Scientific Programming 19 (2011) 47–62
DOI 10.3233/SPR-2010-0303
IOS Press

Programming heterogeneous clusters with accelerators using object-based programming

David M. Kunzman * and Laxmikant V. Kalé
Department of Computer Science, University of Illinois, Urbana, IL, USA

Abstract. Heterogeneous clusters that include accelerators have become more common in the realm of high performance computing because of the high GFlop/s rates such clusters are capable of achieving. However, heterogeneous clusters are typically considered hard to program as they usually require programmers to interleave architecture-specific code within application code. We have extended the Charm++ programming model and runtime system to support heterogeneous clusters (with host cores that differ in their architecture) that include accelerators. We are currently focusing on clusters that include commodity processors, Cell processors, and Larrabee devices. When our extensions are used to develop code, the resulting code is portable between various homogeneous and heterogeneous clusters that may or may not include accelerators. Using a simple example molecular dynamics (MD) code, we demonstrate our programming model extensions and runtime system modifications on a heterogeneous cluster comprised of Xeon and Cell processors. Even though there is no architecture-specific code in the example MD program, it is able to successfully make use of three core types, each with a different ISA (Xeon, PPE, SPE), three SIMD instruction extensions (SSE, AltiVec/VMX and the SPE's SIMD instructions), and two memory models (cache hierarchies and scratchpad memories) in a single execution. Our programming model extensions abstract away hardware complexities while our runtime system modifications automatically adjust application data to account for architectural differences between the various cores.

1. Introduction

The growing popularity of heterogeneous computing has made the subject increasingly important in the realm of high performance computing (HPC). This importance is reflected in the presence of several large heterogeneous clusters including the Roadrunner cluster at Los Alamos National Lab (LANL) [4], the Lincoln cluster [25] at the National Center for Supercomputing Applications (NCSA), and the MariCel cluster [3] at the Barcelona Supercomputing Center (BSC). Graphical processing units (GPUs) and the Cell processors [20] are increasingly being used for general purpose computation. This trend will continue with advances such as the upcoming introduction of Intel's Larrabee [30] device.

As if parallel programming were not considered difficult enough, introducing heterogeneity into systems further complicates matters. Writing applications to execute on heterogeneous clusters can be quite difficult. Often requiring architecture-specific code to be interleaved throughout the application code, heterogeneous systems place a great burden on programmers as they develop and maintain application code. Having to include architecture-specific code reduces portability and distracts programmers by requiring them to focus on various aspects of the hardware's architecture instead of focusing on the application specific code itself. Heterogeneous clusters further complicate code development since some portions of the code may have to be duplicated (reimplemented) for each node type (and even core type within a node). This is particularly true between nodes that have accelerators and nodes that do not, even if the host cores in the various nodes share the same architecture.

We believe that several aspects of the Charm++ programming model make it well suited for programming heterogeneous systems. As we will discuss in Section 2.1, Charm++ encapsulates application data in chare objects and program execution in entry methods. There are also clear communication boundaries where the structure of the application data crossing that boundary is well defined, as described in Section 4. Towards this end, we are extending the Charm++ programming model and adapting the Charm++ runtime system to allow programs to execute on heteroge-

*Corresponding author: University of Illinois (UIUC), Department of Computer Science, 201 N. Goodwin Ave, Urbana, IL 61801, USA. Tel.: +1 217 333 5827; Fax: +1 217 2446306; E-mail: kunzman2@illinois.edu.

neous systems, while at the same time, trying to minimize programmer effort. We have extended the programming model to include *accelerated entry methods*, *accelerated blocks* and a *SIMD instruction abstraction*. These extensions were also described in an earlier paper [22], which is subsumed by the work presented here. In this work, we demonstrate heterogeneous execution using a simple molecular dynamics (MD) code running on a heterogeneous cluster comprised of Xeon-based (x86) and Cell-based nodes. To achieve performance on such a cluster, we introduce a static load balancer to the MD code and demonstrate that the program is able to achieve greater scaling performance using a combination of both Xeon and Cell processors, compared to only using Cell-based nodes which we were limited to in our previous work. Our work enables a programmer to use the Charm++ programming model to write application code for all of the processing cores in a heterogeneous system, including accelerator cores. Using a single programming model, the programmer is able to write their application code once. The application code remains portable between various homogeneous systems and heterogeneous systems that may include accelerators. An underlying runtime system maps the execution of the program across the available processing cores, making use of the architecture specific features of the individual cores, such as SIMD instruction extensions, scratchpad memories, direct memory transfers and so on. When executed using all the core types in our cluster, the MD program makes use of three core types (Xeon cores, PPEs and SPEs, all using different ISAs), three SIMD instruction extensions (SSE on the Xeon cores, AltiVec/VMX on the PPEs and the SPE's own SIMD instructions), and using two memory structures (cache hierarchies in the Xeon and PPE cores, and scratchpad memories in the SPEs). Regardless of these architecture differences, by making use of our programming model extensions and our build process and runtime system modifications, there is no architecture-specific code in the MD program. Furthermore, the MD program, without modification, retains portability to other systems, including typical homogeneous clusters (e.g., a cluster comprised of commodity x86 processors).

The programming model extensions and runtime system modifications that we discuss in this work currently support the Cell processor, and we are working on support for Larrabee devices. However, Larrabee hardware is not publicly accessible, thus we are currently unable to discuss our work on Larrabee in detail. Therefore, our discussion here will focus on our work as it relates to the Cell processor in particular. The reader should keep in mind that our programming model and runtime system modifications to Charm++ are applicable to Larrabee devices as well.

2. Background

This section briefly reviews background information on Charm++ and the supported accelerator technologies as they relate to the discussion of our work.

2.1. Charm++

The Charm++ programming model is typically used to write high performance computing (HPC) and scientific computing applications. Target hardware platforms range from desktop workstations to some of the largest supercomputer clusters currently available. Charm++ has been used to develop several scientific applications, including NAMD [6], OpenAtom [7] and ChaNGa [18].

Charm++ is an object-based message-passing parallel programming model that makes use of an adaptive and intelligent runtime system. Charm++ is based on the C++ programming model. The Charm++ runtime system is implemented as a set of libraries that are linked to the application at build time. Charm++ applications are decomposed into a collection of objects called *chares*. Chare objects are C++ objects with special member functions called *entry methods*. The main differences between entry methods and standard member functions are that entry methods can be remotely invoked and do not return a value. That is, an entry method of one chare object can invoke an entry method on another chare object regardless of whether or not the two chare objects are physically located on the same processing core. The parameters passed to an entry method are packed into a message and delivered to the target chare object (the chare object the entry method is being invoked on). If the chare objects are on different physical processors, the runtime system locates the physical processor that contains the target chare object and then routes the message over the network to the appropriate physical core. Once the message has arrived on the target processor, it is queued for execution in a message queue. The runtime system dequeues messages from the message queue and schedules the associated entry methods for execution on the associated chare objects, based on various factors such as message priorities. There is at most one entry method execut-

ing on any chare object at any given moment. As a result, there is no need to synchronize access to member variables that are accessed by multiple entry methods executing on the same chare object. There are many chare objects on each physical processor (referred to as *over-decomposition* of the Charm++ program, or *virtualization*). Typically at least one chare object has a message waiting for it in the message queue while one or more other chare objects are waiting for messages to arrive. This leads to a natural overlap of communication and computation as messages are moving through the network while the physical cores are executing messages that have already arrived and waiting in the message queue to be executed.

Unlike other parallel programming models that use threads, such as MPI [14], the main programming constructs in Charm++ are chare objects. The application data is distributed amongst the member variables of the chare objects. Entry methods, when invoked on the chare objects, perform their specified operations on the chare object's local data and invoke other entry methods on other chare objects. The chare objects themselves are well encapsulated (chare objects do not access member variables of other chare objects), leading to well defined working sets for entry methods. Furthermore, chare objects can migrate between physical cores under the control of the runtime system for the purposes of dynamic load balancing. More information about the Charm++ programming model, the Charm++ runtime system, or any of the features available to Charm++ programs, such as dynamic load balancing, fault tolerance, and automatic checkpointing, is located on the Charm++ website [9].

2.2. The Cell processor

The Cell processor [20] is a multicore chip composed of two types of processing cores. The first type of processing core is called the Power Processing Element (PPE). The PPE is similar to a standard commodity core in that it has access to both the network and system memory. The second type of processing element is the Synergistic Processing Element (SPE). SPEs are specialized cores designed to perform large amounts of computation. On a single Cell chip, typically there is a single PPE with four to eight SPEs serving as slaves to the PPE.

Unlike commodity multicore processors, where all of the cores are peers, the SPE cores in the Cell processor are slaves to the PPE core. Further, the SPE can only access data contained within their *local stores*. Local stores are 256 KB scratchpad memories that contain all of the data available to the SPEs, including code, heap memory, and stack memory. The local stores are completely under the control of the programmer and require the programmer to use direct memory accesses (DMAs) to transfer data between the local store and system memory. The specialized nature of the SPEs, which contain the great majority of the processing power, make programming the Cell difficult and time consuming. Because the SPEs are so specialized and because they are slaves to the PPE core on the same chip, we refer to the SPEs as being accelerator cores available to the PPE.

2.3. Larrabee

Larrabee [30] was originally designed to perform graphics calculations. However, the architecture is quite flexible and is likely to accelerate many HPC computations. A host core may have one or more Larrabee devices available to it which perform various calculations on the host core's behalf. That is, the Larrabee device itself does not function as the system's host core, unlike the Cell where the PPE can serve as a host core. Our programming model extensions and runtime system modifications described in this paper are also being adapted to support Larrabee hardware. The goal is that the same code, when written using our programming model extensions, will be portable between standard homogeneous systems, heterogeneous systems, and systems that include Cell processors and/or Larrabee devices.

3. Extensions to Charm++

We have extended both the Charm++ programming model and the Charm++ runtime system to support heterogeneous systems with and without accelerator cores. In this section, we discuss our modifications, starting with our extensions to the programming model.

3.1. Programming Model Extensions

We have introduced three extensions to the Charm++ programming model: *accelerated entry methods*, *accelerated blocks* and a *SIMD instruction abstraction*. By using these extensions, Charm++ programs can take advantage of supported accelerator hardware while remaining portable between various

hardware platforms that make use of supported accelerators. While these abstractions are capable of abstracting away hardware differences, we have also provided macros that the programmer can use to identify which core type the code is being compiled for. Thus, we do not inhibit programmers from including architecture-specific code if they so desire (e.g., for performance reasons). However, there is no requirement that the programmer include any architecture-specific code (e.g., the molecular dynamics code we present in Section 5 does not include any architecture-specific code).

3.1.1. SIMD instruction abstraction

The SIMD instruction abstraction is a fairly straightforward abstraction of the commonly used SIMD instructions supported in modern hardware. Examples of SIMD instruction extensions that currently exist in modern processors include AltiVec/VMX, SSE, etc. SIMD instructions perform operations on special registers, called *vector* or *SIMD registers* which contain multiple data values. A typical size for a vector register is 128 bits (i.e., 4 32-bit floating point values, 2 64-bit floating point values, 4 32-bit integers and so on), though the size of a vector register is architecture dependent. Unlike scalar instructions which perform only a single operation per clock cycle, vector instructions perform the same operation on each of the values packed within the vector register in parallel. The result is that the peak performance of the processor (operations completed per cycle) is increased by a factor of the architecture's vector size (e.g., if X 32-bit floating point values fit within the architecture's vector registers, then the peak throughput rate possible for 32-bit floating point operations is increased by a factor of X, when using vector instructions compared to scalar instructions). However, SIMD instruction support varies across architectures, making vectorized code non-portable.

The point of the SIMD instruction abstraction is to make pieces of code portable between architectures while still allowing the program to take advantage of the underlying SIMD instructions supported by the hardware. Naturally, the SIMD instruction abstraction resembles the hardware vector instructions that it abstracts. Therefore, the modifications that programmers make to their code to use the SIMD abstraction resemble the code changes they would have to make when directly using any one of SEE, AltiVec/VMX, etc. The various operations provided by the SIMD instruction abstraction are mapped to the hardware's underlying SIMD instruction extensions. A generic C/C++ implementation is used on unsupported architectures and architectures without SIMD support in hardware.

Figure 1(b) illustrates how the SIMD instruction abstraction could be used to 'vectorize' or 'SIMDize' the code given in Fig. 1(a). This example is not highly optimized (i.e., it does not use loop unrolling, software pipelining, etc.), however, it demonstrates

```
1:   float *A = <pointer to data>, *B = <pointer to data>;
2:   int N = <length of both float arrays, A and B>;
3:   float C = 0.0f;
4:   for (int i = 0; i < N; i++)
5:     C = A[i] * B[i] + C;
```

(a)

```
  1:   float *A = <pointer to data>, *B = <pointer to data>;
  2:   int N = <length of both float arrays, A and B>;
2–1:   vecf *Avec = (vecf*)A, *Bvec = (vecf*)B;
2–2:   int Nvec = N/vecf_numElems;
2–3:   vecf Cvec = vsetf(0.0f);
2–4:   for (int ivec = 0; ivec < Nvec; ivec++)
2–5:     Cvec = vmaddf(Avec[ivec], Bvec[ivec], Cvec);
  3:   float C = vreduceaddf(Cvec);
  4:   for (int i = Nvec * vecf_numElems; i < N; i++)
  5:     C = A[i] * B[i] + C;
```

(b)

Fig. 1. SIMD instruction abstraction example. (a) Standard C++ code. (b) Using SIMD instruction abstraction.

how the SIMD instruction abstraction is used. The code in Fig. 1(a) uses scalar instructions to calculate the dot product between two arrays of floating point numbers of the same length, *A* and *B*. A single loop (lines 4 and 5) iterates over arrays *A* and *B*, multiplying the corresponding elements and summing all the products across all iterations into a scalar, *C*. Figure 1(b) performs the same calculation using two loops. Note that the added lines are numbered 2-*X* and basically duplicate the original code, but with modifications to use vector instructions. The first loop (lines 2–4 and 2–5) has been vectorized. By using SIMD instructions, the vectorized loop performs *vecf_numElems* fused-multiply-add operations each iteration. A fused-multiply-add operation multiplies *Avec[ivec]* and *Bvec[ivec]* and then adds that result to *Cvec*. One iteration of the vectorized loop in Fig. 1(b) performs the same amount of work as *vecf_numElems* iterations of the original loop in Fig. 1(a). In other words, the code is able to more efficiently make use of the processor's functional units and reach a higher percentage of the processor's peak performance by performing more flops per cycle (and more flops per instruction). Once the vectorized loop in Fig. 1(b) has completed, the *Cvec* variable contains *vecf_ numElems* partial sums that are added together, using *vreduceaddf()*. If the length of the *A* and *B* arrays is not an integer multiple of the hardware's vector size (i.e., if the condition '*N % vecf_numElems !*= 0' evaluates to true), then there will be a few remaining elements that the vectorized loop did not compute (less than *vecf_numElems*). These remaining elements will be processed by the second non-vectorized loop (lines 4 and 5) with the final result being placed in the scalar *C*.

While the SIMD instruction abstraction makes pieces of vectorized code portable between cores with differing SIMD instruction extensions, this is not enough to abstract the differences between a host core (e.g., the PPE core on a Cell processor or an x86-based host core) and any available accelerator cores (e.g., an SPE on Cell or a Larrabee core). To further assist the programmer, we have introduced *accelerated entry methods* and *accelerated blocks*.

3.1.2. Accelerated entry methods and blocks

By design, accelerated entry methods are similar to standard entry methods. However, unlike standard entry methods which never execute on an accelerator, accelerated entry methods may or may not execute on a supported accelerator, if one is available. By marking an entry method as accelerated, the programmer is indicating to the runtime system that the code is suitable for executing on an accelerator core.

While accelerated entry methods are similar to standard entry methods, they have some notable differences. First, the entry method is marked with an *accel* keyword. Second, in addition to the passed parameters of standard entry methods, accelerated entry methods have *local parameters*. Local parameters list the member variables of the chare class that the accelerated entry method will require access to, along with the type of access (read-only, read-write or write-only). Third, the function body of the entry method is declared in the interface file instead of the source code file. Interface files are specific to Charm++ and are used by the programmer to declare chare classes and entry methods (similar to header files in C++ being used to declare classes and member functions). Function bodies of accelerated entry methods are declared in the interface files to give the Charm++ build tools, which are currently not capable of manipulating the C++ source code files directly, the ability to manipulate the accelerated entry method's function body. Fourth, there is a *callback function* associated with the accelerated entry method. The callback function is executed on the host core after the accelerated entry method has completed executing on the accelerator core and any updates to the chare object's state (member variables) is reflected in system memory. The callback function is one of the member functions or entry methods on the same chare object. Fifth, the function body of accelerated entry methods are somewhat restricted in their functionality. For example, other entry methods cannot be invoked directly from the function body of an accelerated entry method. Instead, any entry method invocations must be moved to the associated callback function which will be executed on the host core.

Figure 2(a) presents a simple entry method that performs a dot product calculation on two incoming arrays, *A* and *B*, both *N* elements in length. *C* and *targetChare* are both member variables of the *MyChareClass* chare class. *C* will contain the calculated result of the dot product and *targetChare* indicates which chare object will receive the results. In this example, only one chare object will be sent the result, but any number of chare objects could be passed the data.

Figure 2(b) presents an accelerated version of the standard entry method presented in Fig. 2(a). The keyword *accel* has been used to indicate that the entry method is accelerated. In addition to the passed parameters, the local parameter *C* is listed. In this case, *C* is designated *writeonly*. The value of *C* will not be

```
///// Interface File (Charm++ Specific) /////
entry dotProduct(int N, float A[N], float B[N]);

///// Source Code File /////
void MyChareClass::dotProduct(int N, float *A, float *B) {
  C = 0.0f;
  for (int i = 0; i < N; i++)
    C = A[i] * B[i] + C;
  targetChare.sendResult(C);
}
```

(a)

```
///// Interface File (Charm++ Specific) /////
entry [accel] dotProduct(int N, float A[N], float B[N])
      [writeonly : float C <impl_obj->C>] {
  C = 0.0f;
  for (int i = 0; i < N; i++)
    C = A[i] * B[i] + C;
} dotProduct_callback;

///// Source Code File /////
void MyChareClass::dotProduct_callback() {
  targetChare.sendResult(C);
}
```

(b)

Fig. 2. Accelerated entry method example. (a) Standard entry method. (b) Accelerated entry method.

initially read from system memory before the accelerated entry method starts. By default, local parameters are read-write and the programmer may mark them as read-only or write-only to minimize the amount of data being moved between the host and accelerator cores. The function body of the accelerated entry method will be executed on an available accelerator, if an accelerator is present. If no accelerator is present, the function body will be executed on the host core. Once the function body of the accelerated entry method has completed, the callback function, *dotProduct_callback()*, will be executed on the host core. In this example, the invocation of the *sendResult* entry method (on chare object *targetChare* and with passed parameter *C*) must be moved to the body of the callback function since accelerated entry methods cannot directly invoke other entry methods within their own function bodies. The result of the code in Fig. 2(b) is the same as in Fig. 2(a), except that the dot product calculation in Fig. 2(b) may be scheduled by the Charm++ runtime system on any supported accelerator that is present. While it is likely that accelerated entry methods will be executed on available accelerator cores, the runtime system may choose to execute some accelerated entry method invocations on the host core(s) for the purposes of load balancing work between host and accelerator cores. While it is already the case that accelerated entry methods are portable between host and accelerator cores, the development of an automatic and dynamic load balancing scheme to decide what percentage of accelerated entry method invocations should be executed on accelerator cores versus host cores is still future work. Currently, all accelerated entry methods are executed on accelerator cores by default when an accelerator is present.

In addition to accelerated entry methods, we have also introduced *accelerated blocks*. Accelerated blocks are simply blocks of code that exist at the 'global scope' on both the host and accelerator, providing areas where *#defines* and global functions can be declared. For example, consider a case where two accelerated entry methods overlap in functionality. It would be advantageous if the programmer could code the shared functionality into a global function that can then be called, perhaps with varying parameters, by multiple accelerated entry methods. Figure 3 gives a simple example case, building on the previous dot product example from Fig. 2. In this case, there are two accelerated entry methods, *doProduct_two* and *dotProd-*

```
///// Interface File (Charm++ Specific) /////
accelblock {
  float do_dot_product(int N, float *A, float *B) {
    float C = 0.0f;
    for (int i = 0; i < N; i++)
      C = A[i] * B[i] + C;
    return C;
  }
}

entry [accel] dotProduct_two(int N, float A[N], float B[N])
      [writeonly : float C <impl_obj->C>] {
  C = do_dot_product(N, A, B);
} dotProduct_callback;

entry [accel] dotProduct_one(int N, float A[N])
      [writeonly : float C <impl_obj->C>] {
  C = do_dot_product(N, A, A);
} dotProduct_callback;
```

Fig. 3. An example of using a function declared in an accelerated block to avoid duplicating code in multiple accelerated entry methods.

uct_one that both make use of the 'global function' *do_dot_product*. Incidentally, both accelerated entry methods use *dotProduct_callback* as their callback functions, however, there is nothing requiring multiple accelerated entry methods in the same chare class to use the same callback function. Just as multiple member functions of one or more classes may call a global function in C++, our extensions also support multiple accelerated entry methods calling a global function. However, because the build tools need to understand which global functions need to be present on the accelerator device, those global functions need to be declared in accelerated blocks (i.e., accelerated blocks indicate the global pieces of code that should be included on both the host and accelerator cores, just as marking an entry method as accelerated indicates that it needs to be available both on the host and the accelerator cores).

4. Heterogeneous execution

In addition to the programming model extensions that we have introduced into Charm++, we have also made modifications to the Charm++ runtime system and build tools to support heterogeneous execution. In a way, Charm++ programs can already execute on heterogeneous clusters, as long as the host cores have the same architecture (i.e., x86 cores with different clock speeds, cache sizes and/or different generations of the architecture). With the extensions discussed in Section 3, this is also true of Cell processors (i.e., a mixture of PS3s and Cell blades). However, porting the code between the various host cores in heterogeneous clusters of this sort is fairly trivial (not even requiring a recompilation of the application code is most cases). We would like to allow Charm++ programs to support heterogeneous clusters in which the host cores have different architectures (e.g., a mixture of x86 and Cell processors) without requiring the programmer to modify their application code.

Writing code for heterogeneous clusters in which the host cores have differing architectures forces programmers to spend time handling mundane details, such as endianness and pointer size differences. However, in the context of Charm++, the runtime system can take advantage of the clearly defined communication boundaries between chare objects to automatically handle several of these mundane details. Chare objects act on data that they store locally and only pass data between one another through entry method invocations. The structure of data that being passed between chare objects during entry method invocations is clearly defined in the parameter list to the entry method (including data types and array lengths as specified in their interface file declarations). Therefore, the structure of the programming model (chare objects with remotely invoked entry methods) already causes the programmer to provide the build tools and runtime system with the information required to automatically modify application data to account for architectural differences between host cores.

We have modified the Charm++ build tools and runtime system to take advantage of the information being provided by the application code. As entry methods are invoked, Pack-UnPack (PUP) routines [19] are invoked for each of the passed parameters being passed to the entry method. As the sending processor packs the passed parameters into a message, the runtime system adds information to the message that indicates how the data is encoded. After the message is sent to the target processor, the runtime system uses this additional information to modify the passed parameters (application data) to match the target processor's architecture as the parameters are being unpacked from the message. The net result is that the programmer is not concerned if the invoking and invoked entry methods happen to be executing on processors with differing architectures. In writing their application code, programmers automatically provide data type and length information to the runtime system as they define the passed parameters for any entry method. With our runtime system and build process modifications in place, no modification to the application code is required to port between homogeneous and heterogeneous clusters, allowing the programmers to focus on their application code.

5. Performance

In this section, we use a simple molecular dynamics (MD) program to demonstrate the performance of our programming model extensions and runtime system modifications on a heterogeneous cluster, composed of Xeon 5130 cores, IBM QS20 Cell blades, and Sony Playstation 3s. We start by describing the MD program (Section 5.1) and the heterogeneous cluster (Section 5.2). Finally, we present results in Section 5.3.

5.1. Example MD program description

To demonstrate the performance of our programming extensions and runtime system modifications, we have written a simple molecular dynamics (MD) code. The code is included in the Charm++ distribution. The structure of the MD code is modeled after the nonbonded electrostatic calculations in a production molecular dynamics code called NAMD [6], in that it is composed of *Patch*, *PairCompute* and *SelfCompute* chare objects. The compute objects perform the force calculations, one SelfCompute per Patch and one PairCompute per pair of Patches. Once the compute objects calculate the forces, the force values are passed back to the Patch objects, which will then integrate (updating particle positions and velocities) and initiate the next simulation iteration by triggering the compute objects yet again. Compared to NAMD, the MD code is greatly simplified and has been implemented using accelerated entry methods. Our MD code calculates forces by applying Coulomb's Law using a non-cutoff algorithm, which runs in O(N^2) time. Even though our MD code uses an O(N^2) algorithm, we have chosen a problem size (92,160 particles across 144 Patches) that creates less than a factor of two more compute objects than NAMD does for the ApoA1 benchmark system. The number of compute objects in both NAMD and our MD code is directly related to the amount of parallelism in the simulations.

We have also extended the example MD code to include a static weight-based load balancer. The load balancer gives each chare object type a weight for each type of host core (with QS20 PPEs being different that PS3 PPEs). When the MD code is executed, on a per-object-type-basis, the weights for the given object type for each available core are summed. Then, for each host core, the weight for that individual core (for the given object type) is divided by the total sum of weights of all the cores (for the given object type) to calculate the fraction of chare objects of the given type that host core should receive. For example, if there are two host cores, a Xeon and a PS3 PPE, with equal weights for a given object type, then all the objects of that type will be equally divided between the Xeon and the PS3 PPE. However, if the PS3's weight were twice that of the Xeon's weight, then two thirds of the objects of that type will be placed on the PS3 PPE and only one third will be assigned to the Xeon. Using this weight-based static load balancer, we can control how the chare objects, on a per-object-type-basis, are distributed across the various host cores.

5.2. Experimental setup

The performance data presented in Section 5.3 was collected using a heterogeneous cluster comprised of four IBM QS20 Cell Blades (*QS20s*), four Sony Playstation 3s (*PS3s*), and one node with a dual-core Intel Xeon 5130 (*Xeon cores*) connected via a Gigabit Ethernet network. Each QS20 blade has 2 Cell processors running at a clock speed of 3.2 GHz, where each Cell has 8 SPEs with a combined peak flop rate of 204.8 GFlop/s and one PPE with a peak flop rate of 25.6 GFlop/s. The PS3s have a single Cell proces-

sor running at a clock speed of 3.2 GHz. The PS3 Cells have 6 SPEs with a combined peak flop rate of 153.6 GFlop/s and one PPE with a peak flop rate of 25.6 GFlop/s. Each Xeon core runs at a clock speed of 2.0 GHz and has a peak flop rate of 16.0 GFlop/s. The execution times of ten separate executions were averaged to create each data point, unless stated otherwise. We only use one Cell processor per QS20 (the reason for this is related to the network performance of the QS20 Cells in our cluster, as will be discuss in Section 5.3).

5.3. Results

We have divided the results section into three subsections. First, we establish a baseline measurement of the MD program's performance in Section 5.3.1 using only Cell processors (these results update previous results [22]). Then, in Section 5.3.2, we further improve performance of the MD program by applying a static load balancer. While the load balancer improves the program's overall performance, it is less effective as the program scales to a larger number of Cell processors. However, somewhat surprisingly, when Xeon cores are introduced into the hardware configuration, the program scales better compared to a Cell-only hardware configuration. Finally, to better understand why the MD code scales better with a mixture of Xeon and Cell processors compared to using Cell processors alone, we explore the effects of communication on the MD program's performance in Section 5.3.3.

5.3.1. Results: Baseline (without load balancing)

To clearly understand how the load balancer effects the performance, we first measure the performance of the MD program without using the load balancer. The first three plots in Fig. 4 demonstrate the performance of the MD code executing on PS3s, QS20 Cell blades, and a combination of PS3s and Cell blades (*Cell pairs*), respectively. A *Cell pair* is one PS3 Cell and one blade Cell combined (e.g., 4 Cell pairs represents four blade Cells and four PS3 Cells). Ideally, the performance of the *"QS20/PS3 Cell Pairs (no LDB)"* plot would be equal to the performance of the *"PS3 Cells"* and *"QS20 Cells"* plots summed together (the *"QS20 Cells + PS3 Cells"* reference plot). However, because the workload is not being load balanced and the PS3s have a lower peak GFlop/s rate, the measured performance more closely follows the performance of the *"PS3 Cell"* plot multiplied by two (since the QS20 Cells are being underutilized).

To help give the reader some context by which to judge the performance achieved in Fig. 4, we point out that our previous work [22] showed that the maximum performance achievable by the PairCompute's force kernel running on a single SPE with an infinite

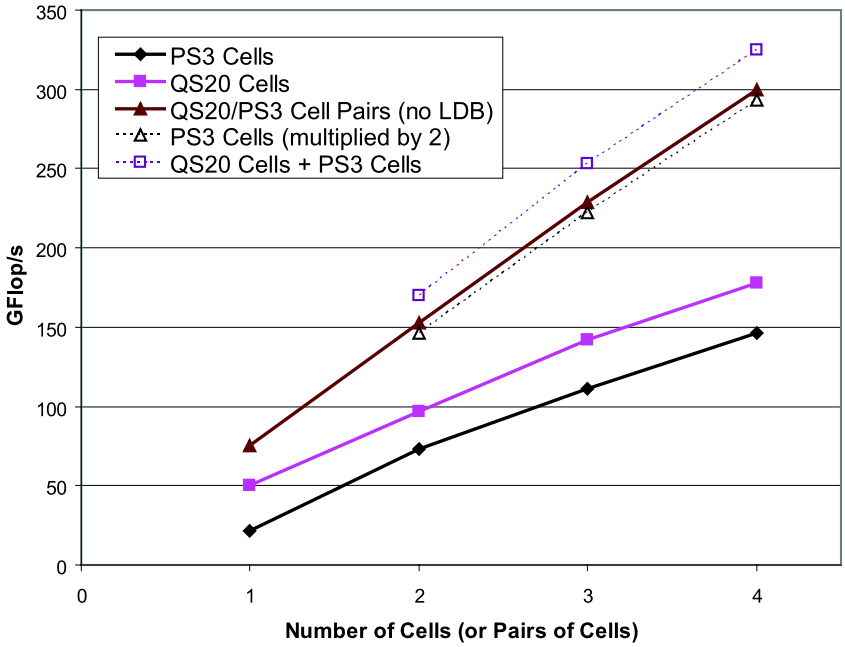


Fig. 4. Performance of the simple MD code executing on QS20 Cells, PS3 Cells and a combination of QS20 and PS3 Cells.

local store is 27.7% of the SPE's peak GFlop/s rate. This upper bound is calculated by analyzing the performance of the inner-most loop of the force calculation, which performs the great majority of the work in the MD program and has been optimized to have a high ratio of flops per cycle. On an SPE, the inner-most loop performs 124 flops in 56 cycles using 54 instructions (average of 2.21 flops/cycle or 27.7% of the peak flop rate). This upper-bound, however, is unachievable as it assumes an infinite local store, ignores data movement between the local store and system memory, and any other overheads actually encountered by the code. For the moment, we will only consider the SPEs because only the SPEs are doing physics calculations. The physics calculations are the only type of calculations that are being included when calculating the total flops performed by the program. Other flops, such as the summing of individual forces for a given particle (contributed by multiple compute objects executing on different SPEs) into a single force, are not being counted towards the total flops performed by the MD program since these flops would not be required in a serial version of the code. In the case of a single QS20 Cell, the code reaches a measured rate of 50.1 GFlop/s. This is 88.4% of the 56.7 GFlop/s that would be seen if the code were to reach the unachievable upper bounds performance (204.8 GFlop/s peak rate across all 8 SPEs multiplied by 27.7%). In the case of 4 Cell pairs without load balancing, the code is achieving 299.7 GFlop/s (20.9%) of the SPEs' combined peak performance or 75.5% of the unachievable upper bounds performance (1433.6 GFlop/s across all 56 SPEs multiplied by 27.7% is 397.1 GFlop/s). If we include the PPEs, which are not doing any of the physics calculations, the single QS20 Cell case reaches 21.7% of the QS20 Cell's peak GFlop/s. The 4 Cell pair case reaches 18.3% of the total peak GFlop/s summed across all 64 cores (8 PPEs and 56 SPEs). Note that 18.4% of the total peak means that the MD program is performing an average of 1.46 flops per core per cycle.

5.3.2. *Results: With load balancing*

The performance of the MD program is further improved by applying the static load balancer discussed in Section 5.1. Figure 5 presents the performance achieved by the MD code after static load balancing has been applied. Reference plots and the non-load balanced *"QS20/PS3 Cell Pairs (no LDB)"* plot from Fig. 4 have been included in Fig. 5 for the sake of comparison. In the *"QS20/PS3 Cell Pairs (Max LDB)"* plot, we use the weight-based static load balancer to distribute the workload across the QS20 Cells and the PS3 Cells based on various factors. The weights were chosen by hand. Because the PairCompute objects represent the great majority of the workload, we initially chose to load balance only the PairCompute objects

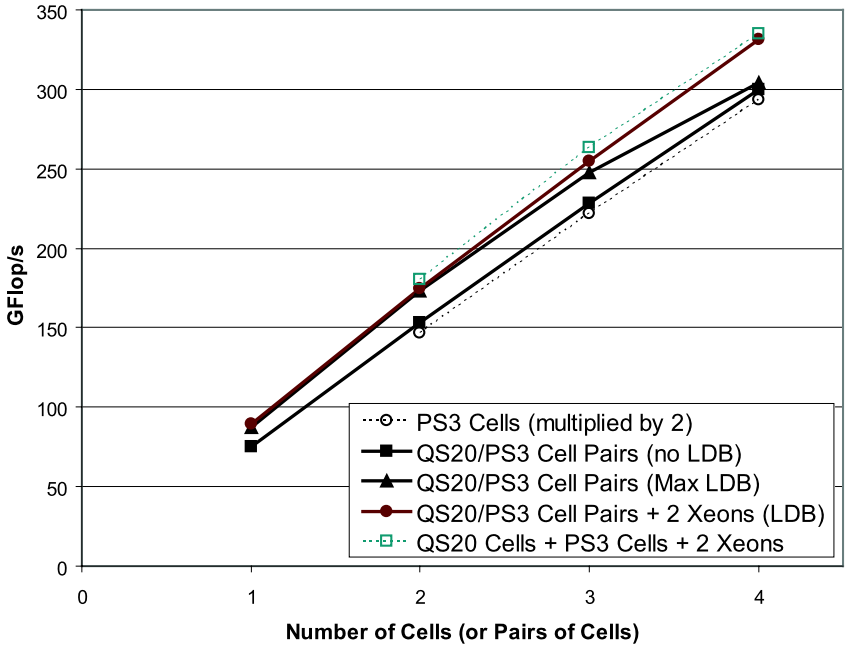


Fig. 5. Performance of the simple MD code executing on combinations of QS20 Cells, PS3 Cells and Xeon 5130 cores.

based on peak flop rates (*LDB Method* 1). While this helped at lower processor counts, it was less effective as the processor count increased. Then, we additionally load balanced the Patch and SelfCompute objects, thinking that second order effects were causing the decrease in the load balancer's effectiveness as the processor count increased (*LDB Method* 2). However, this method also showed a decreased effectiveness at four QS20/PS3 Cell pairs. We further modified the weights related to all three object types so that each object type was weighed differently between QS20s and PS3s, trying many individual weight combinations (e.g., most Patch objects are placed on PS3 Cells while most PairCompute objects are placed on QS20 Cells) (*LDB Method* 3). At each point in the *"QS20/PS3 Cell Pairs (Max LDB)"* plot, we use the load balancing method that resulted in the best performance for the given number of QS20/PS3 pairs (i.e., we varied the load balancing method, weights used, and so on). Using 4 Cell pairs, the MD program achieved 303.9 GFlop/s (only 4.2 GFlop/s greater than the non-load balanced performance). 303.9 GFlop/s represents 18.55% of the total peak performance (including all SPEs and PPEs).

Before discussing why the load balancer does not perform as well as the number of QS20/PS3 pairs increases, regardless of the load balancing method or choice of weights, we first describe the *"QS20/PS3 Cell Pairs + 2 Xeons (LDB)"* plot in Fig. 5 which does not suffer from the same scaling issue. In this plot, we add two Xeon cores to the hardware configuration at each point (i.e., *X* Cell pairs means *X* blade Cells, *X* PS3 Cells and 2 Xeon cores). LDB Method 3 is used for this configuration to place the majority of the Patch objects on the Xeons and the majority of the compute objects on the Cells. This plot closely follows the *"QS20 Cells + PS3 Cells + 2 Xeons"* reference plot (which adds the measured performance values of the *"QS20 Cells"* and *"PS3 Cells"* plots in Fig. 4 with the measured performance of 10.09 GFlop/s on two Xeon cores). Using 4 Cell pairs and 2 Xeon cores, the MD code achieves 331.1 GFlop/s (19.82% of the total peak performance including all SPE, PPE and Xeon cores). Even though the great majority of the peak performance of this hardware configuration is being provided by the Cell processors (1638.4 GFlop/s from Cell processors, compared to the 32.0 GFlop/s provided by the Xeon cores), the presence of the Xeon cores has a significant effect on the MD code's scaling performance. Adding the two Xeon cores increases the performance of the MD program by 27.2 GFlop/s (85% of the Xeon cores' combined peak flop rate) over the load balanced 4 Cell pair case, even though the two additional Xeon cores are not actually performing all of this additional work (as discussed below).

5.3.3. Results: Effects of communication

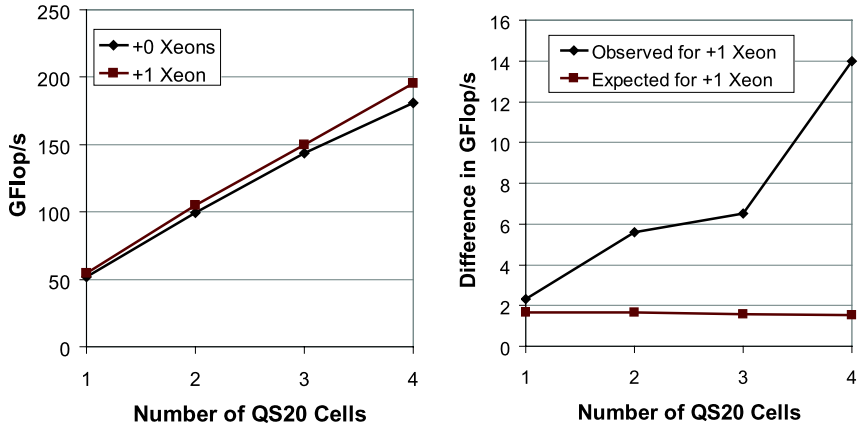
We want to understand why the simple MD program does not scale as well when the Xeon cores are not present, despite our attempts to load balance the workload in Section 5.3.2. Towards this end, we used the Projections [21] performance visualization tool, to analyze the performance details of the 4 Cell pair case from the *"QS20/PS3 Cell Pairs + 2 Xeon (LDB)"* plot in Fig. 5 (4 QS20 Cells, 4 PS3 Cells, 2 Xeon cores). What we noticed was that the message send times for the QS20 Cell Blades in our cluster were significantly greater than message send times for either the PS3 Cells or the Xeons cores. The message send times are presented in Table 1. There are two things to note about the numbers in Table 1. First, the time values include all overhead costs associated with invoking an entry method, including processor lookup times, parameter serialization, network message send time, and so on. Second, the exact same binary was used for both the PS3 and QS20 Cells.

With this communication difference in mind, we simplify the experiment to better understand how this communication difference is affecting the performance of the MD program. To simplify our experiment, we remove the PS3s from our configuration and limit ourselves to one additional Xeon core. Figure 6(a) shows how the MD program scales from one to four QS20 Cells with (*"+1 Xeon"*) and without (*"+0 Xeons"*) an additional Xeon core. Since we are only using QS20

Table 1
Average send times (entry method invocations) when passing a force data message (approximately 7.7 KB) over the network

PE	PE type	Total time (ms)	No. samples	Avg. time (µs)
0	x86	279	1664	168
1	x86	336	1696	198
2	blade	1488	4256	350
3	PS3	967	4480	216
4	blade	2931	4256	689
5	PS3	1078	4480	241
6	blade	2479	4352	570
7	PS3	991	4480	221
8	blade	2705	4320	626
9	PS3	995	4480	222

Note: These times include all associated overheads, such as target processor lookup and serialization of data into the message payload (1 run per sample set).



(a) (b)

Fig. 6. Scaling of the MD code on QS20 Cells, with and without an additional Xeon core.

Cells in the *"+0 Xeons"* plot, the weight values used by the load balancer are inconsequential since all chare objects, regardless of type, will be evenly distributed across the nodes. When the single Xeon is included, the weights simply control how many objects of each type are placed on the Xeon and the remaining objects are evenly distributed across the QS20 Cells. In particular, because the QS20 Cells have long message send times as shown in Table 1, we have set the load balancer weights so that the great majority of the Patch objects (more communication intensive) are placed on the Xeons while most of the compute objects (more computation intensive) are placed on the QS20 Cells. As the figure shows, the configuration that includes the Xeon core not only exhibits an increase in performance for any number of QS20 Cells, but also scales better. In other words, the simplified hardware configuration (only QS20s) exhibits the same improvement in scaling when a Xeon core is included, as was seen in the more complicated hardware configurations of Fig. 5 that used both QS20 and PS3 Cells. To illustrate the difference more clearly, the *"Observed for +1 Xeon"* plot in Fig. 6(b) shows the difference between the two lines in Fig. 6(a). Because we know how many objects of each type are being placed on the Xeon and because the work done by each object type is constant per timestep (due to the regular nature of the all-to-all algorithm), we can precisely calculate the number of GFlops that the Xeon is performing (the *"Expected for +1 Xeon"* plot in Fig. 6(b)). It is not surprising that the expected increase in GFlop/s presented in Fig. 6(b) is less than the observed GFlop/s rate of the MD program executing on a single Xeon core (5.07 GFlop/s) because we are increasing the ratio of communication intensive Patch objects to computation intensive compute objects. That is, the ratio of PairCompute objects to Patch objects on the Xeon core varies depending on the hardware configuration (72.5:1 when using only a single Xeon, slightly less than 3:1 when using one QS20 Cell and one Xeon, and approximately 1:1 when using four QS20 Cells and one Xeon). The Xeon core is clearly not providing all of the additional GFlop/s observed. Instead, most of the additional GFlop/s are coming from the QS20 Cells themselves.

We use Projections to sum the communication time spent across all of the host cores. We are forced to use a single run with a smaller number of total timesteps because of the large amount of timing output we are having Projections create so we can isolate communication time. In the four Cell case, adding a single Xeon core decreases the amount of time the QS20 Cells spend performing communication by 4.5 processor-seconds (4.6% of the total processor-seconds of the QS20 Cells). When the Xeon core is present, the 4 QS20 Cells are spending a higher fraction of their time performing force calculations and providing the majority of the 13.99 GFlop/s performance increase seen in Fig. 6(a). In fact, we can calculate that the Xeon core is averaging 1.55 GFlop/s of actual work, and therefore, the QS20 Cells are providing an additional 12.44 GFlop/s (combined). This simplified experiment strongly indicates that the presence of the Xeon cores is causing the improved scaling performance of the *"QS20/PS3 Cell Pairs + 2 Xeons (LDB)"* plot compared to the *"QS20/PS3 Cell Pairs (Max LDB)"* plot in Fig. 5, even though the conditions are not identical (i.e., the PS3 Cells differ from the QS20 Cells).

Finally, we would like to explicitly point out that we demonstrated the MD program executing on 66 cores

of three different types (2 Xeons, 8 PPEs and 56 SPEs used in the 4 Cell pair case of the *"QS20/PS3 Cell Pairs + 2 Xeon (LDB)"* plot in Fig. 5). Moreover, there are three different ISAs in use, three different SIMD instruction extensions in use (SSE, AltiVec/VMX and the SPE's SIMD extensions), and two different memory structures in use (PPEs and Xeons use cache hierarchies, while the SPEs use scratchpad memories). However, by using our programming model extensions and runtime system modifications, there is no architecture-specific code in the program, nor is there any application code that corrects for architecture differences between the cores.

5.3.4. Results: Summary

To summarize our results, we first showed some baseline results of the MD code scaling from 1 to 4 Cell pairs in Section 5.3.1 without using load balancing. In Section 5.3.2, we added a load balancer to the MD code, increasing the MD code's performance. However, as the number of Cell pairs increased, the effectiveness of the load balancer was diminished. By the time the program was scaled to 4 Cell pairs (64 total cores), the added performance from load balancing was minimal. We further added two Xeon cores to the hardware configurations. When the Xeons were used in conjunction with the load balancer, the scaling performance improved. For example, in the case of 4 Cell pairs, the addition of the Xeon cores only represent an increase of 1.95% in the peak performance (increased from 1638.4 GFlop/s to 1670.4 GFlop/s), while the measured performance of the MD code increases by 8.95% (from 303.9 GFlop/s to 331.1 GFlop/s). To understand why the mixture of Xeon and Cell processors scales better than Cell processors alone, we simplify our experiments in Section 5.3.3. The simplified experiments demonstrate that the Xeon cores exhibit better communication performance, and thus, the workload can be balanced across the cores in a way that most effectively maps the type of work being performed to the type of processor best suited for that type of work (i.e., communication heavy Patch objects on Xeon cores and computation heavy compute objects on Cell processors). Finally, we demonstrated heterogeneous execution of the MD code on a total of 66 cores (2 Xeon cores, 8 PPE cores and 56 SPE cores), using 3 different node types (Xeon-based node, Playstation 3s, and QS20 Cell Blades), 3 different ISAs, 3 different SIMD instruction extensions, and 2 different types of memory hierarchies (cache hierarchies and scratchpad memories). Despite the heterogeneous nature of the hardware configuration, the MD code does not include any architecture-specific code and remains portable to other clusters by making use of our programming model extensions and runtime system modifications.

6. Related work

Many programming models focus on assisting the programmer in writing code for a single node. That is, a single host processor with one or more cores and one or more accelerators that are directly attached. Some models that fall into this category include OpenCL [17], Ct [15], CUDA [26], CellSs [5], Mercury's Multicore Framework [8], RapidMind [24] and Sequoia [13]. While several of these models target single nodes that are heterogeneous, our work differs in that it specifically targets heterogeneous clusters, where the individual nodes are not only internally heterogeneous, but are architecturally different from one another (i.e., an x86-based node and a Cell-based node). Further, in our work, the same programming model and parallel constructs are used to make use of all of the cores, host or accelerator. These programming models require a separate programming model, such as MPI [14] or OS TCP/UDP sockets, to communicate data and express parallelism between multiple nodes (i.e., network communication on a distributed memory cluster).

Some existing parallel programming models have been extended to support accelerators and heterogeneous computing. Two approaches, Open MPI [16] and HeteroMPI [23], extend the MPI programming model to support heterogeneity between host cores. Unlike our work, they do not directly address the presence of accelerators or architectural differences such as different SIMD instruction extensions supported by hardware. However, we do not support network heterogeneity, which is supported by Open MPI. Another approach, MPI microtasks [27], uses microtasks to support accelerators, however, it does not directly address heterogeneity between host cores, as we do in our work. The OpenMP [10] programming model is used by the XLC single source compiler [12] to compile parallel programs for Cell. Once again, this approach does not directly support heterogeneous clusters.

Yet other task-based parallel programming models have been created to address heterogeneous systems that may have accelerators present. In HMPP [11], the programmer creates *codelets* (tasks) which are executed in parallel on the available processing cores, host or accelerator. However, HMPP does not directly

support communication between multiple nodes in a cluster, requiring the programmer to use another programming model, such as MPI, for host-to-host communication. StarPU [1] also makes use of codelets. However, the programmer is required to write a version of the codelet for each core type, forcing the programmer to duplicate functionality. In our approach, an accelerated entry method is written once and executed on any supported core type, host or accelerator. Like our approach, where we mark some entry methods (member functions) as accelerated, in StarSs [2, 28] some functions are marked as *tasks* which may be executed on an available accelerator. However, the tasks in StarSs are arranged into a task hierarchy, requiring that all the child tasks complete before a parent task can complete. In our approach, the chare objects, and their related entry methods, are peers to one another. However, HMPP, StarPU and StarSs all support CUDA-based GPUs. We currently do not support CUDA-based GPUs using accelerated entry methods. Instead, a different approach [31] is currently being used to make use of CUDA-based GPUs in Charm++. Unlike HMPP, StarPU and StarSs, we do provide a SIMD instruction abstraction, allowing SIMDized portions of code to be portable between multiple core types, including both host cores and accelerator cores.

Saletore et al. demonstrated execution of CHARM (a predecessor to Charm++) programs executing on a heterogeneous cluster composed of a mixture of Sun Sparc, HP-PA and IBM RS/6000 workstations [29]. Their work focuses mainly on using dynamic load balancing techniques and introduces an 'IOChare' construct to allow parallel programs to effectively utilize a cluster of workstations. Our work differs from their work in multiple ways. For example, we demonstrate execution of a Charm++ program on a mixture of processor architectures that include both big and little endian data encodings. Our runtime system modifications allow the runtime system to automatically modify application data at runtime to account for differences in those architectures (endianness, data type sizes and so on) as the data passes between the various processing cores. We additionally address the presence of accelerators (i.e., not all processing cores are peers to one another). For example, our programming model extensions use the SPEs on a Cell processor without requiring the programmer to explicitly issue direct memory accesses, manage each SPE's local store and so on. Our work also specifically addresses the presence of hardware support for SIMD instructions. Our SIMD instruction abstraction allows a programmer to write portable code that makes use of the various SIMD instructions available across all of the processing core types.

7. Conclusion

Our work has enabled Charm++ to be used as the programming model for all of the processing cores in a heterogeneous cluster and/or systems that include accelerators. The programmer writes their application code once, using a single programming model and making use of our programming model extensions presented in Section 3. The underlying runtime system maps the execution of the program across the available host and accelerator cores, making use of core specific features such as hardware support for SIMD instructions, scratchpad memories, DMA transfers and so on. This eases programmer burden by allowing them to use a single programming model to write their application code and abstracts away many of the hardware details normally associated with programming heterogeneous clusters. Our extensions allow the application code to remain portable between homogeneous systems, systems that include accelerators, and heterogeneous systems that may include accelerators. We have demonstrated our extensions and modifications by executing an example molecular dynamics (MD) code written using our programming model extensions on a heterogeneous cluster comprised of IBM QS20 Cell Blades, Sony Playstation 3s and Xeon 5130 cores. The MD program was able to achieve 331.1 GFlop/s, which is 19.82% of the peak flop rate.

Currently, our programming model extensions support Cell processors (treating SPEs as accelerators to PPEs). We are currently implementing support for Larrabee devices in Charm++ using the same programming model extensions presented here. However, because the implementation is not complete, we leave Larrabee support as future work. We leave the implementation of a dynamic load balancer as future work. However, our experience with the static load balancer used in this work has given us insight into the types of measurements that the Charm++ runtime system should make and pass along to a future dynamic load balancer.

Acknowledgements

We would like to thank IBM for providing the hardware used in this work through the SUR grant awarded

to the University of Illinois. This work was also supported in part by NIH Grant PHS 5 P41 RR05969-04. We would also like to thank several members of the Parallel Programming Lab for their advice and assistance, including Lukasz Wesolowski, Eric Bohm, Gengbin Zheng, Aaron Becker, Isaac Dooley and Chao Mei.

References

[1] C. Augonnet, S. Thibault, R. Namyst and M. Nijhuis, Exploiting the Cell/BE architecture with the StarPU unified runtime system, in: *SAMOS Workshop – International Workshop on Systems, Architectures, Modeling, and Simulation*, Samos, Greece, July 2009, Lecture Notes in Computer Science, Vol. 5657, Springer, Berlin, 2009, pp. 329–339.

[2] E. Ayguade, R.M. Badia, F.D. Igual, J. Labarta, R. Mayo and E.S. Quintana-Orti, An extension of the StarSs programming model for platforms with multiple GPUs, in: *Proceedings of the 15th International Euro-Par Conference*, Delft, The Netherlands, August 2009, Lecture Notes in Computer Science, Vol. 5704, Springer, Berlin, 2009, pp. 851–862.

[3] Barcelona Supercomputing Center, Prototypes Systems for PRACE, available at: http://www.prace-project.eu/documents/PRACE-Prototypes.pdf.

[4] K.J. Barker, K. Davis, A. Hoisie, D.J. Kerbyson, M. Lang, S. Pakin and J.C. Sancho, Entering the petaflop era: the architecture and performance of roadrunner, in: *SC'08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, IEEE Press, 2008, pp. 1–11.

[5] P. Bellens, J.M. Perez, R.M. Badia and J. Labarta, Exploiting locality on the cell/b.e. through bypassing, in: *SAMOS'09: Proceedings of the 9th International Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation*, Springer-Verlag, Berlin, 2009, pp. 318–328.

[6] A. Bhatele, S. Kumar, C. Mei, J.C. Phillips, G. Zheng and L.V. Kale, Overcoming Scaling challenges in biomolecular simulations across multiple platforms, in: *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008.

[7] E. Bohm, A. Bhatele, L.V. Kale, M.E. Tuckerman, S. Kumar, J.A. Gunnels and G.J. Martyna, Fine grained parallelization of the car-parrinello ab initio MD method on blue gene/L, *IBM Journal of Research and Development: Applications of Massively Parallel Systems* **52**(1,2) (2008), 159–174.

[8] B. Bouzas, R. Cooper, J. Greene, M. Pepe and M.J. Prelle, MultiCore framework: An API for programming heterogeneous multicore processors, Mercury Computer System's Literature Library, available at: http://www.mc.com/mediacenter/litlibrarylist.aspx.

[9] Charm++ website, available at: http://charm.cs.uiuc.edu/.

[10] L. Dagum and R. Menon, OpenMP: An industry-standard API for shared-memory programming, *IEEE Computational Science & Engineering* **5**(1) (1998), 46–55.

[11] R. Dolbeau, S. Bihan and F. Bodin, HMPP: A hybrid multicore parallel programming environment, in: *Workshop on General Purpose Processing on Graphics Processing Units*, October 2007.

[12] A.E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P.H. Oden, D.A. Prener, J.C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao and M. Gschwind, Optimizing compiler for the cell processor, in: *PACT'05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, IEEE Computer Society, 2005, pp. 161–172.

[13] K. Fatahalian, T.J. Knight, M. Houston, M. Erez, D.R. Horn, L. Leem, J.Y. Park, M. Ren, A. Aiken, W.J. Dally and P. Hanrahan, Sequoia: Programming the memory hierarchy, in: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, New York, USA, 2006.

[14] M.P.I. Forum, MPI-2: Extensions to the message-passing interface, 1997, available at: http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[15] A. Gholoum, E. Sprangle, J. Fang, G. Wu and X. Zhou, Ct: A flexible parallel programming model for tera-scale architectures, Technical report, Intel Whitepaper, 2007.

[16] R.L. Graham, G.M. Shipman, B.W. Barrett, R.H. Castain, G. Bosilca and A. Lumsdaine, Open MPI: A high-performance, heterogeneous MPI, in: *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.

[17] T.K. Group, Open computing language, 2008, available at: www.khronos.org.

[18] P. Jetley, F. Gioachin, C. Mendes, L.V. Kale and T.R. Quinn, Massively parallel cosmological simulations with ChaNGa, in: *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, Long Beach, CA, USA, April 2008, pp. 1–12.

[19] R. Jyothi, O.S. Lawlor and L.V. Kale, Debugging support for Charm++, in: *PADTAD Workshop for IPDPS 2004*, Santa Fe, NM, USA, IEEE Press, 2004, p. 294.

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

[21] L.V. Kale, G. Zheng, C.W. Lee and S. Kumar, Scaling applications to massively parallel machines using projections performance analysis tool, in: *Future Generation Computer Systems*, Special Issue on: Large-Scale System Performance Modeling and Analysis, Vol. 22, Elsevier, Amsterdam, 2006, pp. 347–358.

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

[23] A. Lastovetsky and R. Reddy, Heterompi: Towards a message-passing library for heterogeneous networks of computers, *Journal of Parallel and Distributed Computing* **66**(2) (2006), 197–220.

[24] M.D. McCool, Data-parallel programming on the cell be and the gpu using the rapidmind development platform, in: *GSPx Multicore Applications Conference*, Santa Clara, CA, USA, 2006.

[25] National Center for Supercomputing Applications, Lincoln Cluster, available at: http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/.

[26] NVidia, *NVidia CUDA Programming Guide*, Version 2.2.1, May 2009, available at: http://developer.download.nvidia.com/.

[27] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu and T. Nakatani, MPI microtask for programming the cell broadband engine™ processor, *IBM Systems Journal* **45**(1) (2006), 85–102.

[28] J. Planas, R.M. Badia, E. Ayguad and J. Labarta, Hierarchical tasked-based programming with StarSs, *Int. J. High Perform. Comput. Appl.* **23** (2009), 284–299.

[29] V. Saletore, J. Jacob and M. Padala, Parallel computations on the charm heterogeneous workstation cluster, in: *Proc. Third Int'l Symposium on High Performance Distributed Computing*, San Francisco, CA, USA, August 1994, pp. 203–210.

[30] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan and P. Hanrahan, Larrabee: A many-core $\times 86$ architecture for visual computing, *ACM Transactions on Graphics* **27**(3) (2008), 1–15.

[31] L. Wesolowski, An application programming interface for general purpose graphics processing units in an asynchronous runtime system, Master's thesis, Department of Computer Science, University of Illinois, 2008, available at: http://charm.cs.uiuc.edu/papers/LukaszMSThesis08.shtml.