Towards a Framework for Abstracting Accelerators in Parallel Applications: Experience with Cell

David M. Kunzman
University of Illinois
201 N. Goodwin Ave.
Urbana, IL 61801
kunzman2@illinois.edu

Laxmikant V. Kalé
University of Illinois
201 N. Goodwin Ave.
Urbana, IL 61801
kale@illinois.edu

ABSTRACT

While accelerators have become more prevalent in recent years, they are still considered hard to program. In this work, we extend a framework for parallel programming so that programmers can easily take advantage of the Cell processor's Synergistic Processing Elements (SPEs) as seamlessly as possible. Using this framework, the same application code can be compiled and executed on multiple platforms, including x86-based and Cell-based clusters. Furthermore, our model allows independently developed libraries to efficiently time-share one or more SPEs by interleaving work from multiple libraries. To demonstrate the framework, we present performance data for an example molecular dynamics (MD) application. When compared to a single Xeon core utilizing streaming SIMD extensions (SSE), the MD program achieves a speedup of 5.74 on a single Cell chip (with 8 SPEs). In comparison, a similar speedup of 5.89 is achieved using six Xeon (x86) cores.

1. INTRODUCTION

In the era of multicore, there has been a movement to specialized cores that are able to exploit parallelism inherent in various algorithms. These specialized cores, or *accelerators*, come in various forms such as Graphics Processing Units (GPUs), the Cell processor's Synergistic Processing Elements (SPEs), and Field Programmable Gate Arrays (FPGAs). One of the challenges facing the widespread adoption of accelerator technologies is the difficulty in programming these devices. The various accelerator technologies are quite different from one another. There are numerous hardware differences that force the programmer to become an expert in each architecture to effectively use the accelerators. Ideally, a programmer would just use a single programming model to utilize all cores present in the system, accelerator cores included. In the absence of any accelerator, the *host* or *standard core* is used to execute the portions of the code that could have otherwise been executed on the accelerator.

SC09 November 14-20, 2009, Portland, Oregon, USA

Our work extends the Charm++ programming model [12] to include *accelerated entry methods*, making Charm++ applications portable to systems that include accelerators while decreasing programmer burden by abstracting away architecture specific details. Accelerated entry methods are similar to standard entry methods (member functions of C++ classes) in Charm++, with the main difference being that they can execute on an accelerator if one is present. This paper will focus on using the SPEs as *accelerators* available to the Power Processing Element (PPE) within the Cell processor [11], treating the PPE as a *host core*. While the work presented here mainly focuses on Cell-based systems, we are working to generalize the approach to support additional accelerators, including currently available GPUs and the unreleased Larrabee design [18].

Although our work is in the context of the Charm++ programming model, we believe that the ideas presented here can be incorporated into other programming languages and models. Charm++ presents a way of breaking down a program such that it is easy for build tools, such as preprocessors, compilers, and linkers, to manipulate the code. Because the build tools are able to manipulate the code and the underlying runtime system has information on the application data, expressed through the programming model, the program becomes portable between various core types. Additionally, by breaking down the program into *schedulable chunks of computation* (*entry methods* in Charm++), the runtime system is able to schedule these chunks in parallel across the available cores based on actual data dependencies expressed via messages (*entry method invocations* in Charm++).

The main contributions of this paper are as follows. We have developed extensions to the Charm++ programming model which, when used to write application code, allows the code to be portable between Cell-based and non-Cell-based platforms with no changes to the code. Additionally, pieces of code (chare objects, libraries, and modules) which are independently written, are able to share the same SPE or set of SPEs by interleaving the execution of *accelerated entry methods*. We demonstrate that the framework achieves good performance by analyzing the performance of an example molecular dynamics application. Further, because our extensions are compatible with the Projections [13] visualization tool provided by Charm++, the runtime system can provide detailed performance information that the application developer can use to analyze and improve

overall application performance.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 provides some background information for the reader, including a short introduction to the Charm++ programming model. Section 4 introduces our extensions to Charm++ and describes how they are used in a Charm++ program. Section 5 demonstrates the use of accelerated entry methods through an example molecular dynamics code to give the reader a more concrete understanding of their usage. Code examples are given along with performance data on both x86 processors and Cell processors. Finally, sections 6 and 7 discuss conclusions and future work, respectively.

2. RELATED WORK

The CellSs programming model [3] most closely resembles our approach. CellSs allows a programmer to annotate existing code by marking functions as *candidates* to be executed in parallel. The annotations indicate which parameters contain input data and output data generated by the function. A runtime system then determines data dependencies during runtime and schedules these functions on the SPEs based on these data dependencies. These annotated functions in CellSs resemble the *accelerated entry methods* discussed here. However, they differ in that accelerated entry methods do not implicitly synchronize with a calling function (i.e. values are not returned to the caller). More generally, programs written using Charm++ are explicitly written to be parallel while the CellSs approach is more analogous to using OpenMP[7] pragmas to annotate sequential C/C++ programs. In our approach, data dependencies are satisfied via messages (i.e. entry method invocations expressed in the application code) instead of analysis by the runtime system. Programs written using the CellSs model are portable to other platforms, including common multicore architectures by using the SMPSs [1] runtime. However, our approach also includes a SIMD instruction abstraction which allows the programmer to write portable code that takes advantage of various SIMD instruction extensions (e.g. SSE, AltiVec/VMX, and so on). Furthermore, our approach allows programs to run in parallel across multiple Cell processors using the same abstractions (i.e. there is no difference in the code for using one Cell or using a cluster of Cell processors). Our approach also addresses supporting the sharing of SPEs between independently written pieces of code.

Numerous other high-level programming models have also been developed or extended to support the Cell processor. The Sequoia [9] programming model exposes the memory hierarchy to the programmer by decomposing tasks. If a task is too large to fit in a particular level of the memory hierarchy, it is broken down into subtasks that will fit. The subtasks are self-contained and cannot directly communicate with one another. This approach focuses on distributing a single large task across the SPEs while our approach allows for multiple large tasks to be broken down and executed concurrently on the SPEs. In the RapidMind model [16], the programmer specifies operations through *program* constructs. These program constructs are then applied to arrays of data structures in a data parallel manner. In Mercury's Multicore Framework [6], large multi-dimensional arrays of data are broken down into tiles which are then streamed through the SPEs and operated on independently of one another. Both approaches differ from ours in that they operate on array based data structures while our approach allows the programmer to specify multiple arbitrary pieces of code, performing different operations, to be executed on the SPEs. The MPI programming model has also been extended to support Cell by allowing the programmer to use MPI microtasks [17] to program the SPEs.

3. BACKGROUND

In this section, we provide background information on the Cell processor, the Charm++ programming language, and the Offload API.

3.1 Cell Processor

The Cell Broadband Engine Architecture (CBEA) [11], commonly referred to simply as *Cell*, was jointly developed by Sony, IBM, and Toshiba. The Cell processor has been used in several products, including IBM's QS20 and QS22 Cell Blades, Sony's Playstation 3 (PS3) game console, and the SpursEngine by Toshiba. The Cell processor is also used in Roadrunner [2], currently one of the largest supercomputers. The CBEA defines two types of cores in the Cell processors, the Power Processing Element (PPE) and the Synergistic Processor Element (SPE). The PPE is similar to a typical core found in modern Power-based processors. In particular, load and store instructions issued from the PPE access system memory via a typical cache hierarchy. In current Cell processor implementations, there is a single PPE core per chip.

The SPEs are smaller, less complex cores designed to be *accelerator* cores to the PPE. With each Cell processor having several SPEs, the SPEs provide the great majority of the chip's processing power (peak GFlops/sec). However, the SPEs have several architectural differences from the PPEs that make them harder to program. First, the SPEs do not have direct access to system memory. Instead, loads and store instructions access a local scratchpad memory called the *local store*. The local store is 256KB in size, fully controlled by the application code, and must contain all code and data used by the SPE. Second, data is moved between an SPE's local store and the main system memory via Direct Memory Accesses (DMAs) that are explicitly issued via the application code. Third, the SPEs have a different instruction set architecture (ISA) and, as such, use a different binary image than the PPE. A binary image that contains instructions for an SPE is embedded into the PPE's binary image at compile time and loaded into one or more SPEs at runtime by application code. Additionally, similar to many other core designs, the SPEs make use of single instruction multiple data (SIMD) extensions that are necessary to achieve the majority of the core's peak floating point performance. These architectural differences require architecture specific code to be interleaved within the application code, reducing the general portability of the application code. Simply recompiling an application (e.g. a program written using C/C++) for the Cell processor is not sufficient to take advantage of the processor's full potential. The opposite is also true, porting code from a Cell-based system to a non-Cell-based system requires removing all the architecture specific code.

3.2 Charm++ Programming Model

With this work, we are extending the Charm++ programming model to include *accelerated entry methods*. Accelerated entry methods will be discussed in more detail in sections 4.1 and 4.2. The Charm++ programming model has been in use for over a decade and has been used in the development of multiple production scientific simulations, including NAMD [4], OpenAtom [5], and ChaNGa [10]. Furthermore, Charm++ applications account for a significant number of cycles in supercomputing centers. The remainder of this section gives a short description of the Charm++ programming model. A more complete description of Charm++ can be found in Kalé et al. [12] and the Charm++ tutorial.[1]

The Charm++ programming model is based upon C++. In Charm++, the application is decomposed into a set of *objects* called *chare objects* or simply *chares*. Chare classes can be basically thought of as C++ classes, having encapsulated member variables and member functions just like classes in C++. Chare objects are instances of the chare classes. Beyond standard C++ methods, chare classes also have *entry methods*. Entry methods are special member functions of the chare class that can be *invoked asynchronously* by other chare objects, regardless of whether or not the two chare objects are on the same physical processor. When an entry method is invoked, a message is created and passed to the the target chare object with any data needed by the target entry method. Forward progress in a Charm++ application is made as the chare objects receive messages, do work in response to receiving the messages, and send out more messages to other chare objects.

Because this work focuses on creating a new type of entry method, we would like to point out some of the key attributes of standard entry methods. Entry methods are *special* member functions of chare classes. I.e. entry methods do not behave in the same manner as standard member functions in C++ classes. One of the main differences is that entry methods are *scheduled*, not simply *called*. Member functions are called, execute immediately, and return a value once they have completed. C++ code calls a member function in the following way: *myCPPObject.myMemberFunction(...)*. Entry methods are called in a similar way: *myChareObject.myEntryMethod(...)*. However, the call to the entry method does not actually result in the entry method being executed (invoked or called) directly. Instead, a message, containing the parameters passed to *myEntryMethod(...)*, is created and sent to the processor that contains the chare object represented by *myChareObjectProxy*.[2] From the caller's perspective, the entry method returns immediately with no return value. The message, when it arrives at the target processor, is queued by the Charm++ runtime system. Typically, there are many chare objects per physical processor. The runtime system, on a per processor or per core basis, dequeues messages one by one and executes the associated entry method on the target chare object, effectively virtualizing the physical processors.

From the point of view of the programmer, *invoking an entry method* means *scheduling the execution of an action (entry method) on a set of data (chare object's local data and data contained in the message itself) at some point in the future*. In other words, it triggers the target object to start doing a piece of work in parallel with other entry methods executing on other chare objects. Another difference, which has already been mentioned, is that entry methods do not return a value. More to the point, entry methods do not implicitly synchronize in any way with the invoking code. If the invoked entry method needs to send information back to the invoking entry method, the invokee will need to invoke another entry method on the object that originally sent the message. The only way to pass data between chare objects is to send the data by invoking an entry method on the chare object that is to receive the data.

Charm++ does not use special compiler extensions or precompiler macros to implement the programming model. The Charm++ runtime system and other related frameworks are implemented as libraries that are linked to the application. Member functions that are also entry methods are specified, with their signatures (i.e. parameter types), in *interface files*. Interface files also specify which classes are chare classes along with various message types, readonly global variables, and so on. The information in the interface files is used by the Charm++ build tools to generate various pieces of code that are included into the application, such as code to pack and unpack (serialize) entry method parameters to and from messages.

3.3 The Offload API

Prior to this work, some Charm++ applications were ported to Cell-based platforms by directly making use of the Offload API [14, 15]. The Offload API is a C library that allows a programmer to submit *work requests* to the SPEs. The work requests represent *independent chunks of computation* in a manner that is similar to the accelerated entry methods described in section 4.1. In this sense, the Offload API is a C library that can be used by general C/C++ programs to make use of the ideas presented in this paper. It should be noted that our modifications to the Charm++ build process automatically take care of some extra programmer effort that will not be handled automatically by simply using the Offload API directly, as discussed in section 4.5. For example, the Charm++ build process incorporates a mechanism for *registering function indexes* (used by the Offload API) dynamically at program startup. This registration mechanism, along with the nature of accelerated entry methods, allows for independently written pieces of code (chare objects, libraries, modules, etc.) to effectively share a single SPE by interleaving accelerated entry methods with one another. The accelerated entry methods that are presented in this work make use of the Offload API behind the scenes. The functionality of the Offload API could be used by other programming models.

[1] http://charm.cs.uiuc.edu/tutorial/

[2] Because objects only exist on a single processor, they have *proxy objects*, or just *proxies*, that *represent* them on other processors. These proxy objects simply relay the messages or entry method invocations to the actual chare object, on whichever processor that object happens to actually exist. From the programmer's perspective, a proxy for a chare object can be treated as if it were the object itself for the purposes of invoking entry methods.

```
module myModuleName {
  array[1D] myChareArray {
    entry [accel] void myAccelEntryMethod(int passedParam1, float passedParam2[passedParam1])
                        [  readonly : int localParam0 <impl_obj->memberVar0>,
                           readwrite : float localParam1[localParam0] <impl_obj->memberVar1> ] {
      // entry method code has access to passedParam1, passedParam2, localParam0, and localParam1
    } myAccelEntryMethod_callback;
    ...
```

Figure 1: Structure of an accelerated entry method.

4. EXTENSIONS TO CHARM++

In this section, we describe the extensions we made to the Charm++ programming model, build process, and runtime system to support this work. Our extensions include *accelerated entry methods*, *accelerated blocks*, and an abstraction for SIMD instructions commonly supported by modern processors.

4.1 Accelerated Entry Methods

We have extended the Charm++ programming model by introducing *accelerated entry methods*. Accelerated entry methods are entry methods which can be executed on accelerator hardware when such hardware is present. Otherwise, when no accelerator is present, the accelerated entry methods will be executed using the same host cores as the rest of the *non-accelerated* entry methods in the application.

By using accelerated entry methods, Charm++ programs can be written once without the programmer having to be concerned about the underlying hardware architecture. If the program is compiled for a Cell-based platform, the accelerated entry methods are compiled both for the PPEs and SPEs. Currently, while both the PPE and SPE versions of the accelerated entry methods are created, only the SPE version is actually used by the application. In the future, we plan to allow the runtime system, through the help of a load balancing framework, to make dynamic decisions on how much work should be pushed to the SPEs and how much work should remain on the PPEs. Furthermore, if the SPE's local store cannot hold all the data associated with an accelerated entry method, the accelerated entry method could be executed on the PPE instead. However, this failover mechanism has not yet been fully implemented in the Charm++ runtime system.

An additional advantage of this approach is that the programmer does not need to be concerned about the interactions of various portions of the code and how independent pieces of code are going to share the available accelerators. That is to say, the programmer does not have to worry about time-sharing the accelerator between different portions of the code. Instead, the programmer simply programs each portion of the application to use accelerated entry methods when possible. During runtime, the Charm++ runtime system will schedule the entry methods on the accelerator. This allows entire portions of the application (i.e. modules and libraries) to be written independently from one another and to still utilize the accelerator. As various modules, objects, libraries, etc. are brought together to form the overall application, the programmer need not worry how the components will interact or even how many accelerators are available to the application. Programmers can then use the idea of accelerated entry methods to develop *accelerated libraries*. Applications can be programmed to use the accelerator directly and link to multiple libraries that all make use of the same accelerator. All of these software components then share the accelerator resources without burdening the programmer to explicitly code the components to share the accelerator resources.

Accelerated entry methods are meant to be used for the computationally intensive portions of the application. This follows directly from the fact that accelerators are designed to work on the computationally intensive portions of an application. In our implementation, the accelerated entry methods do not currently support entry method invocation. That is to say, other entry methods cannot be invoked directly from accelerated entry methods. For one chare object to pass data generated by an accelerated entry method to another chare object, the accelerated entry method needs to fill in a buffer which can later be sent via a message in the accelerated entry method's callback function (a member function of the object that is called when the accelerated entry method completes). Invoking other entry methods directly from accelerated entry methods may be supported in the future as these extensions continue to develop.

So that the build tools provided by Charm++ can manipulate the code included in *accelerated entry methods* (discussed in section 4.1), the function bodies for accelerated entry methods are specified along with the entry method declaration in the interface file. Other languages with specialized compilers and/or compiler extensions could have the function bodies for accelerated entry methods in the source code file itself. This is a current drawback of our specific implementation, but is a result of the Charm++ build process and not a limitation of accelerated entry methods themselves. The interface files are processed by the Charm++ build tools during compile time, at which point the build tools generate the required code needed by the accelerated entry methods. This generated code is then included with the application code written by the programmer.

4.2 Structure of Accelerated Entry Methods

Figure 1 illustrates the general structure of an accelerated entry method. The code section is from an interface file which declares a single chare class, *myChareArray*. This particular chare class is declared as a chare collection (specifically, a one dimensional array of chare objects). The *myChareArray* chare class has a single accelerated entry method declared, called *myAccelEntryMethod*.

There are several differences between declaring a standard entry method and an accelerated entry method. First, the

```
// Interface File: Declare prototype so the runtime system understands what the parameters are
entry void accumValues(int inArrayLen, align(sizeof(vecf)) float inArray[inArrayLen]);

// Source File: Accumulate the incoming floating point values into the local array of values
// NOTE: localArrayLen and localArray are member member variables of ChareObjClass
void ChareObjClass::accumValues(int inArrayLen, float* inArray) {
  if (inArrayLen != localArrayLen) return;
  vecf *inArrayVec = (vecf*)inArray, *localArrayVec = (vecf*)localArray;
  int arrayVecLen = inArrayLen / vecf_numElems;
  // Add as many elements using SIMD operations as possible
  for (int i = 0; i < arrayVecLen; ++i)
    localArrayVec[i] = vaddf(localArrayVec[i], inArrayVec[i]);
  // Add remaining elements via scalar operations (if inArrayLen % vecf_numElems != 0)
  for (int i = arrayVecLen * vecf_numElems; i < inArrayLen; ++i)
    localArray[i] = localArray[i] + inArray[i];
}
```

(A) Standard Entry Method Version

```
// Chare Class Declaration
class MyChareClass : public CBase_MyChareClass { ... friend class CkIndex_MyChareClass; ... };

// Interface File: Accumulate the incoming floating point values into the local array of values
// NOTE: localArrayLen and localArray are member member variables of ChareObjClass
entry [accel] void accumValues(int inArrayLen, align(sizeof(vecf)) float inArray[inArrayLen])
                        [ readonly : int localArrayLen <impl_obj->localArrayLen>,
                          readwrite : float localArray[localArrayLen] <impl_obj->localArray> ] {
  // Function body same as Standard Entry Method Version (not repeated for brevity)
} accumValues_callback_function;
```

(B) Accelerated Entry Method Version

Figure 2: Comparison of a standard entry method version and an accelerated entry method version of the same operation.

accel keyword is used to declare the entry method as being accelerated, as shown in figure 1. Second, the return value of an accelerated entry method must be *void*. Third, in addition to the *passed parameters*, accelerated entry methods have *local parameters*. Standard entry methods only declare passed parameters and resemble parameters being passed by value as one function calls another function in C/C++. Accelerated entry methods also have local parameters which list the local member variables of the chare object that the entry method code will access as it executes. Fourth, the function body of the accelerated entry method needs to be located in the interface file instead of the source file so the Charm++ build tools are able to manipulate it. Both the third and fourth requirements stem from the fact that the Charm++ build process does not have a custom C/C++ compiler which could otherwise do these tasks automatically for the programmer. Finally, a *callback function* is specified. The callback function is a member function of the chare class that will be called on the same chare object when the accelerated entry method completes. Callback functions are executed on the host cores.

The local parameter list contains additional information that the passed parameter list does not contain (refer to figure 1). First, an optional access specifier indicates if the data will be accessed in a *readonly*, *readwrite* (the default), or *writeonly* manner. All passed parameters are readonly. This specifier is used to direct how the data should be moved between system memory and the accelerator's local memory. Next, the data type for the local parameter is specified. After the data type, the programmer indicates the local parameter name as it will be used in the function body. If the local parameter is an array, the size of the array is indicated in square brackets directly after the local name. Finally, the member variable of the chare class that will be associated with the local parameter name is indicated. Even though the local parameter and member variable names do not match in figure 1 (e.g. *localParam0* and *memberVar0*), there is no restriction requiring the names to be different. The *impl_obj* (implied object) keyword is similar to the *this* keyword in C/C++ and *points* to the instance of the chare object that the entry method is operating on. The function body of an accelerated entry method should only access the passed and local parameters it specifies, along with any local variables that are declared within the scope of the function body itself. This is what we mean when we refer to the accelerated entry methods being *self contained and clearly defined*.

When an accelerated entry method completes executing, a *callback member function* is called on the same object on the *host core*. The callback entry method that is to be called is specified at the end of the declaration of the accelerated entry method and may invoke other entry methods. In figure 1, the callback entry method *myAccelEntryMethod_callback* will be called on the same chare object when *myAccelEntryMethod* finishes executing on the chare object.

Figure 2 presents another example of an accelerated entry method which makes use of the SIMD instruction abstraction discussed in section 4.4. Both versions of the entry method are invoked in the same way, *myChareObj.accumValues(arrayLen, arrayData)*, by another entry method, accelerated or not. Only one version of the entry method needs to be included in the application code, however, the accel-

erated entry method version can be executed on an accelerator. Both versions of the entry method perform the same operation which is to add the contents of an array passed to the object to an array of the same length local to the same object. As can be seen from the figure, the difference in the code between the two versions is fairly minor, mainly in the entry method declaration itself, demonstrating how accelerated entry methods resemble standard entry methods in application code. It should be pointed out, however, that not all translations from standard entry method to accelerated entry method will result in code that is as similar as the versions presented in figure 2 (i.e. it is not always the case that the function bodies will be identical for both versions, which is the case in this particular example). There is one additional difference pointed out by figure 2 (B), which is the declaration of the friend class so that the generated code can access the data within the chare object.

A disadvantage of our extensions is that they currently do not have any mechanism that supports storing data in the SPEs' local stores across accelerated entry method executions. Currently, if persistent data is required, the programmer is required to declare a static data structure in an accelerated block (described in section 4.3) and use Offload API work requests to fill in the data structures. Although possible, it would be more natural for the programmer if our programming model extensions incorporated mechanisms to support this type of activity directly. However, the development of this mechanism is further complicated by the fact that we are targeting other accelerators in addition to the Cell's SPEs, such as currently available GPUs and the future Larrabee GPU. As such, we leave the development of this mechanism as future work. Another disadvantage of our approach is that we currently leave the granularity of the accelerated entry methods to the programmer (vs. the Sequoia model [9] which automatically determines task grain size based on hardware constraints).

4.3 Accelerated Blocks

In addition to accelerated entry methods, we have also introduced *accelerated blocks* to the Charm++ programming model. Accelerated blocks are simply blocks of code that are accessible to the accelerated entry methods. For example, one could write one or more functions in an accelerated block of code. These functions are then available to be called from an accelerated entry method. Accelerated blocks can also be used to include external files and define macros to be used by multiple accelerated entry methods. Figure 3 has an example of an accelerated block which simply includes a header file containing defines used by the *doCalc()* accelerated entry method. Section 5 will discuss the specifics of this example code in more detail.

4.4 Abstraction of SIMD Instructions

To fully take advantage of accelerators such as the SPEs on the Cell, we also had to develop an abstraction for the SIMD instructions supported by both the accelerators and host cores. While the idea of an abstraction for portability of SIMD code, by itself, is not a novel one, this work represents the introduction of a SIMD instruction abstraction to the Charm++ programming model. Various types are defined that represent packed data elements such as four single-precision floating point numbers, two double precision floating point numbers, four 32-bit integer values, and so on. Operations such as addition, subtraction, multiplication, shift, insert, extract, fused-multiply-add, and so on that operate on one of these types are also defined. When possible, these types and operations are mapped to the SIMD instructions supported by the underlying hardware such as AltiVec/VMX on Power (including the Cell's PPE), the SIMD instructions supported by the SPEs, and streaming SIMD extensions (SSE) on x86 processors. Generic C/C++ versions of all these types and operations are defined and used by default on platforms that do not directly provide support for SIMD instructions. Figure 2 illustrates some of the data types and operations that are provided (e.g. *vecf* which represents a packed set of single-precision floating point values). It should also be pointed out that the code in figure 2 works regardless of how many float point values are packed into the *vecf* data structure (i.e. *vecf_numElems*).

4.5 Scheduling and Execution

Scheduling an accelerated entry method occurs as a two stage process. The first stage of the process is the same for all entry methods. The second stage involves scheduling the accelerated entry method on the accelerator device (SPE in this case) and is specific to the type of accelerator being used. For the Cell processor, we leverage our previous work on the Offload API along with additional modifications we have made to the Charm++ build process in this work.

The first stage of scheduling an accelerated entry method involves queueing the message. For accelerated entry methods, this stage is the same as the scheduling process for other standard entry methods in general. The Charm++ runtime system keeps a *message queue* of all the *yet to be executed* messages. For standard entry methods, as a message reaches the head of the message queue and is dequeued, the associated entry method is executed on the associated object with the message data being passed into the entry method. The programmer may influence the ordering of the messages in the message queue by assigning them a *message priority*.[3] For accelerated entry methods, if no accelerator is present, the accelerated entry methods are executed on the host core as they are pulled from the message queue, just as standard entry methods are executed. However, if an accelerator is present, the accelerated entry methods enter into a second stage of scheduling.

In the second stage of scheduling, the information provided by the programming model, including our extensions, is used by the runtime system to identify the data that will be required by the accelerator. Our modifications to the build process create two versions of the accelerated entry method, one for the PPE and one for the SPE. In the future work, as part of the dynamic load balancer, we plan to have the runtime system make dynamic decisions about load balancing the work between the PPE and the SPE cores. However, at this time, we currently execute all accelerated entry methods on the SPEs. The second stage of scheduling is managed by the Offload API directly. To understand how this occurs,

[3]Other methods can be used to influence the order of message execution, such as indicating a queueing scheme (first-in-first-out (FIFO) ordering, last-in-first-out (LIFO) ordering, and others) and using *immediate* or *expedited* messages.

one needs to understand how we have modified the build process.

Our modifications to the build process generate code associated with the accelerated entry methods and make use of the Offload API to execute the accelerated entry methods at runtime. The generated code includes (1) PPE code to create an Offload API work request at runtime that encapsulates the execution of the accelerated entry method on the accelerator, (2) a lookup function required by the Offload API, (3) registration functions and tables, and (4) a SPE function that will do the actual work of the accelerated entry method. First, the generated PPE code, using the information supplied by the programming model, creates an Offload API work request which encapsulates both the data to be operated on (in our case message and object data) and the function that will be applied (an SPE function generated from the accelerated entry method's function body). For the purposes of this discussion, the terms *accelerated entry method* and *work request* can be used interchangeably. Once the work request has been created, the Offload API takes over for scheduling the work request on one of the accelerators (the second stage of scheduling). The Offload API decides which SPE the work request will be executed on, currently in a round-robin manner, with a fixed maximum number of work requests specifically assigned to each SPE at any given moment. Excess work requests are queued on the PPE and later assigned to the SPEs as the SPEs finish previously issued work requests. A small runtime on each SPE identifies the work requests that are currently assigned to that SPE and moves each work request through a simple state machine which basically allocates memory in the local store for the work request, DMAs input data into the work request's memory, executes the work request (accelerated entry method), DMAs output data back to system memory, releases the work request's local store memory, and finally notifies the PPE that the work request has been completed and any resulting output data is in system memory. The Offload API itself, along with how the SPE runtime handles data movement and work request scheduling, is discussed in more detail in Kunzman 2006 [14] and Kunzman et al. 2006 [15]. The overlap of data movement to and from the SPE's local store while the SPE executes work requests naturally follows from the fact that, at any given moment, multiple work requests are moving through different stages of the state machine on an SPE. As work requests move through the state machine, some are transferring data (input or output) while others are ready for execution. Because the Offload API is a library, when a work request is ready to execute, the SPE runtime requires a lookup function (number 2 above) to call into application code. The lookup function we generate at build time uses a lookup table generated at application startup by our generated registration functions (number 3 above) which uniquely identifies the various accelerated entry methods via function pointers. The function pointers are used to call into the generated SPE code which executes the accelerated entry methods' function bodies (number 4 above). The net result is that accelerated entry methods, along with their data, are streamed through the available SPEs (i.e. self contained tasks are streamed through the SPEs).

To summarize, our modifications extend the Charm++ programming model to include accelerated entry methods along with related extensions. We have modified the Charm++ build process to generate code to execute the accelerated entry methods on host and accelerator cores, making accelerated entry methods portable between systems with and without accelerators. The SIMD instruction abstraction further increases the portability of entry methods, accelerated or not, that make use of SIMD instruction extensions. For the Cell processor, our build process modifications generate code that leverages the SPE runtime provided by the Offload API to stream accelerated entry methods through the available SPEs as messages are dequeued from the general message queue. Furthermore, we generate code that registers accelerated entry methods at startup, allowing independently written accelerated entry methods to time-share the available accelerators. In section 5, we discuss an example molecular dynamics code used to illustrate the performance achieved by an application that utilizes accelerated entry methods.

5. MOLECULAR DYNAMICS EXAMPLE

To illustrate how accelerated entry methods would be used in an actual application, we developed a simplified molecular dynamics (MD) code. Charm++ has been used in the development of production molecular dynamics applications, including NAMD [4] and OpenAtom [5], and, as such, we initially chose an MD code for testing our extensions. We plan on developing other Charm++ applications that make use of accelerated entry methods in the future. The source code for our MD example program, along with other examples, can be found in the Charm++ distribution. For brevity, only a small portion of the program is presented along with performance results.

5.1 Description of MD Example Code

The structure of the MD example program is based on the nonbonded force computation in NAMD [4], a production molecular dynamics code. However, our MD example is quite a bit less complex. There is a three dimensional volume of space with a fixed number of particles randomly placed within the space. During each *timestep* of the simulation, each particle interacts with every other particle in the overall system according to Coulomb's Law (electrostatic force between two charged particles). Each timestep advances the simulation time by one femtosecond. The list of particles in the overall system is divided into a set of objects called *patches*. Unlike NAMD, the particles are divided evenly between the patches with no regard to their spatial position. The electrostatic forces between the particles are calculated by *compute* objects. Compute objects come in two varieties: *self computes*, which calculate the forces among particles within a single patch, and *pair computes*, which calculate the forces between particles in two different patches.

In the MD example, because all particles interact with one another each timestep, the number of compute objects grows at a rate of $O(N^2)$, where N is the number of patch objects. NAMD uses a cutoff-based calculation (i.e. particles only interact with particles within a certain cutoff distance) and thus the number of compute objects grows at a rate of O(N), where N is the number of patch objects. However, we have chosen a problem size, as described in section 5.2, comparable to the ApoA1 benchmark commonly used by NAMD

```
module pairCompute {
  accelblock { #include "md_config.h" };
  entry [accel] void doCalc()[  readonly : int numParticles <impl_obj->numParticles>,
                                readonly : float p0_x[numParticles] <impl_obj->particleX[0]>,
                                readonly : float p1_x[numParticles] <impl_obj->particleX[1]>, ...
                               writeonly : float f0_z[numParticles] <impl_obj->forceZ[0]>,
                               writeonly : float f1_z[numParticles] <impl_obj->forceZ[1]>
                              ] {
    // Cast float arrays to vector float arrays
    vecf* p1_x_vec = (vecf*)p1_x; int p1_x_vec_length = numParticles / vecf_numElems;
    ...
    // Calculate r and r^2 between the particles
    vecf p_x_diff_vec = p1_x_vec[i] - p0_x_vec[i], p_y_diff_vec = ...
    vecf p_x_diff_2_vec = p_x_diff_vec * p_x_diff_vec, p_y_diff_2_vec = ...
    vecf r2_vec = p_x_diff_2_vec + p_y_diff_2_vec + p_z_diff_2_vec;
    vecf r_vec = vsqrtf(r_2_vec);
    ...
  } doCalc_callback;
  ...
}
```

Figure 3: Portion of the pair compute object's doCalc() accelerated entry method.

which keeps the number of compute objects relatively similar. In particular, NAMD using a default configuration for the ApoA1 benchmark has 92224 particles divided across 144 patches and creates 6017 compute objects to perform the force calculation (with PME). For our MD program, we use a problem size of 92160 particles divided evenly across 144 patches, resulting in 10440 compute objects (less than a factor of 2 more than ApoA1 in NAMD). We simply point this out to put the problem size into some context given the fact that the MD example code has $O(N^2)$ computes compared to NAMD's O(N) computes. The amount of parallelism in both applications is directly related to the number of compute objects created, which execute one entry method each per timestep.

A given timestep is started by each of the patch objects updating the compute objects with the particles' position data. This is done via *proxy patches*. Each patch has a *representative* or *proxy* on each of the physical processors. The patch first sends the updated particle position data to all of its proxies. The proxy patches then send the position data to any compute objects local to that processor that require the patch's data. This technique, essentially an object-based multicast, reduces the total amount of data being sent over the interconnect. Once a compute object has received all the particle data it requires (from one patch for self computes, from two patches for pair computes), the compute object calculates the forces experienced by the particles that it is responsible for. It then passes this force information back to the patches via the patch's proxy objects. That is, the proxy patches combine all forces for a patch local to a particular processor and then send the combined set of forces back to the patch. Once the patch has received force data from all of the compute objects that it initially sent position data to, via its proxy patches, it updates the velocity and position data of its particles, effectively ending the timestep. If there are more timesteps to be calculated, the patch starts the process again by sending out the updated particle data to the compute objects.

It should be noted that the patch objects, the self compute objects, and the pair compute objects all make use of accelerated entry methods for their respective work (force integrations for patches, force calculation for computes). There is no coordination between the code for these chare classes. For example, whether or not the patch object uses the accelerator via accelerated entry methods, the code for the compute objects remains exactly the same. All the chare classes are written independently and internally make use of accelerators when available. In the same manner, libraries which make use of accelerators can be developed independently of application code or other libraries without regard to whether or not the code it will eventually be linked to also makes use of accelerators. The runtime system takes care of scheduling the accelerated entry methods on the available accelerators.

Figure 3 contains a small portion of code from the MD example program. In particular, this is the declaration of the pair compute object's *doCalc()* accelerated entry method along with a few lines of code from the function body to illustrate the SIMD instruction abstraction. First, there is an accelerated block which includes the file *md_config.h*. This gives the accelerated entry methods access to all the *#defines* and other macros that are declared in the md_config.h header file. Second, the doCalc() accelerated entry method itself is declared (within the PairCompute chare class, which is not shown). The declaration of doCalc() defines no passed parameters and several local parameters that will be accessed by the entry method. There are several readonly buffers that contain the particle's position data from both patches for each of the three dimensions (not all shown). There are also several writeonly buffers that will contain the resulting force data for both patches for each of the three dimensions (not all shown). The body of the accelerated entry method takes advantage of the SIMD abstractions described in section 4.4. Using the SIMD abstraction allows this code to be portable between core types while still allowing it to take advantage of the different SIMD instructions supported by the various cores.

5.2 Testing Methodology

Two systems were used to test the performance of the MD example program, an x86-based system and a Cell-based

system. The x86-based system is a single workstation comprised of two quad core processors (two Intel Xeon E5320 chips running at 1.86GHz). The Cell based system is a cluster of four IBM Cell Blades (QS20s; each having two Cell chips running at 3.2GHz and 8 SPEs per chip) and four Playstation 3s (PS3s; each running at 3.2GHz and using 6 SPEs per chip). In addition to running on the Cell processors individually, we make runs which utilize both the QS20s and the PS3s in a single execution. The Blades and PS3s are connected via a gigabit Ethernet network.

To illustrate the performance, we made several runs on the hardware configurations described. The parameters used to make the runs are outlined in figure 4. The problem size chosen mimics the commonly used ApoA1 benchmark.

Number of Patches	144 patch objects
Number of Particles	640 per patch (92160 total)
Self Computes	144 SelfCompute objects
Pair Computes	10296 PairCompute objects
Number of Timesteps	128 (128 femtoseconds)
Total Flops (128 steps)	16851 GFlops

Figure 4: Simulation Parameters

All optimizations in the application code are applied to both systems since a single set of source code files is being used with no architectural specific code. This helps keep the comparison between the x86-based and the Cell-based systems as fair as possible, in addition to being more convenient for the application programmer since the application code is only written once. The first optimization is reducing the number of messages being sent between processors by using the proxy patch objects as described in section 5.1. The second optimization is loop unrolling and applying the SIMD instruction abstraction to the computes' force calculation code and the patch objects' integration code. Additionally, the pair compute force calculation code's innermost loop, which makes up the majority of the work performed by the program, was software pipelined.

5.3 Performance Results

We achieve a speedup of 5.74 when using a single QS20 Cell chip compared to a single x86 core. This is similar to the speedup of 5.89 achieved when using six x86 cores compared to a single x86 core. That is to say, a single Cell processor with eight SPEs is performing on par with six x86 cores for this example MD program. Once again, both platforms are taking advantage of SIMD instructions. Because of this, the main differences between an x86 core and an SPE are the clock speeds, characteristics of the pipelines, and the memory/cache hierarchies. We believe these to be good results considering both software characteristics of the MD example program and architecture differences between the SPE and x86 cores, as discussed in the remainder of this section.

Figure 5 contains these data points along with others to illustrate the general performance trends of the MD example program. The single PS3 case is particularly slow. Upon closer examination, we noticed that the program was swapping pages out to disk, which is likely causing the poor performance. As a result, for the PS3 runs, speedup is normalized against the two PS3 case instead of the single PS3 case. For all other hardware configurations, the speedups are normalized to the single chip/core case for the same hardware configuration. There is a second speedup reported, *Speedup (vs. 1 x86 core)*, which can be used to relate the performance of the x86 configurations to the Cell configurations. In addition to these speedups, figure 5 also lists the number of cores (SPEs or x86 cores), execution time for the simulation (averaged execution time of multiple runs), average GFlops per second (GFlops/sec) over the course of the simulation, and the average percent of peak GFlops/sec achieved. The GFlops count flops that go towards useful physics calculations. That is, it does not include flops from reduction of force data or any other non-physics-flops the code performs. The execution time is from the start of the simulation to the end of the simulation. It does not include startup time for the application. However, it does include all code related to the simulation steps themselves such as sending of messages over the network, Charm++ runtime system overheads, all synchronization, and so on.

To better understand our speedups between the Cell-based and x86-base hardware configurations, we consider the differences between the two types of cores: SPE and x86. On the Cell-based platforms, we focus on the SPEs since the SPEs are doing all of the physics calculations. The SPEs have a clock speed of 3.2 GHz, have dual-issue in-order pipelines, and can only access their associated local store. The x86 cores have a clock speed of 1.86 GHz, have 4-issue out-of-order pipelines, and can access system memory via a cache hierarchy. The SPEs are designed to be simple cores that can execute a well structured binary code efficiently. Because the SPEs have simpler in-order pipelines, it is important for performance that the compiler optimizes the code well [8]. The x86 cores are designed to be more complex cores that can speed up the sequential portions of the application code (the entry methods in our case) quite well.

To get a more concrete picture of the performance observed from the Cell-based hardware configurations, we examined the compiler output for the pair compute object's force calculation code. This code makes up the great majority of the work performed by the application. The innermost loop, which represents four particle-to-particle interactions, has a total of 54 instructions. Of the 54 instructions, 29 are SIMD single precision floating-point instructions, only two of which are fused-multiply-adds, representing 124 total flops per iteration of the inner loop. By using the SPE timing tool provided in the Cell SDK, we can see that these 54 instructions will take 56 cycles to execute[4]. We can then see that the sequential physics code as optimized by the SPE compiler achieves approximately 2.2 flops/cycle on average (124 flops / 56 cycles). If the SPE had infinite memory (no DMAs, all data in the local store, etc.) and could continuously run the inner loop of the pair compute code for the entire problem, the code output by the SPE compiler would reach approximately 27.7% of the SPE's peak performance.

[4]This assumes that the first instruction for iteration i will be issued the cycle after the branch instruction for iteration *i-1*. In other words, we assume perfect branch prediction and are thus giving the SPE as much credit as possible. 56 cycles represents the number of cycles between the first instructions being issued for two consecutive iterations.

# PS3 Cells	# SPEs	Execution Time	GFlops/Sec	% Peak	Speedup	Speedup (vs. 1 x86 core)
1	6	782.56	21.53	14.02%	0.59	2.47
2	12	229.79	73.33	23.87%	2.00	8.41
3	18	151.78	111.03	24.09%	3.03	12.73
4	24	114.94	146.61	23.82%	4.00	16.81
# QS20 Cells	# SPEs	Execution Time	GFlops/Sec	% Peak	Speedup	Speedup (vs. 1 x86 core)
1	8	336.45	50.09	24.46%	1.00	5.74
2	16	174.39	96.63	23.59%	1.93	11.08
3	24	118.66	142.01	23.11%	2.84	16.28
4	32	94.55	178.22	21.76%	3.56	20.43
# QS20 & PS3 Cells	# SPEs	Execution Time	GFlops/Sec	% Peak	Speedup	Speedup (vs. 1 x86 core)
1 & 1	14	225.34	74.78	20.87%	1.00	8.57
2 & 2	28	112.48	149.82	20.90%	2.00	17.17
3 & 3	42	75.34	223.68	20.80%	2.99	25.64
4 & 4	56	58.57	287.69	20.07%	3.85	32.98
# x86 Cores (SMP)	# Cores	Execution Time	GFlops/Sec	% Peak	Speedup	Speedup (vs. 1 x86 core)
1	1	1931.75	8.72	58.56%	1.00	1.00
4	4	485.93	34.68	58.20%	3.98	3.98
6	6	327.70	51.42	57.53%	5.89	5.89
8	8	257.95	65.33	54.82%	7.49	7.49

Figure 5: Performance Data for an Example MD Program. The results for each configuration (row) are calculated by averaging the execution times of 10 separate runs.

For an entire QS20 chip, this means that the sequential code can reach, at most, approximately 56.7 GFlops/sec. When we add in all the overhead actually encountered by the program (such as DMA overheads, runtime overheads, etc.) we actually observe 50.1 GFlops/sec, or 24.46% of the chip's peak GFlops/sec. This is approximately 88.4% of the maximum performance one could achieve if the SPEs had infinite memory and could continuously run the pair compute code. This shows that our framework allows the application code to effectively utilize the SPEs with little additional overhead.

Another more general point to consider is a comparison between the peak GFlops/sec for each type of core. A single QS20 Cell has peak rate of 204.8 GFlops/sec achievable by issuing one SIMD fused-multiply-add (MADD) each cycle. A single x86 core has a peak rate of 14.9 GFlops/sec achievable by dual-issuing one SIMD multiply (MUL) and one SIMD addition (ADD) in a single cycle. This limits the possible speedup of any application using SIMD instructions to 13.76 on a QS20 Cell when compared to a single x86 core. However, it is sometimes difficult for application code to take advantage of MADD instructions which require related multiply and addition operations. This is not the case for the x86 core since the MUL and ADD instructions do not have to operate on related data.[5] For any code that is not able to take advantage of the MADD instructions, the maximum speedup achievable by the code is automatically halved to 6.88 when moving from a single x86 core to a QS20 Cell processor. The example MD program presented here falls into this category of applications. The pair compute code in our MD example program only has 2 MADD instructions per 29 SIMD instructions which will nearly halve the maximum speedup for the MD example program on Cell compared to a single x86 core (to an estimated 7.36,[6] since 7% of SIMD instructions are MADD instructions). The actual observed speedup of the example MD program including all overheads is 5.74.

5.4 Visualizing the Program

The accelerated entry methods also work with Projections [13]. Projections is a performance visualization tool that can be used to visualize various aspects of a Charm++ program. Figure 6 shows one of the supported views, called a *timeline graph*. In a timeline graph, each core (PPE, SPE, and/or x86 core) is represented by a separate row. For example, the rows labeled as PE 0-3 in the upper portion of figure 6 are the four PPEs in the 4 QS20 hardware configuration. The rows labeled as PE 1-8 in the lower portion of figure 6 are the eight SPEs associated with PE 1 in the upper portion of the figure (i.e. the SPEs on the second of four Cell chips).

A timeline graph is a detailed view of what each core is doing as the program progresses. For each row or core in figure 6, time increases from left to right. Figure 6 show the computation on the cores for almost seven timesteps. Each block of color on a row represents the execution of a single entry method on that associated core. In the case of the SPEs, each entry method is an accelerated entry method. The thinner white blocks represent idle time[7] (the PPEs are idle for the majority of the time). Black represents time that the core is doing something related to the runtime system. The timeline graph, along with other graphs that are available in Projections, can give the programmer deeper insight into various performance characteristics of a program.

All Charm++ programs, including programs that use accelerated entry methods, can make use of the Projections tool

[5] In codes where the data is related, techniques such as instruction scheduling and software pipelining can be used to hide the latency between the data dependent instructions.

[6] (13.76 * 0.07) + ((13.76/2) * 0.93)

[7] Currently, for SPEs, both idle time and runtime overheads are considered the same thing and are displayed as black.

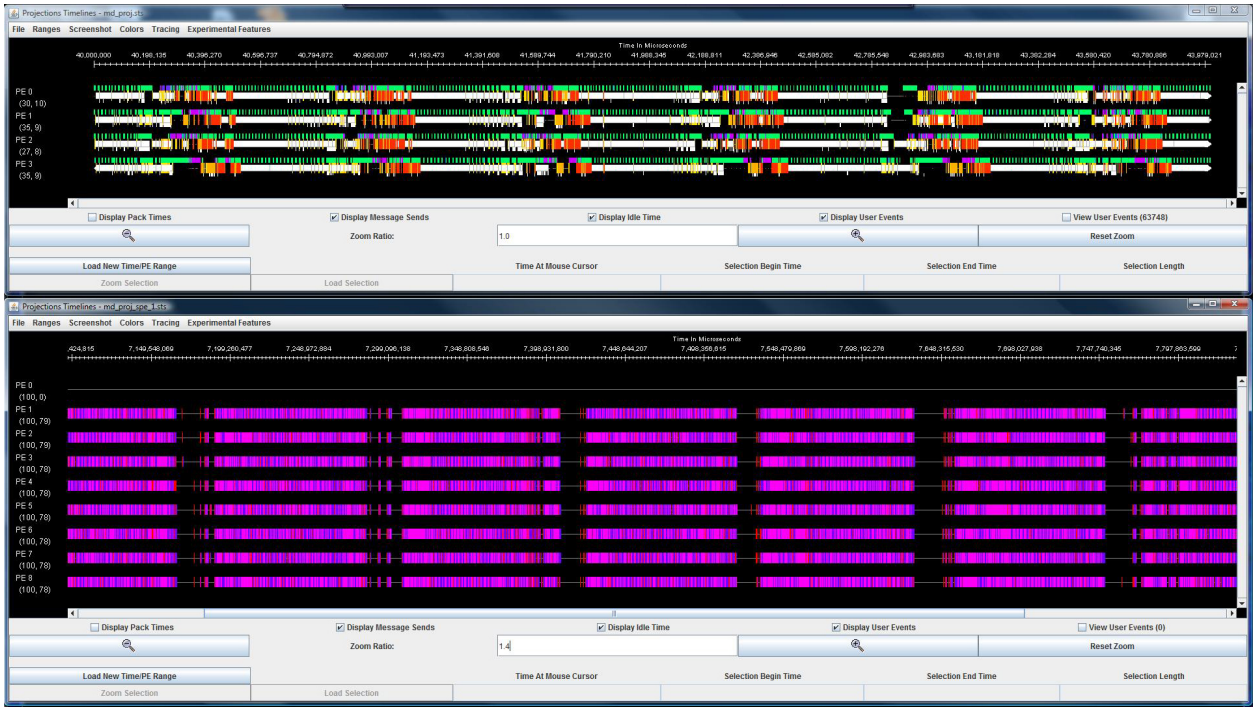

Figure 6: Screenshot of timeline graph used when analyzing performance of the MD example code. The upper window shows all four of the PPEs while the lower window shows the eight SPEs associated with PPE 1 (labeled as *PE 1* in the upper window).

by simply linking an extra module to the application during link time. No additional effort is required by the programmer to allow Projections to collect performance data related to the entry methods. The code can be further instrumented to get sub-entry-method information giving performance data related to specific sections of code within their entry methods.

6. CONCLUSIONS

In this paper, we have presented a high level overview of the extensions we made to the Charm++ programming model. The extensions include *accelerated entry methods*, *accelerated blocks*, and a SIMD instruction abstraction. When using these extensions, a programmer is able to easily utilize the SPEs on a single Cell processors or a cluster of Cell processors. Furthermore, code written with these extensions is portable between various processor architectures (x86-based and Cell-based systems presented here) without changing the application's source code. We also demonstrated that our extensions are able to achieve good performance by presenting an example molecular dynamics program which utilizes these new extensions. The example MD program, running on a single QS20 Cell, reached 50.1 GFlops/sec of the maximum 56.7 GFlops/sec possible for this application running sequentially on an SPE with infinite memory, as discussed in section 5.3.

Beyond portability and performance, our extensions allow a programmer to write code that can make use of available accelerators in a modular way. Independently written pieces of code written using our extensions can make use of accelerated entry methods. Each portion of code can make use of accelerated entry methods regardless of whether or not the other portions of code are also doing so. At runtime, after the pieces of code have been linked together to form a single application, the runtime system can interleave the accelerated entry methods on the available accelerators. Furthermore, accelerated entry methods are compatible with Projections. As a Charm++ application executes, the runtime system can be instructed to gather detailed performance information. An application developer can later analyze the performance data using the Projections visualization tool to better understand the performance characteristics of the application, including accelerated entry methods executing on accelerators.

7. FUTURE WORK

The framework extensions added and the model in general are still under development. For example, one additional feature we would like to add is the ability of the runtime system to automatically load balance work between host cores and accelerator cores. Currently, on Cell-based platforms the accelerated entry methods are compiled for both the PPEs and SPEs. We are working on adapting the runtime system to use dynamic performance information to direct a fraction of the accelerated work to the PPEs when the PPEs have idle time to spare, potentially providing a small speedup.

We are also working towards being able to execute Charm++ programs on heterogeneous systems. The example Charm++ programs that use accelerated entry methods can already be executed on a heterogeneous cluster comprised of Cell-based and x86-based nodes with the x86 cores and the PPE cores

being treated as peers. The goal of this work is to allow for the development of Charm++ programs on heterogeneous clusters such as Roadrunner [2] at Los Alamos National Lab. However, the Charm++ load balancing framework does not currently 'understand' some architectural differences between cores.

8. ACKNOWLEDGMENTS

We would like to thank Gengbin Zheng, Lukasz Wesolowski, Eric Bohm, Aaron Becker, Isaac Dooley, and Chao Mei of the Parallel Programming Lab for their assistance with this work. We would also like to thank IBM for the SUR grant awarded to the University of Illinois which provided the Cell hardware used for work presented in this paper. The work was supported in part by the NIH grant PHS 5 P41 RR05969-04.

9. REFERENCES

[1] Barcelona Supercomputing Center. *SMP Superscalar (SMPSs) User's Manual*, July 2007. http://www.bsc.es/media/1002.pdf.

[2] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. 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.

[3] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the ACM/IEEE SC 2006 Conference*, November 2006.

[4] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale. Overcoming Scaling Challenges in Biomolecular Simulations across Multiple Platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[5] E. Bohm, A. Bhatele, L. V. Kale, M. E. Tuckerman, S. Kumar, J. A. Gunnels, and G. 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.

[6] B. Bouzas, R. Cooper, J. Greene, M. Pepe, and M. J. Prelle. MultiCore Framework: An API for Programming Heterogeneous Multicore Processors. Mercury Computer System's Literature Library (http://www.mc.com/mediacenter/litlibrarylist.aspx).

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

[8] A. E. Eichenberger, K. O'Brien, K. 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, and M. 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.

[9] K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R. Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[10] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn. Massively Parallel Cosmological Simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[11] 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, 2005.

[12] L. V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, Madrid, Spain, February 2004.

[13] L. V. Kale, G. Zheng, C. W. Lee, and S. 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.

[14] D. Kunzman. Charm++ on the Cell Processor. Master's thesis, Dept. of Computer Science, University of Illinois, 2006. http://charm.cs.uiuc.edu/papers/KunzmanMSThesis06.shtml.

[15] D. Kunzman, G. Zheng, E. Bohm, and L. 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.

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

[17] 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.

[18] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.