Charm++, Offload API, and the Cell Processor

David Kunzman, Gengbin Zheng, Eric Bohm, Laxmikant V. Kale
Parallel Programming Lab
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave.
Urbana, IL 61801
kunzman2,gzheng,ebohm,kale@uiuc.edu

ABSTRACT

As multicore processor designs become more mainstream, it becomes more important to both expose and efficiently exploit parallelism in a clear and straight forward manner. We believe that the Charm++ paradigm is a good fit for the Cell processor (a heterogeneous multicore processor). Many aspects of the Charm++ runtime system allow it to take advantage of the benefits provided by the Cell processor in a clear and natural manner. These aspects include being able to peek ahead in the message queue, effective prefetching of data, virtualization, encapsulation of data, etc. In adapting the Charm++ runtime system to be able to use the Cell processor, we have developed the Offload API. The Offload API is a general purpose API, which can be used independently of Charm++, that allows a program to easily take advantage of the Cell's potential. Additionally, Charm++ applications are portable between many existing platforms in common use today. By adapting the Charm++ runtime system to utilize the Cell processor without modification to the user's application code, Cell programs written using Charm++ will automatically be portable to Cell-based platforms. Finally, we will discuss some initial efforts in porting the popular molecular dynamics program NAMD to the Cell processor.

1. INTRODUCTION

The Cell processor [1] jointly developed by IBM, Sony, and Toshiba deviates from standard processor design. While it has more computational power than most processors, it also presents many challenges. One of the challenges that it presents is how hard it can be to program in an efficient manner. Efficiently programming for the Cell processor requires the programmer to explicitly manage the resources available to each of the SPEs (see *The Cell Processor* section below). This, of course, also greatly hinders the portability of code when moving between a Cell-based platform and a non-Cell-based platform.

Charm++ [2] can provide portability to the Cell. Charm++ applications can already be ported across platforms comprised of many different combinations of processors and interconnects by simply recompiling the code without modification. In practice, Charm++ applications mainly fall within the realm of High Performance Computing (HPC). We are currently in the process of applying some of the abstractions and techniques we have developed to Cell-based platforms. In the course of this process, we have developed the *Offload API*. The Offload API is independent of Charm++ and allows C/C++ based programs to more easily utilize the SPEs on a Cell.

We believe that the Charm++ paradigm can also help solve many of these programability issues. Charm++ is a message driven paradigm. As messages arrive on a processor they are queued. The Charm++ runtime system can peek ahead in this queue telling the runtime what code will be executed along with what data will be needed in the future. Through virtualization, a technique already widely used in Charm++, the SPE communication overhead can be overlapped with useful computation effectively hiding overhead. On other platforms, Charm++ automatically handles many other aspects of a parallel program, such as measurement-based dynamic load-balancing. These techniques can be applied to the Cell, both between multiple Cell processors and between the SPEs on a single Cell processor. See the *Charm++* section below for more details.

2. THE CELL PROCESSOR

The Cell processor has been jointly developed by IBM, Sony, and Toshiba. Commonly referred to simply as *Cell*, it presents a departure from mainstream processor design. The processor has nine cores. There is a *main* core called the Power Processor Element (PPE). This core can be thought of as a standard 2-way SMT processor. The other eight cores, called Synergistic Processor Elements (SPEs), are specialized cores. The SPEs do not have direct access to main memory. Instead, each SPE has a private memory called a local store (LS). The local store is 256KB in size and contains all data needed by the SPE during execution (including code, heap, and stack). DMA transactions are explicitly issued by the program to move data between the local stores and main memory. These DMA transactions are controlled by the hardware. That is to say, the PPE and SPEs queue the DMA transactions and then continue processing while the hardware asynchronously takes care of moving the data.

Information on the Cell processor and the Cell Broadband Engine Architecture (CBEA) in general, is publicly available at http://www-128.ibm.com/developerworks/power/cell/.

3. OFFLOAD API

The Offload API provides a simple interface that a user can use to *offload* chunks of computation onto the SPEs making it easier to program for the Cell processor. Each chunk of computation is referred to as a *work request*. The idea is to have each work request be a separate, self-contained

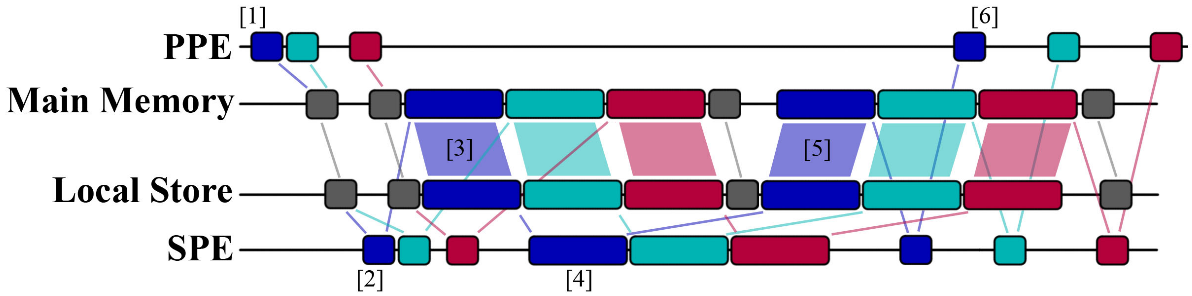

Figure 1: Timeline of SPE Runtime (not to scale): [1] The PPE makes a work request and the Offload API decides which SPE should execute the work request. [2] The SPE receives the work request through its work request list and initiates a DMA-Get to retrieve the needed input data. [3] The DMA controller asynchronously brings the input data from main memory into the local store. [4] Once the input data has arrived, the SPE executes the work request. Once the work request has completed execution, a DMA-Put is issued to move the results back to main memory. [5] The DMA controller asynchronously moves the results to main memory. [6] Once the data has been placed into main memory, the SPE notifies the PPE that the work request has completed. Please note that this image is not to scale; ideally, the execution time for a work request ([4]) should be much longer than the time to transfer the data ([3] + [5]).

unit of work (i.e. - no direct data dependencies between concurrently executing work requests). The work request is then passed into the Offload API. The Offload API is responsible for: moving the data buffer(s) associated with the work request to one of the SPEs, executing the code associated with the work request, moving the result(s) back to main memory, and then notifying the PPE that the work request has completed. Between the time the PPE code makes the work request and the SPE notifies the PPE that the work request has finished, the PPE is free to do other work (including submit more work requests to the Offload API).

As a simple example to help illustrate the idea, consider adding two matrices, $A + B = C$, where all the matrices have the dimensions M by N. Assuming the matrices are stored in row major format, the data for each row is stored in consecutive memory. The addition of each set of rows can form a single work request. That is, there will be M work requests each performing the computation $A(i,*)+B(i,*) = C(i,*)$ where $0 \leq i < M$, i has a unique value for each work request, and $X(i,*)$ denotes the ith row of matrix X.

Section 8 contains an example "Hello World" program (used for brevity) that demonstrates the use of the Offload API.

3.1 Making a Work Request

There are two components needed to make a work request: what code to execute and what data to use. Currently, the Offload API does not support migration of SPE code between the local stores and main memory. All SPE code must be statically compiled into a SPE binary that placed onto the SPE when the Offload API initializes. Each *work request type* is given a unique number. When the PPE makes a work request, it specifies the work request type which basically identifies what user defined code on the SPE will be executed. At the time the work request is made, numerous data buffers can be specified. Each buffer can be identified as either a *read/write* buffer, a *read-only* buffer, or a *write-only* buffer. Read/write buffers are moved to the local store before the work request is executed and moved back to main memory after the work request has finished. Read-only buffers are moved to the local store before the work request is executed and then their contents are discarded by the SPE after the work request has finished. For write-only buffers, a buffer of equal size is set aside in the SPE but no data is transferred before the work request executes. The work request can produce results to the write-only buffer. Once the work request has finished executing, the contents of the write-only buffer are transferred to its counterpart in main memory. The read-only and write-only buffer types are meant to reduce pressure on the enhanced interconnect bus (EIB).

3.2 SPE Runtime

Once a work request has been made by the PPE and the Offload API has assigned it to a particular SPE, all of the coordination for the work request is taken care of by the SPE owning the work request. Each SPE has a *SPE Runtime* executing on it to handle the coordination of work requests assigned to it along with moving data buffers in and out of the SPE's local store. Figure 1 illustrates how the SPE Runtime works. Each work request has an associated state while it is being processed by the SPE Runtime. There are no ordering restrictions between work requests. That is to say, the order in which work requests are submitted may not match the order in which they are completed. It is up to the code calling the Offload API to ensure that work requests are acting on independent sets of data. A work request is considered *completed* when it's results have been stored to main memory. For example, if there are two work requests submitted at almost the same time, one that needs quite a bit of input and one that does not need any input, it is likely that the work request with no input will execute first regardless of the actual submission order.

3.3 Notification of Work Request Completion

The PPE code is notified using one of two schemes as chosen by the user. The first is by callback. When the Offload API is initialized, a callback function can be specified. This callback function will be called each time a work request has completed. If no callback function is specified at initialization, then the user's PPE code must explicitly check if a work request has finished (the second scheme). Each time a work request is made, a *work request handle* is returned. The user's PPE code may poll the Offload API to check if the work request associated with that handle has completed. Alternatively, the user's code can also block until a work request associated with the handle has finished.

Once a work request has been completed the PPE is notified by the SPE. This does require the PPE to do some work as it must check for notifications from the SPEs. The Offload API provides a *progress* call that takes care of this. Periodically, the PPE code needs to *make progress on the Offload API* to clear finished work requests and possibly send more work requests to the SPEs (that were not be sent immediately because all SPE work request lists were full at the time). Additionally, some Offload API function calls also *make progress on the Offload API* when they are called.

3.4 Current Status and Availability

As stated, the Offload API has been developed to be used by the Charm++ runtime system. However, it does not rely on Charm++ to function and can be used as a standalone API in C/C++ applications. The Offload API is included in the Charm++ distribution available for download at the Charm++ website (http://charm.cs.uiuc.edu). While it is available for use, it is still under development with new features being added.

4. CHARM++

Charm++ is an asynchronous message passing paradigm. The program is broken up into objects called *chares*. Each chare, individually, does a portion of the overall computation. They pass messages between one another to coordinate and perform the entire computation. Each chare has one or more *entry methods*. Basically, entry methods are member functions that act as receiving points for messages. When one chare sends a message to another chare, it specifies both the message and the entry method that will receive the message (as if it were just doing a normal member function call on the receiving chare object). The chares themselves are spread out over all of the processors during the execution of a Charm++ program. Typically, there are many chares per processor. Each processor has a Charm++ runtime system that controls the execution, load-balancing, sending and receiving messages, etc. for all the chares located on that processor.

We believe that there are several aspects of the Charm++ paradigm that could be useful for Cell. These are covered in the subsections below. Additionally, Charm++ can provide portability to Cell-based platforms. The overall Charm++ applications can remain the same. When the program is *compiled*, our interpreter will automatically generate the code needed by the Offload API for those entry methods that are deemed *safe* to execute on the SPE. Here, *safe* would be defined as *self-contained* (as discussed above) and only requiring features of the Charm++ runtime that are supported by the SPE Runtime. This will provide portability as the application code will not need modification to move between Cell-based and non-Cell-based platforms, only recompilation.

4.1 Message Queuing, Encapsulation, and Virtualization

While the local stores are small and contain the only memory that the SPEs can directly access, they do provide two advantages. First, they are completely under the control of the programmer. Second, they provide a constant latency during load and store operations. The second advantage allows for better scheduling of instructions during code compilation. However, for the programmer to see these benefits, the programmer must be able to preemptively DMA the data into the local store before it is needed. This presents two challenges. The programmer must be able to predict what data will be needed before it is needed with enough time for a DMA transaction to complete (provided by the Charm++ model). Also, the programmer must explicitly manage the local store (the SPE Runtime handles this for the programmer).

The Charm++ model can assist with both of these challenges. First, remember that the overall computation performed by the program is broken up into chunks. These smaller chunks of computation are encapsulated within the chares. For the most part, the computation within a single chare only accesses the data contained in the message that was just received and the data contained within the chare itself. Because of this, the common case is that the data, and more importantly the location of the data, that will be used is known prior to its actual use. Each message has a known entry method that will *receive* it. This means that the code to be execute is also known prior to its execution. As the Charm++ runtime system receives messages for the chares, it queues the messages. The runtime system can *peek ahead* in the message queue and preemptively start DMAing the required data and code needed by an entry method before the entry method is executed. By the time the message reaches the head of the queue, that is, by the time the chare needs to do computation in reaction to receiving the message, the required data and code has already been moved into the SPE's local store. This is similar to our Out-of-Core work that has already been done in [4].

This process will clearly add some overhead. To help hide this overhead, Charm++ relies on virtualization. In this context, virtualization refers to each processor having multiple chares (sometimes referred to as virtual processes). While, one chare's entry method is being executed, the runtime system can be DMAing data for other chares into/out-of the SPE's local store. This effectively creates a double buffering effect (but typically with more than two buffers) at the granularity of entry methods. Concurrently, data buffers for pending entry methods are being DMAed into the local store, the SPE is executing another entry method, and results for previously executed entry methods are being DMAed back to main memory. This overlap of data movement with entry method execution will effectively hide the overhead needed to move data around within the Cell processor without requiring the programmer to explicitly write Cell specific code to handle DMAs, double buffering, etc.

However, it is important to make sure the PPE is providing enough work requests to the SPE's to make sure they continuously have work to do.

4.2 Load-Balancing

The Charm++ runtime system already has various load-balancers and also allows users to create their own load-balancers. The load-balancers can take several metrics into consideration including processor load, communication latencies, network topology, etc. In terms of the Cell, we plan to develop a load-balancer that takes the multilevel nature of the overall system into account. This is useful for multi-core platforms in general, not just Cell-based platforms. In the case of Cell, the interconnect between processors is one level and the EIBs on each Cell processor are another level. The communications costs of sending the same amount of data over each of these *interconnects* differs greatly. The load-balancing framework in the Charm++ runtime system can dynamically measure these effects and present a load-balancer with this information.

The load-balancing framework within Charm++ is modular and flexible. This will allow the same application to use a platform specific load-balancing technique geared towards the platform it is currently running on without modification to the application code. This allows portability between platforms without having to include platform specific code in the application or sacrificing performance.

4.3 Charm++ and the Offload API

The adaption of the Charm++ runtime system to use the Offload API has been broken down into three phases. The first phase is to simply get the already existing Charm++ runtime executing on the PPE. The second phase is to create the Offload API and provide a mechanism in Charm++ that will allow the user's code to utilize the Offload API. The third phase is to modify the Charm++ interpreter (charmxi) to automatically generate the code needed by the Offload API. The overall goal of the third phase is to allow Charm++ code to be able to execute on any processor, including the Cell, with no modification. Charm++ code can already be execute efficiently on a wide variety of platforms and interconnects with no modification to the user's code. The third phase will add Cell-based platforms to this list.

The first phase has been completed. Currently, the second phase is well underway and has already produced usable results. Example Charm++ programs using the Offload API already exist and are publicly available with the Charm++ distribution (http://charm.cs.uiuc.edu). We are continuing to add features to the Offload API. We plan to start work on the third phase in the near future.

4.4 NAMD on Cell

Work has already begun on modifying NAMD [3] to use the Offload API. NAMD is a popular molecular dynamics code written using Charm++. It is written and maintained by both the Theoretical and Computational Biophysics Group at UIUC's Beckman Institute and the Parallel Programming Lab. It is used by biophysicists throughout the world with over an estimated 10,000 unique downloads. It runs on many platforms ranging from small clusters using commodity hardware to larger systems including BlueGene, XT3, Altix, etc. Currently, using the Offload API along with the initial Charm++ adaptions, portions of the non-bonded electrostatic computations are already being computed by the SPEs. For each non-bonded electrostatic computation sent to an SPE as a work request, there are two lists of atoms as input. The basic idea is that each atom in the first list interacts with every atom in the second list (i.e. - if one list has N atoms and the other list has M atoms, there are $N * M$ total interactions). The SPE calculates the resulting forces and passes an array of force vectors back to the PPE. In the future, we plan to move more work to the SPEs (bonded force calculations, etc). For more information regarding NAMD, please visit it's homepage at http://www.ks.uiuc.edu/Research/namd.

5. CONCLUSION

The Offload API is a useful tool for helping programmers to easily utilize the SPEs on a Cell processor. Additionally, current modifications along with our planned modifications will allow Charm++ applications to easily take advantage of the SPEs on a Cell platform while maintaining portability to other non-Cell-based platforms. We have shown how the Charm++ paradigm fits well with the Cell processor. Example Charm++ programs already exist and are freely available. We are currently porting NAMD to the Cell using the Offload API.

All development to date has been done using the Cell simulator provided by IBM. We hope to have actual performance measurements in the near future of the Offload API, Charm++, and NAMD to present once we gain access to Cell hardware. One possible drawback of this approach is that the PPE may become a bottleneck as it must *feed* work requests to all of the SPEs. If the grainularity of the work requests being executed on the SPEs is small, the SPEs may finish the work requests that has been passed to them faster than the PPE can issue the work requests. While the simulator models the SPEs in great detail, the performance of our approach will rely on the performance of the PPE, system memory, and the EIB. Because of this, we are waiting for access to actual Cell hardware before presenting any performance results.

6. FUTURE WORK

6.1 Offload API

Due to the limited size of the local store when compared to the possible amount of data, we would like to migrate both the code and data associated with a work request instead of just the data. Currently, the code executed by a work request is statically linked with the SPE Runtime. We plan to allow the code to move to the SPE along with the work request needing it. This will allow for more SPE code in an application than is possible with just static linking.

We would also like to allow *affinity* to be specified for both code and data. By this we mean, if a particular data buffer or section of code is needed by multiple work requests both of those work requests should be passed to the same SPE. This will help reduce pressure on the EIB by transferring the data/code less often. Additionally, if multiple work requests will be accessing and/or modifying the same data buffers, the Offload API will need a mechanism that will allow the calling code to identify data dependencies between work requests. For example, consider the case where a work request, A, produces data to a buffer and then another work request, B, consumes that data. B must wait for A to complete before it can be executed, however, both A and B should be executed on the same SPE. The calling code should be able to clearly and easily indicate this (same SPE, execution order) to the Offload API.

6.2 Charm++

While using the Offload API is not difficult, at the moment, it still requires users to modify their Charm++ applications to use it. This hinders the portability benefits possible with the Charm++ model. With this in mind, we will be modifying the Charm++ interpreter to automatically generate the code needed by the Offload API while the Charm++ program is being compiled. This will allow Charm++ applications to be portable between Cell-based and non-Cell-based platforms with little to no modification.

7. REFERENCES

[1] 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.

[2] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[3] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[4] M. Potnuru. Automatic out-of-core exceution support for charm++. Master's thesis, University of Illinois at Urbana-Champaign, 2003.

8. OFFLOAD API CODE EXAMPLE

```
///// hello_shared.h (PPE + SPE) ////////////////
#ifndef __HELLO_SHARED_H__
#define __HELLO_SHARED_H__
#define FUNC_SAYHI    1
#endif //__HELLO_SHARED_H__

///// hello.cpp (PPE Only) /////////////////////
#include <stdio.h>
#include <string.h>
#include <spert_ppu.h>    // Offload API Header
#include "hello_shared.h"
#define NUM_WORK_REQUESTS  10

int main(int argc, char* argv[]) {

  WRHandle wrHandle[NUM_WORK_REQUESTS];
  char msg[] __attribute__((aligned(128)))
    = { "Hello" };
  int msgLen = ROUNDUP_16(strlen(msg));

  InitOffloadAPI();

  // Send some work requests
  for (int i = 0; i < NUM_WORK_REQUESTS; i++)
    wrHandle[i] = sendWorkRequest(FUNC_SAYHI,
                                  NULL, 0,
                                  msg, msgLen,
                                  NULL, 0
                                 );

  // Wait for the work requets to finish
  for (int i = 0; i < NUM_WORK_REQUESTS; i++)
    waitForWRHandle(wrHandle[i]);

  CloseOffloadAPI();
  return EXIT_SUCCESS;
}

///// hello_spe.cpp (SPE Only) //////////////////
#include <stdio.h>
#include "spert.h"        // SPE Runtime Header
#include "hello_shared.h"

inline void sayHi(char* msg) {
  printf("\"%s\" from SPE %d...\n",
         msg, (int)getSPEID());
}

#ifdef __cplusplus
extern "C"
#endif
void funcLookup(int funcIndex,
    void* readWritePtr, int readWriteLen,
    void* readOnlyPtr, int readOnlyLen,
    void* writeOnlyPtr, int writeOnlyLen,
    DMAListEntry* dmaList) {

  switch (funcIndex) {

    case SPE_FUNC_INDEX_INIT:  break;
    case SPE_FUNC_INDEX_CLOSE: break;
    case FUNC_SAYHI:
      sayHi((char*)readOnlyPtr);
      break;
    default:
      printf("ERROR :: Invalid funcIndex (%d)\n",
             funcIndex);
      break;
  }
}

///// Output ///////////////////////////////////
"Hello" from SPE 0...
"Hello" from SPE 7...
"Hello" from SPE 4...
"Hello" from SPE 5...
"Hello" from SPE 6...
"Hello" from SPE 2...
"Hello" from SPE 3...
"Hello" from SPE 0...
"Hello" from SPE 1...
"Hello" from SPE 1...
```