© 2007 Chao Mei

A PRELIMINARY INVESTIGATION OF EMULATING APPLICATIONS THAT USE PETABYTES OF MEMORY ON PETASCALE MACHINES

BY

CHAO MEI

B.S., Fudan University, 2004

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Adviser:

Professor Laxmikant V. Kale

Abstract

With the advent of petascale computing era, a typical petascale machine is usually installed with hundreds of thousands of processors. To predict the performance of existing applications running on such a machine, a simulation-based approach faces various unprecedented challenges. One of them is the memory constraint posed by emulating the petascale application that uses petabytes of memory on an existing parallel machine that typically only has hundreds of terabytes of memory or less. With an attempt to address this challenge, we present a preliminary investigation of utilizing the out-of-core concept in the context of an existing performance prediction framework BigSim. We have designed three basic out-of-core schemes and explored the optimizations that could be applied to the basic schemes. The additional theoretical analysis models both the performance of the emulation with and without out-of-core support, points out the very importance of the optimizations for the basic out-of-core schemes and offers suggestion to its implementation. By resolving multiple issues across the software stack, we implemented a prototype of the basic out-of-core support and integrated it into the emulation component of BigSim. The experimental evaluation shows the prototype works correctly for both Charm++ and MPI applications, and a reasonable emulation performance regarding a MPI kernel benchmark.

To my parents.

Acknowledgements

There are many people whom I need to thank for their help and support with this thesis. I apologize if I have missed out some names.

I would like to thank my advisor Prof. Laxmikant V. Kalé for his guidance, advice, motivation and continued support throughout my stay at University of Illinois at Urbana Champaign, without which this thesis would not have been possible.

Many thanks to Gengbin Zheng who helped me to set my foot in the BigSim framework, and for his insights on various problems I encountered during the system implementation.

I also want to thank several past and current PPL members Chao Huang, Sayantan, Fillipo, Eric and Celso for their help with various things.

Finally, thanks to my friends for their support and my parent for their confidence in me.

Table of Contents

List of Tables . . . . . **vii**

List of Figures . . . . . **viii**

Chapter 1 Introduction **1**

Chapter 2 Existing Software Infrastructure **4**
2.1 Converse 4
2.1.1 Scheduling in Converse 5
2.1.2 Threads in Converse 5
2.2 Charm++ and AMPI 6
2.3 BigSim 8

Chapter 3 Design and Analysis of the Out-of-core Approach **12**
3.1 Design Space of the Out-of-core Scheme 12
3.1.1 Granularity Choice 13
3.1.2 Options of Basic Schemes 14
3.1.3 Optimizations for Basic Schemes 17
3.2 Theoretical Analysis 20
3.2.1 Large Computation Chunk 21
3.2.2 Small Computation Chunk 22

Chapter 4 Implementation of the Out-of-core Approach **25**
4.1 Implementation Overview 25
4.2 Packing/Unpacking Framework 28
4.3 Implementation Issues 29
4.3.1 Charm++ Runtime Related Changes 29
4.3.2 AMPI Runtime Related Changes 31
4.3.3 Modifications to Load Balancing Framework 36

Chapter 5 Experimental Results **39**
5.1 Correctness Verification 40
5.1.1 Verification by Charm++ Programs 40
5.1.2 Verification by a MPI 3D-Jacobi Program 41
5.2 Performance Results 41

5.2.1 Results and Analysis of Normal Tests 42
5.2.2 Results and Analysis of Stress Tests 44

Chapter 6 Summary and Future Work 45

References 48

List of Tables

5.1 Performance comparison between emulation with and without out-of-core support 42

List of Figures

2.1 Virtualization in Charm++ 7
2.2 Implementation of AMPI virtual processes 8
2.3 Functional view of a target node 9
2.4 A simple work flow of the idea behind BigSim 10

3.1 The basic idea of all out-of-core schemes 15
3.2 Timeline in the case of large computation chunk 22
3.3 Timeline in the case of small computation chunk 23

4.1 The implementation skeleton of the first basic out-of-core scheme . . 26
4.2 The implementation skeleton of the last two basic out-of-core schemes 27
4.3 The issue of one data structure shared by multiple pointers 36
4.4 The issue of recording load information incorrectly for a chare object 37

Chapter 1

Introduction

At the door of petascale computing era, people have to face unprecedented challenges arising both for the construction of petascale machines and the application development. On one hand, the designers of the petascale machine are making architectural choices; they would like to make choices and trade-offs that will benefit the mix of applications that are likely to run on this machine. But how can they determine the efficacy of each choice? On the other hand, the application programmers have the daunting task of scaling existing parallel science or engineering code to the petascale machine typically installed with hundreds of thousands of processors. They also would like to have their application ready for the new machine when it is deployed. However, since such machine is still being designed or under construction, how can they effectively tune their application without the resource?

A simulation-based approach is envisioned as one approach to surmount these challenges. Its feasibility has been demonstrated with a proof-of-principle system called BigSim[1] where the IBM BlueGene machine is the study target. The system has two major components: one is the emulator providing an environment in which users can run a parallel application as if they are running on the target machine with its full set of processors; the other is the simulator, with input of information such as the traces generated by the emulator, produces detailed performance data for the application running on the simulated (target) machine.

The attempt of using BigSim for petascale machines brought additional issues into focus. One specific issue involves the scalability of emulation so that applica-

tions running on a very large number of processors can be emulated using a much smaller system. It is expected that petascale machines have petabytes of physical memory available which can be fully used by parallel applications. To emulate such memory intensive applications on much smaller parallel machines, the memory required for emulation is highly demanding because one emulating processor typically emulates tens to hundreds of target processors, implying that each emulating processor has to hold tens to hundreds of instances of application data in memory. It is difficult, if not impossible, to find an existing parallel machine with sufficient memory if one is simulating real applications. The virtual memory mechanism can satisfy the memory requirements of these applications to some extent. But this approach typically performs poorly with random data access patterns, such as those in BigSim due to switching between targeting processors.

Therefore, based on the out-of-core concept, we explore the schemes suitable for the emulation component of BigSim which will not require specific knowledge of the applications being emulated. We have designed three basic out-of-core schemes, and investigated the possibility of optimizations that could be applied on these schemes by taking advantage of the fact that the scheduler, responsible for processing message, has the knowledge of the messages that are going to be active in the near future. The theoretical analysis of the schemes and possible optimizations tells the very importance of the optimization in order to achieve good emulation performance in the case of out-of-core execution as well as the advice on its implementation. By implementing the prototype of basic out-of-core schemes and integrating it into the BigSim framework, we resolved multiple issues related with the Charm++/AMPI runtime system. Both Charm++ and MPI programs have been tested to validate the correctness of the new emulation component augmented with basic out-of-core support. And an experimental evaluation on a MPI 3D-Jacobi kernel benchmark shows a reasonable performance for our initial implementation.

The contributions of this thesis are:

- The design space is explored for the out-of-core technique in the context of the BigSim framework.
- A theoretical analysis is presented for the performance of out-of-core schemes and its effectiveness is demonstrated by the experimental results.
- The scalability of emulation is preliminarily addressed as a prototype to support out-of-core emulation has been integrated into the existing BigSim framework.

This thesis comprises six chapters. Chapter 2 provides an overview of the existing software infrastructures, including Converse, Charm++/AMPI and BigSim, on which this work is built. Design space exploration, together with theoretical analysis are presented in Chapter 3, followed by Chapter 4 focusing on the discussion of the implementation issues for the out-of-core approach. Afterwards, Chapter 5 shows the results of our experimental evaluation on the prototype for out-of-core support. Finally, we conclude the thesis and describe the future work in Chapter 6.

Chapter 2

Existing Software Infrastructure

The work of this thesis is built upon a sophisticated and substantial software infrastructure. The emulation of petascale applications leverages the existing version of BigSim[1, 2]–a framework to help predict performance for petascale machines and applications. As for the applications studied in this thesis, they are mainly MPI programs enabled by the Adaptive MPI (AMPI) system[3, 4].

BigSim depends in a critical way on the object-based virtualization capabilities provided by the Charm++[5] and Adaptive MPI (AMPI) systems. In addition, the emulation utilizes the thread library provided by the Converse runtime[6], and this library is indispensable for the execution of MPI programs with the AMPI system. Converse is also the underlying software layer on which Charm++ system is built. Therefore, a brief review of these systems is given to help readers familiarize the background of this work. According to the dependencies among these systems, we first begin with a description of Converse, followed by the introduction of Charm++ and AMPI systems. Finally, the current status and approach of BigSim are touched.

2.1 Converse

Converse is a multi-language, interoperable runtime framework which support different programming paradigms such as the message-driven programming style, thread-based programming style etc. It provides efficient and portable implementation of all the functions typically needed by a parallel language library. This runtime envi-

ronment has been running on most popular operating systems such as Windows and Linux as well as on different supercomputers. Both Charm++ and AMPI systems are built on top of this framework.

2.1.1 Scheduling in Converse

Each processor in the Converse runtime runs a scheduler which is responsible for receiving all messages, and repeatedly picking and processing messages. The Converse scheduler is based on the notion of schedulable entities, called *Generalized Messages*. A generalized message is an arbitrary block of memory with two types. The first type is related with message-driven execution. The first few bytes of the memory block specify a function pointer or an index into a table of functions that will handle the message and the rest contains the user data. The scheduler dispatches this type of generalized message by invoking the function and passing the user data as the function parameter in the form of a message pointer. Any function that is used for handling messages must first be registered with the Converse environment. The second type of the generalized message is related with thread scheduling. In this case, the message is a scheduler entity for a ready thread or a token object which contains a pointer to a ready thread. The latter entity is used when threads do migration or do checkpointing where the thread would be taken out of memory. Once this thread is brought back into memory again and ready to execute, the token object which is associated with this thread will update its pointer to the thread. In this way, the scheduler will correctly dispatch the thread indirectly through the token object.

2.1.2 Threads in Converse

Thread-related functions are worth noting as they are necessary for the implementation of AMPI system and the emulation of applications in BigSim. Converse provides

an architecture-independent interface to most thread functions, thread scheduling and synchronization of variables. For instance, CthYield() stands for the function of yielding a thread. As mentioned in the above section, threads in Converse use the same scheduling queue as the one used for Converse messages. Normally, when a thread is awakened, it goes into this primary ready-queue. However, to make the scheduling more flexible, the Converse system provides scheduling hooks for developers to make threads go into a different ready-queue, whether it is a FIFO or LIFO, or a queue with any complicated order desired. More details can be found in the Converse manual[7].

2.2 Charm++ and AMPI

Charm++[5] is a portable C++ based parallel programming language and runtime system. It is based on the idea of migratable objects and resultant virtualization of processors[8, 9]. In this approach, an application developer decomposes a problem into N components, a.k.a *virtual processes (VP)* which are implmented as C++ migratable objects or user-level threads. The N VPs will execute on P processors where N is independent of P though ideally $N >> P$. The programmer's view of the programs is of VPs and their interactions; the underlying runtime system keeps track of the mapping of VPs to processors and performs any remapping tat might be necessary at runtime. This idea is illustrated by figure 2.1. In Charm++, VPs are known as *chares*. Chares are C++ objects with special *entry* methods that are invoked asynchronously from other chares. A group of chares performing on a same parallel task can be organized into an indexed group called a *chare array*. A Charm++ program typically consists of one or more chare arrays.

Charm++ applies a *message-driven execution* model to determine which chare gets control on a processor. An advantage of this approach is that no chare can

hold a processor idle while it is waiting for a message. Since $N > P$, there may be other chares on the same processor that can overlap their computation with the communicating chare.

The object-oriented based programming style, together with the virtualization of underlying real physical processors also enable the transparent migration of chares (including both application code and data) and provides the property of location independent. In other words, chares can migrate from processor to processor freely. This migration of chares provides opportunities for the runtime sytem of dynamically shifting chares from overloaded processors to underloaded procesors, thus achieving better load balancing.

MPI is the de-facto standard of writing parallel applications. Therefore, Adaptive MPI or AMPI[3, 4] is designed and implemented, based on Charm++. AMPI implements virtual MPI processes, or VPs, using migratable user-level threads, several of which may be mapped to a single physical processor shown by figure 2.2. As a result, MPI now becomes a special case of AMPI when exactly one VP is mapped to a physical processor. AMPI inherited many advantages from Charm++ system such as adaptive overlapping of communication and computation, automatic load balancing by migrating user-level threads, flexibility of running on an arbitrary number of processors.

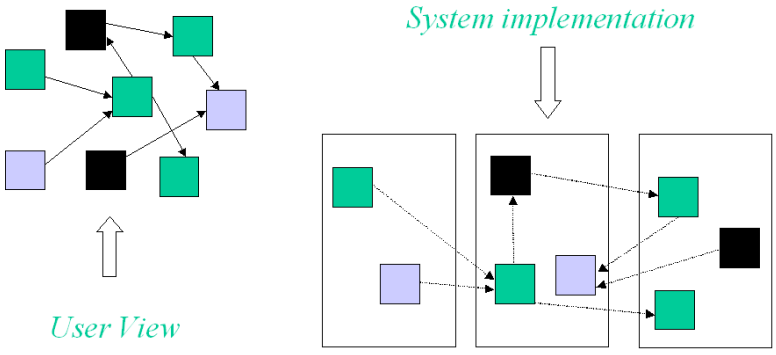

Figure 2.1: Virtualization in Charm++

One point is worth mentioning here: although the targets of our emulation system are mainly normal MPI applications, applications written in the Charm++ language are also included, such as NAMD[10] and LeanCP[11] etc.

In the forthcoming era of petascale computing when supercomputers will typically be installed with hundreds of thousands of processors, we believe that the message-driven execution model used in Charm++ and AMPI is suitable to exploit the full computation power provided by petascale machines. In addition, the virtualization of processors featured by both systems eases the parallel programming for petascale machines as developers only focus on the logical view of the problem.

2.3 BigSim

The BigSim project software has evolved into two components: a generic emulator of applications on petascale machines, and a simulator that allows for performance prediction and analysis of the emulated program in a variety of situations, including the presence of a detailed network contention model. In this work, the emulator is extended to support the emulation of applications that use petabytes of memory. Therefore, the emulator is the focus in the following introduction.

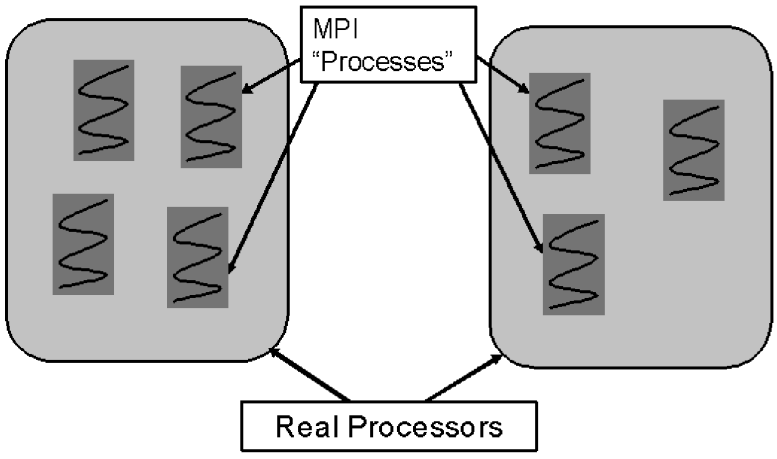

Figure 2.2: Implementation of AMPI virtual processes

The emulator provides an environment in which users can run a parallel application as if they are running on the target machine with its full set of processors. In order to support high degree of parallelism, a low level abstraction of petascale machines is designed to provide access to machines's capabilities. In the programmer's view, each node consists of a number of hardware-supported threads with common shared memory. Those threads continuously monitor the incoming buffer for arriving messages, which drives the progress of the application. The header of each message encodes a handler function, which is extracted and invoked by the threads on the destination node. This low level abstraction of the petascale architectures is general enough to embody various parallel machines with different numbers of processors and co-processors on each node.

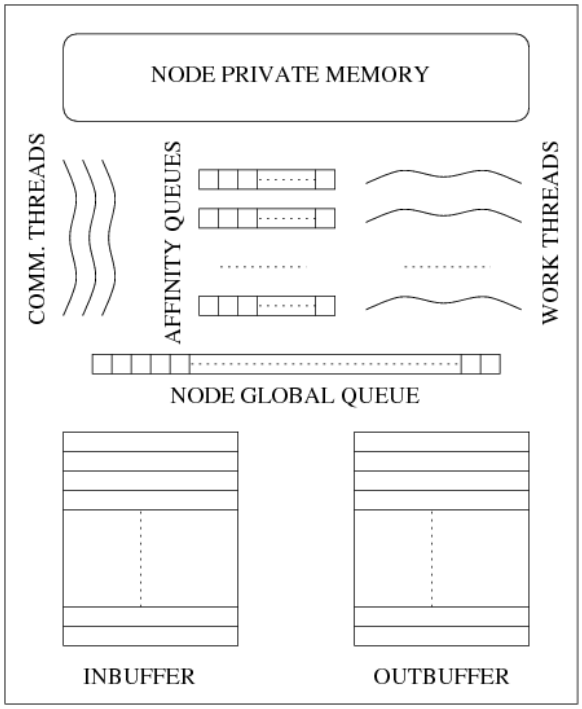

Figure 2.3: Functional view of a target node

Users of this emulator can easily configure it for specific target machines. Figure 2.3 [12] describes the functional view of an emulated node (*target node*). Within the node, processors or cores (*target procesors*) are viewed as a number of threads, which

are divided into communication threads and worker threads. Communication threads check incoming messages from the network. If the workload specified by the incoming message is lower than a threshold, the communication thread will process this message by itself. Otherwiese, it will put the messages in either a worker's queue or the node's global queue. Concurrently with communication threads, worker threads repeatedly retrieve messages from the queues and execute the handler functions associated with the messages. Application's code are trapped and executed inside the handler function. Noting that each worker thread and communication thread has its own message scheduler. The reasons for this choice are: firstly, it maintains a consistent view of target processors; secondly, it has the advantage of scalability because there is no central control point in the emulation.

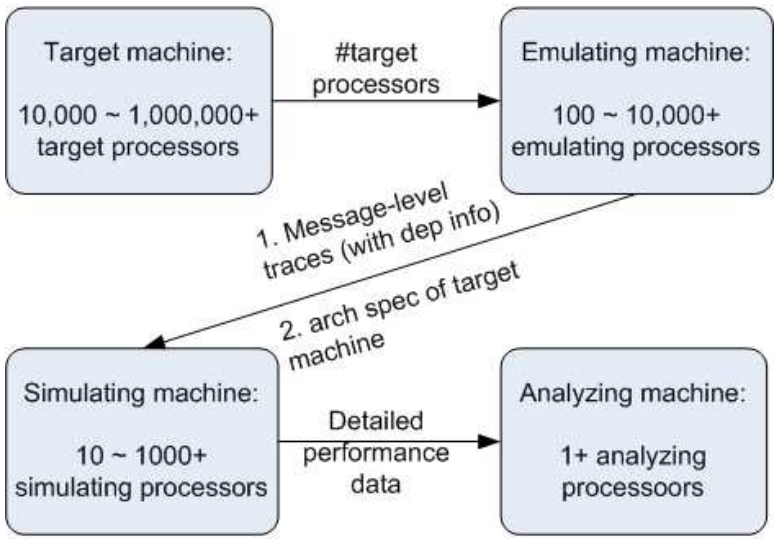

Figure 2.4: A simple work flow of the idea behind BigSim

With such emulation, application developers can already test for bugs, non-scalable data structures etc. The other purpose of emulation is to obtain the message-level execution traces. Along with message-dependence information, the traces are passed to the simulation component which also takes the architectural specification of the machine as its input, and then produces detailed performance data for the appli-

cation running on the simulated (target) machine. The whole process of performance prediction can be simply illustrated by the figure 2.4.

Chapter 3

Design and Analysis of the Out-of-core Approach

In this chapter, the ideas of enabling the emulation of petabytes of memory are discussed. The approach we adopt is to use the out-of-core technique which has been explored by Mani in the context of Charm++ system[13]. Out-of-core execution is a general technique to handle the scenario when there is not enough memory for the application. Since the out-of-core execution is controlled by the application developer who has better knowledge of the application's memory access pattern than the operating system does, the virtual memory (VM) system in the OS is not considered here as a performance-wise technique to solve the problem of emulation of petabytes of memory. Furthermore, the execution of the emulation is like the parallel discrete event simulation (PDES) whose memory access pattern is not as normal as to be predictive. The VM system in the OS cannot handle this case efficiently. Therefore, the out-of-core technique is chosen as our current approach to emulating applications that use petabytes of memory on petascale machines.

3.1 Design Space of the Out-of-core Scheme

Before diving into the exploration of design spaces for the out-of-core scheme, one assumption of the memory usage by the emulation has to be stated so that potential users of this work would have clear ideas of under what the circumstances the out-of-core technique can be turned on in the emulation. The emulation aims at the application running on the petascale machine which consists of around hundreds of

thousands of target processors. This ultra-scale machine will be emulated on thousands of processors available on current supercomputers. Therefore, each emulating processor is responsible for emulating hundreds of target processors. In other words, hundreds of instances of the application are running on each emulating processor. Assuming each process consumes a reasonable amount of memory, hundreds of them on a single processor pose a memory constraint. In a word, to emulate petabytes of memory, the scenario we assume to face is that *each application process consumes a reasonable amount of memory (usually a small fraction of the total memory available on the processor), but adding them up on a single emulating processor will exceed the memory available on that processor.* How to deal with the scenario when a single application process consumes more memory than physically available on the emulating processor is not the focus of this work.

3.1.1 Granularity Choice

The above assumption in turn suggests that the granularity of the preliminary out-of-core emulation could be in the level of swapping one target processor at a time. As for Charm++ programs, such granularity means either all the chares (including the chare array elements) on a target processor stay in memory (in-core) or all of them stay outside the memory (out-of-core). When it comes to AMPI programs, the granularity means either all the virtual processes (VPs) on a target processor are in-core or all of them are out-of-core. In most cases, we expect only one VP will be on a target processor when we are emulating a MPI program by using AMPI system because the performance of the MPI program to be emulated at this time is of no interest, but the multiplexing of multiple VPs is still supported on a target processor.

In addition,the current implementation of the emulation component of BigSim also affects the choice of the granularity of out-of-core execution in this preliminary work. As described in section 2.3 where we briefly introduce the BigSim system, each target

processor, i.e., each worker/communication thread runs its own message scheduler for the sake of scalability and a logically consistent view of target processors. To maintain these two advantages, the out-of-core execution engine is better to be put inside each worker thread's message scheduler (the communication thread is not considered here because it is not responsible for processing large work load which usually contains the allocation of large memory space). Inside the scheduler of each worker thread, it is simpler from the perspective of logic, not to mention from the perspective of implementation, to determine whether the thread as a single entity is in-core or not. This is harder with the small units such as chares because the emulation system does not know which small units are needed to be in-core when processing the next message. Such information can either be obtained from the application itself or from the Charm++ runtime on top which the application runs. It is complex to fetch such cross-layer information, and may even require the change of application source codes. Furthermore, it is far beyond the design purpose of emulation component of BigSim. As a result, the granularity of the out-of-core execution is better chosen in the level of swapping one target processor at a time, partly due to the current design and implementation of the emulation component of BigSim.

3.1.2 Options of Basic Schemes

Currently, we have designed three basic out-of-core schemes. They share the same very basic idea, illustrated by figure 3.1, in which before processing a message, the corresponding target processor must be in memory.

The following shows these three schemes,ordering from the simple one to the relatively complex one, and gradually becoming more flexible.

1. *Per message based scheme.* In this scheme, before a worker thread (i.e. a target processor) processes a message, we will check whether itself is in-core or not. If

it is out-of-core, then the worker thread will be brought into the memory. Then, immediately after processing the message, the worker thread will be taken out of the memory.

(**Note**: the above description sounds self-contradicting as if the worker thread is in execution mode, how could it not be in core? In the context, the "in-core" means that all the user-level data on this target processor are in memory. For Charm++ programs,the user-level data refers to the chares; for AMPI programs, this refers to the threads that represent MPI processes. If and only if the user-level data is in memory, the message could be correctly processed. The "out-of-core" has the opposite meaning of "in-core". Therefore, when we say bringing a worker thread or a target processor into memory, we mean bringing all the user-level data on this target processor into memory. Similarly, taking a worker thread out of memory refers to taking all the user-level data out of memory.)

This idea behind this scheme is simple and it is designed to speedup the implementation of adding out-of-core functionality into the emulation component of BigSim without considering the associated performance issues. Furthermore, its implementation will lay a solid foundation for the implementation of other out-of-core design schemes.

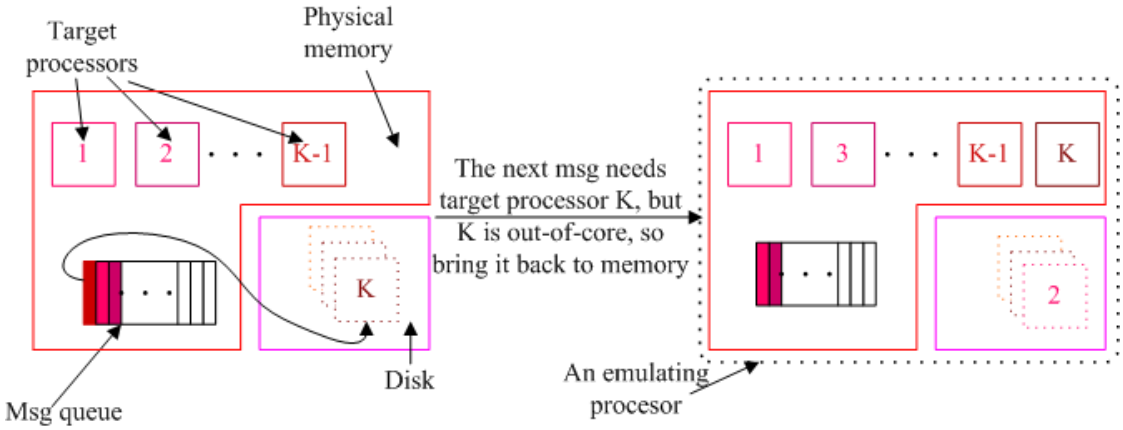

Figure 3.1: The basic idea of all out-of-core schemes

2. *Multiple target processors based scheme.* In this scheme, the emulation component allows multiple target processors or worker threads staying in-core. The number of target processors allowed in-core is a tunable parameter but fixed during the execution time. The first out-of-core scheme could be viewed as a specific case of this scheme where only one target processor is allowed in-core at a time. Before a worker thread processes a message, we will again check whether itself is in-core or not. If in-core, we do nothing and begin to process the message. Otherwise,we will first check if the number of worker threads allowed in memory has reached the limit. If yes, a worker thread that is currently in-core will be taken out of the memory based on an eviction policy. For example, if we use the least-recently-used (LRU) policy, the worker thread that has not processed a single message for the longest time will be evicted from the memory. Afterwards, the worker thread that is going to process the message is brought into memory and the message is processed. Advantages of this approach are that it is easy to implement and delivers reasonable performance.

3. *Memory based scheme.* In this scheme, the emulation component allows a dynamic number of worker threads to stay in-core based on the maximum amount of memory to be devoted to the emulation. To make it more flexible, the maximum amount of memory could be specified by the user, or be set to the amount of memory totally available on the emulating processor. This is a evolved version of the second scheme as it allows a dynamic rather than static number of target processors in-core. The working flow of this scheme is similar to the other two schemes. Firstly, we still check whether the worker thread is in-core before it processes a message. If not, we estimate the amount of memory required by this worker thread based on its execution history, and compare the estimated amount against the remaining unoccupied memory. If less, we bring this

worker thread into memory. Otherwise, we continue to evict currently in-core worker threads from memory until the left amount of empty memory satisfies the requirement of the this worker thread. The eviction of worker threads is also based on an eviction policy such as the LRU one mentioned in the second scheme. The advantage of this scheme is that it will potentially deliver the best performance as it maximizes the usage of available memory. The side-effect is the relatively complex implementation.

3.1.3 Optimizations for Basic Schemes

The optimizations discussed in this section mainly target the last two schemes.

- **Choosing the most appropriate eviction policy**. Both schemes share a common problem: which worker thread should be evicted from the memory when there are resource constraints, i.e., either the maximum number of target processors allowed in-core has been reached or the remaining empty memory is not enough for the worker thread that is going to be brought into memory. This refers to the eviction policy, choice of which first exposes opportunities for optimizing the basic schemes to achieve better performance. In the description of the design schemes for out-of-core emulation, we have cited one option of this policy as the LRU. The LRU may not be an appropriate policy in this emulation context after a second thought. The progress of the parallel application usually depends on the consumption of messages. For instance, the message received contains the data that is required for this worker thread to continue its computation. In the context of message-driven execution model as used by the Charm++ and AMPI system, this claim is more applicable. The least recently used worker thread is likely to be waiting for an arrival of a message to continue its execution while the most recently used worker thread may have just finished

one piece of work and could have to wait for the next message delivered by some other worker thread. According to this reasoning, the least recently used worker thread should not be evicted as its expected message should arrive very soon after the longest waiting time among all the in-core worker threads. In this sense, the most recently used worker thread should be evicted out of the memory instead.

A further tuning of the eviction policy takes advantage of the emulation runtime information owned by each emulating processor. Currently, each emulating processor only knows how many target processors (worker threads) are emulated (running) on itself, but does not track the order of the execution of these worker threads. The latter information is only known by the Converse scheduler which schedules both the messages and the threads. If we trap the "yielding", "suspending" and "awakening" points of these worker threads on each emulating processor before directing these three thread scheduling functionalities to the Converse scheduler, we can keep a record of the execution order of these worker threads on each emulating processor. Making use of this scheduling information, we have the ability of looking into the future to precisely know which worker thread is going to be invoked to process a message. Therefore, the eviction policy based on this lookahead scheme will be perfect since no cases will occur that the just-evicted worker threads are going to process a message in the very near future.

- **Overlapping the message processing with bringing the next worker thread into memory**. The speed of disk I/O lags far behind that of memory and CPU. Therefore, frequent disk I/O operations will severely under-utilize the computation power provided by the CPU, and become the performance bottleneck of the emulation. To mitigate the high latency of disk I/O, based on

the lookahead information described above, prefetching the contents of the next target processor could be applied so that the time consumed on the bringing the next worker thread into memory is at least partially overlapped with that of processing messages by the current worker thread. If overlapping perfectly, the performance achieved by this optimization will be the same with that as if all the target processors are in-core. There are two different schemes to perform the prefetch. The first one is just touching the virtual addresses of all the contents on a target processor but without filling in the actual data. In this approach, only the physical memory is pinned down. Differently from the first light-weight approach, the second one is heavy-weight in that not only the physical memory is pinned down but also the actual data is filled in. Similar schemes are also mentioned in Mani's work [13] for supporting out-of-core execution in Charm++ runtime system.

Both prefetch schemes will encounter the problem of choosing an appropriate eviction policy if the memory size requirement exceeds the remaining empty memory size. In this case, the eviction policy based on the lookahead scheme described previously is utilized. To implement the prefetch optimization, we can either use a separate thread to perform the job or use asynchronous I/O (AIO)[14] which becomes a standard support on Linux 2.6 kernel. The former is expected to be more complicated since all the functions related with finishing a disk I/O have to be explicitly managed by the emulation component. In contrast, in the AIO approach, the underlying OS has provided a good encapsulation and exposed a nice library interface to the software layer above it.

3.2 Theoretical Analysis

In this section, we present theoretical analysis to show how the straightforward out-of-core technique will affect the emulation performance. The analysis will also shed some light on to what extent the prefetching optimization will improve the performance and how it should be implemented to help the emulation achieve best performance.

Before starting the analysis, we assume the execution pattern of the out-of-core emulation is abstracted as *if using the out-of-core technique, the emulation repeatedly performs computation work intermixed with out-of-core operations; if not, the emulation's computation work is intermixed with the page fault service provided by virtual memory (VM)* where one "*out-of-core operation*" refers to *an operation, either taking a target processor out of memory or bringing a target processor back to memory.* In addition, we use the following symbols to facilitate the discussion.

- N: the number of computation chunks
- T_c: the average execution time of each computation chunk
- T_o: the average operation time of one out-of-core operation
- f_o: the frequency of the out-of-core operation, i.e., the number of out-of-core operations per computation chunk
- T_p: the average time of page fault service by the operating system.
- f_p: the frequency of the page fault service, i.e., the number of page fault service per one computation chunk
- T_{wo}: the total execution time of the emulation without out-of-core support
- T_w: the total execution time of the emulation with out-of-core support

- P_r: the performance ratio calculated by T_w/T_{wo}. $P_r > 1$ indicates the out-of-core emulation is slower than the normal emulation while $P_r < 1$ indicates the opposite.

We divide the discussion into two cases according to the size of the computation chunk:

3.2.1 Large Computation Chunk

The assumption in this scenario is $T_c \geq T_o$. The emulation with/without basic out-of-core support is illustrated by (a) and (b) respectively in figure 3.2. Accordingly, the execution time of the emulation can be expressed by equation 3.1 and equation 3.2 respectively.

$$T_{wo} = N * (T_c + f_p * T_p) \tag{3.1}$$

$$T_w = N * (T_c + f_o * T_o) \tag{3.2}$$

From the two equations, P_r is shown by equation 3.3

$$P_r = \frac{T_c + f_o * T_o}{T_c + f_p * T_p} \tag{3.3}$$

When the emulation exceeds the physical memory available on the emulating processor, which motivates the out-of-core support in emulation, we expect $T_o \approx T_p$. Considering the memory page size is usually smaller than the memory image size of the target processor, and the fact of the granularity of our initial out-of-core support, we expect $f_p < f_o$. Therefore, we will have $P_r > 1$ meaning the emulation with the basic out-of-core support will be slower than the one without the support.

In the extreme case when the emulation totally fits into the physical memory and we still do the very basic out-of-core emulation (say adopting the second basic design scheme), T_{wo} now is reduced to $N * T_c$ as $f_p = 0$ (i.e., there is no page fault), hence

leading to the new performance ratio shown by equation 3.4:

$$P_r = \frac{T_c + f_o * T_o}{T_c} = 1 + \frac{f_o * T_o}{T_c} \tag{3.4}$$

Now we examine T_w with the prefetch optimization described in section 3.1.3. Since $T_c \geq T_o$, we expect to see the perfect overlapping between the computation and the out-of-core operation as illustrated by (c) in figure 3.2. Accordingly, we have the following two equations:

$$T_w = N * T_c$$

$$P_r = \frac{T_c}{T_c + f_p * T_p}$$

Observing that $P_r < 1$ unless the emulation totally fits into the physical memory, the performance of the emulation with the optimized out-of-core support is always better than the one without the support. This also demonstrates the importance of doing the prefetch optimization, otherwise the out-of-core emulation will show no benefit.

3.2.2 Small Computation Chunk

The assumption in this scenario is $T_c < T_o$. This case has the identical equations for T_w, T_{wo} and P_r as those in the large computation chunk scenario if the emulation

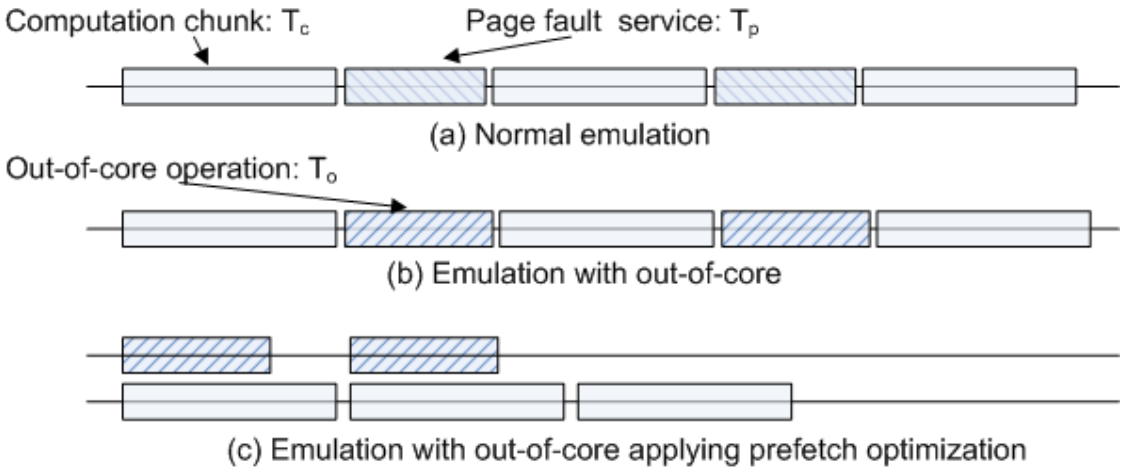

Figure 3.2: Timeline in the case of large computation chunk

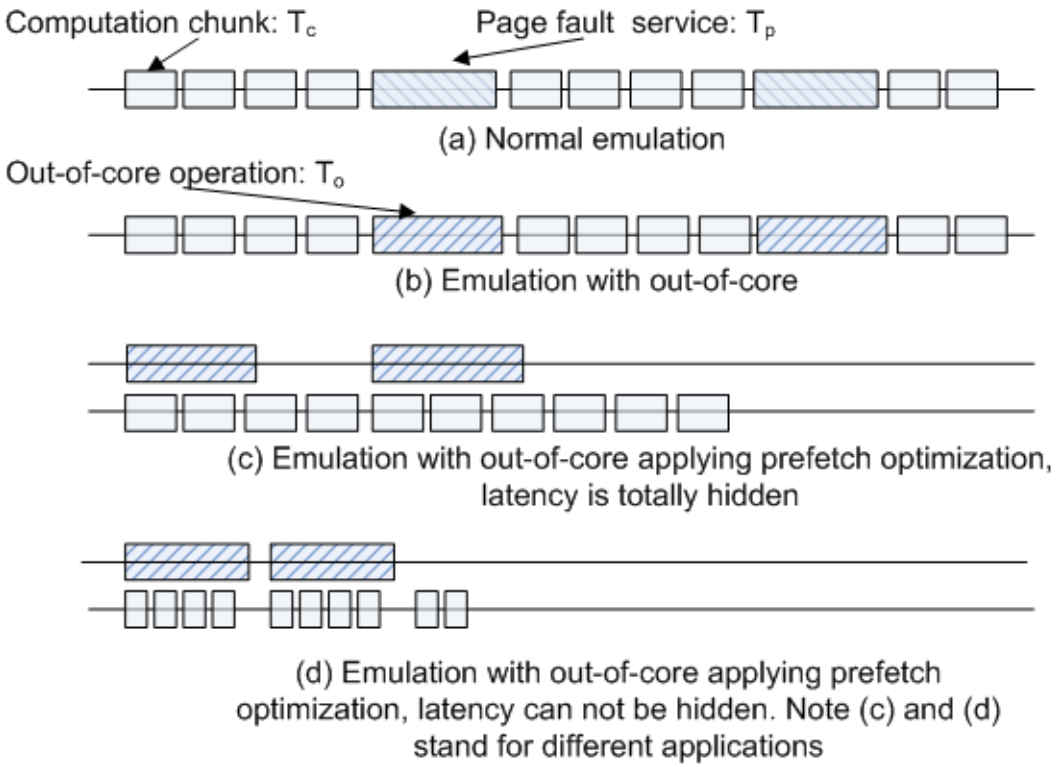

Figure 3.3: Timeline in the case of small computation chunk

is performed with the basic out-of-core support. So it has the same performance analysis as the basic out-of-core support. However, its execution pattern is a little bit different as illustrated by the figure 3.3. Now more computation chunks (which is abstracted in figure 3.3 as multiple versus one in figure 3.2) can be executed before one page fault service or one out-of-core operation.

In this case, the computation chunk is small which implies the data size (i.e. the memory footprint) associated with this computation chunk is small, therefore, the memory could hold more computation chunks before the necessity of performing page fault service or out-of-core operation. Assuming the average number of these computation chunks is K, the following analysis shows the performance of the prefetch optimization scheme is affected by this number K. There are two possibilities:

- $K * T_c \geq T_o$: the execution pattern under this circumstance is illustrated by (c) in figure 3.3. Actually this pattern is the best case we will have during emulation

as the computation totally hides the latency of one out-of-core operation. This happens if and only if we begin to prefetch the data of the $\lceil \frac{T_o}{T_c} \rceil$th computation chunk after the first computation chunk that just starts to execute. In other words, the lookahead has to be at least $\lceil \frac{T_o}{T_c} \rceil$ chunks. This conclusion can be viewed as a piece of advice to implement the prefetch optimization for basic schemes if we wish to achieve the minimal emulation time. Practically, this is very difficult to achieve. The following equation gives the ideal execution time and performance ratio that can be obtained.

$$T_w = N * T_c$$
$$P_r = \frac{T_c}{T_c + f_p * T_p}$$

- $K * T_c < T_o$: the execution pattern under this circumstance is illustrated by (d) in figure 3.3. The analysis is almost the same with that of the other possibility except that the latency of the out-of-core operation can not be hidden even in the ideal case. Now the ideal execution time and performance ratio are different as follows:

  $$T_w = N * (f_o * T_o)$$
  $$P_r = \frac{f_o * T_o}{T_c + f_p * T_p}$$

  Based on the equations, the emulation with the optimized out-of-core support may or may not outperform the normal emulation.

In conclusion, based on the above analysis, the basic out-of-core scheme will not buy anything compared with page fault service provided by the virtual memory system unless the prefetch optimization is applied. In addition, when it comes to the small computation chunk, the prefetch optimization will have to be more complicated in terms of implementation in order to have the least emulation time since the lookahead is larger than one.

Chapter 4

Implementation of the Out-of-core Approach

This chapter introduces the implementation of the basic schemes discussed in chapter 3.

4.1 Implementation Overview

So far, the three basic schemes presented in the previous chapter have been implemented and integrated into the latest BigSim version. The optimization scheme that applies the prefetch technique to improve the performance of out-of-core emulation is still in progress.

From the perspective of software engineering, the implementation of the out-of-core schemes should be modular so that this component can be easily decoupled from the existing BigSim framework, and further enhancements to BigSim, such as adding a new functionality, will not need to be mixed with the implementation of the out-of-core component. According to this, we made our best to mix as few code fragments of the out-of-core implementation as possible with the current code of BigSim, and gathered all out-of-core related implementation into a separate module. In addition, bringing the target processor in-core and taking it out-of-core resembles the functionality of checkpointing or chare object migration which has already been implemented in the Charm++ system. Therefore, we try to implement the out-of-core emulation by maximizing the usage of existing software framework. The packing/unpacking framework described later (in section 4.2) is such an example

which is the fundamental software module in the Charm++ system to help implement object-migration related functionalities.

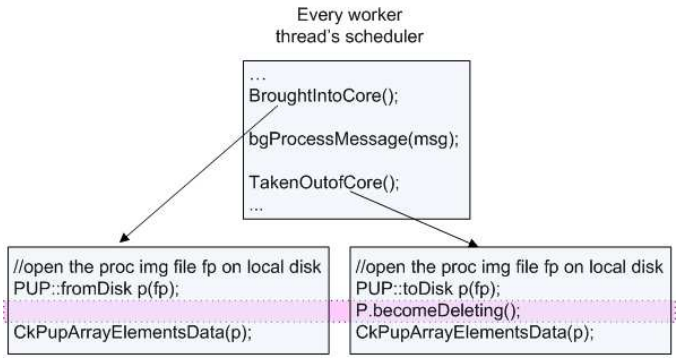

Figure 4.1: The implementation skeleton of the first basic out-of-core scheme

The initial implementation is straightforward with the design schemes described in chapter 3. The implementation skeleton of the first design is shown in figure 4.1. Inside each worker thread's scheduler code, the message processing function "bgProcessMessage(msg)" is now embraced with a pair of functions that are respectively responsible for bringing a worker thread in-core by "broughtIntoMem()" and taking a worker thread out-of-core by "takenOutofMem()". The two functions have almost the same work flow. They both utilize the packing/unpacking framework in the Charm++ system implemented by the use of "PUP::toDisk" and "PUP::fromDisk" classes. "CkPupArrayElementsData" means packing/unpacking all chare array elements on a target processor. The difference as shaded in figure 4.1 lies in the pupper instance having an additional state of "deleting" when the target processor is evicted. The cause will be mentioned in section 4.3.2 where AMPI related implementation issues are described.

The implementation skeletons of the last two designs are shown in figure 4.2. In this case, the only change to the scheduler code is adding one function call "bgOutOfCoreSchedule(this)", inside which two different schemes are implemented. Both contain almost the same work flow but differ at three points (the first two are illus-

trated by shaded area in figure 4.2). Firstly, they differ at the part of selecting and evicting a worker thread when a certain condition is encountered. For the multiple target processors based scheme (the second one), this refers to the condition that the limit of the number of worker threads allowed in-core has been reached while in the memory based scheme (the third one), this refers to the condition that the maximum amount of memory allowed to use has been reached. Secondly, the third scheme may require evicting multiple target processors to spare enough empty memory while the second does not need to. Lastly, they differ at the organization of multiple worker threads. In the second scheme, the multiple worker threads are organized as a table since the number of worker threads is fixed while in the third scheme, the worker threads are organized as a list because the number of worker threads is dynamically changing. Regarding the common points shared by both schemes in terms of implementation, we can see inside function "bgOutOfCoreSchedule()", functions "broughtIntoMem()" and "takenOutofMem()" are reused as shown by the skeleton

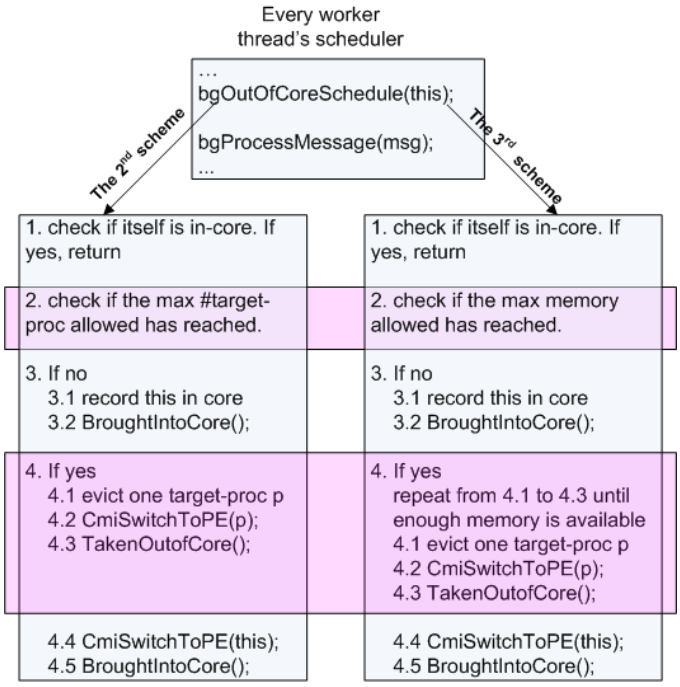

Figure 4.2: The implementation skeleton of the last two basic out-of-core schemes

codes. This demonstrates that the implementation of the first scheme is a foundation for the implementation of other relatively complex design schemes. The other common point worth mentioning is that when taking a worker thread out of memory inside the current context of worker thread, we have to switch to the context of that worker thread by calling "CmiSwitchToPE(int)" in order to pack data correctly, and afterwards switch back to the current context.

4.2 Packing/Unpacking Framework

As mentioned before, the packing/unpacking framework facilitates the implementation of migration-related functionalities. This framework, a.k.a "pup" framework[15] is a collection of efficient and elegant classes that enable the chare arrays of Charm++ to be migrated from one processor to another processor or to the disk (as in the case of checkpointing or the out-of-core execution). It is designed to describe the in-memory layout of an object. The framework can be extended to provide services to any operation that requires a traversal of the object state (typically a traversal over the object data members).

The object migration is usually handled in a way that requires each class to implement "pack" and "unpack" methods. The "pack" method serializes the object's data into a buffer before migration while "unpack" method de-serializes the object's data from the buffer upon the completion of migration. The functionality of both methods is similar in nature, i.e., the object data is packed in a particular order and unpacked in the same order as it is packed. Therefore, code duplication is resulted from a common code skeleton shared by both methods, while differing only in the actual operation that the method performs on the data members. The pup library is designed for the purpose of preventing such code duplication. The developer of a particular class only needs to implement a single "pup" method which takes a single

parameter of class "pupper". The role of this method is to perform a traversal of the data members inside an object, i.e., to describe the memory layout of data members. The actual operations that need to be performed on the data members are executed by the pupper.

The following describes the classes contained in the pup library that are used in this work:

- class PUP::er: this class is the abstract superclass of all the other classes in the library. The "pup" method of a particular class takes a reference to an instance of this class as parameter. This class has methods for handling all the basic C++ data types. Additionally, this class provides function to query its operation mode such as in packing or unpacking mode.

- class PUP::toDisk: the role of this class is to save the data members of the object it operates on into a disk file. It is usually initialized with a file pointer.

- class PUP::fromDisk: the role of this class is just the opposite of the previous class PUP::toDisk.

Sometimes it is quite tedious to write the "pup" method for a class, hence a tool to automatically implement this method for each class is a convenient feature. Recently, this has been realized by one of the Charm++ developers who utilized a source-to-source translator to provide an initial support[16].

4.3 Implementation Issues

4.3.1 Charm++ Runtime Related Changes

Although the out-of-core emulation is viewed as a similar case of object migration in the Charm++ system, it is different from the real migration in that the object

taken out of the memory is still on the same processor, opposed to migrating to other processors. This difference requires changes to the Charm++ runtime system described as follows:

- *Changes to chare array manager*: In the real migration case, if one chare array element migrates from one processor to another processor, the chare array manager will regard this element dies on the first processor, thereby deleting its related information in the array location manager, i.e., claiming this processor now no longer owns this array element. This is not true for the out-of-core emulation since the array element is just out-of-core, not actually dying. Therefore, in the implementation of out-of-core emulation, we make changes to bypass this part of code. The reason that we do not recover this state when the element is brought into memory is because of the overhead incurred by the recovery.

  The array manager's listener scheme for real migration suffers from this difference as well. The listener scheme is invoked through the function "CkPupArrayElementsData" which implies the semantic meaning that the object migration just happens. Therefore, in the real migration case, the listener is notified some array element is migrating, and keeps waiting for the arrival, which will never happen in the out-of-core emulation. If not disabling this listener scheme in the out-of-core emulation case, we would see a hang of the program.

  There are other places in the chare array management code sharing a similar problem because of not differentiating the real migration from the out-of-core emulation. A simple solution we used in our initial implementation is to add a new state (i.e., the out-of-core state) for the chare array element to represent this difference.

- *Reduction problem*: This difference also affects the correctness of reduction implementation in the Charm++ runtime system. The reduction on a chare

array is performed through a spanning tree. Each processor acts as a tree node in this spanning tree. The processor uses a counter to record the number of reduction messages it needs to receive. When that number equals the number of elements of this array that stay on this processor, the processor will immediately perform the operation associated with the reduction and pass the result to the upper tree node. In the real migration case, the counter is modified to reflect the fact that some array element has migrated to other processor, thus there is no need in waiting for its reduction message. However, in the out-of-core emulation case where the array element still stays in the original place, this counter should not be modified. Otherwise, the processor will perform the reduction operation earlier, thinking that itself has already collected all the reduction messages. But the fact is that it will miss one reduction message, hence obviously invalidating the whole global reduction operation. The new state added to the chare array element is again used here to differentiate the case.

4.3.2 AMPI Runtime Related Changes

Regarding the out-of-core emulation of MPI programs, the AMPI thread which represents a MPI process will also be taken out of and brought into memory as it is a chare array element on the target processor. With this pair of operations, the starting virtual address of this AMPI thread is changed. In other words, if we use a pointer variable in C++ to point to this thread, the value of this pointer variable is changed when this thread is brought into memory again. More broadly speaking, all objects that go through the pup operation (i.e., going through the out-of-core and in-core operations) usually have their memory location changed. Such memory location change is the root cause of various modification to the current implementation of the AMPI system.

1. *Scheduling problem of AMPI threads*: In the original implementation of emulation component of BigSim, the worker threads and the AMPI threads are both scheduled by the Converse. This scheme works correctly for normal emulation. But when it uses the out-of-core technique, this scheme fails because of the following reason. When the worker thread (the target processor) is evicted from memory, the AMPI threads staying on this processor are simultaneously evicted out. However, assuming one of these AMPI threads has been queued in the Converse scheduler as a general message, and it is going to be the immediate message to be processed, the execution of this thread will be incorrect since this thread's context information is not in-core. What is worse, this message is consumed, i.e., the message to drive the execution of this AMPI thread is lost when it is brought into memory next time. Therefore, the program will make no progress at all!

   The solution to this problem is to utilize the scheduling hooks as mentioned in the introduction of Converse system in chapter 2. Through this scheduler hooks, the AMPI thread is scheduled by the scheduler of the target processor it belongs to. In other words, the invocation of an AMPI thread is now triggered by the function "bgProcessMessage(msg)" inside the worker thread's scheduler. This ensures the context of the AMPI thread must be in-core, either if it is just brought into memory before processing the message or if the worker thread is already in-core. Remember that the thread invocation is done through a token object (described in section 2.1.1), therefore, the pointer inside this token which points to the to-be-invoked AMPI thread has to be updated when the thread is brought into memory. With these changes, the emulation component is now correctly executing AMPI threads even when the out-of-core technique is used. Another benefit of this scheduling mechanism change for AMPI threads in the emulation component of BigSim is that now it sounds more logically correct in

terms of scheduling for AMPI threads as they are now scheduled by the worker thread (i.e., the target processor) they belong to rather than the Converse layer which is shared by all worker threads on the emulating processor.

2. *MPI_Init problem*: The "MPI_Init" call in the AMPI system is designed to create the required AMPI threads (its number is usually equal to the number of target processors in the emulation) and other related chare array objects. The creation of these entities (including threads and chare array objects) is asynchronous as it is invoked by Charm++ entry methods. Since the creation of some chare array objects depends on the creation of AMPI threads, i.e., only after AMPI threads have been created and initialized, the chare array object can be created, a semaphore is used to guarantee this order. This semaphore contains pointers to every other AMPI thread. Therefore, the problem of memory location change of AMPI mentioned in the beginning of this section invalidates the semaphore because the thread pointers in the semaphore are not updated in the original implementation.

   To solve this problem, one approach is to add a new data structure in the AMPI thread to record which semaphore this AMPI thread participates in. Once this AMPI thread gets pupped, we update the corresponding pointer in the semaphore obtained through this new data structure. Additionally, when the semaphore gets pupped, the new data structure in the AMPI thread should also be updated to ensure the thread can obtain the correct semaphore. This approach is quite complicated in terms of implementation.

   The solution we adopted in the current implementation is not to use out-of-core technique until the "MPI_Init" call is finished. This method is simple to implement and also very reasonable based on the fact that "MPI_Init" is frequently the first function call in MPI programs and at that time, the real

application work has not started yet, hence there is very little consumption of memory.

3. *Isomalloc-related problem*: Isomalloc is used in AMPI system to help migration related functions[3]. It ensures that even after the AMPI thread migrates to another processor, the thread's stack will stay at the same virtual address in memory that it had on the old processor so that the pointers embedded in the stack will still work properly. As out-of-core is viewed as a special migration, its implementation has to deal with this memory management module where we identified two related problems.

   First of all, in the case of emulating hundreds of AMPI threads on a single physical processor, the virtual address space will be used up on 32-bit machines for isomalloc. As a result, we naturally shift to use 64-bit machines to perform the emulation as it is expected that a much larger virtual address space is supported. However, the emulation still reported the virtual address space was used up. It turns out the original isomalloc implementation used a 32-bit variable to track the size of virtual address space that had been allocated. Therefore, no matter it is on a 32-bit machine or a 64-bit machine, it will definitely overflow when the size becomes very large, which signals the error that the virtual address space is used up. By changing the variable to a machine-dependent integer type (i.e., the size of the type is 32-bit (64-bit) when it is on a 32-bit (64-bit) machine, the problem is resolved.

   Secondly, regarding the pupper instance used in taking a target processor out of memory, we found we need to set its state to be "deleting" which is shown in the implementation of "takenOutofMem()" function (refer to figure 4.1). This was identified through a segmentation fault we encountered when we were emulating a MPI program with the out-of-core support. After many efforts

of debugging, this error was narrowed down to the memory allocation of a variable. It was very strange that the virtual address of this variable returned by the isomalloc module was only a few bytes away from the starting virtual address of this AMPI thread's stack. The root cause of this strange behavior is that we forgot to set the state of the pupper instance to "deleting" when swapping out an AMPI thread. The isomalloc manages both AMPI thread's heap and stack. From the point view of isomalloc, these two structures have no semantic differences. Both heap and stack are part of the list of memory blocks the isomalloc maintains. If the state of the pupper instance is not set to "deleting", when taking a thread out of memory, the thread's stack is not freed at all. When it comes to the pupping procedure of isomalloced memory blocks associated with the thread, the blocks belonging to the stack are treated as part of the heap. Considering the thread is back to memory again, the stack space is treated as the free memory space as indicated by the pupping procedure. Therefore, the allocation of a variable (should be a heap data) afterwards will be very likely created on the stack space, thus corrupting the stack of the thread, and encountering a segmentation fault.

4. *Problems of data structure sharing by multiple pointers*: This kind of problems may be possibly quite specific to our implementation of the AMPI system. The root cause of this kind of problems is mentioned at the beginning of this section. One real example of this kind is the implementation of "MPI_Irecv".

   Abstractly, this kind of problem is generalized as follows: assume we have two pointers A and B, both pointing to a data structure C. The program progress relies on checking the state changes inside C, which is performed through pointer B. But changing the state of C is done through pointer A. During the out-of-core emulation, C is pupped through pointer B. Therefore, B is always updated

with the latest memory address of C, but A is not. When the state change of C is triggered by pointer A, the state change of C is lost as it operates on a fake C as shown by the figure 4.3. As a result, the program will never make progress through the reference of B as it sees no state change in C. The basic approach to solving this problem is to have only one pointer to C. Assume B is the only pointer to C, and the access to C through pointer A is replaced by either directly or indirectly through pointer B. "directly" refers to the change in the codes that A is directly replaced by B, while "indirectly" refers to the method of obtaining B through operating on A. We have to be cautious if we have to use the indirect method. In this case, the value of A should not record the memory address of B. Otherwise, when B gets updated through the pupping routine, how does A knows the new memory address of B? Considering the example of resolving such problem in the implementation of "MPI_Irecv", we have to use the indirect approach. In that scenario, B is one element of an array which is globally accessible, and its index is fixed during execution. Therefore we choose A to record the array index of B, leading to an indirect access to C.

4.3.3 Modifications to Load Balancing Framework

Automatic load balancing is a very important feature of the Charm++ runtime system. To enable this feature in the case of out-of-core emulation, there are some

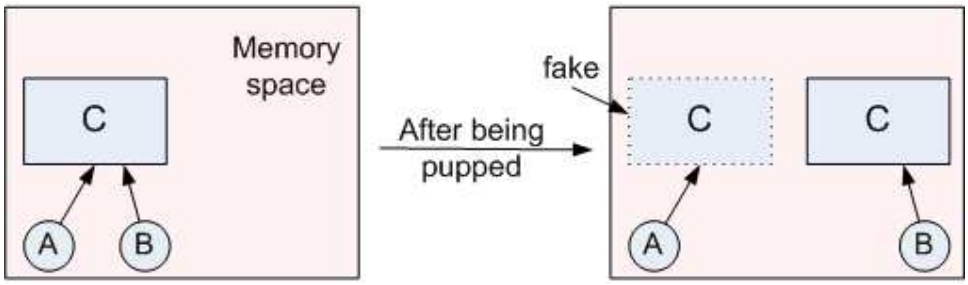

Figure 4.3: The issue of one data structure shared by multiple pointers

subtle issues to be resolved. Firstly, each chare array element is associated with a "Loc_rec" instance in the Charm++ runtime. The load balancing framework on each processor maintains an array with each element representing the load of a "Loc_rec" object. We refer to the array as "load array". Since the "Loc_rec" is associated with a chare array element, the load balancing framework can automatically access the chare array element through the reference of "Loc_rec" and perform the migration operation on it. Once a new "Loc_rec" instance is created, the load balancing framework will automatically assign the first free slot in the "load array" to represent the load information of this new "Loc_rec" object.

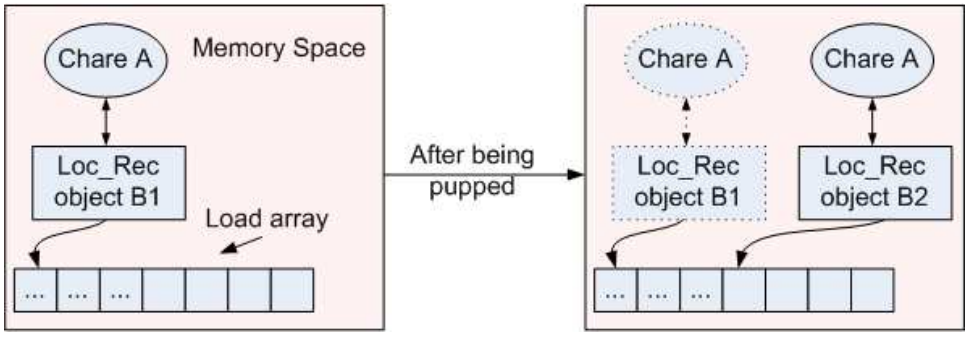

Figure 4.4: The issue of recording load information incorrectly for a chare object

Considering one scenario of out-of-core emulation as shown by figure 4.4, the new "Loc_rec" object B2 will be created upon the back to memory from disk of a chare array element A. The load balancing system regards "Loc_rec" object B2 as a new one although it represents an old chare array element A whose load information has been recorded through the old "Loc_rec" object B1. As mentioned before, the "Loc_rec" object B2 now occupies a new slot in the "load array". The following load information of chare array element A will be recorded in the new slot rather than in the old slot that "Loc_rec" object B1 holds. Therefore, the load information of a chare array element is not correctly collected, thereby leading to a wrong automatic load balancing strategy. The fix to this issue is to insert the "Loc_rec" object B2 to the old slot as the object B1 holds in the "load array".

The second issue is related with the periodic built-in load balancing in the Charm++ runtime system. Once it is periodically triggerred, if it identifies that there is no chare objects, hence no workload, on its processor, a round of load balancing is performed. The identification is done through a variable named "client_count" in the "LocalBarrier" class. When this variable reaches 0, meaning there is no chare objects on the processor, the "LocalBarrier" object will send a message to the converse scheduler to trigger a round of load balancing across the whole system. During the out-of-core emulation, this periodic load balancing may happen if the emulation runs at least as long as a period of this built-in load balancing. Considering the follow scenario, after taking all the chare objects on a target processor from memory, the "client_count" variable is reduced to 0. The built-in load balancing happens to occur immediately after this, before the target processor is back to memory. Therefore, a round of false built-in load balancing takes place as the chare objects on that target processor do not really disappear. To prevent this from happening, we simply avoid decreasing the value of variable "client_count" when the chare object is in the state of being taken out of memory.

Chapter 5

Experimental Results

We first present the results of verifying the correctness of using the out-of-core scheme in the emulation, and then the performance evaluation of the initial implementation of our out-of-core scheme on a kernel benchmark written with MPI.

Charm++ and MPI programs are the targets for our emulation, therefore, both types of applications are chosen for the correctness verification part. But as for the performance evaluation part, the MPI program is the focus because firstly, MPI is the de-facto standard of programming model for writing large scale parallel programs, and is expected to be still used on petascale platforms. In addition, MPI programs are enabled by the AMPI system which is built-on Charm++ and has its own complexity of implementation to be in accordance with the MPI specification. Therefore, we believe MPI programs are good examples for evaluating out-of-core schemes comprehensively for the emulation part of BigSim.

The testing environment is a dual quad-core Intel Xeon machine installed with a 64-bit version of Linux and 4GB memory in total. Although the performance evaluation is not done on the existing supercomputers, the specification of this machine is very similar to a single node on those supercomputers. So we believe it is reasonable to use this machine as the evaluation platform for our initial implementation of out-of-core schemes.

5.1 Correctness Verification

The main method of verifying the correctness of the out-of-core emulation is through comparing the output of two emulation runs of the same application, one running with out-of-core technique enabled and the other running without. As there are three schemes as described in chapter 3, we have verified each of them in detail. But once we ensure the correctness of the first scheme "per message based scheme", since it lays the foundation for the other two schemes, we could regard the out-of-core emulation to be correct.

5.1.1 Verification by Charm++ Programs

First of all, we tested the so-called "hello world!" program in the Charm++ system. This program first creates a chare array and then invokes a "sayHi" function on each array element one by one. This program covers the test of the basic chare array management inside the Charm++ runtime system in the case of out-of-core emulation. By resolving the chare array manager related issues mentioned in section 4.3, we successfully passed this entry-level test.

Afterwards, we tested a reduction Charm++ program. This program again first creates a chare array, each element of which has an integer variable member and is initialized with a value equal to the array index of this element. Then, the program does a global reduction on this chare array by summing up the values of each element's integer variable member and output the final result. Therefore, if there are k elements, the reduction result will be $\sum_{i=0}^{k-1} i$. In the real test, we specified the array size to be 101 and emulated the reduction program on 8 target processors. The reduction result output by the out-of-core emulation is correct since it is equal to 5050 as it should be.

5.1.2 Verification by a MPI 3D-Jacobi Program

The MPI 3D-Jacobi program is a kernel benchmark program which performs a 7-point stencil computation with 3D decomposition. In the program, every chunk of data communicates with its six neighbors in three dimensions via the "MPI_Irecv" and "MPI_Send" pair. After the Jacobi relaxation computation, the maximum error is calculated via "MPI_Allreduce" among all local errors. Even more, after a ceratin number of iterations, "MPI_Migrate"(a MPI API extension on AMPI for explicitly performing load balancing by migrating AMPI threads across the whole system) is called to trigger the load balancing. In spite of the simplicity, this kernel benchmark could be viewed as a relatively comprehensive test for the correctness of out-of-core emulation in the context of AMPI system. The journey of passing this kernel benchmark excercised almost all the implementation issues we have described in the section 4.3. At first, we deliberately disabled the call to "MPI_Migrate" to test the standard MPI functionalities, by comparing the output which shows the relaxation error of each iteration. We verified the correctness of the out-of-core emulation for this purpose. Secondly, to verify the load balancing strategy is handled correctly, we explicitly specify the strategy as "RotateLB" meaning the AMPI threads migrate in a round-robin manner after each call of "MPI_Migrate" so that we know statically where each AMPI thread will migrate to. To facilitate the checking, the program does "MPI_Migrate" in each iteration, and each AMPI thread output its position at the beginning of each iteration. With such output information, we verified the load balancing system functioned correctly.

5.2 Performance Results

The performance evaluation is focused on the kernel benchmark–MPI 3D-Jacobi program. We have performed two different tests with different purposes in mind. One is

intended to obtain the performance of the initial implementation of out-of-core compared with the normal emulation case even the emulation totally fits into physical memory. The other is a stress test where the normal emulation will use up all the physical memory available on the emulating processor. This is the technically real test for out-of-core implementation.

5.2.1 Results and Analysis of Normal Tests

We begin with the description of the result of the first test. The configuration is as follows: 1 emulating processor, 8 target processors and only 1 AMPI thread (i.e., one MPI process) on each target processor. We selected the second one (described in section 3.1.2 as the out-of-core scheme, and specified that only two target processors are allowed in memory. Table 5.1 shows the performance comparison between the emulation with and without out-of-core support.

problem size per MPI process	image size of target proc (MB)	emulation time s/step		slowdown ratio
		w/o out-of-core	with out-of-core	
10^3	1.03	0.024	0.755	31.46
50^3	2.31	0.146	1.716	11.75
100^3	10.01	1.826	25.110	13.75
200^3	67.55	15.728	184.816	18.11

Table 5.1: Performance comparison between emulation with and without out-of-core support

The problem size in the first column of the table refers to the total number of elements (of type "double") the target processor will work on. An analysis of the source code shows that the asymptotic execution time of one computation chunk is about linear with the problem size for this benchmark. The second column shows the corresponding image size of a target processor. The target processor will be stored as a disk file when it is out-of-core, thus the image size is obtained through monitoring when the size of this disk file becomes stable during emulation. Remember

that the granularity of this out-of-core support in emulation is performed at a level of per target processor, so the image size is an indicator of the cost of one out-of-core operation (i.e., bringing one target processor into memory or taking it out of memory). The slowdown ratio presented in the last column is quite expected as the normal emulation (without out-of-core) has far fewer disk I/O operations than the one with out-of-core has.

The trend of slowdown ratio with regard to the problem size is interesting but explainable based on our theoretical analysis in section 3.2. The slowdown ratio in this test could be approximately modelled by equation 3.4 ($P_r = 1 + (f_o * T_o)/T_c$). Since the application computation and communication pattern is fixed in all four cases of problem size and the out-of-core scheme used is identical, f_o should remain unchanged. Therefore, the change of slowdown ratio depends on the $\Delta T_o/\Delta T_c$. T_c is changed linearly with regard to the problem size, but T_o is not with regard to the image size of one target processor. T_o not only includes the time spent on disk I/O which could be viewed asymptotically linear with the image size of the target processor, but also includes the time spent on re-laying out the memory for the target processor upon its back into memory (i.e., the pupping routines of chare objects). We think T_o should change asymptotically more dramatically than linear with regard to the image size. From problem size 10^3 to 50^3, the T_c will change by 125 times, far outweighing the T_o change as the image size of the target processor is changed by about twice. Therefore, we have $\Delta T_o/\Delta T_c < 1$, hence a drop in the slowdown ration. Now considering the change of problem size from 50^3 to 100^3, the T_c will change by 8 times, and the image size of the target processor changes by around 5 times. Considering that T_o will change more dramatically than linear as mentioned above, we have $\Delta T_o/\Delta T_c > 1$ but not much, resulting in a slight increase in the slowdown ratio. The same applies to the increase of slowdown ration from problem size 100^3 to 200^3.

5.2.2 Results and Analysis of Stress Tests

Finally, we present the result of the stress test. In this case, the configuration is as follows: 1 emulating processor, 512 target processors and still 1 AMPI thread on each target processor; the problem size per MPI process is chosen as 200^3. Based on the image size of target processor in table 5.1, although the total emulation memory footprint calculated by $512 * 67.55\text{MB} \approx 3.5\text{GB}$ is less than 4GB, the actual memory footprint monitored through "top" command actually shows the emulation uses up all the 4GB memory available. This is understandable because extra memory usage such as by the OS, communication buffer and memory padding is not considered at all in the calculation above. The second out-of-core scheme is also used. The number of target processors allowed in memory is 2 in one case and 128 in another case. Both cases showed the very close slowdown ration of 1.77 which is much smaller compared with that in the first test. In this test, the slowdown ratio agrees with the equation 3.3, and the difference in ratio change from the first test also aligns with the previous explanation of the first test results.

Chapter 6

Summary and Future Work

Due to the various challenges brought by the petascale machines, the application development for such ultra-scale machines becomes a daunting task for developers. In particular, the traditional gap between the software deployment and the machine deployment should be bridged so that the computation power of the petascale machines could be fully utilized. Therefore, it is very important to have a software system to predict the performance and identify the potential bottlenecks of applications that will run on petascale machines before the actual machine is deployed. The BigSim system is designed with the intent to provide such support. One specific challenge in developing this system is the emulation of petabytes of memory that an application will use on petascale machines. This thesis presents a preliminary investigation of addressing this challenge by taking advantage of the out-of-core technique. We have presented the design space for the out-of-core technique and described three basic schemes in the context of the emulation component of the current BigSim system. We also explored the possible optimizations for the basic schemes such as a lookahead based eviction policy to precisely know which should be the next target processor to be brought into memory, and a prefetch technique to allow the overlapping between the message processing and the I/O operation of bringing a target processor into memory. The theoretical analysis afterwards points out not only the importance but the complexity of the prefetch optimization in order to achieve good performance. Besides the exploration of design space and the theoretical analysis of the design schemes and optimizations, we showed our initial implementation of the

three basic schemes and the issues we have encountered during the implementation regarding the Charm++ programs and MPI programs enabled by the AMPI system. The experimental evaluation of this initial implementation is presented as well, to show the correctness in functionality of the out-of-core technique and the initial performance results. Currently, considering the emulation for a MPI 3D-Jacobi kernel benchmark, there is about 13 times slowdown on average, which is understandable, by using the initial implementation of out-of-core support compared with the normal emulation case where the whole emulation fits into memory.

Regarding the future work, we continue working on testing the basic out-of-core schemes on real world applications such as a Turbulence program which is a candidate petascale application. This will make new features in BigSim more mature for production mode usage.

The current performance gap leaves a lot of space for future work from the perspective of performance tuning. First of all, we could finish the implementation of the prefetch optimization mentioned in the thesis. Secondly, we could investigate the ways of doing out-of-core emulation at a finer granularity level so that the disk I/O cost is reduced as much as possible. The current implementation is done at the level of swapping one target processor at a time partially due to the constraints of existing software infrastructure. For example, we could design a clean interface between the Charm++ runtime system and the emulation component of BigSim in order to expose finer-granularity information, such as the chare object level, to the emulation. With this information, we may do the out-of-core emulation at a finer granularity level.

The future work also includes exploring other approaches to the emulation of petabytes of memory such as modelling the memory footprint implicitly. It is observed that many parallel applications produce instances which contain large data segments which are either replicated, or differ only in content (say, different floating point values). Furthermore, variations in this data generally do not influence the flow of

control. The execution time to compute using that data is solely governed by its size, thereby resulting in the observation that if the object of study is execution time, rather than obtaining the actual numerical result, the difference in content is irrelevant. Thus, the same data segment may be shared across target processors. Furthermore, the replicated read-only data segments can be shared as well across target processors without incurring any correctness penalty.

References

[1] Gengbin Zheng, Gunavardhan Kakulapati, and Laxmikant V. Kalé. Bigsim: A parallel simulator for performance prediction of extremely large parallel machines. In *18th International Parallel and Distributed Processing Symposium (IPDPS)*, page 78, Santa Fe, New Mexico, April 2004.

[2] Gengbin Zheng, Terry Wilmarth, Praveen Jagadishprasad, and Laxmikant V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, volume 33, pages 183–207, 2005.

[3] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003), LNCS 2958*, pages 306–322, College Station, Texas, October 2003.

[4] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance evaluation of adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

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

[6] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[7] Parallel Programming Laboratory. *Converse Programming Manual*. Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. http://charm.cs.uiuc.edu/manuals/html/converse/manual.html.

[8] Laxmikant V. Kalé. The virtualization model of parallel programming : Runtime optimizations and the state of art. In *LACSI 2002*, Albuquerque, October 2002.

[9] Laxmikant 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.

[10] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[11] Eric Bohm, Glenn J. Martyna, Abhinav Bhatele, Sameer Kumar, Laxmikant V. Kale, John A. Gunnels, and Mark 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 (to appear)*, 2007.

[12] Gengbin Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

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

[14] Free Software Foundation. *The GNU C Library*. http://www.gnu.org/software/libc/manual.

[15] Parallel Programming Laboratory. *The Charm++ Programming Language Manual*. Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL. http://charm.cs.uiuc.edu/manuals/html/charm++/manual.html.

[16] Isaac Dooley. Automated source-to-source translations to assist parallel programmers. Master's thesis, Dept. of Computer Science, University of Illinois, 2006. http://charm.cs.uiuc.edu/papers/DooleyMSThesis06.shtml.