Charm++ for Productivity and Performance

A Submission to the 2011 HPC Class II Challenge

Laxmikant V. Kale‡

Anshu Arya, Abhinav Bhatele, Abhishek Gupta, Nikhil Jain, Pritish Jetley, Jonathan Lifflander, Phil Miller, Yanhua Sun, Ramprasad Venkataraman‡, Lukasz Wesolowski, Gengbin Zheng

Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801
‡{kale, ramv}@illinois.edu

We present our implementation of the HPC Challenge Class II (productivity) benchmarks in the Charm++ [1] programming paradigm[1]. Our submission focuses on explaining how over-decomposed, message-driven, migratable objects enhance the clarity of expression of parallel programs and also enable the runtime system to deliver portable performance. Our submission includes implementations of three required benchmarks: Dense LU Factorization, FFT, and Random Access. We also include two additional benchmarks that represent relevant scientific computing algorithms of some complexity: Molecular Dynamics and Barnes-Hut. We believe our implementations demonstrate that a high-level productivity oriented model can also deliver portable performance via an intelligent runtime. Our code-size results can be seen in Table 1 and a summary of the performance metrics can be seen in Table 2.

1. PROGRAMMING MODEL

We describe relevant aspects of the Charm++ programming model in order to set the context for explaining the benchmark implementations.

1.1 Salient Features

1.1.1 *Object-based*

Parallel programs in Charm++ are implemented in an object-based paradigm. Computations are expressed in terms of work and data units that are natural to the algorithm being implemented and not in terms of physical cores or processes executing in a parallel context. This immediately has productivity benefits as application programmers can now think in terms that are native to their domains.

The work and data units in a program are C++ objects, and hence, the program design can exploit all the benefits of object-oriented software architecture. Classes that participate in the expression of parallel control flow (*chares*) inherit from base classes supplied by the programming framework. They also identify the subset of their public methods that are remotely invocable (*entry methods*). This is done in a separate interface definition file described in subsection 1.2. Charm++ messages can also be C++ classes defined by the program. Any C++ class (be it a work unit, data unit, or message) that is transmitted across processes has to define a serialization operator, called `pup()` for Pack/UnPack. This allows the transmission of complex message types, as well as the movement of work / data units across processes during execution.

Chares are typically organized into indexed collections, known as *chare arrays.* Chares in an array share a type, and hence present a common interface of *entry methods*. A single name identifies the entire collection, and individual elements are invoked by subscripting that name. Code can broadcast a message to a common *entry method* on all elements of an array by invoking that method on the array's name, without specifying a subscript. Conversely, the elements of an array can perform asynchronous reductions whose results can be delivered to an *entry method* of the array itself or of any other object in the system.

In essence, Charm++ programs are C++ programs where interactions with remote objects is realized through an inheritance and serialization API exposed by the runtime framework.

1.1.2 *Message-Driven*

Messaging in Charm++ is one-sided, sender-driven and asynchronous. Parallel control flow in Charm++ is expressed in the form of method invocations on remote objects. These invocations are generally *asynchronous*, in that control returns to the caller before commencement or completion of the callee, and thus no return value is available. If data needs to be returned, it can flow via subsequent remote invocation by the callee on the original caller, be indirected through a callback, or the call can explicitly be made synchronous.

These semantics immediately fit well into the object-based paradigm. Each logical component (*chare*) simply uses its *entry methods* to describe its reactions when dependencies (remote events or receipt of remote data) are fulfilled. It is notified when these dependencies are met via remote invocations of its *entry methods*. Upon such invocation, it can perform appropriate computations and also trigger other events (*entry methods*) whose dependencies are now fulfilled. The parallel program then becomes a collection of objects that trigger each other via remote (or local) invocations by sending messages. Note that these *chares* do not have to

[1] This document was released by Lawrence Livermore National Laboratory for an external audience under release number LLNL-MI-508671.

Code	C++	CI	Working Subtotal	Driver	Total
LU	1231	418	1649	476	2125
FFT	112	47	159	22	181
RandomAccess	155	23	178	n/a	178
MD	645	128	773	n/a	773
Barnes-Hut	2871	56	2927	9861	13130

Table 1: Lines of code in each benchmark. 'C++' and 'CI' refer to Charm++ code necessary to solve the specified problem. 'Driver' refers to additional code used for setup and verification. All numbers were generated using David Wheeler's SLOCCount.

Code	Machine	Max Cores	Best Performance
LU	Cray XT5	8K	67.4% of peak
FFT	IBM BG/P	64K	2.512 TFlop/s
RandomAccess	IBM BG/P	64K	22.19 GUPS
MD	Cray XE6	16K	1.9 ms/step (125K atoms)
	IBM BG/P	64K	11.6 ms/step (1M atoms)
Barnes-Hut	IBM BG/P	16K	27×10^9 interactions/s

Table 2: Performance summary: largest scale of execution and the best performance achieved

explicitly expect a message arrival by posting receives. Instead, the arrival of messages triggers further computation. The model is hence message-driven.

Its worthwhile to note that the notion of processes / cores has not entered this description of the parallel control flow at all. This effectively separates the algorithm from any processor-level considerations that the program may have to deal with; making it easier for domain experts to express parallel logic.

1.1.3 *Over-decomposed*

Divorcing the expression of parallelism (work and data units) from the notion of processes / cores allows Charm++ programs to express much more parallelism than the available number of processors in a parallel context. The fundamental thesis of the Charm++ model is that the application programmer should over-decompose the computations in units natural to the algorithm, thereby creating an abundance of parallelism that can be exploited.

1.1.4 *Runtime-Assisted*

Once an application has been expressed as a set of over-decomposed *chares* that drive each other via messages, these can be *mapped* onto the available compute resources and their executions managed by a runtime system. The programming model permits an execution model where the runtime system can:

- maintain a queue of incoming messages, and deliver them to *entry methods* on local *chares*.

- overlap data movement required by a *chare* with *entry method* executions for other chares.

- observe computation / communication patterns, and move *chares* to balance load and optimize communication.

- allow run-time composition (interleaving) of work from different parallel modules.

A list of advantages of the Charm++ programming and execution model can be found at `http://charm.cs.illinois.edu/why/`.

1.2 Interface Definitions

The Charm++ environment strives to provide a convenient, high-level interface for parallel programmers to use in developing their applications. Rather than requiring programmers to do the cumbersome and error-prone bookkeeping necessary to identify the object and message types and associated *entry methods* that make up their program, Charm++ provides code generation mechanisms to serve this purpose. When building a Charm++ program, the developer writes one or more interface description files (named with a `.ci` extension), listing the following:

1. System-wide global variables set at startup, known as *read-only* variables,

2. Types of messages that the code will explicitly construct and fill, specifying variable-length arrays within those messages, and

3. Types of *chare* classes, including the prototypes of their *entry methods*.

Interface descriptions can be decomposed into several modules that make up the overall system, generated declarations and definitions for each of which will be placed in a separate file (named with `.decl.h` and `.def.h` extensions, respectively). A Charm++ program's starting point, equivalent to `main()` in a C program, is marked as a *mainchare* in its interface description. The runtime starts the program by passing the command-line arguments to the mainchare's constructor.

1.3 Structured Dagger

One effect of describing a parallel algorithm in terms of individual asynchronous *entry methods* is that the overall control flow is split into several pieces, whose relationship can become complicated in any but the simplest program. For instance, one pattern that we have observed is that objects will receive a series of similar messages that require some individual processing, but overall execution cannot proceed until all of them have been received. Another pattern is some *entry method* must execute in a particular sequence, even if their triggering messages may arrive in any order. To facilitate expressing these and other types of higher-level control flow within each object, Charm++ provides a syntax known as Structured Dagger [2]. Structured Dagger lets the programmer describe a sequence of behaviors in an object that can be abstractly thought of as a local task dependence DAG (from which the name arises).

The basic constructs in Structured Dagger are drawn from the traditional control-flow constructs in C-like languages: conditional execution based on `if/else`, and `for` and `while` loops. The constructs operate as in common sequential C or C++ code. In addition to basic C constructs, Structured Dagger provides the `when` clause to indicate that execution beyond that point depends on the receipt of a particular message. Each `when` clause specifies the *entry method* it is waiting for, and the names of its argument(s) in the body of the clause. A `when` clause may optionally specify a *reference number* or tag expression limiting which invocations of the associated *entry method* will actually satisfy the clause. A reference number expression appears in square brackets between an *entry method 's* name and its argument list. Definitions for *entry methods* named in `when` clauses are generated by the interface translator. Though not used in this submission, Structured Dagger also provides the overlap construct for divergent control flow, akin to a task fork/join mechanism specifying that a sequence of operations must complete, but not an ordering between them. Because a chare is assigned to a particular core at any given time, these operations do not execute in parallel, avoiding many problems encountered by concurrent code.

The various Structured Dagger constructs are implemented in generated code by a collection of message buffers and trigger lists, and have negligible overhead, comparable to a few static function calls. If the body of a Structured Dagger method is viewed as a sequential thread of execution, a `when` can be seen as a blocking receive call. However, rather than preserving a full thread stack and execution context, Structured Dagger saves just the current control-flow location, and returns to the scheduler. Local variables to be preserved across Structured Dagger blocks are encoded as member variables of the enclosing chare. In contrast, thread context switches to provide the same behavior with a less sophisticated mechanism can be much more costly. Charm++ does provide mechanisms to run *entry methods* in separate threads when blocking semantics are needed for other purposes.

2. DENSE LU FACTORIZATION

2.1 Productivity

Charm++ is a general and fully capable programming paradigm and our LU implementation does not employ any linear algebra specific notations. Our implementation is very succinct and presents several distinct benefits.

2.1.1 Composable Library

The implementation is presented as a modular library that can be composed into client applications. By composition, we imply both modularity during program design and seamlessness during parallel execution. Factorization steps can be interleaved with other computations from the application (or even with other factorization instances of the same library!) on the same set of processors.

2.1.2 Flexible Data Placement

The placement of matrix blocks on processes is completely independent of the main factorization routines; is encapsulated in a sequential function, and can be modified with minimal effort.

2.1.3 Block-Centric Control Flow

For block sizes on which `dgemm` calls perform well, we typically need hundreds of blocks assigned to each processor core to meet the memory usage requirements. Such over-decomposition is also necessary for load balance. By elevating these over-decomposed, *logical*, entities to become the primary players in the expression of parallelism, Charm++ enables a succinct representation of the factorization logic. Additionally, Structured Dagger allows the control flow for each block to be directly visible in the code in a linear style effectively independent of other activity on the same processor.

2.1.4 Separation of Concerns

The factorization algorithm has been expressed from the perspective of a matrix block. However, processor-level considerations (e.g, memory management) are implemented as separate logic that interacts minimally with the factorization code. This demonstrates a clear separation of concerns between application-centric domain logic and system-centric logic. Such separation enhances productivity of both application domain experts and parallel systems programmers. It also allows easier maintenance and tuning of parallel programs.

2.2 Performance

We have scaled our implementation to 8064 cores on Jaguar (Cray XT5 with 12 cores and 16GB per node) by increasing problem sizes to occupy a constant fraction of memory (75%) as we increased the number of cores used. We obtain a constant 67% of peak across this range. We also demonstrate strong scaling by running a fixed matrix size ($n = 96,000$) from 256 to 4096 cores of Intrepid (IBM Blue Gene/P with 4 cores and 2GB per node). The matrix size was chosen to just meet the requirements of the spec (occupying 54% of memory) at the lower end of the strong scaling curve (256 cores). Our results are presented in Figure 1. Extensive experiments with LU are an expensive proposition as the amount of computation increases as n^3 (where n is the matrix size). We fully expect the implementation to scale to much larger partitions and demonstrate high performance on multiple architectures.

2.2.1 Adaptive Lookahead

Our implementation provides completely dynamic, memory-constrained lookahead so that panel factorizations are over-

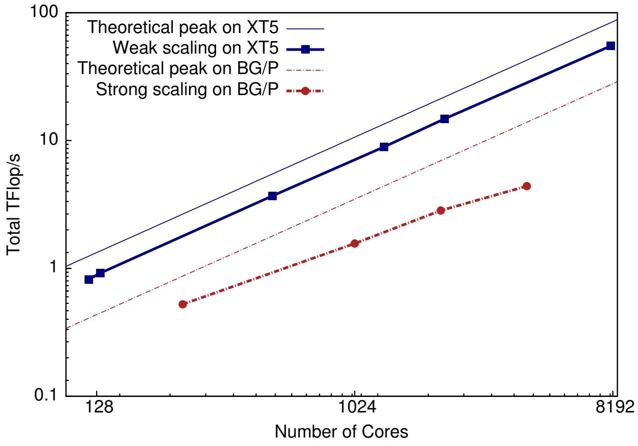

Figure 1: Weak scaling (matrix occupies 75% of memory) from 120 to 8064 processors on Jaguar (Cray XT5). Strong scaling ($n = 96,000$) from 256 to 4096 processors on Intrepid (IBM BG/P).

lapped as much as memory usage limits will allow. In keeping with its library form, applications can choose to restrict the factorization to use only a fraction of the available memory.

2.2.2 *Asynchronous Collectives*

Charm++ collective operations are also asynchronous like its other messaging semantics and can be overlapped with other work. For example, this allows asynchronous pivot identification reductions to be overlapped with updating the rest of the sub-panel. Masking such latencies allows this implementation a wider choice of data placements.

2.3 Implementation

We use a typical block-decomposed algorithm for the LU factorization process. Our focus in this effort has been less on choosing the best possible factorization algorithm than on demonstrating productivity with a reasonable choice.

The input matrix of $n \times n$ elements is decomposed into square blocks of $b \times b$ elements each. We delegate underlying sequential operations to an available high performance linear algebra library, typically a platform-specific implementation of BLAS and perform partial pivoting for each matrix column.

2.3.1 *Data Distribution*

Matrix blocks are assigned to processors when the library starts up, according to a *mapping* scheme, and are not reassigned during the course of execution. Charm++ facilitates the expression and modification of the data distribution scheme by encapsulating the logic into a simple sequential function call that uses the block's coordinates to compute the process rank it should be placed on:

$$process\ rank = f(x_{block}, y_{block}) \qquad (1)$$

This is a standard feature in Charm++ and is available to all indexed collections of *chares* (chare arrays). This allows library users to evaluate data distribution schemes that may differ from the traditional two-dimensional block-cyclic format.

2.3.2 *Asynchrony and Overlap*

In our implementation, each block is placed in a message-driven object, driven by coordination code written in Structured Dagger [2]. The coordination code describes the message dependencies and control flow from the perspective of a block. Thus, every block can independently advance as it receives data and we avoid bulk synchrony by allowing progress in the factorization when dependencies have been met. With many blocks per processor, overlap is automatically provided by the Charm++ runtime system.

2.3.3 *Block Scheduler*

Our solver implements dynamic lookahead, using a dynamic *pull-based* scheme to constrain memory consumption below a given threshold. To implement the pull-based scheme, we place a *scheduler object* on each processor in addition to its assigned blocks. The scheduler object maintains a list of the blocks assigned to its processor, and tracks what step they have reached. Within the bounds of the memory threshold, it requests blocks from remote processors that are needed for local triangular solves and trailing updates. An earlier technical report [3] describes the dependencies between the blocks and how the scheduler object uses this to safely reorder the selection of trailing updates to execute. We include this in our submission to demonstrate that the programming model allows separation of concerns in a parallel context.

2.4 Verification

Our LU implementation conforms fully to the spec and passes the required validation procedures for all the results presented here. We have supplied a test driver with the library that generates input matrices, invokes the library for the factorization and solves, and validates the results while also measuring performance. Performance and validation statistics are printed at the end.

3. GLOBAL FFT

3.1 Productivity

Charm's virtualization allows us to trivially over-decompose a parallel FFT and take advantage of overlapping communication and computation by changing an input parameter. At core counts less than 1024, some amount of over-decomposition improves performance by up to 50% with no modification to the code. At higher core counts, virtualization results in smaller serial FFTs and slower performance, so we simply choose a decomposition equivalent to the number of processors. Additional code to split messages or explicitly handle overlapping communication and computation is unnecessary.

3.2 Performance

Runs were performed on the ALCF Surveyor Blue Gene/P machine. Performance scales well up to one mid-plane (2048 cores) and then stagnates due use of a simple point-to-point communication pattern. However, software routing via the Mesh Streaming library in Charm allows the FFT to scale up to at least 64k cores. ESSL was used to perform serial FFTs. Figure 2 and table 3.2 summarize the results.

3.3 Implementation

Our implementation of Global FFT takes input size N and performs a complex 1D FFT on an NxN matrix where

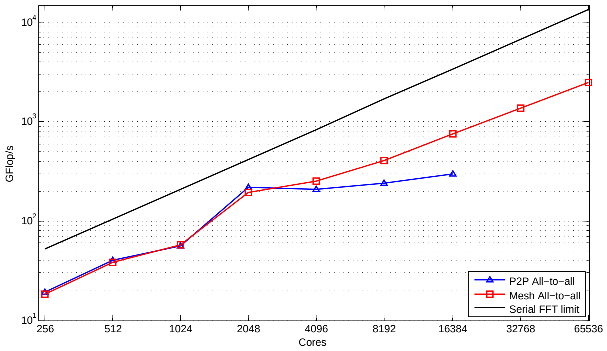

Figure 2: Performance of Global FFT on Surveyor (IBM BG/P)

Cores	Gflops	Size
256	19.386	32768^2
512	39.897	46080^2
1024	57.315	65536^2
2048	219.155	92160^2
4096	252.337	131072^2
8192	409.359	180224^2
16384	756.228	262144^2
32768	1379.728	360448^2
65536	2512.835	524288^2

Table 3: Global FFT - Results from Surveyor (BG/P)

subsequent rows are contiguous data elements of a double precision complex vector. Three all-to-all transposes are required to perform the FFT and unscramble the data. All-to-all operations are executed via point-to-point messages and external libraries (FFTW or ESSL) perform serial FFTs on the rows of the matrix.

3.4 Verification

Verification is performed via two methods: (1) the benchmark code self-validates by executing an inverse FFT on the output result and (2) the output is dumped to a file and an inverse FFT is performed on the dumped data by a separate MPI program that invokes parallel FFTW.

4. RANDOM ACCESS

4.1 Productivity

4.1.1 Quiescence Detection

Implementations of RandomAccess typically contain code for determining when all updates are completed globally. This task is simplified in Charm++ by using its built in quiescence detection feature, which monitors the global number of messages sent and delivered and identifies the point when all messages have been delivered. The interface to detect quiescence in Charm++ consists of a single call which specifies the method to execute upon reaching a quiescent state.

4.1.2 Automatic Topology Determination

The Charm++ Topology Manager [4] automates the task of determining physical network topology for a partition assigned to a job. We use it to provide topology information for a routing and message aggregation library.

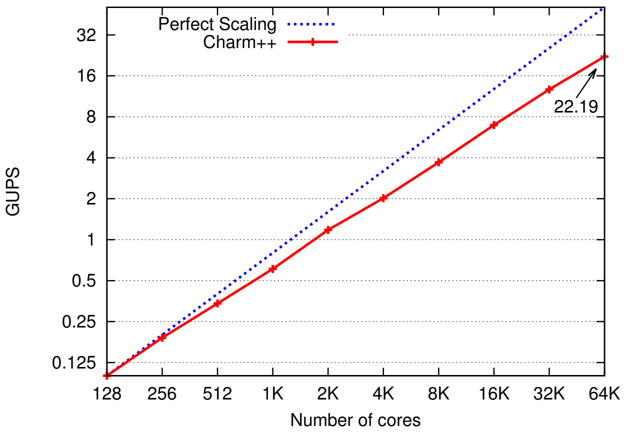

Figure 3: Performance of random access on Intrepid (IBM BG/P)

Cores	GUPS	Memory (TB)
128	0.10	0.031
256	0.19	0.062
512	0.34	0.125
1024	0.61	0.250
2048	1.18	0.500
4096	2.02	1.000
8192	3.70	2.000
16384	6.96	4.000
32768	12.69	8.00
65536	22.19	16.00

Table 4: Performance of random access on Intrepid (IBM BG/P)

4.1.3 Communication Libraries

Charm++ contains libraries for improving network communication performance for various scenarios. For this benchmark we use the Mesh Streamer library, further described below, for optimizing all-to-all communication on small messages.

4.2 Performance

The results on Blue Gene/P are presented in Figure 3 as well as in Table 4.

4.3 Implementation

Using Charm++ the global table is evenly distributed across all nodes. Each core is responsible for generating a subset of the random numbers and updating its local table. The notion of a group in Charm++, which can be thought of as a chare array with one chare per core, is used, where the global table is distributed among the chares in the group. Each group member takes charge of generating the 64-bit numbers and updating its local table in turns. The Charm++ runtime system controls the sending and receiving of messages. Termination of the updating process is carried out by using Charm++ quiescence detection, which helps to simplify the code.

The small size of individual data items in this benchmark makes it prohibitively expensive to send each item as a separate message. To improve performance, we use a Charm++ message aggregation and routing library called

Mesh Streamer. Mesh Streamer assumes a three dimensional mesh layout of the processors. The separate Charm++ Topology Manager library assigns dimensions of this mesh to match the physical network topology of the partition assigned for the run. Each message submitted by the user to the library arrives at its destination processor after at most three communication steps, one along each dimension of the mesh. At each step, messages are buffered so that different messages traveling to the same plane, column, or specific processor are placed in the same buffer. For the purposes of this benchmark, Mesh Streamer was modified to buffer at most 1024 data elements at each core.

4.4 Verification

The verification is done by repeating the Random updates. In our implementation memory is distributed among the members of the group. Each region of memory is only accessed by one group member, making the implementation thread-safe. There were no errors found.

5. MOLECULAR DYNAMICS

LeanMD is a molecular dynamics simulation program written in Charm++. This benchmark simulates the behavior of atoms based on the Lennard-Jones potential, which is an effective potential that describes the interaction between two uncharged molecules or atoms. The computation performed in this code mimics the short-range non-bonded force calculation in NAMD [5, 6] and resembles the miniMD application in the Mantevo benchmark suite [7] maintained by Sandia National Laboratories.

The force calculation in Lennard-Jones dynamics is done within a cutoff-radius, r_c for every atom. In LeanMD, the computation is parallelized using a hybrid scheme of spatial and force decomposition. The three-dimensional (3D) simulation space consisting of atoms is divided into cells of dimensions that are equal to the sum of the cutoff distance, r_c and a margin. In each iteration, force calculations are done for all pairs of atoms that are within the cutoff distance. The force calculation for a pair of cells is assigned to a different set of objects called computes. Based on the forces sent by the computes, the cells perform the force integration and update various properties of their atoms – acceleration, velocity and positions.

5.1 Productivity

Our implementation of LeanMD takes only 773 lines of code while offering capabilities that are sometimes not matched by production molecular dynamics applications. In comparison, miniMD from the Mantevo benchmark suite, which nurtures similar objectives of representing real applications, requires just under 3000 lines of code [7] but does not offer many of the capabilities of LeanMD.

Below, we present several Charm++ features that have been exploited in LeanMD that significantly improve programmer productivity without sacrificing performance and in some cases, such as load balancing, lead to performance improvements.

5.1.1 *Dense and Sparse Charm++ Arrays*

Indexed collection of objects in Charm++ provides an elegant and easy to understand abstraction for representing dissimilar but related work units. Different phases and computation in an application can be assigned to different chare arrays. Cells are a dense 3D chare array that represent a spatial decomposition of the 3D simulation space. They are responsible for sending positions and collecting the forces for their atoms. Computes, on the other hand, form a sparse 6-dimensional array of chares. Such a representation makes it convenient for a pair of cells with coordinates (x1, y1, z1) and (x2, y2, z2) to use a compute with coordinates (x1, y1, z1, x2, y2, z2) to calculate forces for their atoms.

5.1.2 *Ability to run variable size jobs*

Charm++ enables users to run applications on any number of cores without any restrictions on the size or shape of the processor partitions that are used. Additionally, it provides the freedom to choose a convenient number of work units, depending on the simulation, independent of the number of cores the application is run on. Note that, this freedom does not come at the expense of performance which either improves or follows the general trend seen in the more restricted environment.

5.1.3 *Structured Dagger*

LeanMD requires substantial amount of control flow information to be exchanged between the two chare arrays, cells and computes. Structured Dagger provides a lucid and easy to use mechanism to trigger dependent events and maintain the application control flow. It obviates the need for user maintained flags, counters and temporary variables and vastly improves the readability of the code.

5.1.4 *Automatic Load Balancing*

A measurement-based load balancing framework in Charm++ enables strong scaling with minimal effort from the application programmer. It is critical in an application like LeanMD, which can suffer from substantial load imbalance because of the variation in sizes of computes resulting from a spherical cutoff. Measurement-based load balancing is well suited for applications where the recent past is a predictor of the near future. This works well for LeanMD because atoms migrate slowly and hence the load fluctuations are gradual.

5.1.5 *Communication Libraries*

Charm++ provides a set of communication libraries which supports efficient multicast and reduction operations. In each iteration of LeanMD, the atoms contained in a cell are sent to every compute that needs them. Also, the resultant forces on atoms in a cell are obtained by a summation of the forces calculated by each compute that received those atoms. LeanMD exploits the ability of the runtime to generate efficient spanning trees over arbitrary sets of processors.

5.2 Performance

We present performance numbers for LeanMD on two machines: 1) Intrepid - an IBM Blue Gene/P installation at ANL and 2) Hopper - the Cray XE6 at NERSC. LeanMD was run using two molecular systems - one consisting of $125,000$ atoms and a second larger system of 1 million atoms. Note that the runs on Hopper were done on non-power of two cores so as to use all the cores allocated to the job (on Hopper, allocation happens in multiples of 24 cores).

Figure 4 presents the results for the molecular system of $125,000$ atoms on Intrepid and Hopper. For this small system of atoms, without load balancing, we observe that the performance saturates at 1024 cores. However, if load bal-

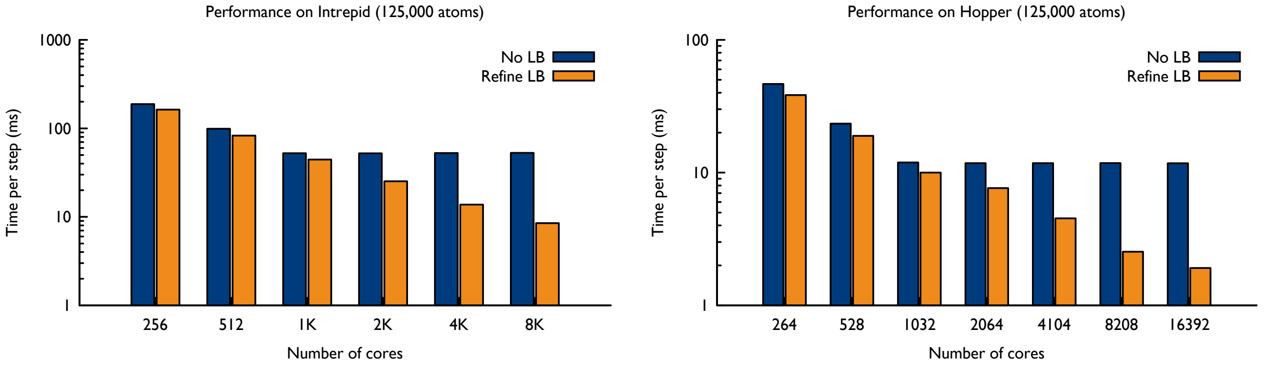

Figure 4: Performance of LeanMD for a $125,000$-atom system on Intrepid (IBM BG/P) and Hopper (Cray XE6)

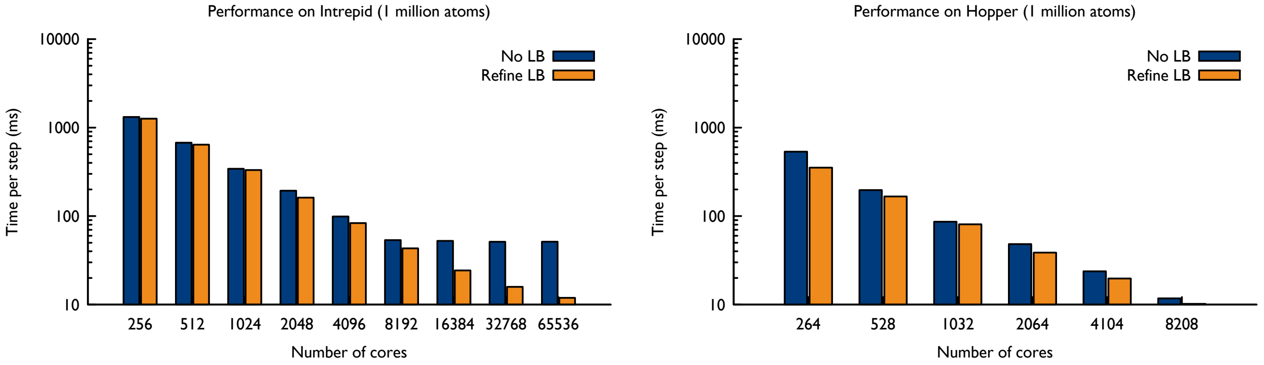

Figure 5: Performance of LeanMD for a 1 million atom system on Intrepid (IBM BG/P) and Hopper (Cray XE6)

ancing is used, the performance keeps on improving till 8192 cores after which the gains are small because of the relatively small size of the system. The parallel efficiency at 8192 cores compared to the 256-core performance is 60% on Intrepid and for a similar increase in the number of cores, it is 50% on Hopper. Note that, on Hopper, the time per step at 16392 cores is less than 2 ms which is very good considering that each step of LeanMD has two way data communication and a good amount of computation.

Performance results for the 1 million atom system are presented in Figure 5. On Intrepid, using load balancing, the performance of LeanMD improves by 52x (efficiency of 83%) as the number of cores is increased from 256 to 16384. Beyond $16,384$ cores, reduced performance gains are observed possibly due to the topology oblivious nature of load balancers used. The performance using load balancing shows a super linear speed up on Hopper as it improves by 33x as the number of cores increases by 32x. This can be attributed to a smaller work set per core that results in better cache performance.

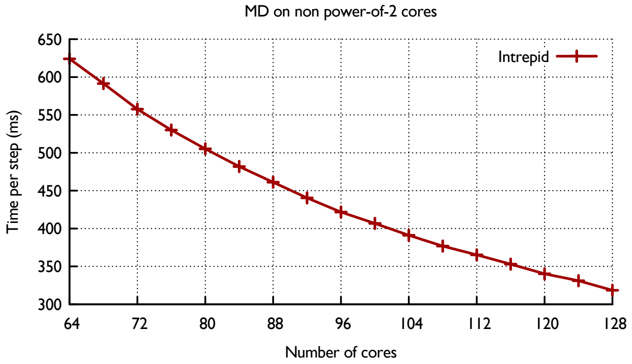

Figure 6: Performance of LeanMD for non power of 2 cores on Intrepid (IBM BG/P)

Figure 6 presents the performance numbers for running LeanMD on allocation with non-powers of two cores. The number of cores was increased from 64 to 128 with an interval of 4 cores between consecutive runs. Two important points are demonstrated via this experiment: 1) Charm++ can be used to run applications on arbitrary number of cores and 2) Charm++ preserves the performance characteristics of an application for any arbitrary number of cores.

5.3 Implementation

In the Charm++ implementation the computation is parallelized using a hybrid scheme of spatial and force decomposition. The three-dimensional (3D) simulation space consisting of atoms is divided into cells of dimensions that are equal to the sum of the cutoff distance, r_c and a margin.

In each iteration, force calculations are done for all pairs of atoms that are within the cutoff distance. The force calculation for a pair of cells is assigned to a different set of objects called computes.

At the beginning of each time step, every cell multicasts the positions of its atoms to the computes that need them for force calculations. Every compute receives positions from two cells and calculates the forces. These forces are sent back to the cells which update other properties of the atoms. Every few iterations, atoms are migrated among the cells based on their new positions. Structured Dagger is used to control the flow of operations in each iteration and trigger dependent events. The load balancing framework is invoked periodically after a certain number of iterations to redistribute computes and cells among the processors. In the submitted version, the parallel control flow is described in the `run` functions of each chare in `leanmd.ci`. The reduction for forces computed by computes is in `physics.h`.

5.4 Specification and Verification

For a pair of atoms, the force can be calculated based on the distance by,

$$\vec{F} = \left(\frac{A}{r^{13}} - \frac{B}{r^{7}}\right) \times \vec{r} \tag{2}$$

where A and B are Van der Waals constants and r is the distance between the pair of atoms. Table 5 lists a set of parameters and their values used in LeanMD.

Parameter	Values
A	1.6069×10^{-134}
B	1.0310×10^{-77}
Atoms per cell	150
Cutoff distance, r_c	12 Å
Cell Margin	2 Å
Time step	1 femtosecond

Table 5: Simulation details for LeanMD

The benchmark computes kinetic and potential energy and uses the principle of conservation of energy to verify that the computations are stable. Users can choose to run the benchmark for as many timesteps as desired, and verification statistics are printed at the end.

6. BARNES-HUT

The N-body problem involves the numerical calculation of the trajectories of N point masses (or charges) moving under the influence of a conservative force field such as that induced by gravity (or electrical charges). In its simplest form, the method models bodies as *particles* of zero extent moving in a collision-less manner. The objective is to calculate the net force incident on every particle at discrete time steps. These forces are then used to update the velocity and position of each particle, leading into the next time step, where the net force on each particle is calculated once more, etc. In general, the force may be long-range in nature (as is the case with gravity), so that interactions between distant particles must also be calculated. Thus, in order to obtain a good approximation to the actual solution of a system, $O(N^2)$ computations must be performed. Given its quadratic complexity, the amount of work done by this *all-pairs* method makes it infeasible for systems with large N.

Barnes and Hut [8] devised a hierarchical N-body method that performs significantly fewer computations but at the cost of a greater relative error in the computed solution. The method relies on the spatial partitioning of the input system of particles, thereby imposing a tree-structure on it. Particles that are close to each other in space are grouped into closely related nodes of the tree. This allows the approximation of forces on a particle due to a *distant group* of particles through the multipole moments of that group. Note that applying such an approximation to points relatively close to the group will result in gross errors of calculation. In such a case, sub-partitions within the group are tested for proximity to the point. This technique, applied systematically, yields an expected complexity of $O(NlgN)$, making it suitable for large systems of particles.

6.1 Implementation

Below we describe the structure of the Barnes-Hut method in greater detail. We also discuss the *distributed* tree data structure used to partition particles and detail its construction.

6.1.1 Space partitioning trees

The Barnes-Hut algorithm relies on the organization of particles into a spatial tree. The leaves of such a tree represent particles, or small groups of particles called *buckets*. Each node of the tree represents a spatial partition enclosing a certain number of particles. Space partitioning trees may be constructed to have different properties and structures. For example the *k*d-tree attempts to balance the number of particles within each child of a parent node. Here, we focus on the distributed binary *space* partitioning tree as the underlying data structure for the Barnes-Hut algorithm. This data structure has several desirable properties, such as good aspect ratio of partitions (leading to a reduction in the surface area per unit volume, and therefore total communication) and flexibility in deciding the communication grain size. However, by itself the binary space partitioning tree does not guarantee an even distribution of particles or computational load across partitions. The tree is constructed recursively in the following fashion. Given a node that represents a particular partition of space, if the node has more than a threshold number of particles, it is split along an axis (the axis is chosen in a round-robin fashion) to create two new children of equal size, maintaining the axis-aligned nature of the children.

When constructing the tree in parallel on a distributed memory machine, it is customary to divide the procedure into two distinct phases, namely *domain decomposition* and *tree construction*. In the Charm++ implementation that we discuss here, both phases are managed by a PE-level entity (i.e. a *group*) called the DataManager. Therefore, we will refer to DataManagers (DMs) and PEs interchangeably.

6.1.2 Domain decomposition

The objective of the domain decomposition phase is to assign particles to members of a one-dimensional chare array of *TreePieces*. This step is similar to the sorting of keys in parallel, both in structure and effect. In fact, the iterative *master-worker* structure used here closely resembles that of histogram sort [9]. Each PE (i.e. DM) begins by loading its share of particles from an input file. The PE then performs a local sort on its particles. An iterative distributed histogramming phase follows, in which a master PE obtains the total number of particles in each *active* node. A node is active if the histogramming procedure is currently determining the number of particles in it through a global reduction. The reduction operation is done on arrays of counts contributed by worker PEs. If it is determined that an active node has more than a threshold number of particles, it is removed from the active list and its children are made active. This new list of active nodes is broadcast to the workers, each of which performs a local partitioning of the parents' particles among the new, active children. On the other hand, if a node is determined to be within the user-specified threshold of particles, it is removed from the active list and a corresponding TreePiece is created. At this point, each PE *flushes* the subset of its particles that lie beneath that node to the TreePiece. When there are no active nodes remaining, domain decomposition is complete. The master also broadcasts a list of key ranges to the workers, thereby informing all PEs of the range of particles held by each TreePiece. This information is needed in order to label the nodes of a local tree with *ownership* data in the tree construction phase (described below).

6.1.3 *Tree construction*

Upon receiving all the particles intended for it, each TreePiece submits its particles to the DataManager on its PE. This allows all particles on a PE to be agglomerated by the DataManager, resulting in a larger locally accessible tree for all TreePieces on the PE. The DataManager calculates the moments (in our simple implementation these are the total mass and center of mass) of all nodes that are exclusively on its PE. However, in general each PE holds only a subset of the particles in the system, so that the global tree cannot be recreated in its entirety on any PE. Therefore, in order to enable access to remote portions of the tree the DataManager annotates it with ownership information. In effect, the information about the (disjoint) range of particles held by each TreePiece is used to calculate the range of owners of each node. Note that nodes, especially those at shallower levels of the tree, can be shared among many TreePieces since they may enclose particles assigned to a number of TreePieces. In particular, the root is shared by all TreePieces.

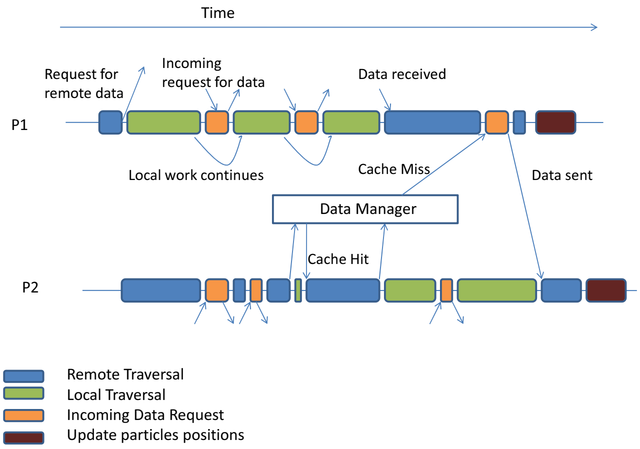

Figure 7: Time Progression view showing 2 processors executing an iteration of Barnes-Hut

6.1.4 *Tree traversal*

The computation of gravitational forces is preceded by a traversal of the distributed Barnes-Hut tree by each TreePiece. The traversal can be defined recursively, and is at the heart of the $O(N lg N)$ complexity of the algorithm. Given a target particle, on which net force is to be calculated, and a source node (initially the root), the traversal checks whether the distance between the particle and node is large enough to apply the Barnes-Hut approximation. If so, an interaction between the particle and the moments of the node is computed and the node is discarded. If the node is not far enough, the children of the node are considered in turn. In the Charm++ version of the code the DataManager provides TreePieces with seamless access to remote nodes. If the node is a leaf, pairwise interactions between its particles and the target are performed. In our implementation, the cost of traversing the tree is amortized over several local particles by grouping them into buckets. Of course, by group particles into buckets, we are forcing interactions that needn't be performed, but this extra computation is insignificant when compared to the benefit of reduced memory accesses due to fewer traversals. For each bucket of local particles the global tree is traversed in two parts – a *local* traversal is conducted on that portion of the Barnes-Hut tree that is local to the PE whereas a *remote* traversal operates on the remainder of the distributed Barnes-Hut tree, leading to communication between tree pieces in the form of requests for remote nodes and particles. By assigning greater precedence to remote traversals than local ones, we can use the Charm++ feature of **automatic computation-communication overlap** to accelerate the critical path: in effect, the latency of high-priority remote data communication can be overlapped with low-priority local work. This effect is illustrated by figure 7.

Requests for remote data are funneled through the DataManager so as to merge requests for the same particles and nodes from local TreePieces. This optimization in itself can substantially reduce the volume of communication. We combine this technique with a software-managed remote data cache to increase the amount of data reuse. Upon receipt of remote data, the DataManager maintains a copy of the data so that it may be reused by other TreePieces that require it.

6.1.5 *Advancing particles*

Once the net force on each particle has been calculated, we integrate the kinematical equations of motions for it over the duration of a single time step. We use a single-timestepping second-order leapfrog integration technique for this purpose. Note that this translation can only be performed once it is guaranteed that all traversals have completed. This serves as the boundary for an iteration; following it, particles are once again decomposed onto TreePieces, this time based on the new positions of the particles.

6.2 Performance and Productivity Benefits

This subsection briefly enumerates some of the key features of the Charm++ implementation of the Barnes-Hut algorithm and how they were facilitated by the Charm++ programming model and runtime system.

6.2.1 *PE-level Software cache*

Charm++ provides entities called *groups* to enable management and aggregation tasks at the level of PEs. In our Barnes-Hut code we use the DataManager group to aggregate inter-processor communication and improve reuse of remotely fetched data.

6.2.2 *Automatic Overlap between computation and communication.*

The asynchronous message-driven model of Charm++ provides explicit opportunities for the overlap of computation and communication. Asynchronous communication does not require participation from the recipient, so that message latency can be overlapped with useful computation done after communication. Adaptive overlap helps to improve the performance of the Barnes-Hut code in several phases: (1) to overlap particle exchange latencies with decomposition work for remaining active nodes following domain decomposition; (2) in the tree building phase, requests for remote moments may be fired before attending to local calculation of multipole moments and (3) to mask latency of communication associated with remote data requests and responses in the tree traversal phase.

6.2.3 *Prioritization to accelerate critical path*

Charm++ has explicit support for best-effort prioritization of messages. Such support is of use in giving precedence to critical tasks, on which other objects are dependent. In the case of the Barnes-Hut code, priorities are used in two contexts: (1) To give precedence to remote data requests, since the requesting TreePieces may be waiting to receive the associated data and (2) to encourage automatic overlap of communication intensive remote traversals with computation intensive local traversals.

6.2.4 *ORB Load Balancing*

The Charm++ run time system incorporates a dynamic, measurement-based load balancing infrastructure, thereby separating the issues of decomposing data units onto work units and mapping the work units onto the processing elements. This removes the burden of dynamic load balancing from programmers, allowing them to use generic or customized strategies for load balancing. Such a separation is of great value in N-body simulations, where the appropriateness of balancing strategies can depend greatly on the distribution of data. N-body codes written in other paradigms, for instance, must perform decomposition with an eye to maintaining load balance. See, for instance, the cost-zone partitioning used by the authors of the SPLASH benchmarks [10].

6.3 Performance

We present performance results obtained by running our implementation on *Intrepid*, the IBM Blue Gene/P Supercomputer located at Argonne National Laboratory. Figure 8 presents the results for Barnes-Hut on synthetic 10- (*10m*) and 50-million (*50m*) particle systems generated according to the Plummer model for star clusters. Using the ORB load balancing strategy, the code scales up to 16k cores with the 50m dataset, yielding a parallel efficiency of 69% relative to performance at 2k cores. We expect that the code will demonstrate better scaling behavior with (a) a more sophisticated load balancer, (b) larger input data size and (c) a more uniform distribution of particles, the kind of which are used in large-scale cosmological simulations.

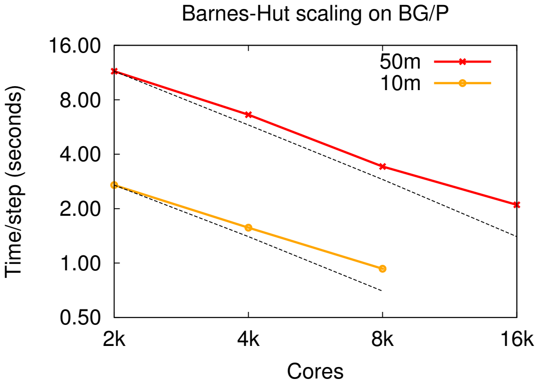

Figure 8: Performance of Barnes-Hut for 10 and 50 million particle systems on Intrepid (IBM BG/P)

6.4 Specifications and Verification

We provide a brief description of the simulation method and input distribution so that the reader may recreate the conditions of our experiments.

1. Input system. The one and ten million particle data sets used as input were generated using the Plummer model for initial conditions of star clusters. These generate a fairly non-uniform distribution of particles, thereby testing the load balancing infrastructure. The particle data set was generated using code adapted from the *barnes* benchmark of the SPLASH suite.

2. Our code evaluates itself for correctness by verifying the conservation of total energy of the input system. The total energy is calculated and reported at iteration boundaries.

3. We used standard values for several simulation parameters. Among these, the opening angle, θ, was set to 0.5 and the softening parameter ϵ was set to 0.05. Load balancing was initiated every 10 iterations, so as to provide enough instrumentation time to the load balancer. Finally, the tree was constructed to have a maximum of 8 particles per leaf.

7. CONCLUSION

Charm++ is an extremely general-purpose parallel programming paradigm capable of high performance. It is suitable for a wide spectrum of parallel programs. Over the years it has attempted to present abstractions and semantics that are derived as generalizations of successful domain-specific solutions. The benchmarks presented here do not use any domain-specific languages tailored to the problem. However, its possible to envision domain-specific languages or targeted, parallel abstractions deployed atop this general model for further productivity benefits. Our implementations should underscore the productivity impact of the programming model and the benefits of the approaches we have discussed in section 1.

8. READING ORDER

We suggest reading the submitted applications in the order listed below, in order to obtain a coherent introduction to the various features of Charm++ that they use.

Global FFT.

1. fft1d.ci – contains control flow in the doFFT() function.
2. fft1d.cc – contains all serial code and initialization routines
3. verify.cc – contains residual calculation and verification initialization

Molecular Dynamics.

1. leanmd.ci – contains the parallel control flow; focus on run() function of each chare.
2. Cell.cc – contains important functions of Cell; focus on sendPositions() and updateProperties().
3. Compute.cc – interact() function that does the force computation
4. Main.cc – start point of application; passes control to run()

Dense LU Factorization.

1. lu.ci – Control flow for the factorization and solve process is described here, starting from the steps that every block executes, then proceeding to the factorization steps taken by blocks in different active panel positions (above diagonal, below diagonal, on diagonal). The methods used during solving and startup follow. Implementations of sequential methods called from lu.ci can be found in order of reference in lu.C. These include the data copying and sending used in pivoting, and routines to set up BLAS calls.
2. mapping.h – Some data distributions available for use with the factorization/solver library. These can be set independently of the algorithm's computation and communication logic.
3. benchmark.C – Setup and validation code.
4. scheduler.C – Logic to determine when blocks of remote data should be retrieved based on what work local blocks are ready to do, within the bounds of a memory constraint. Also tracks when latency-sensitive active panel work is occurring in order to defer bulk trailing updates.

Random Access.

randomAccess.ci should be read first followed by randomAccess.cc

Barnes-Hut.

barnes.ci contains the description of the parallel interfaces. Examine the interfaces used for the different phases in the computation (decomposition, tree building, traversal) and use this as a launch point to examine the rest of the code. All relevant parallel code is in the DataManager and TreePiece files.

9. REFERENCES

[1] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[2] L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[3] Jonathan Lifflander, Phil Miller, Ramprasad Venkataraman, Anshu Arya, Terry Jones, and Laxmikant Kale. Exploring partial synchrony in an asynchronous environment using dense lu. Technical Report 11-34, Parallel Programming Laboratory, August 2011.

[4] Abhinav Bhatele, Eric Bohm, and Laxmikant V. Kale. Optimizing communication for charm++ applications by reducing network contention. *Concurrency and Computation: Practice and Experience*, 23(2):211–222, 2011.

[5] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. Overcoming scaling challenges in biomolecular simulations across multiple platforms. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[6] 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.

[7] Michael A. Heroux, Douglas W. DoerïňĆer, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving performance via mini-applications. Technical report, Sandia National Laboratories, September 2009.

[8] J. E. Barnes and P. Hut. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324, 1986.

[9] L. V. Kale and Sanjeev Krishnan. A comparison based parallel sorting algorithm. In *Proceedings of the 22nd International Conference on Parallel Processing*, pages 196–200, St. Charles, IL, August 1993.

[10] J.P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, 20(1):5–44, March 1992.