Charm++

Migratable Objects + Active Messages + Adaptive Runtime

=

Productivity + Performance

Laxmikant V. Kale‡
Anshu Arya, Nikhil Jain, Akhil Langer, Jonathan Lifflander, Harshitha Menon
Xiang Ni, Yanhua Sun, Ehsan Totoni, Ramprasad Venkataraman‡, Lukasz Wesolowski
Parallel Programming Laboratory
Department of Computer Science, University of Illinois at Urbana-Champaign
‡{kale, ramv}@illinois.edu

1. SUMMARY

Charm++ is an elegant, general-purpose parallel programming model backed by an adaptive runtime system [1]. This combination yields portable performance and a spectrum of real-world productivity benefits that have been demonstrated in production applications. Our submission to this year's HPC Challenge (Class II) comprises six benchmarks that demonstrate these practical benefits available to application developers today. These include: (*1*) interoperability with MPI (*2*) automatic, dynamic load balancing (*3*) effortless checkpointing and restart on a different number of processors, and (*4*) application progress in the presence of failing processes. The submission also explains 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 code-size (line counts) and a summary of the performance results can be seen in Table 1.

1.1 Submitted Benchmarks

- Required
  - Global FFT
  - Random Access
  - Dense LU Factorization
- Optional
  - Molecular Dynamics
  - Sparse Triangular Solver
  - Adaptive Mesh Refinement

1.2 Demonstrated Capabilities

- Automatic communication-computation overlap
- Asynchronous, nonblocking collectives
- Interoperability with MPI
- Automated, dynamic load balancing
- Checkpointing to save application state
- Resilience to failing processes
- Modular, yet composable parallel libraries

2. PROGRAMMING MODEL

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

2.1 Salient Features

The Charm++ programming model has several fundamental characteristics that enable high performance while providing high productivity.

2.1.1 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 2.2. Charm++ messages can also be C++ classes defined by the program. Any work or data unit that is transmitted across processes has to define a serialization operator, called `pup()` for Pack/UnPack. This allows the transmission of complex objects 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

	Productivity					Performance		
Code	C++	CI	Benchmark Subtotal	Driver	Total	Machine	Max Cores	Performance Highlight
Required Benchmarks								
1D FFT	54	29	83	102	185	BG/P BG/Q	64K 16K	2.71 TFlop/s 2.31 TFlop/s
Random Access	76	15	91	47	138	BG/P BG/Q	128K 16K	43.10 GUPS 15.00 GUPS
Dense LU	1001	316	1317	453	1770	XT5	8K	55.1 TFlop/s (65.7% peak)
Additional Benchmarks								
Molecular Dynamics	571	122	693	n/a	693	BG/P BG/Q	128K 16K	24 ms/step (2.8M atoms) 44 ms/step (2.8M atoms)
AMR	1126	118	1244	n/a	1244	BG/Q	32k	22 steps/sec, 2d mesh, max 15 levels refinement
Triangular Solver	642	50	692	56	748	BG/P	512	48x speedup on 64 cores with helm2d03 matrix

Table 1: Performance and line count summaries for 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. Performance data is simply a summary highlight. Detailed performance results can be found in the following sections

interactions with remote objects are realized through an inheritance and serialization API exposed by the runtime framework.

2.1.2 Message-Driven

Messaging in Charm++ is 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*, i.e., the 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 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.

2.1.3 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.

2.1.4 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/.

2.2 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.

2.3 Describing Parallel Control Flow

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*s 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 notation 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 for describing parallel control flow 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 much like common sequential C or C++ code. In addition to basic C constructs, the when clause indicates 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 numerical 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, the overlap construct describes 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 constructs are implemented in generated code by a collection of message buffers and trigger lists that have negligible overhead comparable to a few static function calls. If the body of a 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, only the current control-flow location is saved, and control is returned to the scheduler. Local variables to be preserved across blocks are encoded as member variables of the enclosing chare. In contrast, thread context switches that 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. Also, unlike linear chains of dependencies in threads, Structured Dagger allows one to express a DAG of dependencies between messages and local computations.

2.4 Demonstrated Capabilities

2.4.1 Automated Dynamic Load Balancing

As HPC hardware architectures and applications become more complex, and execution scales continue to grow larger, load imbalances will intensify further. Making decisions such as when to balance load, and what strategy to use are already quite challenging in the context of large applications. Multiple application runs may be needed to tune for the best values for all the parameters that control the application's execution and utilization profiles. Charm++ has a mature measurement-based load balancing framework, which relies on the principle of persistence [3]. During application execution, the load balancing framework continuously monitors the performance profile, and on invocation, it redistributes the load without any requirement from the application. We have demonstrated beneficial load-balancing strategies for several scientific applications.

The latest addition to the Charm++ load balancing framework is the Meta-Balancer, which automates decision making related to load balancing: when to invoke the load balancer and what load balancing strategy to use. Meta-Balancer uses a linear prediction model on the collected information about the application, and predicts the time steps (or iterations) at which load balancing should be performed for optimal performance. By observing the load distribution, the idle time on processors and the volume of communication during an application run, it also identifies a suitable balancing strategy. This relieves application writers from conducting theoretical analysis and multiple test runs to tune the load balancing parameters for the best performance. Given the increased complexity of applications and systems, such a analysis may not even be feasible. Meta-Balancer has been shown to be effective for various real-world applications and benchmarks [4], and has been used to improve the performance of LeanMD in this submission.

2.4.2 Checkpointing Application State

For HPC applications running for days and hours, it is

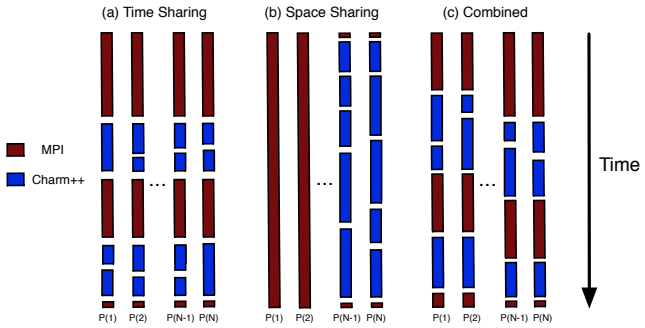


Figure 1: Supported modes of interoperation between Charm++ and MPI

```
1   //MPI_Init and other basic initialization
2   { optional pure MPI code blocks }
3
4   //create a communicator for initializing Charm++
5   MPI_Comm_split(MPI_COMM_WORLD, peid%2, peid, &
        newComm);
6   CharmLibInit(newComm, argc, argv);
7
8   { optional pure MPI code blocks }
9
10  //Charm++ library invocation
11  if(myrank%2)
12    fft1d(inputData,outputData,data_size);
13
14  //more pure MPI code blocks
15  //more Charm++ library calls
16
17  CharmLibExit();
18  //MPI cleanup and MPI_Finalize
```

Figure 2: Sample code showing a Charm++ library (FFT1D) being used from an MPI program

difficult to get the allocations needed at one time. Charm++ provides support to checkpoint the application state to disk for split execution. The application can even be restarted using the checkpoint file on a different number of processors than the original run (e.g., a 10000 processors run can be checkpointed and restarted on 9000 processors). To perform checkpointing, the user only needs one extra line of code:

CkStartCheckpoint("log",callback)

Here "log" is the filesystem path to the checkpoints. `callback` is called when the checkpoint (or restart) is complete.

2.4.3 Tolerating Process Failures

Applications running on large scale machines may face a high failure frequency as the number of components in every generation of supercomputers keeps increasing. It will become increasingly difficult for applications to complete execution without any fault tolerance support. Charm++ offers a double in-memory fault tolerance mechanism for applications running on unreliable systems.

The in-memory based checkpointing algorithm helps reduce the checkpoint overhead in comparison to the file system based checkpointing scheme by storing the checkpoint in memory. It therefore avoids any congestion caused by simultaneous access to the file system.

In the scheme, periodically at application synchronization points, two copies of checkpoints are stored; one in the local memory of the processor and the other in the memory of a remote processor (the "buddy" processor of the local processor). Charm++ automatically detects failures using a periodic hearbeat, and chooses a new processor to replace the crashed processor. This new processor can either be a previously unused spare processor, or a currently used processor that is already part of the execution. The most recent checkpoint is then copied to the chosen replacement processor from the buddy of the crashed processor. Thereafter, every processor rolls back to the last checkpoint, and the application continues to make progress. The fault tolerance mechanism provided by Charm++ can be used by adding only one line of code:

CkStartMemCheckpoint(callback)

`callback` is called when the in-memory checkpoint(or restart) is complete. To enable this feature, the `syncft` option should be added to the configuration when compiling Charm++.

We inject failures into the application by randomly selecting a process and forcing it to become completely silent and unresponsive. The "failed" process simply ceases to communicate with any other process in the execution. All application and runtime state on this process is lost to the rest of the execution. The runtime system, upon failing to receive heartbeat signals from the "failed" process, flags it as a process failure and commences recovery protocol. The silencing functionality is encapsulated behind a CkDieNow() function that is called on the randomly selected failing process.

2.4.4 Interoperating with MPI

Charm++ provides support for multi-paradigm programming where only a subset of application modules are written in Charm++, while the remaining use MPI (or MPI + OpenMP). Any legacy MPI application can use a library implemented in Charm++ in the same way that it uses any external MPI library. This also provides a gradual migration path to existing applications that wish to adopt the Charm++ programming model.

Charm++ supports interoperability in three different modes as shown in Figure 1. (*1*) In the time sharing mode, the control transfers between Charm++ and MPI in a coordinated manner for all the processors in the execution. (*2*) In the space sharing mode, a set of processors always execute Charm++ code, while the rest execute the MPI program. (*3*) In the general case, time sharing and space sharing can be used simultaneously by dividing the MPI ranks using MPI communicators during initialization, and then, making calls to Charm++ libraries as desired.

A simple interface function is added to the Charm++ library to permit invocation from MPI. In Figure 2, we present example usage of a one-dimensional FFT library written in Charm++, from an MPI program. The statements of interest are the additional calls to CharmLibInit and CharmLibExit, which are required once at the beginning and end of execution, respectively. After such initialization, Charm++ libraries can be invoked at any time using the interface methods.

2.4.5 Nonblocking Collective Communication

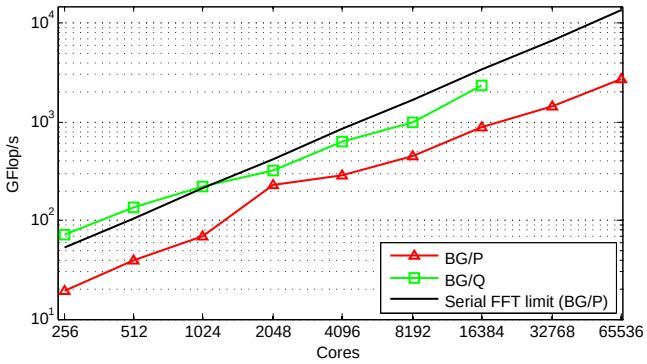


Figure 3: Performance of Global FFT on IBM's BG/Q and BG/P

Sizes m	BG/P Cores	BG/P GFlops	BG/Q Cores	BG/Q GFlops
32768^2	256	19.4	128	43.5
46080^2	512	39.9	256	72.5
65536^2	1024	69.1	512	133.8
92160^2	2048	229.2	1024	220.7
131072^2	4096	284.1	2048	325.5
180224^2	8192	454.4	4096	634.1
262144^2	16384	867.9	8192	974.7
360448^2	32768	1430.8	16384	2306.2
524288^2	65536	2708.9	-	-

Table 2: Performance of Global FFT 1D on Intrepid (IBM BG/P) and Vesta (IBM BG/Q)

Charm++ supports a number of collective communication patterns using high performance implementations. Some collectives, such as broadcast, are expressed by overloading basic operations on proxies. Broadcast, as an example, is expressed using the same function invocation syntax as point-to-point sends, with the distinction that the entry method is invoked on a collection instead of a single chare or processing element. The intuitive expression of broadcasts is an example of the productivity benefits of Charm++.

For more complex cases, Charm++ provides communication libraries which support the most commonly used collectives: broadcasts, reductions, multicasts, and many-to-many/all-to-all. All collective communication in Charm++ is naturally asynchronous and non-blocking, and overlaps with other computation and communication that may involve the same cores or objects.

Some Charm++ libraries are more subtle than the typical collective calls of a message passing library. One of these specialized libraries, which we found useful in improving performance and elegance of two of our benchmarks (FFT and Random Access) is Mesh Streamer. Mesh Streamer is a library for accelerating all-to-all communication. By aggregating small units of communication into larger messages, and routing these messages over the dimensions of a virtual mesh comprising the processing elements involved in the run, Mesh Streamer improves fine-grained all-to-all communication performance. The library is asynchronous and highly flexible in supporting communication patterns that are not fully all-to-all. As an example, the various processing elements participating in the collective are free to send different numbers of messages, with varying message sizes, as part of the same *streaming step*. Processing elements may even avoid participating in the collective completely. Further, many-to-many exchanges are also supported, and a single streaming step can even support numerous localized many-to-many exchanges in the same streaming step, and combine intermediate messages which are not part of the same many-to-many operation. In many ways, Mesh Streamer typifies the flexibility with which the Charm++ programming model can handle dynamic communication patterns which do not belong to commonly accepted patterns.

The library also permits clean expression of communication in the application. For example, in the random access benchmark, the receipt of the update messages from remote processors is expressed as follows:

```
1 void process(const T &key) {
2     CmiInt8 localOffset = key & (localTableSize − 1);
3     // Apply update to local portion of table
4     HPCC_Table[localOffset] ^= key;
5 }
```

The code snippet looks like it is receiving individual updates one at a time. However, the library handles the data aggregation and routing completely invisibly under the application, producing very succinct application code.

3. GLOBAL FFT

3.1 Performance

Runs were performed on Intrepid (IBM's Blue Gene/P) and Vesta (Blue Gene/Q). Performance scales well up to at least 64K cores of Blue Gene/P, in large part due to the use of our message aggregation and software routing communication library, Mesh Streamer (also used in the Random Access benchmark). ESSL was used to perform serial FFTs. Figure 3 and table 2 summarize the results.

3.2 Implementation

Our implementation of Global FFT takes input size N and performs a complex 1D FFT on an NxN matrix where 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 done via a general Charm software routing library, Mesh Streamer, and external libraries (FFTW or ESSL) perform serial FFTs on the rows of the matrix.

3.2.1 Interoperable with MPI

The FFT library submitted as part of this code can also interoperate with programs written in MPI, and the library in Charm++. The general concept, and the simplicity of use of interoperation in Charm++ was described in Section 2.4.4. In order to use the FFT library with MPI, one should build Charm++ using MPI as the base layer. For more details, one can refer to the README and the interface code that has been submitted as part of this submission for example

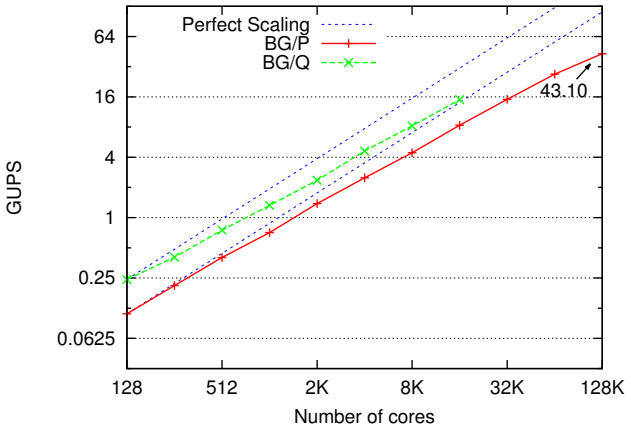


Figure 4: Performance of random access on Intrepid (IBM BG/P) and Vesta (IBM BG/Q)

Cores	BG/P		BG/Q	
	GUPS	Mem(TB)	GUPS	Mem(TB)
128	0.11	0.031	0.242	0.062
256	0.21	0.062	0.403	0.125
512	0.40	0.125	0.751	0.250
1K	0.71	0.250	1.327	0.500
2K	1.38	0.500	2.360	1.000
4K	2.49	1.000	4.623	2.000
8K	4.43	2.000	8.204	4.000
16K	8.34	4.000	15.001	8.000
32K	15.13	8.00	-	-
65K	26.94	16.00	-	-
128K	43.10	32.00	-	-

Table 3: Performance of random access on Intrepid (IBM BG/P) and Vesta (IBM BG/Q)

usage. Note that the same library code has been used by both the drivers (in MPI and in Charm++).

3.3 Verification

The benchmark code self-validates by executing an inverse FFT and the residual is printed.

4. RANDOM ACCESS

4.1 Productivity

4.1.1 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.

4.1.2 Automatic Topology Determination

The Charm++ Topology Manager automates the task of determining physical network topology for a partition assigned to a job. We use it to provide topology information for Mesh Streamer.

4.2 Performance

Performance results on IBM BG/P and BG/Q are presented in Figure 4 as well as in Table 3. We achieve over 35 GUP/s when running on 128K cores of BG/P.

4.3 Implementation

We used a Charm++ group to partition the global table across the nodes in a run. A group can be thought of as a chare array with one chare per Processing Element (PE), which corresponds to either a core or a thread in the system. For example, on Blue Gene/P, we ran using a single PE per core, while on Blue Gene/Q, we ran in a mode employing one PE per thread, and up to 4 threads per core. Each element of the group allocates its part of the global table, generates random update keys, and sends the updates to the appropriate destination.

The small size of individual update messages in this benchmark made it prohibitively expensive to send each item as a separate message. To improve performance, we used a Charm++ message aggregation and routing library called Mesh Streamer. Mesh Streamer routes messages over an N-dimensional virtual topology comprising the PEs involved in the run. The topology needs to be specified when creating an instance of the library. To facilitate the specification of the virtual topology, we used the Charm++ Topology Manager library, which provides an interface to determine the network topology for the current run. In typical cases, we found that using virtual topologies with dimensions matching the network representation of the current run led to good performance.

In the context of Mesh Streamer, each processor is limited to sending and receiving messages from a subset of the processors involved in the run. When determining where to send a particular data item (in our case table update), Mesh Streamer selects a destination from among its peers so that data items always make forward progress toward the destination. Items traveling to the same intermediate destination are combined into larger messages. This approach achieves improved aggregation and lower memory utilization compared to schemes which aggregate separately for each destination on the network.

Mesh Streamer allows specifying a limit on the number of items buffered by each local instance of the library. We set the buffering limit to 1024 to conform to the benchmark specifications.

4.4 Verification

Verification is done by performing a second phase with the same updates as in the timed run. In the absence of errors, this has the effect of returning the global table to its initial state. To verify correctness, we check the state of the table to ensure its final state is the same as when the benchmark is started. 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. As such, our implementation does not allow for even the small number of errors permissible

according to the benchmark specification.

5. DENSE LU FACTORIZATION

5.1 Productivity

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

5.1.1 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.

5.1.2 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. We have utilized this ability to explore novel mapping schemes, demonstrating that deviating from a traditional block-cyclic distribution can increase performance for modern multicore architectures [5].

5.1.3 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.

5.1.4 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.

5.2 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 5. 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.

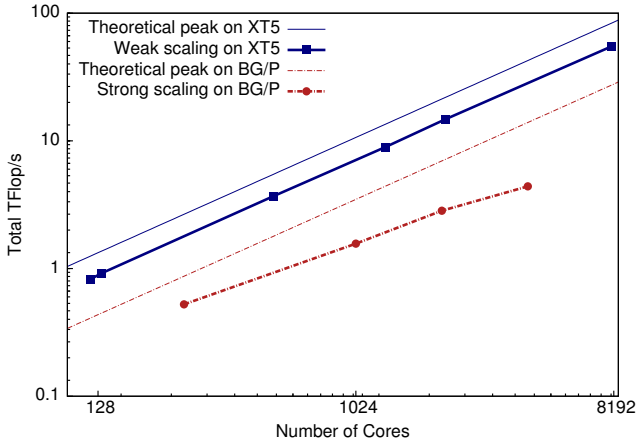


Figure 5: Weak scaling (matrix occupies 75% of memory) from 120 to 8064 processors on Jaguar (Cray XT5). Strong scaling ($n = 96,000$) from 256 to 4096 processors on Intrepid (IBM BG/P).

5.2.1 Adaptive Lookahead

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

5.2.2 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.

5.3 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.

5.3.1 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 distri-

bution 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:

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

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.

5.3.2 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.

5.3.3 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 [6] 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.

5.4 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.

6. 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 [7, 8] and resembles the miniMD application in the Mantevo benchmark suite [9] 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.

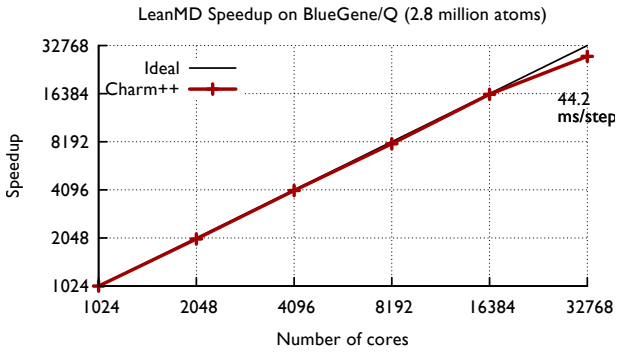


Figure 6: Scaling of LeanMD for the 2.8 million atoms system on Vesta (IBM BG/Q)

6.1 Productivity

Our implementation of LeanMD takes only 693 lines of code while offering capabilities that match some of the 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 [9] 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.

6.1.1 Automatic Load Balancing with Meta-Balancer

Charm++'s fully automated measurement-based load balancing framework 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. To enable automatic load balancing decisions using Meta-Balancer, the user simply specifies a flag, *+MetaLB*, at the command line, and the run-time system will automatically identify a suitable load balancing period. Measurement-based load balancing is well suited for applications where the recent past is a predictor of the near future. In such situations, load balancing decisions taken by Meta-Balancer based on such predictions are often correct. This works well for LeanMD because atoms migrate slowly and hence the load fluctuations are gradual.

6.1.2 In-disk and in-memory Checkpointing

Applications that perform physical simulation are executed for a long period (tens of hours to a few days). In such a scenario, providing fault-tolerance mechanisms are critical for sustained execution. LeanMD has been used to demon-

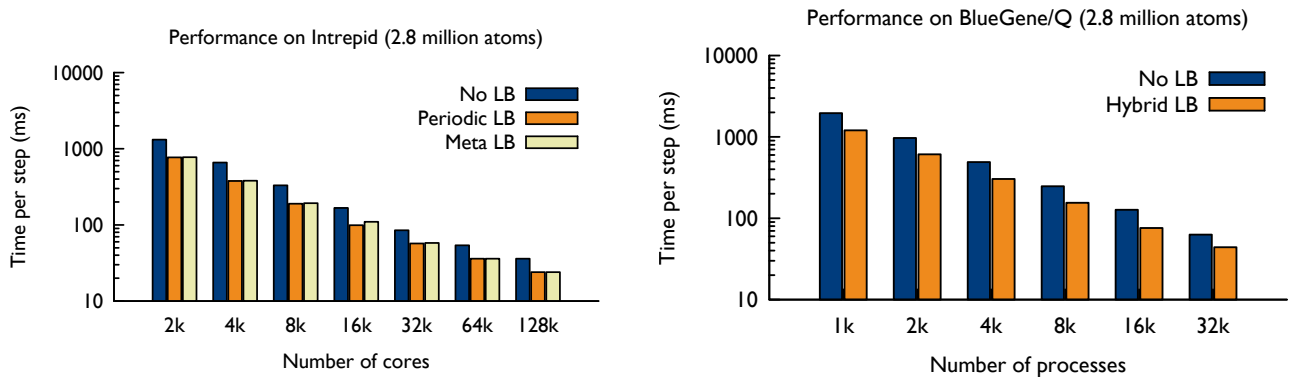


Figure 7: Performance of LeanMD for the 2.8 million atoms system on Vesta (IBM BG/Q) and Intrepid (IBM BG/P)

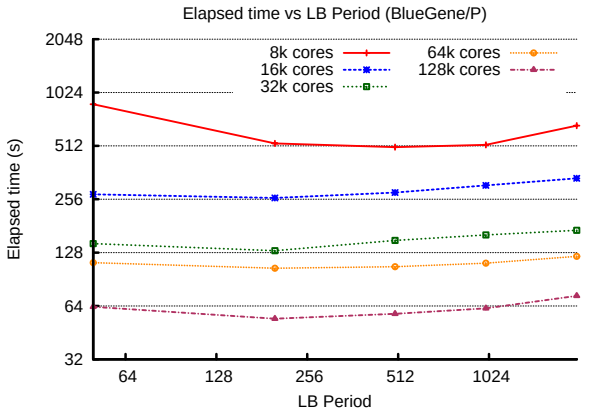


Figure 8: Execution time of LeanMD: variation in LB Period for 2.8 million atoms system on Intrepid (IBM BG/P)

Cores	No LB (s)	Periodic LB (s)	Meta-Balancer (s)
8k	666	504	413
16k	336	260	277
32k	171	131	131
64k	122	104	100
128k	73	54	52

Table 4: LeanMD application time with various load balancing strategies

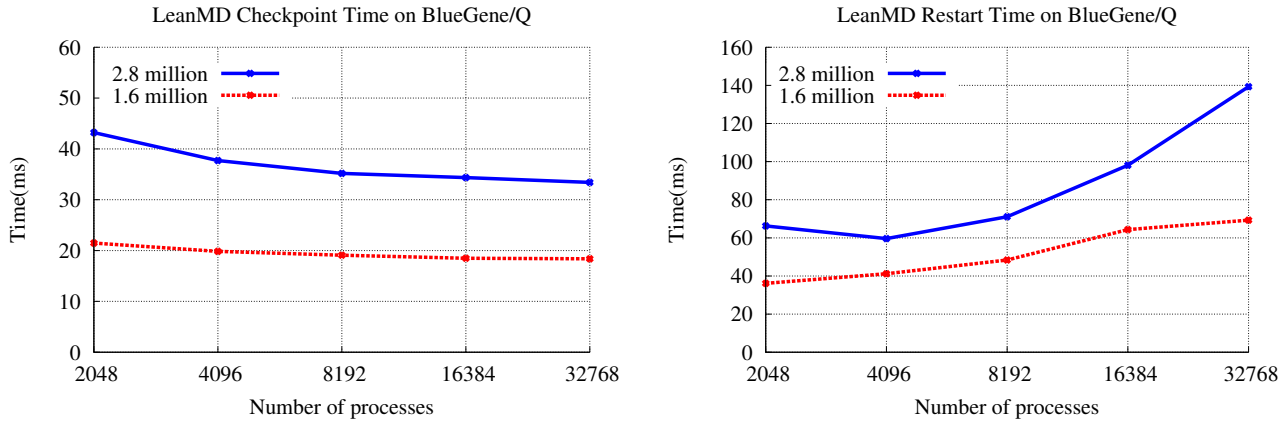


Figure 9: Checkpoint and restart of LeanMD for a 2.8 million atoms system on Vesta (IBM BG/Q)

strate two important checkpointing mechanisms provided by Charm++ - checkpointing to disk for split execution and checkpointing to memory for fault tolerance support. As mentioned earlier, selection of a checkpointing scheme in Charm++ is as simple as making a different function call.

6.1.3 Dense and Sparse Chare Arrays

Indexed collection of objects in Charm++ provide 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.

6.1.4 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. It also allows an application to checkpoint for a particular number of processors, and restart on a different number of processors. 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.

6.1.5 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. In Charm++, creation of such spanning trees is performed dynamically using asynchronous message passing without imposing a global barrier.

6.2 Performance

We present performance numbers for LeanMD on two machines: 1) Vesta - an IBM Blue Gene/Q installation at ANL, and 2) Intrepid - an IBM Blue Gene/P installation at ANL. LeanMD was run using a molecular system that consist of 2.8 million atoms distributed in a uniformly random manner in the space.

We experimented within a range of LB periods (50, 200, 500, ..., 2000) to find the period at which periodic load balancing gives the best performance. Figure 8 shows the application run time using these LB periods on various core counts. If the load balancing is done very frequently, the overheads of load balancing overshoots the gains of load balancing, and results in bad performance. On the other hand, if load balancing is done very infrequently, the load imbalance in the system reduces the gains due to load balancing.

In Table 4, a comparison of total execution time of LeanMD for the following three cases is presented - without load balancing, best periodic load balancing and Meta-Balancer. We observe that in all the cases Meta-Balancer either improves the best performance obtained by periodic load balancing, or matches it. Meta-Balancer successfully identifies that there is an initial load imbalance in LeanMD and hence calls load balancing at the very beginning. Thereafter, the frequency of load balancing decreases as the change in loads of individual chare is slow. Given the superiority of Meta-Balancer, all load balancing decisions were automated using Meta-Balancer in the remaining experiments.

Figure 7 presents performance results for execution of LeanMD on Vesta and Intrepid. On Vesta, time per step decreases linearly as the process count is increased. The speed up for the runs on Vesta is shown in Figure 6. The linear decrease in the time per step to 44 ms/step for a 2.8 million atoms system on just 16384 is a result of accurate automated decision making by Meta-Balancer. On Intrepid, the scaling results are shown from 2K cores to 128K cores. It is to be noted that although LeanMD does not scale linearly on BG/P for large core counts, the benefits of load balancing are always seen. On 128K cores, performing load balancing reduces time per step from 36 ms to 24 ms, which is a performance gain of 33%.

Figure 9 shows the checkpoint and the restart time for a 2.8 million atoms and a 1.6 million atoms system on Bluegene/Q. For the 2.8 million atoms system, as the number of processes increase from $2K$ to $32K$, the checkpoint time decreases by 25% from 43 ms to 33 ms. Such a drop is observed because the checkpoint data size per process decreases as the number of processes is increased, and it overshadows the communication overheads. Note that the time to checkpoint decreases slowly after $8K$ cores when the synchronization overheads becomes the main bottleneck.

The restart time, which is measured from the time a failure is detected to the point where the applications is recovered and is ready to continue execution, is shown in Figure 9. Since several barriers are used to ensure consistency until the crashed processor is recovered, the complexity of the recovery process is $O(logP)$ (due to the use of reduction tree for synchronization). The restart time increases from 66 ms on $4K$ cores to 139 ms on $32K$ cores for the 2.8 million system. Note that the efficiency of the restart process is partly due to the fact that the double in-memory fault tolerance protocol allows the application to restart from the last checkpoint in the local memory. Hence, an application can restart without a full stop, and thus job turn-around time and a new job submission are avoided.

6.3 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

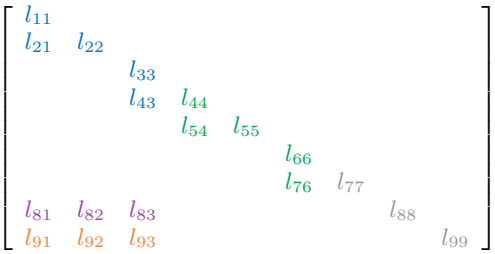


Figure 10: Sparse matrix divided into blocks.

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 addition, the fault tolerance scheme performs a periodic checkpointing. 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`.

6.4 Specification and Verification

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

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

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	1.1328×10^{-133}
B	2.23224×10^{-76}
Atoms per cell	700
Cutoff distance, r_c	26 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.

7. SPARSE TRIANGULAR SOLVER

Solution of sparse triangular systems of linear equations is a performance bottleneck in many methods for solving more general sparse systems, such as many iterative methods with preconditioners. It is notoriously resistant to parallelism, however, and existing parallel linear algebra packages appear to be ineffective in exploiting much parallelism for this problem. We chose this benchmark to show the ability of our system for challenging sparse linear algebra computations. The algorithm we implement is described in detail

```
1  // if this chare has some diagonal part of matrix
2  if (onDiagonalChare) {
3    // schedule the independent computation with lower priority
4    serial { thisProxy[thisIndex].indepCompute(...) }
5    // "while" and "when" can happen in any order
6    overlap {
7      // while there are incomplete rows, receive data
8      while (!finished) {
9        when recvData(int len, double data[len], int rows[len])
10         serial {if(len>0) diagReceiveData(len, data, rows);}
11     }
12     // do serial independent computations scheduled above
13     when indepCompute(int a) serial {myIndepCompute();}
14   }
15   // if chare doesn't have diagonal part of matrix
16 } else {
17   // wait for x values
18   when getXvals(xValMsg* msg) serial {nondiag_compute();}
19   // while there are incomplete rows, receive data
20   while (!finished) {
21     when recvData(int len, double data[len], int rows[len])
22     serial {nondiagReceiveData(len, data, rows);}
23   }
24 }
```

Figure 11: Parallel flow of our triangular solver in Structured Dagger

elsewhere [10], and a brief summary follows. The matrix is divided into blocks of columns, as shown with blue, green and grey colors in Figure 10. Each block is analyzed to find its independent rows for computation. If there are dense regions below the diagonal section (purple and orange in Figure 10), they are divided into new blocks. Each diagonal block starts the computation with its independent parts and waits for required messages from the left. Nondiagonal blocks wait for the solution values from their corresponding diagonal block, and then start their computation (and receipt of other messages). This flow is expressed in the Structured Dagger code of Figure 11.

7.1 Productivity

Using different features of Charm++, we improve both performance and productivity, comparing to the state-of-the-art packages such as SuperLU_DIST. We implement a

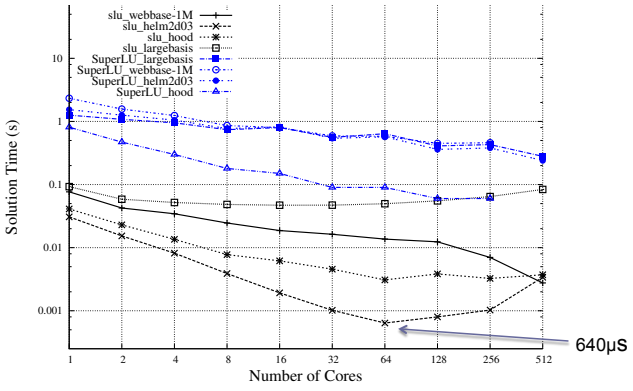


Figure 12: Performance of sparse triangular solver on Intrepid (IBM BG/P) for different sparse matrices. Performance is also competetive with SuperLU_DIST

more complicated algorithm with only 692 total SLOCs, comparing to 879 SLOCs of the triangular solver in SuperLU_DIST. For better load balance and overlap of communication and computations, we divide the columns into many blocks (more than the number of processors) and map them in round-robin fashion to processors. This performance optimization feature is essential for this solver and managing the blocks manually is burdensome for the programmer. Thus, virtualization in Charm++ improves productivity significantly for this implementation. In addition, having parallel units independent of the processors and creating them dynamically allows the solver to adapt to the structure of the input matrix (create new blocks for denser nonzero regions) and benefit from more parallelism. Furthermore, message-driven nature of the algorithm matches perfectly to the Charm++ programming paradigm, so the overall flow is easily expressed in Structured Dagger with high performance and less programming cost. As an example, doing the same message-driven computation in MPI would require MPI_Iprobe calls that cost both performance and productivity. Also, easy usage of priorities improves the performance with minimal programming effort.

7.2 Performance

Figure 12 compares the performance of our implementation against the triangular solver of SuperLU_DIST package for different matrices on a Blue Gene/P. As shown, we achieve much better performance and scaling for all the compared cases. For example, SuperLU_DIST is about 18.5 times slower than the best serial performance on 64 cores for matrix slu_helm2d03, whereas our solver achieves a speedup of more than 48. SuperLU_DIST uses a simple 2D decomposition approach for parallelism, which is inefficient. Our scaling and performance is made possible by a better algorithm and its effective implementation in Charm++.

7.3 Implementation

In the Charm++ implementation, column blocks are assigned to elements of a Chare array and mapped in round-robin fashion into processors. The denser regions (with many nonzeros) below the diagonal sections are also divided into new blocks and new Chare elements are created for them. Different Chares start their independent computation but with a lower priority than the messages they receive, with the intention of accelerating the critical path. Note that since there are multiple Chares per processor, a Chare may receive some messages before starting its independent computation, while other Chares are using the processor. If there are some subdiagonal Chares corresponding to a diagonal Chare, it will multicast its results for them (using CkMulticast library for faster communication). These messages have the highest priority in the system, since they enable new computations and accelerate the critical path.

7.4 Verification

We verify the solution by computing a residual as described by the HPL specification.

8. ADAPTIVE MESH REFINEMENT

Traditional AMR algorithms phrase the design in terms of processors that contain many blocks. Instead, we promote blocks to first-class entities that act as a virtual processor. As the mesh is refined or coarsened in AMR, the number of blocks will change and traditional algorithms require synchronization at these points. However, to enhance productively, we abstract the blocks as a collection that dynamically expands and contracts over time. Refinement decisions can then be local to each block and propagated as far as algorithmically required so blocks are always within one refinement level of their neighbors. Because remeshing occurs only at discrete points in the simulation time, instead of using collective communication that is proportional to the depth of the recursive refinement, we use a scalable termination detection mechanism built into our runtime to globally determine when all refinement decisions have been finalized. Previous collective methods require $\mathcal{O}(d)$ rounds of collective communication, where d is the refinement and consume $\mathcal{O}(P)$ memory per processor to store the results. By utilizing termination detection, we consume a negligible amount of memory and communicate no data. Besides termination detection, blocks execute completely asynchronously, communicating only with neighboring blocks when required.

Traditional AMR implementations store the quad-tree instance on each process consuming $O(P)$ memory and taking $O(\log N)$ time for neighbor lookup. We organize the blocks into a quad-tree but address each block by their location in the tree using bit vectors to represent quadrants recursively. It requires only $O(\#blocks/P)$ memory per process and $O(1)$ lookup time. It also frees the programmer from having to know where the block lies; instead, the underlying runtime system manages the physical locations of each block and provides direct, efficient communication between them. The runtime system can then redistribute the blocks periodically without any change to the logic.

8.1 Implementation

In our implementation, the collection of blocks is organized as a chare array that is indexed by a custom array index. This index is a bit vector used to describe the block's hierarchical location in the quad tree. More information about the algorithm and implementation is available [11]. The parallel control flow is expressed succinctly for the remeshing algorithm which consists of two phases: reaching the consensus on granularity of each chare in phase 1, and performing the actual refinement or coarsening in phase 2.

8.2 Performance

We present strong scaling results for a finite-difference simulation of advection. A first-order upwind method in 2D space was used for solving the advection equation (figure 13). The benchmark attains efficiencies of 99%, 95%, 65%, 55% at $2k$, $8k$, $16k$ and $32k$ ranks, respectively (figure 14a) when the dynamic depth of the mesh is allowed to vary from 5 to 15. In order to evaluate our remeshing algorithm, we graph the distribution of remeshing latencies (Figure 14b) of each block. The remeshing latency is the time spent in remeshing without any computation overlap, calculated as the duration between the last processor starting remeshing and the beginning of the next time step.

9. CONCLUSION

Charm++ is a general-purpose parallel programming paradigm capable of high performance, and suitable for a wide spectrum of parallel algorithms. Over the years it has presented abstractions and semantics that are have evolved as generalizations of successful domain-specific solutions. The bench-

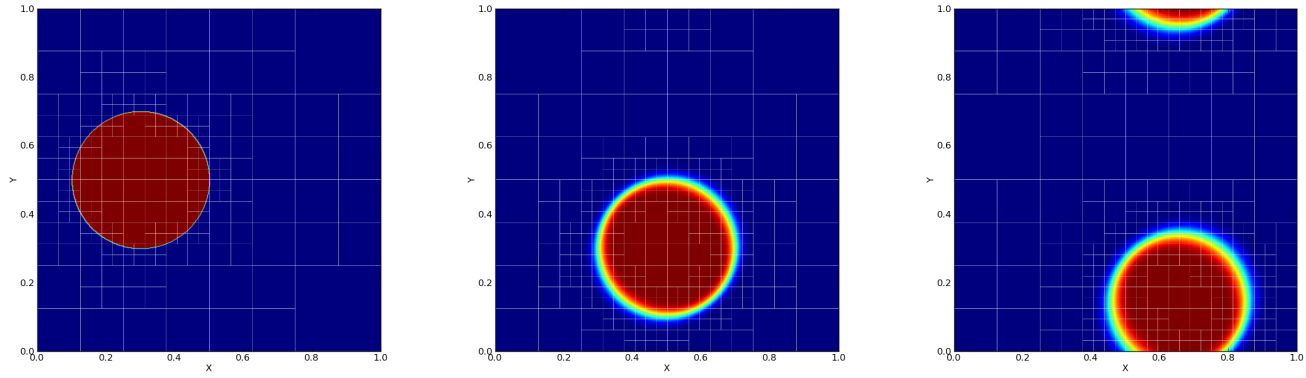
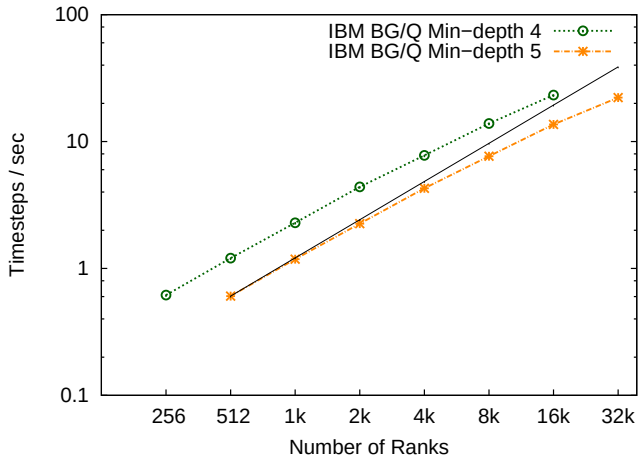
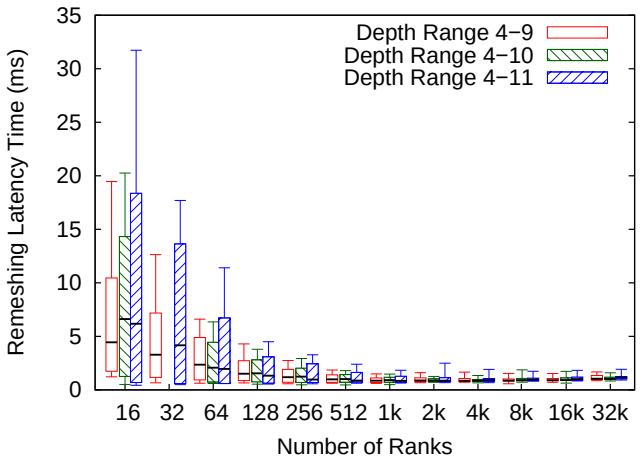


Figure 13: AMR benchmark: example simulation of a circular fluid advected by a constant velocity field. From left to right, the figure displays the fluid density after 2, 572, and 1022 iterations respectively. The white squares demonstrate how the mesh is evolving over time.



(a) Timesteps per second strong scaling on IBM BG/Q with a max depth of 15.



(b) The non-overlapped delay of remeshing in milliseconds on IBM BG/Q. The candlestick graphs the minimum and maximum values, the 5th and 95th percentile, and the median.

Figure 14: Performance results for the AMR benchmark.

marks presented here do not use any domain-specific languages tailored to the problem. However, it is 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 2.

10. READING ORDER

We suggest the following reading order, as a progressive introduction to the various features of Charm++ used.

Global FFT.

1. main.ci – contains control flow for the driver
2. main.cc – contains serial code and initialization routine
3. fft.ci – contains control flow for FFT calculation
4. fft.cc – contains all serial code for doing an FFT
5. data.cc – contains the the data initializer, container, and residual calculation routine

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; invokes run()

Sparse Triangular Solver.

1. sparse_solve.ci – contains the parallel control flow; focus on start() function.
2. sparse_solve.C – reads and analyzes input matrix, initializes solver and verifies solution

Random Access.

1. randomAccess.ci – contains chare, group, and entry method declarations
2. randomAccess.cc – contains the driver, rng, table update sends / receives, and verification code

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 adaptively fetch remote data and schedule block computations while remaining within memory constraints. Also tracks when latency-sensitive active panel work is occurring in order to defer bulk trailing updates.

AMR.

1. advection.ci – contains the parallel control flow for the remeshing algorithm
2. Advection.C – contains the methods that describe the local refinement propagation algorithm and decision-making state machine
3. QuadIndex.C – contains the data type for indexing the parallel collection via a hierarchical bit vector

Acknowledgements

Results presented here were obtained via experiments on Intrepid (IBM BG/P ALCF), Vesta (IBM BG/Q ALCF), and Jaguar (Cray XT5, OLCF). We used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, and the Oak Ridge Leadership Computing Facility at Oak Ridge National Laboratory. These were supported by the OfĩňĄce of Science of the Department of Energy under contracts DE-AC02-06CH11357 and DE-AC05-00OR22725, respectively.

11. 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] Laxmikant V. Kale. Some Essential Techniques for Developing Efficient Petascale Applications. July 2008.

[4] Harshitha Menon, Nikhil Jain, Gengbin Zheng, and Laxmikant V. Kalé. Automated load balancing invocation based on application characteristics. In *IEEE Cluster 12*, Beijing, China, September 2012.

[5] Jonathan Lifflander, Phil Miller, Ramprasad Venkataraman, Anshu Arya, Terry Jones, and Laxmikant Kale. Mapping dense lu factorization on multicore supercomputer nodes. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2012*, May 2012.

[6] 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.

[7] 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.

[8] 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.

[9] Michael A. Heroux, Douglas W. DoerïñĆer, 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.

[10] Ehsan Totoni, Michael T. Heath, and Laxmikant V. Kale. Structure-adaptive parallel solution of sparse triangular linear systems. October 2012.

[11] Akhil Langer, Jonathan Lifflander, Phil Miller, Kuo-Chuan Pan, , Laxmikant V. Kale, and Paul Ricker. Scalable Algorithms for Distributed-Memory Adaptive Mesh Refinement. In *Proceedings of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2012). To Appear*, New York, USA, October 2012.