and Systems Software for High-End Computing (P2S2)*, August 2009.

[15] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[16] Brian Bouzas, Robert Cooper, Jon Greene, Michael Pepe, and Myra Jean Prelle. MultiCore Framework: An API for Programming Heterogeneous Multicore Processors. Mercury Computer System's Literature Library (http://www.mc.com/mediacenter/litlibrarylist.aspx).

[17] A. Branover, D. Foley, and M. Steinman. Amd fusion apu: Llano. *Micro, IEEE*, 32(2):28 –37, march-april 2012.

[18] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, August 2004.

[19] Alfredo Buttari, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, and George Bosilca. Scop3: A rough guide to scientific computing on the playstation 3, May 2007.

[20] R. K. Cavin, P. Lugli, and V. V. Zhirnov. Science and engineering beyond moore's law. *Proceedings of the IEEE*, 100(Special Centennial Issue):1720 –1749, 13 2012.

[21] Wu chun Feng and Kirk W. Cameron. The green500 list - november 2011, November 2011.

[22] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the intel scc many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525 –532, july 2011.

[23] International Business Machines Corp. Ibm techlib: Cell broadband engine, 2005.

[24] Intel Corporation. Intel avx.

[25] Intel Corporation. Many integrated core (mic) architecture - advanced.

[26] Intel Corporation. Scc external architecture specification (eas), revision 0.94, May 2010.

[27] International Business Machines Corporation. Ibm bladecenter qs20 blade with new cell be processor offers unique capabilities for graphic-intensive, numeric applications. `http://www-01.ibm.com/common/ssi/rep_ca/7/897/ENUS106-677/ENUS106-677.P%DF`, September 2006.

[28] NVIDIA Corporation. Nvidia's next generation cuda compute architecture: Fermi, 2009.

[29] NVIDIA Corporation. Tesla m2050/m2070 gpu computing module: Supercomputing at 1/10th the cost, April 2010.

[30] Cray. Cray xk6 specifications.

[31] M. Daga, A.M. Aji, and Wu chun Feng. On the efficacy of a fused cpu+gpu processor (or apu) for parallel computing. In *Application Accelerators in High-Performance Computing (SAAHPC), 2011 Symposium on*, pages 141 –149, july 2011.

[32] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Computational Science & Engineering*, 5(1), January-March 1998.

[33] Romain Dolbeau, Stephane Bihan, and Francois Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

[34] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, 45(1):59 –84, 2006.

[35] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for the cell processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[36] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[37] Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[38] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49(2.3):195 –212, march 2005.

[39] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

[40] Khronos OpenCL Working Group. The opencl specification, version 1.2, November 2011.

[41] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *Micro, IEEE*, 26(2):10 –24, march-april 2006.

[42] Hiroo Hayashi. Spursengine: A high-performance stream processor derived from cell/b.e. for media processing acceleration, August 2008.

[43] A. Heinecke, M. Klemm, and H. Bungartz. From gpgpu to many-core: Nvidia fermi and intel many integrated core architecture. *Computing in Science Engineering*, 14(2):78 –83, march-april 2012.

[44] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108 –109, feb. 2010.

[45] Chao Huang and Laxmikant V. Kale. Charisma: Orchestrating migratable parallel objects. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2007.

[46] Chao Huang, Gengbin Zheng, and Laxmikant V. Kalé. Supporting adaptivity in mpi for dynamic parallel applications. Technical Report 07-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[47] National Supercomputing Center in Tianjun. Tianhe-1 pflop supercomputer, 2009.

[48] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[49] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, 2010. IEEE Computer Society.

[50] D.R. Johnson, M.R. Johnson, J.H. Kelm, W. Tuohy, S.S. Lumetta, and S.J. Patel. Rigel: A 1,024-core single-chip accelerator architecture. *Micro, IEEE*, 31(4):30 –41, july-aug. 2011.

[51] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell Processor. *IBM Journal of Research and Development: POWER5 and Packaging*, 49(4/5):589–604, July 2005.

[52] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[53] Laxmikant V. Kale, Abhinav Bhatele, Eric J. Bohm, and James C. Phillips. NAnoscale Molecular Dynamics (NAMD). In D. Padua, editor, *Encyclopedia of Parallel Computing (to appear)*. Springer Verlag, 2011.

[54] Laxmikant V. Kale, Eric Bohm, Celso L. Mendes, Terry Wilmarth, and Gengbin Zheng. Programming Petascale Applications with Charm++ and AMPI. In D. Bader, editor, *Petascale Computing: Algorithms and Applications*, pages 421–441. Chapman & Hall / CRC Press, 2008.

[55] Laxmikant V. Kalé, David M. Kunzman, and Lukasz Wesolowski. Accelerator Support in the Charm++ Programming Model. In Jakub Kurzak, David A. Bader, and Jack Dongarra, editors, *Scientific Computing with Multicore and Accelerators*. CRC Press (Taylor and Francis Group), December 2010.

[56] Laxmikant V. Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, pages 347–358, February 2006.

[57] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton. Programming the linpack benchmark for roadrunner. *IBM Journal of Research and Development*, 53(5):9:1 –9:11, sept. 2009.

[58] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26(3):10 –23, may-june 2006.

[59] Michael Kistler, John Gunnels, Daniel Brokenshire, and Brad Benton. Petascale computing with accelerators. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 241–250, New York, NY, USA, 2009. ACM.

[60] Sameer Kumar. An adaptive job scheduler for timeshared parallel machines. Master's thesis, Dept. of Computer Science, University of Illinois, 2001. http://charm.cs.uiuc.edu/papers/AdaptiveJobThesis01.html.

[61] David Kunzman. Charm++ on the Cell Processor. Master's thesis, Deptartment of Computer Science, University of Illinois at Urbana-Champaign, 2006. http://charm.cs.uiuc.edu/papers/KunzmanMSThesis06.shtml.

[62] David Kunzman, Gengbin Zheng, Eric Bohm, and Laxmikant V. Kalé. Charm++, Offload API, and the Cell Processor. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism*, Seattle, WA, USA, September 2006.

[63] David M. Kunzman and Laxmikant V. Kalé. Towards a framework for abstracting accelerators in parallel applications: experience with cell. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[64] David M. Kunzman and Laxmikant V. Kalé. Programming Heterogeneous Clusters with Accelerators using Object-Based Programming. *Journal of Scientific Programming*, 19(1):47–62, 2011.

[65] J. Kurzak, A. Buttari, P. Luszczek, and J. Dongarra. The playstation 3 for high-performance scientific computing. *Computing in Science Engineering*, 10(3):84 –87, may-june 2008.

[66] Alexey Lastovetsky and Ravi Reddy. Heterompi: towards a message-passing library for heterogeneous networks of computers. *J. Parallel Distrib. Comput.*, 66(2):197–220, 2006.

[67] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: a compiler framework for automatic translation and optimization. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '09, pages 101–110, New York, NY, USA, 2009. ACM.

[68] Jonathan Lifflander, G. Carl Evans, Anshu Arya, and Laxmikant Kale. Dynamic scheduling for work agglomeration on heterogeneous clusters. In *Proceedings of (PLC'12) Multicore and GPU Programming Models, Languages and Compilers Workshop at IPDPS 2012*, May 2012.

[69] Chris Lomont. Introduction to intel advanced vector extensions, May 2011.

[70] Steven D. Mackay. Virginia tech unveils hokiespeed, a powerful new supercomputer for the masses, January 2012.

[71] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, July 2003.

[72] Michael D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In *GSPx Multicore Applications Converence*, 2006.

[73] A. McCormick. An engineering approach to solving hpc problems using fpgas. In *Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on*, pages 295 –300, aug. 2007.

[74] Chao Mei, Yanhua Sun, Gengbin Zheng, Eric J. Bohm, Laxmikant V. Kalé, James C.Phillips, and Chris Harrison. Enabling and scaling biomolecular simulations of 100 million atoms on petascale machines with a multicore-optimized message-driven runtime. In *Proceedings of the 2011 ACM/IEEE conference on Supercomputing*, Seattle, WA, November 2011.

[75] Esteban Meneses, Xiang Ni, and L. V. Kale. A Message-Logging Protocol for Multicore Systems. In *Proceedings of the 2nd Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS)*, Boston, USA, June 2012.

[76] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. Top500 supercomputer sites: November 2011, November 2011.

[77] Phil Miller, Aaron Becker, and Laxmikant Kalé. Using shared arrays in message-driven parallel programs. *Parallel Comput.*, 38(1-2):66–74, January 2012.

[78] National Center for Supercomputing Applications. Dell NVIDIA Linux cluster Forge, April 2012. http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/DellNVIDIACluster/.

[79] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *PThreads Programming: A POSIX Standard for Better Multiprocessing*. First edition edition, September 1996.

[80] NVIDIA. *NVIDIA CUDA C Programming Guide, Version 4.2*, April 2012. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CU%DA_C_Programming_Guide.pdf`.

[81] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the cell broadband engine^{TM}processor. *IBM Syst. J.*, 45(1):85–102, 2006.

[82] OpenACC-Standard.org. The openacc application programming interface, November 2011.

[83] J.M. Perez, R.M. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142 –151, 29 2008-oct. 1 2008.

[84] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 184 –592 Vol. 1, feb. 2005.

[85] Judit Planas, Rosa M. Badia, Eduard Ayguad, and Jesus Labarta. Hierarchical tasked-based programming with StarSs. June 2009.

[86] S.K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming simd extensions on the pentium iii processor. *Micro, IEEE*, 20(4):47 –57, jul/aug 2000.

[87] James Reinders. *Intel Threaded Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc., first edition edition, July 2007.

[88] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, 2005.

[89] V.A. Saletore, J. Jacob, and M. Padala. Parallel computations on the charm heterogeneous workstation cluster. pages 203 –210, aug 1994.

[90] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, August 2008.

[91] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. *SIGARCH Comput. Archit. News*, 24(2):191–202, May 1996.

[92] J. Tyler, J. Lent, A. Mather, and Huy Nguyen. Altivectm: bringing vector technology to the powerpctm processor family. In *Performance, Computing and Communications Conference, 1999 IEEE International*, pages 437 –444, feb 1999.

[93] Lukasz Wesolowski. An application programming interface for general purpose graphics processing units in an asynchronous runtime system. Master's thesis, Dept. of Computer Science, University of Illinois, 2008. http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.

[94] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2):50 –59, march-april 2011.

[95] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-cpu, gpu and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264 –266, feb. 2011.

[96] M. Yuffe, M. Mehalel, E. Knoll, J. Shor, T. Kurts, E. Altshuler, E. Fayneh, K. Luria, and M. Zelikson. A fully integrated multi-cpu, processor graphics, and memory controller 32-nm processor. *Solid-State Circuits, IEEE Journal of*, 47(1):194 –205, jan. 2012.

[97] Gengbin Zheng. *Achieving high performance on extremely large parallel machines: performance prediction and load balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[98] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance Evaluation of Automatic Checkpoint-based Fault Tolerance for AMPI and Charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.