Model for Programming Heterogeneous Systems

David M. Kunzman
University of Illinois
Urbana, IL 61801
Email: kunzman2@illinois.edu

Laxmikant V. Kalé
University of Illinois
Urbana, IL 61801
Email: kale@cs.uiuc.edu

I. Introduction

Many recent systems being used for high performance computing (HPC) are heterogeneous. The heterogeneity occurs on multiple levels ranging from the chip level (e.g. the Cell processor and AMD Fusion) to the workstation or node level (e.g. commodity cores supplemented with graphical processing units (GPUs) or Larrabee) to entire clusters (e.g. Roadrunner cluster at Los Alamos National Lab (LANL) and the Lincoln cluster at NCSA). While these systems have great potential for performance, it is usually at the cost of programmer productivity.

Programming these systems is often considered difficult as the programmer is required to have intimate knowledge of the hardware and often has to mix architecture specific code with application specific code to achieve good performance. In this work, we put forward a general scheme for programming these heterogeneous systems. While our implementation is in the context of the Charm++ programming model, we believe the ideas are generally applicable to many programming languages. In particular, we extend the Charm++ programming model to include *accelerated entry methods* and *accelerated blocks*. We further include a SIMD instruction abstraction allowing C/C++ programs to utilize SIMD instructions supported by various hardware architectures, such as SSE or AltiVec, without having to write architecture specific code. By using these abstractions, Charm++ programs are able to efficiently utilize the hardware resources while maintaining portability between architectures. The work presented here mainly focuses on using these extensions to program the Cell processor, though future work will include other types of architectures such as Larrabee and GPUs. Other programming models that support the Cell or have been developed for the Cell processor include CellSs [2], Sequoia [7], the RapidMind Development Platform [12], the Multicore Framework [5], and MPI microtasks [13].

II. Background

The Cell processor[9] presents a fairly different hardware architecture than standard commodity cores of the past. There are two types of cores: the Power Processing Element (PPE) and the Synergistic Processing Element (SPE). PPEs, which we treat as host cores, are similar to cores in commodity processors. In particular, the pipeline interacts with memory through a cache hierarchy. SPEs, which we treat as *accelerator* cores, are fairly different. The pipelines are simpler (in-order, dual-issue, etc.) and load/store instructions do not access system memory. Instead, load/stores issued by the SPEs access a local store memory that contains all code and data immediately accessible to the SPE. Direct Memory Accesses (DMAs) explicitly written by the programmer are required to move data between the local store and system memory. These factors contribute to the common perception that the Cell is *hard to program*. The programmer has to explicitly control many asynchronous activities (tasks on the PPE, tasks on the SPEs, managing the SPE's local store memory, and the DMAs). This additional code distracts the programmer from focusing on the application specific code. Furthermore, this additional code is not required on and not portable to other architectures.

The Charm++ programming model is an object-based message passing parallel programming paradigm. Charm++ has been in use for over a decade and has been used in the development of multiple production scientific simulations, including NAMD [3], OpenAtom [4], and ChaNGa [8]. Furthermore, Charm++ applications account for a significant number of cycles in supercomputing centers. The Charm++ runtime and programming model are discussed in detail elsewhere: [10] and the Charm++ tutorial.[1]

Charm++ programs are broken down into a collection of objects called *chare objects* or simply *chares*. Chare objects contain private data (member variables) and have entry methods (member functions that can be invoked asynchronously by other chare objects). Several chare objects are located on each physical processor. As entry methods are invoked, the Charm++ runtime system schedules the execution of the invoked entry methods on the physical processors.

III. Extensions to Charm++

Accelerated Entry Methods. In this work, we introduce *accelerated entry methods* into Charm++. Accelerated entry methods are entry methods that *can be* executed on an accelerator if an accelerator is present. In the absence of an accelerator, accelerated entry methods are executed on the host core. In an x86-based system, the x86s cores are the host cores. In a Cell-based system, the PPEs are the host cores and the SPEs are accelerator cores.

Figure 1 illustrates the general structure of a *standard* entry method in Charm++. Figure 2 illustrates the general structure of an accelerated entry method. Accelerated entry methods and standard entry methods are invoked the same manner. However, declarations for accelerated entry methods are different in a few ways. First, the *accel* keyword is used in the *options* section of the declaration to indicate that the entry method is an accelerated entry method. Second, in addition to the passed parameters, local parameters are declared. Local parameters indicate which data, local to the object that the entry method is invoked on, will by accessed by the accelerated entry method. For example, if the accelerated entry method reads or writes data contained in or pointed to by a member variable of the object, that member variable is listed. This allows the runtime system to *understand* what data will be accessed and how it will be accessed, allowing the runtime system to move the data as needed. Third, the function body of the entry method is included in the entry method declaration itself. This allows the build process to manipulate the code and is a result of the Charm++ build process (not a limitation of the general approach). Finally, a callback function is listed. The callback function is a member function that will be called on the same object when the accelerated entry method has finished computing.

Just as standard entry methods get scheduled on host cores, accelerated entry methods are scheduled on the accelerator cores by

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

```
Declaration:
  entry [options] void myEntryMethod
    ( passed_params );
Invocation:
  myObject.myEntryMethod(passed_params);
```

Fig. 1. Structure of a standard entry method in Charm++.

```
Declaration:
  entry [accel] void myAccelEntryMethod
    ( passed_params ) [ local_params ]
    { function_body } callback_function;
Invocation:
  myObject.myAccelEntryMethod(passed_params);
```

Fig. 2. Structure of an accelerated entry method in Charm++.

the runtime system. First, the application code invokes an accelerated entry method on the target chare object, at which point, the runtime system takes over. First, a message containing the passed parameters is sent to the processor which actual contains the target chare object. Before the accelerated entry method is executed, all data referenced in the local and passed parameters is migrated to SPE via DMA transactions. Once the data is present, the accelerated entry method is scheduled for execution and eventually executes. The application code is executed, generating output data which is pushed back to the host processor by the runtime system. Once the data is back on the host, the application code is notified via the specified callback function which can then invoke other entry methods. All data movement, entry method scheduling, and overlapping of computation and communication is handled by the runtime system automatically, allowing the programmer to focus on the application specific code (not architecture specific code).

The details of how the accelerated entry methods are scheduled on the accelerators is platform dependent. In this work, we focus on utilizing Cell processors. On Cell-based platforms, the Charm++ runtime system uses the Offload API[11], creating a work request for each accelerated entry method. The passed and local parameters are passed in to and out of the local store via the buffers associated with the work request (with all the passed parameters defaulting to read-only). The details of how work requests are scheduled, how data transfers to/from the local stores are overlapped with computation on the SPEs, how buffers are specified, and so on, are discussed in [11].

SIMD Abstraction. Accelerated entry methods are designed to be used for the compute intensive portions of the application code. As such, the function body for an accelerated entry method should issue many floating point operations. On many architectures, this involves using SIMD instructions (e.g. SSE on Intel cores). However, for the code to be portable, the programmer must write SIMD code specific to each architecture that the application may be compiled for and then conditionally include the correct version at compile time. This is quite burdensome to the programmer. To ease this burden, we have developed a SIMD instruction abstraction that the programmer can use to write application code. At compile time, the operations and data structures in this abstraction are mapped to the hardware specific implementations of the SIMD operations supported by the specific architecture. For example, an accelerated entry method, when compiled on a Cell platform, will make use of the SIMD instructions available to the SPEs. The same accelerated entry method, with no application code changes, when compiled on an x86 platform will make use of SSE instructions. In some cases, the compiler for the platform in question may be able to do this automatically. However, in many cases, programmers wish to do this by hand for performance reasons. Using this abstraction will make this programmer optimized SIMD code portable while still allowing the compiler to apply any optimizations for the target architecture it would otherwise apply. When using the SIMD abstraction, entry methods, accelerated or not, are portable between hardware architectures even if they are optimized to utilize SIMD instructions.

Sharing Accelerators. Because the accelerated entry methods are basically small *tasks* that are scheduled on the SPEs, multiple pieces of independently written code can share the SPEs. The runtime scheduler is free to interleave accelerated entry methods from various chare objects with one another. SPEs are not preallocated to any particular kind of computation (e.g. 3 SPEs for audio and 5 SPEs for video). Instead, all SPEs are available to any accelerated entry method. Furthermore, multiple object files with accelerated entry methods can be linked together and scheduled together, allowing completely independently written libraries to fully share the SPEs. When only one library is active, it has full use of all the SPEs. Once another library becomes active, it will automatically begin sharing the SPEs with the already running library. The runtime system makes this possible because of the structure and self-contained nature of accelerated entry methods.

Accelerated Blocks. *Accelerated blocks* have also been introduced into the Charm++ programming model. Accelerated blocks are blocks of code that are also available to the accelerated entry methods. For example, one can declare a function in an accelerated block and then call that function from multiple accelerated entry methods. Additionally, *#include*s can be used to include shared code, *#define*s, and macro declarations.

IV. Performance

To demonstrate the performance achieved by using the extensions described in section III, we have written a simple molecular dynamics (MD) simulation that uses accelerated entry methods.[2] This code is loosely based on the nonbonded force calculation in NAMD[3] and we use a problem size similar to the ApoA1 benchmark. In this simulation, 92160 charged particles are simulated using Coulomb's law for 128 timesteps (1 femtosecond per timestep). This list of particles is divided into equal parts referred to as *patch objects*. *Compute objects* calculate the forces between the charged particles (*pair computes* calculate forces between two patches while *self computes* calculate forces within a single patch). The force data calculated by the compute objects is passed back to the patch objects where it is combined and used to update the particles' positions and velocities. All three object types (patches, self computes, and pair computes) use accelerated entry methods to perform the physics calculations with the force calculations in the compute objects making up the great majority of the work. All calculations are done using single-precision floating point calculations.

When compared to a single x86 core using SSE instructions, a single chip on a QS20 achieves a speedup of 5.74. In both cases, the code is making use of SIMD instructions provided by the hardware architecture. To better understand the meaning of this speedup number, we take a closer look at the calculation itself. A total of approximately 16851 GFlops are performed during the simulation with the Cell processor averaging 50.1 GFlops/sec (24.5% of the peak of all SPEs combined). The inner-most loop of the force calculation is comprised of 54 instructions performing 124 Flops in

[2]The code for the example MD program is available in the Charm++ distribution (the nightly build).

56 cycles.[3] If the SPEs had infinite memory resources and could just continuously execute force calculations without having to issue any instructions related to DMA transactions, the force calculation as compiled and optimized by the SPE's compiler can reach, at most, $124/56 = 2.2 Flops/cycle$ (approximately 27.2% peak). This shows that the MD program is averaging approximately 88.4% of the peak floating point rate possibly attainable by the force calculation as compiled, demonstrating that our framework allows the program to achieve good performance.

To compare the single Cell chip configuration to other configurations, we made additional runs. When compared to a single x86 core, six x86 cores achieves a speedup of approximately 5.89, which is roughly equivalent to a single QS20 Cell processor. In this comparison, the Cell configuration uses 8 SPEs while the x86 configuration uses 6 x86 cores. However, the x86 cores are more complex cores designed to speed up single thread execution with out-of-order pipelines, 4-way instruction issue per clock, and so on (compared to the in-order pipelines, dual-issue, required DMAs, and so on of the SPEs). We also ran the MD program on a combination of four Playstation 3s and four QS20 Blades (one chip per blade), which have a combined sum of 56 SPEs. This configuration, which used a Gigabit Ethernet network, averaged 286.7 GFlops/sec (20.07% peak) for a speedup of 32.98 over a single x86 core using SSE. This is not quite as efficient as a single QS20 blade, however, this configuration is not load balancing the work. By default, the chares objects are spread evenly between all the processors, thus giving the Playstation 3 Cells (6 SPEs each) the same amount of work as the QS20 Cells (8 SPEs each). Typically, the Charm++ runtime system would dynamically migrate objects to balance the load, however, we are still working on adapting the load balancing framework to *understand* the notion of accelerators (SPEs in the Cell case). As a result, no load balancing was used for this configuration and the work was not balanced between the processors.

V. Heterogeneous Systems

A more interesting configuration than the configurations presented in section IV is a case where the nodes making up a cluster are of very different architecture types, which is the case for clusters like Roadrunner[1]. A Charm++ application that makes use of the extensions described in section III is already portable between x86-based and Cell-based platforms with no modification to application code. It naturally follows that the same code could run across nodes with different processor architectures with no modification to the application code itself.

At the application level, allowing programs to run on heterogeneous clusters requires the manipulation of application data. Through the use of Pack-UnPack (PUP) routines, the Charm++ runtime system can automatically relocate application data (passed parameters into entry methods, chare object data, and so on). PUP routines also implicitly describe the structure of the data in an indirect way (e.g. a single float or an array of integers). This additional information allows the runtime system to manipulate the application data to automatically correct for endianness, pointer sizes, etc. when passing data between processors with substantially different architectures. Please see the Charm++ Tutorial for a description of PUP routines. If an x86 processor invokes an entry method on an object located on a Cell processor, the passed parameters need to be corrected for endianness differences between the platforms. From the point of view of the programmer, an entry method is invoked on the calling side with all data in the local format. When the entry method is actually called on the receiving side, the data has already been transformed by the runtime system to the local format of the target processor and is ready to be directly used by the invoked entry method. We can already demonstrate simple programs running on an arbitrary combination of x86 and Cell processors, however, there is still much work to be done on the runtime system itself. Therefore, we leave this type of heterogeneous configuration as future work.

[3]Assuming perfect branch prediction and measured using the SPE timing tool included in the Cell SDK.

VI. Conclusions

We have described extensions to the Charm++ programming model that support programming for heterogeneous systems, including those with accelerators. In particular, the work focuses on the Cell processor, a heterogeneous chip, with multiple core types, PPEs and SPEs. We have shown that these extensions are portable between various chip architectures, such x86-based and Cell-based platforms, with no modification to application code. Calculations using this framework are able to efficiently use the underlying hardware as demonstrated in section IV via the example molecular dynamics code.

References

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

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

[3] 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.

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

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

[6] A. Eichenberger, K. O'Brien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the cell processor. pages 161–172, Sept. 2005.

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

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

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

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

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

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

[13] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband enginetm processor. *IBM Syst. J.*, 45(1):85–102, 2006.