AUTOMATIC OUT-OF-CORE EXECUTION SUPPORT FOR CHARM++

BY

MANI SRINIVAS POTNURU

B.Tech., Regional Engineering College,Warangal,India, 2001

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, 2003

Urbana, Illinois

Abstract

Many of the computationally intensive parallel applications are also memory intensive. The Operating System's Virtual Memory System will let the application run even when the required memory is beyond the available physical memory. However this comes at a substantial cost of taking a page fault, whenever there is a memory access for the swapped out pages. Normally, an application does not have any control of this memory system. But since the application will have a better idea of what pages will be needed in near future, out-of-core techniques for each application are developed seperately to deal with this issue. We present a generic, application independent, technique as part of the application's runtime library to improve the paging performance of these memory-intensive parallel applications. We exploit the message-driven execution style of parallel programming along with the virtualization provided by Charm++ objects. The data driven objects provide the prediction mechanism necessary for an effective prefetching scheme. The implementaion of this automatic out-of-core execution technique inside the Charm++ runtime libray is described along with experimental data using a real-world application.

To my Dad.

Acknowledgments

First and foremost, I would like to express my gratitude to my advisor, Professor Laxmikant Kale for his generous support, guidance, encouragement and understanding. He was a constant source of insight and motivation to strive for betterment. This work would not have been possible without him.

Thanks to my senior collegues at Parallel Programming Laboratory: Gengbin Zheng, Orion Lawlor and Sameer Kumar. They have provided valuable inputs to this work. I thank Orion for helping me understand charm++ internals and guiding me in the required modifications. I particularly thank Gengbin for his enormous patience in answering my innumerable questions related to Converse. Without his help in the final stages, this work would not have finished in time.

I would also like to thank my colleagues in the PPL for their insight. Specifically, I would like to thank Sayantan, Cheewai, Terry for the fun times in PPL.

Throughout my years at the University of Illinois I have met many influential and supportive colleagues and friends. All of whom I would like to thank for their support and friendship and without which, my experience at the University of Illinois would have truely been lacking. In particular, I would like to thank Sindhura for her endless support and encouragement in my difficult times.

A special thanks to my parents who have always encouraged me throughout my studies.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel programs often deal with complex large problems. Besides being computation and communication intensive, often parallel applications have huge memory requirements. In spite of the relatively large amount of memory available in today's computer systems, more and more parallel applications are being developed that require more memory than the available physical memory. This problem is more perceivable in Cluster's environment, where the nodes are off-the-shelf computer systems which typically contain much less memory compared to the supercomputers.

Operating System's Virtual Memory mechanism satisfies the memory requirement of these applications by paging the required pages of memory from physical memory. So the operating system tries to figure out which pages of the memory are used less frequently and which are used more often, and swaps out the less frequently used pages to the disk. If there is a memory access for any of these swapped out pages, the access takes a page fault, which essentially involves a interrupt generated by the hardware and a access to the swapped out pages on the disk inside the interrupt handler. Hence these page faults take several milliseconds compared to a few nanoseconds for memory accesses to the in-memory (in-core) pages. In addition, the application's process will be then removed from the runnable processes queue, while the page-fault handler is executed and the application looses some more time due to the process context switch. In case there are no other competing processes, the CPU sits idle during this peroid. Therefore, reducing or hiding page faults is crucial in achieving high performance. We aim at improving performance of such applications by exploring and analyzing the concept of Out-of-Core execution.

Software Prefetching is a technique for tolerating memory latency by explicitly getting data into memory before it is actually required. Thus when the program makes a memory access to a page, the page is likely to be available instantaneously instead of going through a expensive page fault. This thesis explores a multi-threaded approach for data prefetching. It creates one or more threads that will fetch the data into memory. When one thread is blocked on a page fault, system passes control to other threads [1]. Thus the performance loss due to paging can be reduced by overlapping it with computation.

One step beyond the above *prefetching* scheme, is managing the application's pages by itself, rather than reling on the OS's virtual memory system. In this approach either the application or the runtime system maintains data structures similar to the OS's page tables and does the page management by itself. In a way, in this approach the application takes complete charge of its memory management.

Both of the above approaches present a major challenge. To fetch the required data or to effectively manage the pages, the program should correctly predict what data is going to be accessed in near future. Data-driven object oriented paradigms overcome this challenge of predicting memory-access pattern easily. In such systems, the execution of object's methods is triggered by the availability of messages (method invocations) under the control of a prioritized *scheduler*. Thus the scheduler has the knowledge of the objects that are going to receive messages in near future. So the Out-of-Core schemes can deal with at the objects level instead of at the pages level. Though this is a lot coarser than the page management, we show that for our purposes this will be enough. In the following sections, we discuss one such data-driven object oriented paradigm and the above mentioned schemes for supporting *Automatic Out-Of-Core Excecution* in detail.

1.1 Thesis Contributions

The contributions of this thesis are:

- Application independent *Automatic-Out-Of-Core* execution support is added to the Converse runtime system and the interface between the Charm++ language system and the Converse system is modified appropriately.

- Exisiting applications can use the Out-of-Core support without any modifications to the code. The application can even control the parameters of the runtime system through the new set of parameters possible to pass to the *Charmrun*(which is used to load the executable onto the parallel machine).

1.2 Thesis Organization

This thesis comprises of eight chapters. Chapter 2 provides a basic foundation of the Charm++ programming language. Chapter 3 gives a overview of Converse runtime system and Chapter 4 discusses the schemes for Out-of-Core support in detail. The Pack/Unpack framework is discussed in Chapter 5. Details of the implementation are provided in Chapter 6. Chapter 7 contains a performance analysis of Out-of-Core schemes. Finally, conclusions and future work are the topics of Chapter 8.

Chapter 2

Charm++

Charm++ is an object-oriented portable parallel language [5],[2] based on C++ . What sets Charm++ apart from traditional programming models such as message passing and shared variable programming is that the execution model of Charm++ is message-driven. Therefore, computations in Charm++ are triggered based on arrival of associated messages. These computations in turn can fire off more messages to other (possibly remote) processors that trigger more computations on those processors.

Charm++ is based on a concept of parallel objects called *Chares*, which are similar to a C++ object; however, a *Chare* object may be accessed remotely from other processors. Collection of *Chares* is also available for the programmer's use in the form of *Chare Groups* and *Chare Arrays*. A Group has one *Chare* on each processor in the system; whereas, chare arrays are a collection of arbitary number of chares where each chare has its own index but all share the same global identifier. The distribution of the chares in a chare array among processors is upto the Charm++ runtime system, which provides the notion of *Virtualization*. This notion allows the runtime system to use different load balancing strategies for improved performance without the programmer's conscious effort. Tough individual chares in a group or array still function as an individual parallel unit, they are organized in a collection in order to improve the clarity of applications developed in Charm++ .

Each *Chare* contains a number of *entry methods*, which are methods that can be invoked from remote processors. The Charm++ runtime system has to be explicitly told about these methos, through an *interface* in a seperate file. Charm++ provides system calls to asynchronously create remote chares and to asynchronously invoke entry methods on remote chares by sending messages to those chares. This asynchronous message passing is the basic interprocess communication mechanism in Charm++ . This allows to reduce the processor's idle time by overlapping computation with communication delay. The latency is also minimized by allowing the sending process to continue on its own work rather than waiting for the actual delivery of the message to the remote processor or the execution of the message's corresponding entry method. Charm++ also lets the users associate priorities to the messages, so that high priority messages can be handled before the lower priority ones.

Chapter 3

Converse

Converse [4] is a multi-lingual, interoperable runtime framework. It supports the SPMD programming style, message-driven programming, parallel object-oriented programming and thread-based paradigms. Converse provides portable, efficient implementations of all the functions typically needed by a parallel language library. For example, Converse provides an architecture-independent interface to most thread functions, thread scheduling, sycnchronization of variables, and message passing. This runtime enviroment can run on any of many different operating systems, including Linux, Solaris, Irix, Dec-alpha, Windows NT as well as many specialized super computers like IBM's SP, SGI Origin 2000, etc. Charm++ is built on top of this runtime framework.

3.1 The Scheduler

Each processor in the Converse runtime system runs a scheduler, which is responsible for receiving all the messages and scheduling them. The converse scheduler is based on the notion of schedulable entities, called *Generalized Messages*. A generalized message is an arbitrary block of memory, with the first few bytes specifying a function that will handle the message and the rest containg the user data. The scheduler dispatches a generalized message by invoking its handler with the message pointer as a parameter. The function may be specified by a direct pointer or by an index into a table of functions. Any function that is used for handling messages must first be registered with the converse environment. The scheduler's(Figure 3.1) job is to repeatedly pick and process the messages in the order of their priority. A generalized message may be used as a message sent from a remote processor or as a scheduler entry for a ready thread or object.

There are two kinds of messages in the system waiting to be scheduled–messages that have come from the network, and those that are locally generated. The scheduler's job is to repeatedly deliver these messages to their respective handlers. Since issues related to buffer-management demand timely processing of messages sitting at the network interface, the scheduler first extracts as many messages as it can from the network, calling the handler for each of them. These handlers may enqueue the messages for scheduling (optionally with a priority) if they desire such a functionality. Once there are no more network messages, the scheduler dequeues one message from its priority queue (Figure 3.2) and delivers to the corresponding handler. This process continues until the Converse function to terminate the scheduler is called by the user program. The scheduler's queue is written as a seperate entity so that users can plug-in different queueing strategies. The handler for a particular message may be a user-written function, or a function in the runtime system of the particular language.

In Figure 3.1 the *SCHEDULE_IDLE* gives some time to the Converse Machine Layer to poll the network for any incoming messages. This is called whenever there are no messages to schedule and the scheduler is effectively in idle state. Similarly *CsdPeriodic* takes care of any periodic call-backs on the objects need to be made in the system.

```
void CsdScheduleForever(void)
{
  int isIdle=0;
  SCHEDULE_TOP
  while (1) {
  msg = CsdNextMessage(&state);
    if (msg) { /*A message is available-- process it*/
      if (isIdle) {isIdle=0;CsdEndIdle();}
      SCHEDULE_MESSAGE
    } else { /*No message available-- go (or remain) idle*/
      SCHEDULE_IDLE
    }
    CsdPeriodic();
  }
}
```

Figure 3.1: The main Converse Scheduler (CsdSchduleForever) Implementation

```
void *CsdNextMessage(CsdSchedulerState_t *s) {
   void *msg;
   if (NULL!=(msg=CmiGetNonLocal())
       || NULL!=(msg=CdsFifo_Dequeue(s->localQ))) {
       CpvAccess(cQdState)->mProcessed++;
       return msg;
   }
#if CMK_NODE_QUEUE_AVAILABLE
   if (NULL!=(msg=CmiGetNonLocalNodeQ())) return msg;
   if (!CqsEmpty(s->nodeQ) &&
      !CqsPrioGT(CqsGetPriority(s->nodeQ), CqsGetPriority(s->schedQ))) {
      CmiLock(s->nodeLock);
      CqsDequeue(s->nodeQ,(void **)&msg);
      CmiUnlock(s->nodeLock);
      if (msg!=NULL) return msg;
     }
#endif
   CqsDequeue(s->schedQ,(void **)&msg);
   if (msg!=NULL) return msg;
   return NULL;
}
```

Figure 3.2: CsdNextMessage Implementation

Chapter 4

Schemes for Out of Core Execution Support

In this section we will discuss two software prefetching schemes - *Object Touching* and *Object Management* in detail. In particular we will discuss the feasibility of each scheme, the motivation behind the scheme and provide some theoretical support for it. We also discuss the two schemes using a model of the Converse scheduler.

4.1 Object Touching

The main function of the Converse scheduler is to pick up a message from the Queues, and execute the message on an object. In the *Touching* scheme a *Prefetch Thread* is created in addition to the main *Scheduler Thread* in the Converse scheduler. The *Prefetch Thread* dequeues a message using *CsdNextMessage.* Then it tries to prefetch the object that belongs to the message and enques that message into a temporary Producer-Consumer queue. This queue is mainted between the *Prefetch Thread* and the *Scheduler Thread* to hold the *prefetched-messages.* The main *Scheduler Thread* picks up a message from the above Producer-Consumer Queue and schedules the message to do the required computation on the already prefetched object. Prefetch thread maintains a *leash* such that it will prefetch objects only when the queue size is less than the leash size. This can be used as a programmable parameter to study the effect of different leash sizes on prefetching. The psuedo code for both the threads is shown in Figure 4.1.

```
void* PrefetchThread (void *info)
{
  while (true)  {
   msg = CsdNextMessage();
     if (msg != NULL) {
       if (PCQueueLength(bufQ) >= Leash) {
        sched_yield();
        continue;
       }
     objPtr = getObjectPtr(msg);
     TouchObject(objPtr);
     PCQueuePush(bufQ,msg);
    }
  }
}


void* SchedulerThread (void *info)
{
   while (true)  {
     if (PCQueueEmpty(bufQ) {
       sched_yield();
       continue;
     }
     msg = PCQueuePop(bufQ);
     Schedule_Message(msg);
   }
}
```

Figure 4.1: Pseudo Code for Scheduler with Object Prefetching

The main idea behind the above approach is that as the *Prefetch Thread* accesses the objects corresponding to the messages, any page faults will be taken by it. As the *Computaion Thread/Scheduler Thread* comes to a point to access that object, it is already brought into memory and hence it will not incur a page fault (assuming that object is not paged out by the time it is accessed). Keeping the leash size to some reasonable value such as 10-30, one can statistically assume that pages will not be swapped out between the time *Prefetch Thread* brings it into memory and *Scheduler Thread* access them. Especially, if the page replacement algorithm used in the operating system is LRU (Least Recently Used),it becomes even less likely. When the queue gets filled up to the leash size,*Prefetch Thread* yields to the *Scheduler Thread* and similarly when the queue is empty, scheduler thread yields to prefetch thread.

The Producer-Consumer Queue between the two threads allows some work to be queued up for the *Scheduler Thread* . Whenever the *Prefetch Thread* is blocked in the page fault, the OS (or the thread package scheduler) will schedule other available threads. Meanwhile, *Scheduler Thread* can keep working on the already prefetched objects in memory. If the page fault time and the computation time overlap exactly, the process performs as though there are no page faults, and achieves optimum performance.

4.1.1 Theoretical Discussion

In this section we try to analyze how paging effects the system performance and how the *Object Prefetching* schemes can improve it. Let us make following assumptions about the access pattern of an application which repeatedly accesses some object and performs some computation on it. (This section is taken from [6], for the sake of completeness of this document)

- N = The number of objects
- S = Size of each object in bytes
- M = The available physical memory
- t = Total completion time
- t_c = The computation time per object
- f_p = Page fault frequency
- t_p = Page fault service time.
- $t_o = \frac{t}{N}$, Effective time taken per object.

We use above eight parameters to discuss the different scenarios possible in this scheme. Two major scenarios are one with large computation time with respect to the page fault service time and the other one being the computation time smaller than the page fault service time.

4.1.2 Large Computation Time

In this scenario the computation time per object is larger compared to page fault time, i.e. $t_c > t_p$.

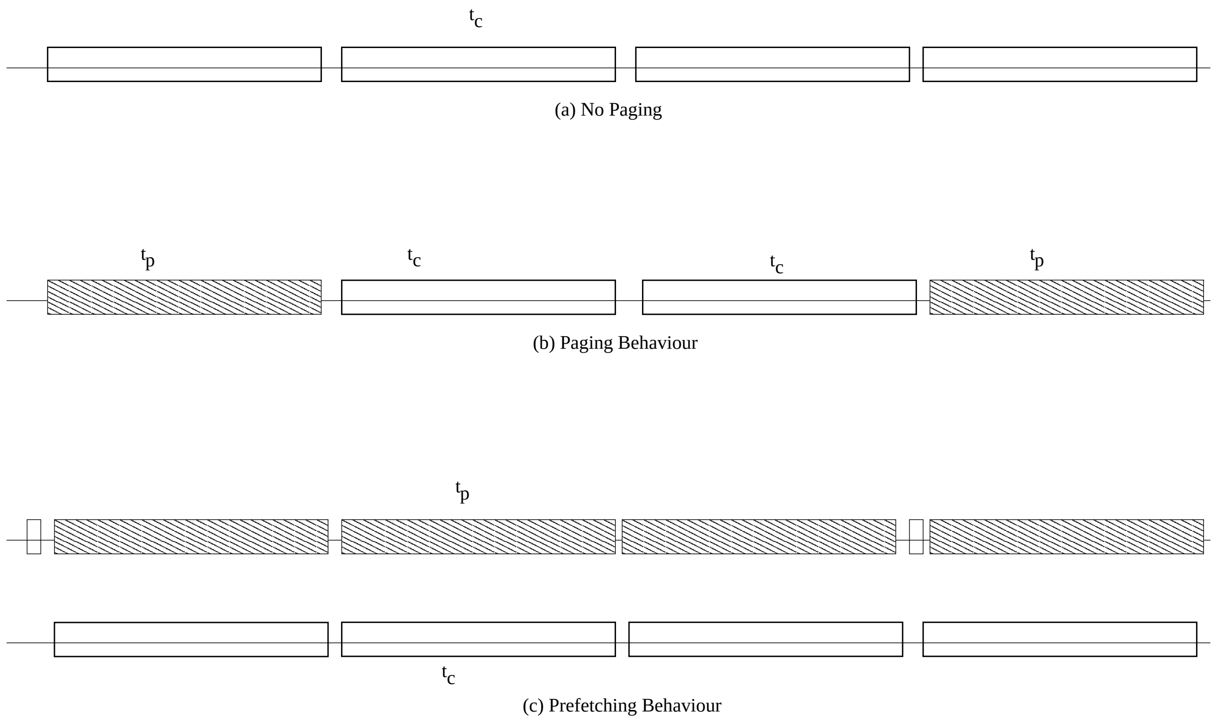

Figure 4.2: Timeline: Computation time higher than the page fault servicing time

In the ideal case when all the necessary data fits into the available physical memory, access time for the data (Object) wil be very small compared to the computation time. When most of the time is spent in computation, the behaviour looks like in Figure 4.2 (a).

The total time taken for the completion of the program can be given by the equation

$$\begin{aligned} t &= N.t_c \\ t_o &= \frac{t}{N} \\ &= t_c \end{aligned} \tag{4.1}$$

But as the memory usage of the application increases, some of the data will not fit in memory and the application might start paging. When a page fault occurs, the access time will increase and becomes significant with respect to the computation time of the object. This behaviour degrades the overall performance of the application. Figure (b) models this kind of paging behaviour of the application.

$$\begin{aligned} t &= N(t_c + f_p.t_p) \\ t_o &= \frac{t}{N} \\ &= t_c + f_p.t_p \end{aligned} \tag{4.2}$$

We can estimate f_p as follows:

$$\begin{aligned} RequiredMemory &= S.N \\ AvailableMemory &= M \end{aligned}$$

When $S.N < M$, there will not be any page faults and every access will be in memory.

$$f_p = 0, \quad S.N < M \tag{4.3}$$

One can define the probability of a page not being in physical memory as,

$$f_p = \frac{S.N - M}{S.N}$$

$$= 1 - \frac{M}{S.N}, \quad S.N > M \tag{4.4}$$

So as the amount of required memory asymptotically increases to infinity, page fault frequency approaches to 1.

$$f_p \rightarrow 1, \quad S.N \rightarrow \infty \tag{4.5}$$

Combining equations 4.2, 4.3, 4.4 and 4.5 we have

$$\begin{aligned} t_o &= t_c, \quad S.N < M \\ &= t_c + (1 - \frac{M}{S.N}) \ . \ t_p, \quad S.N > M \\ &\rightarrow t_c + t_p, \quad S.N \rightarrow \infty \end{aligned} \tag{4.6}$$

From equation 4.6 we can conclude that effective time per object remains constant at t_c when the available memory is sufficient to fit the given problem, but starts increasing with f_p for sometime and then finally asymptotically approaches $t_c + t_p$ for infinetely large problems. The equations can be plotted as shown in the Figure 4.3

As seen in Figure 4.2 (b), computation thread has to wait for the data to be brought into memory. In Figure 4.2 (b), if one can overlap page access times with some actual computational time, peformance can be improved. To achieve that, a new *prefetch thread* is introduced whose only work is to access the data before computation takes place on it. At any point, when the prefetch thread gets blocked on a page fault, *computaion thread* can still continue to perform computation on the data prefetched earlier. If this happens to be as in Figure 4.2 (c), most of the page fault servicing time is overlapped with useful computation time, as computation thread need not wait for the next object access as in Figure 4.2 (b). In essence, in this scenario the computation time t_c domintes the total time taken, thus one

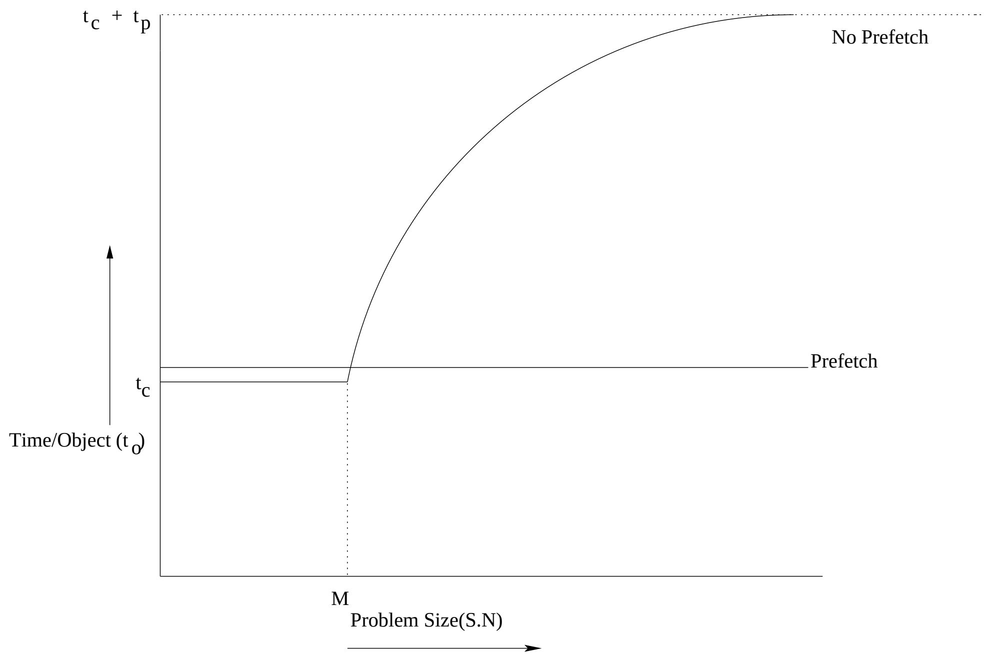

Figure 4.3: Large Computation Time: Time per Object Vs Problem Size

can conclude that

$$t_o \;=\; t_c \tag{4.7}$$

From equations, 4.6 and 4.7, we can conclude that asymptotically, t_o will reach $t_c + t_p$ without prefetching, while with prefetching it remains constant at t_c. Figure 4.3 shows the above analysis pictorially.

In the above figure the slight difference between t_o for prefetching and t_c is due to overheads of thread creation,thread context switches and thread synchronization.

Some of the second-order effects ignored in the above analysis are:

- Even if the average page fault rate remains constant, the prefetch thread can encounter multiple page faults in succession. This can make the computation thread wait for a while.

- A prefetched page can get swapped out by the operating system, before it is accessed by the computation thread.

4.1.3 Small Computation Time

The discussion in the previous section assumed that $t_c > t_p$. This section analyzes the other scenario when the computation time is small compared to page fault time. This section argues that prefetching has advantages even in this scenario.

Figure 4.4 models the application behaviour with small computation time. When there is no prefetching the equation 4.6 still holds in this case.

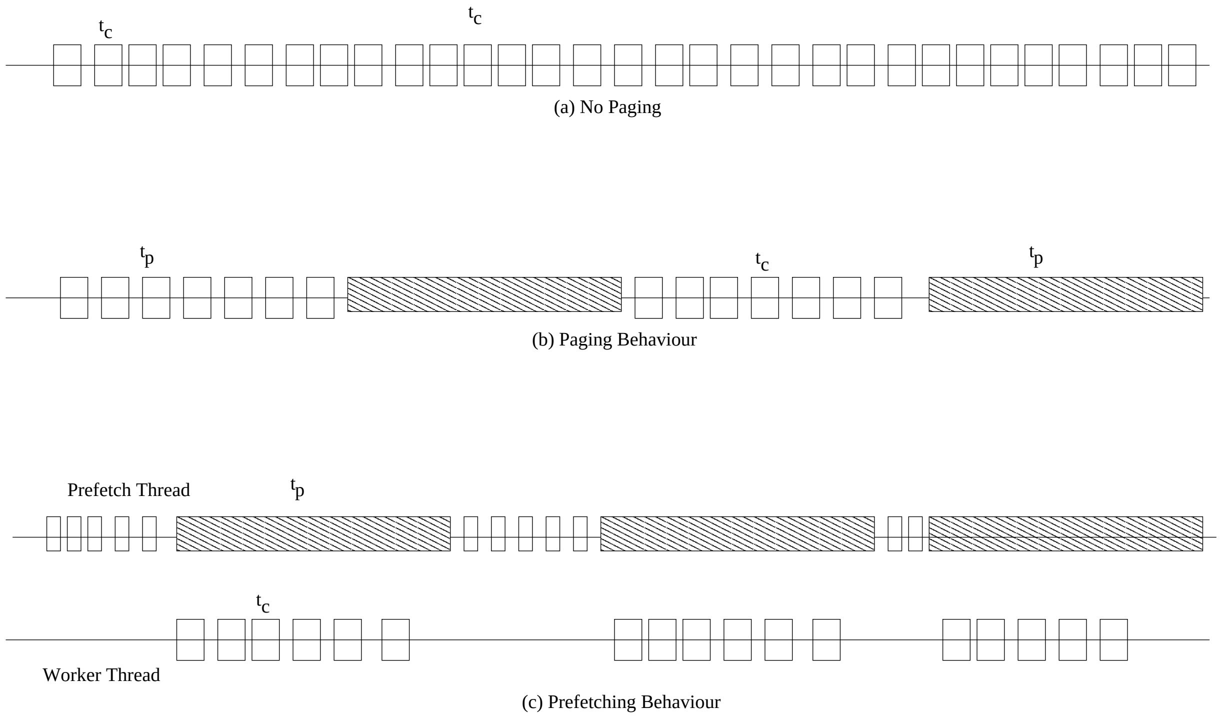

Figure 4.4: Timeline: Computation time smaller than the page fault servicing time

When prefetching is introduced in this scenario, an overlap can be achieved between the page fault service time and the computation time as shown in Figure (c). If $f_p.t_p < t_c$, by the time the computation thread finishes executing K objects in time $K.t_c$, the prefetch thread incurs $f_p.K$ page faults, and taking $K.f_p.t_p$ time units. If we assume that the access time spent by the prefetch thread is negligible and so is the context switching time, the computation thread will never be kept waiting by the prefetch thread. We can summarize as,

$$\begin{aligned} t &= N \,.\, t_c, \quad if\ f_p \,.\, t_p < t_c \\ t_o &= t_c, \qquad if\ f_p \,.\, t_p < t_c \end{aligned} \tag{4.8}$$

The above equation does not account for the second-order effects as in the previous section.

On the other hand, if $f_p.t_p > t_c$, the total time taken will be decided by the page fault time rather than the computational time. For K accesses, the prefetch thread incurs $K.f_p.t_p$ time in servicing the page faults. The computation thread can finish its work in time $K.t_c$, which is less than the all page faults time together,$K.f_p.t_p$. Hence,

$$\begin{aligned} t &= N \,.\, (f_p \,.\, t_p), \quad if\ f_p.t_p > t_c \\ t_o &= (f_p \,.\, t_p), \qquad if\ f_p.t_p > t_c \end{aligned} \tag{4.9}$$

If we rewrite the condition $f_p \,.\, t_p < t_c$ as,

$$\begin{aligned} f_p \,.\, t_p &< t_c \\ (1 - \frac{M}{S.N}) \,.\, t_p &< t_c \\ i.e.\ S \,.\, N &< M(\frac{t_p}{t_p - t_c}) \end{aligned} \tag{4.10}$$

Since we assumed that $t_c < t_p$, the quantity $(\frac{t_p}{t_p - t_c})$ is greater than 1. When f_p approaches 1,

$$\begin{aligned} t_o &\rightarrow t_c + t_p, \quad Without\ Prefetching \\ t_o &\rightarrow t_p, \quad With\ Prefetching \end{aligned} \tag{4.11}$$

Thus, when computation time is high ($t_c \approx t_p$), prefetching makes the program twice as fast as the original program. The expected performace can be plotted as show in Figure 4.5 below.

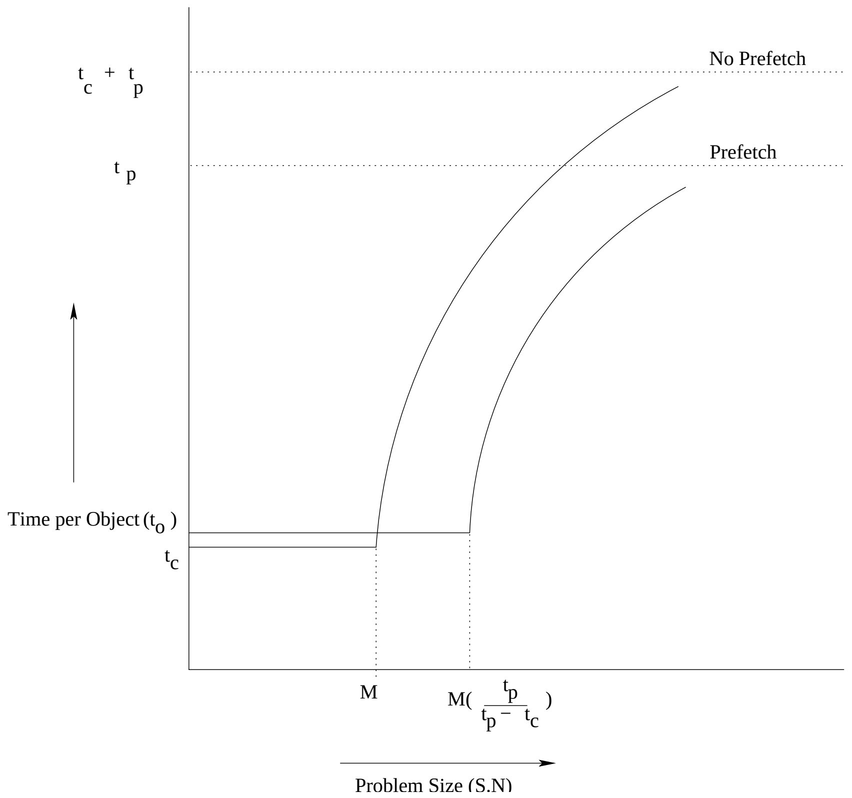

Figure 4.5: Small Computation Time: Time per Object Vs Problem Size

4.1.4 Asymptotic Behaviour

By keeping N constant at a very large value such that $f_p \to 1$, if we vary t_c, the graph shown in Figure 4.6 can be plotted.

Without prefetching, effective time per object is given as $t_o = t_c + t_p$, which is plotted as a straight line in Figure 4.6. With prefetching introduced, t_o depends on the relative values of computation time per object and page fault service time. When $t_c < t_p$, effective time taken is dominated by the page fault service time, $t_o = t_p$, which is independent of t_c. When $t_c > t_p$, t_o is dominated by the computation time, $t_o = t_c$.

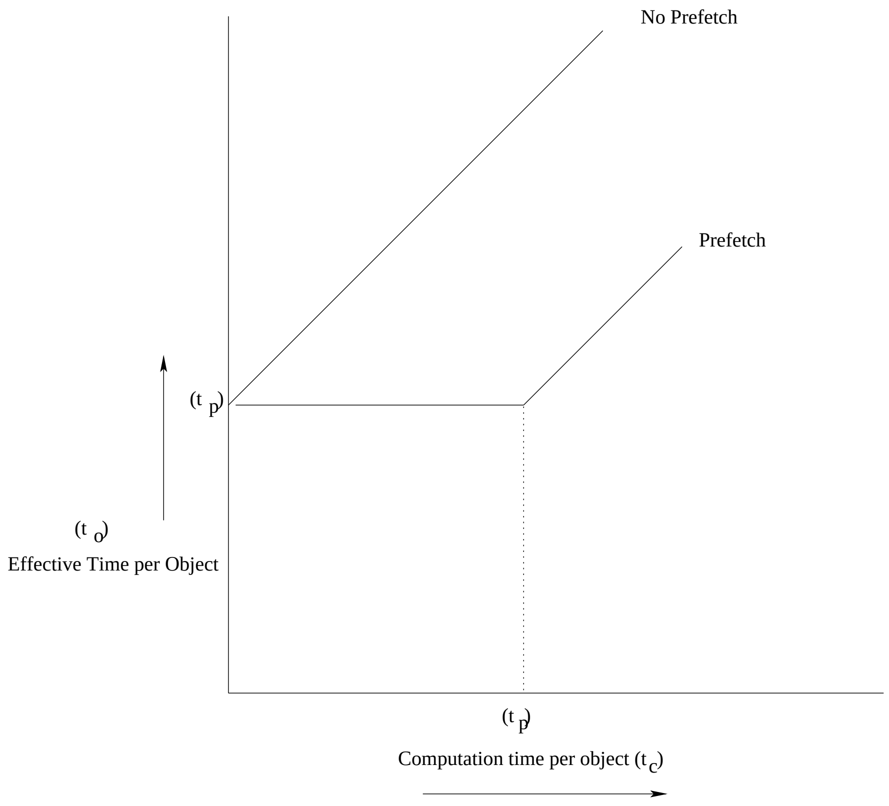

Figure 4.6: Asymptotic analysis for different computation time per object

4.1.5 Different Leash Sizes

Prefetch thread maintains a *leash* so that even when it is blocked on a page fault, the computation thread can continue working. If the leash is 1, computation thread has to wait for every message until the prefetch thread enqueues the message into the Prefetch Queue. As the leash is increased, computation thread will have more work to do when the prefetch thread is waiting on a page fault. Thus, computation thread needs to wait little or no time for work to be enqueued. But at the same time, leash size should not be selected so high that prefetch thread prefetches too much data which might get paged out before it is used, thus causing even more page faults.

4.2 Object Management

As described earlier in section 4.1, *Object Touching* method does not have complete control over which objects will be in memory and which will be paged out since it pretty much relies on the virtual memory for this. Object Management mechanism tries to gain explicit control of the objects. This is one step ahead on the previous prefetching mechanism. Objects are serialized and stored on the disk. To prefetch an object, the object is read, deserialized and brought into memory. Since all these operations take place in the preferch thread, the computational thread is not effected and it does not have to wait for work, by maintaining a leash between the two threads.

Since the available physical memory is limited, whenever a new object is brought into memory another less frequently used object may have to be replaced. This problem is similar to the Operating System's virtual memory management component. Here instead of the pages, we are using replacement algorithms to deal with objects. In effect, the application (or the runtime system on behalf of the application) is taking control of its memory, since at this time there is no mechanism to convey information to the OS about the application's memory access patterns. Any of the OS virtual memory page replacement algorithms can be used to do the object replacement. Whereas the memory accesses to different pages provide hints for the page replacement algorithms, the messages for different objects provide the useful hints for the object replacement algorithms. But since the objects are much bigger than a page, our replacement mechanism works at a much coarser level compared to the OS's virtual memory system.

As shown in the Figure 4.7 the *prefetch thread* differs from the prefetch thread in *object touching* mechanism in the call to *MakesureInMemory* instead of the *TouchObject*. The *MakesureInMemory* method runs the object replacement algoritm, swaps out a object (if necessary) and brings in the required object as needed. The discussion in the Section 4.1.1 is still valid for this scheme, since the time taken by *MakesureInMemory* is similar to *Tou-*

```
void* PrefetchThread (void *info)
{
  while (true)  {
   msg = CsdNextMessage();
     if (msg != NULL) {
       if (PCQueueLength(bufQ) >= Leash) {
        sched_yield();
        continue;
       }
     objPtr = getObjectPtr(msg);
     MakesureInMemory(objPtr);
     PCQueuePush(bufQ,msg);
    }
  }
}


void* SchedulerThread (void *info)
{
while (true)  {
if (PCQueueEmpty(bufQ) {
sched_yield();
continue;
}
msg = PCQueuePop(bufQ);
Schedule_Message(msg);
}
}
```

Figure 4.7: Pseudo Code for Scheduler with Object Management

chObject. So t_p can still represent the page fault service time and hence all the equations and the graphs are still valid.

Chapter 5

The Pack/Unpack Framework

As discussed in the previous section, the *Object Touching* scheme needs a mechanism of touching all the pages belonging to the object. Similarly the *Object Management* mechanism needs, a way of serializing and deserializing the object to and from the disk. For this functionality the *PUP* framework in the Charm++ system is used and it is described at length in this section.

The pack/unpack or "pup" framework [2]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 (checkpointing). The pup framework can be extended to provide services to any operation that requires a traversal of the object state (typically a traversal over the objects data members).

To migrate concurrent objects such as chare array elements the following steps need to be done,

1. The state of the object must be 'packed' into a memory buffer.
2. The memory occupied by the object should be released on the processor where the object resided.
3. The object state should be transported to the new processor where the object is to be migrated.
4. The object should be recreated at the new location.

The state of sequential objects associated with the Chare Arrays is subsumed by the state of the concurrent objects since, in a Charm++ program, every sequential object is a member of or is pointed to by a member of a concurrent object. Chare Arrays may contain dynamically allocated data the size of which varies at runtime. All of this data needs to be packed at the time of migration from the source processor and unpacked at the destination processor.

To checkpoint a Charm++ program, we need to save the state of all the concurrent objects in the system on the disk. We chose to view checkpointing as a variant of migration. When a checkpoint is made, the Charm++ objects are seen as 'migrating' to disk, and upon restart they return to their respective processors. To migrate an object, its data needs to be 'packed', i.e. serialized either into a memory buffer or to disk, and then 'unpacked' into memory.

Migration for objects can be handled in several ways. A possible approach is to require each class to implement **pack** and **unpack** methods. If an object is required to migrate to another processor while the program is executing, the method **pack** is invoked on the object. The pack method allocates a memory buffer large enough to hold all the object's data and then proceeds to serialize the objects data into the memory buffer. The memory buffer is then inserted into a message and sent to the processor where the object is to be migrated. When this message containing the object state is received by the new processor, a new instance of the class is created by calling a special *migration constructor*. The migration constructor's task is to simply create an uninitialized instance of the class. The **unpack** method is invoked with a pointer to the migration message. The unpack method proceeds to stuff the new object with data from the old one. At the end of the unpack method, migration is complete.

The problem with this approach is that most of the functionality in the pack and unpack is similar in nature, i.e., in the pack function, the data is copied to a serial buffer in a particular order and in unpack the data is copied from the serial buffer in the same order as it was packed. Thus there is code duplication as both the methods share a common skeleton, with only the actual operation that the methods perform on the data members being different. The pup library has been designed with the intent of preventing this code dupplication. The programmer of a particular class only needs to implement a single method, called *pup*. The *pup* method takes a single parameter, which is an instance of a *packer/unpacker* or *pupper*. The nature of puppers shall be dealt with subsequently. The role of this method is to perform a traversal of the object state. The actual operations that need to be performed on the data members are executed by the pupper.

The pup library contains the following important classes:

- `class PUP::er` - This class is the abstract superclass of all the other classes in the system. The *pup* method of a particular class takes a reference to a *PUP::er* as parameter. This class has methods for dealing with all the basic C++ data types. All these methods are expressed in terms of a generic pure virtual method. Subclasses only need to provide the generic method.

- `class PUP::packer` - The abstract superclass of all classes that 'pack' objects. It is not clear here what packing means, but it may be considered as any operation that performs a non-destructive transformation on the objects state, i.e. the 'packing' operates on the data that constitutes the object state and creates a different representation of that state. The object does not change as a result of this operation. This class implements additional methods, `isPacking` and `isUnpacking`, that may be used to query the class to determine its mode of operation

- `class PUP::unpacker` - The abstract superclass of all classes that 'unpack' objects. Unpacking is the opposite of packing as described in the previous item. An unpacking operation works with a 'raw' (uninitialized) object and some representation of the object state. The process of unpacking involves a traversal of the object state, at each step of the traversal, part of the object state is 'converted' from the given representation into a piece of memory holding the right bit pattern. When the unpacking is complete, the entire object state has been recovered.

- `class PUP::sizer` - This is a subclass of the `PUP::er` class. Its function is to determine the size (in bytes), of the object that it operates on.

- `class PUP::toMem` - This is a subclass of the `PUP::packer` class. The role of this class is to pack the object it operates on into a preallocated contiguous memory buffer. The most general way to pack an object into a memory buffer is to invoke `pup` on the object using an instance of `PUP::sizer` to determine the size of the object, then a buffer of the required size is allocated and `pup` is invoked again with an instance of `PUP::toMem` that has been initialized with the allocated buffer.

- `class PUP::fromMem` - This is a subclass of the `PUP::unpacker` class. The role of this class is to unpack the state of the object it operates on from a given contiguous memory buffer.

- `class PUP::toDisk` - This is a subclass of the `PUP::packer` class. The role of this class is to save the state of the object it operates on into a disk file. To serialize an object to disk, `pup` is invoked on the object with an instance of `PUP::toDisk` that has been initialized with a file pointer.

- `class PUP::fromDisk` - This is a subclass of the `PUP::unpacker` class. The role of this class is to unpack the state of the object it operates on from a given disk file.

Figure 5.1 shows a class declaration that includes a `pup` method:

The routine in Figure 5.2 presents an example of how an instance of the *foo* class may be packed and unpacked from a memory buffer.

Figure 5.3 shows more complex example of how an instance of the *bar* class may be serialized to memory and then recovered from it. The *bar* class has an instance variable of type *foo*. To pack/unpack or checkpoint/recover an object of type bar we must apply the same operation to the instance variable `foo f`. This is accomplished by having the `pup`

```
class foo {
  private:
    bool isBar;
    int x;
    char y;
    unsigned long z;
    float q[3];
  public:
    void pup(PUP::er &p) {
      p(isBar);
      p(x);
      p(y);
      p(z);
      p(q,3);
    }
};
```

Figure 5.1: A simple class declaration showing the `pup`

method of the bar class invoke `pup` on the member of type foo with the *pupper* passed to it. Figure 5.3 shows the declaration of the *bar* class, including the `pup` method.

In the *Object Touching* scheme, a class (*Prefetcher*) similar to the *PUP:sizer* is used in the *TouchObject* method, to touch all the pages of the object's data members. Similarly in the *Object Management* scheme, to swap out a object *PUP:toDisk* is used and to bring a object into memory *PUP:fromDisk* is used.

```
int main()
{
  //Build a foo
  foo f;
  f.isBar=false;
  f.x=102;
  f.y='y';
  f.z=1234509999;
  f.q[0]=(float)1.2;
  f.q[1]=(float)2.3;
  f.q[2]=(float)3.4;
  //Collapse f into a memory buffer Allocate a buffer for the foo object
  PUP::sizer s;
  f.pup(s);
  void *buf=(void *)malloc( s.size() );
  //Pack f into preallocated buffer
  {
    PUP::toMem m(buf);
    f.pup(m);
  }

  //Unpack the foo object
  foo f2;
  {
    PUP::fromMem m(buf);
    f2.pup(m);
  }
}
```

Figure 5.2: Packing and unpacking a *foo* object

```
class bar {
  public:
    foo f;
    int nArr;//Length of array below
    double *arr;//Heap-allocated array
    bar() { }
    bar(int len) {
      nArr=len;
      arr=new double[nArr];
    }
    void pup(PUP::er &p) {
      f.pup(p);
      p(nArr);
      if (p.isUnpacking())
         arr=new double[nArr];
      p(arr,nArr);
    }
};
```

Figure 5.3: Declaration of the *bar* class showing the `pup` method.

Chapter 6

Implementation

In this section we discuss how the previously explained schemes for providing automatic Out-of-Core execution support are actually implemented in the Charm++ system. First we will discuss how the interface between Charm++ and Converse is extended for Out-of-Core support. Next we discuss the changes made on Charm++ side and Converse side respectively. We will also discuss the limitations of the current implementation.

6.1 Converse - Charm++ Interface

As Charm++ and other languages are implemented on top of the Converse run-time system, Charm++ utilizes a lot of services from Converse . But at the same time, Charm++ provides some services to Converse by registering some handler functions with Converse in the startup phase. Charm++ objects communicate with each other through the use of messages and these messages contain the handler to be executed on the target processor upon the arrival of the message. On the target processor, the Converse scheduler picks up the message from the network or local queue of messages and executes the handler function corresponding to the one in the message. This handler function is implemented in Charm++ runtime system and the execution of the message on the appropriate Object is taken care of by the charm langugae runtime system.

In either of the Out-of-Core schemes, when the Converse scheduler's prefetch thread picks up a message it needs to know to which object the message belongs to, inorder to prefetch the object into memory. But Converse has no knowledge of the language (Charm++) specific objects. But at the same time, only Converse has the scheduler, which can look into the message queue to prefetch the objects. Hence the Charm-Converse interface is extended to accomodate for this.

In the new extended interface, Converse maintains a datastructure with information regarding the Charm++ objects. When the Charm++ langugae runtime system creates the *Chare Array* objects, it also registers them with converse by passing a pointer to the object and size of the object. In return Converse returns an Object Id to Charm++ , which is an index of the object into the Converse data strucutres. Charm++ runtime system stores this objectId index (from now on called *prefetchObjId*) in its own datastructures, the purpose of which will be explained soon. Along with the registration of the object, Converse also provides a way of deregistering the object. This can happen if the object is destroyed or when the Object gets migrated to a different processor. The case of migration will be explained little later in this section. In essence, Converse provides a bunch of new services to charm, for charm to communicate the information about its objects to converse.

One of the services Converse needs from Charm++ is the information about the object that is going to be accessed by a message. Converse looks only at the envelope of the message and it has no idea about the rest of the contents of the message, since the interpretation of this data is upto the language's own runtime system. Converse needs similar services from Charm++ to deal with serializing and deserializing the objects.

Charm++'s *Chare Array* objects are managed by a module called *Array Manager*. To implement the new services in Charm++ , the array manager is extended to act also as a *Prefetch Manager*. The functions implemented by the prefetch manager are summarized below:

- **int (* msg2ObjId) (void *msg)**;

  Returns the Out-of-Core object Id (i.e., prefetchObjId) of the object that the message will access. If the message is a system message and will not access any object, it returns -1. The Prefetch Manager obtains the *prefetchObjId* at the time of registering the corresponding object with Converse .

- **void (* touchObject) (void *objptr);**

  In the *Object Touching* scheme the prefetch thread in the Converse scheduler has to touch all the pages belonging to the object's data. Though Converse has a pointer to the object itself, it has no way of traversing the pages of the object's data members. Hence as explained in section 5, a PUP::er object will be used to traverse the object's data. But since the Converse has no idea about the object's class and the *pup* routine of the object can not be called on *void ** pointer, this service is provided by charm++.

- **void (* readFromSwap) (FILE *swapfile,void *objptr);**

  In the *Object Management* scheme the prefetch thread in the Converse scheduler may have to bring in an object belonging to a message, from the disk. As discussed earlier, the PUP:fromDisk object is used to read in the data related to the object from the disk and recreate the data memebers of the object. For this purpose, the object's *pup* method has to be called with PUP:fromDisk object as an argument. But to call the *pup* method on the object pointer, converse has no idea of the class of the object. This service does this functionality for Converse .

- **void (* writeToSwap) (FILE *swapfile,void *objptr);**

  In the *Object Management* scheme the prefetch thread may have to swap out some object to bring in some other object. Similar to the *readFromSwap* functionality, here the object's *pup* method has to be called with a PUP:toDisk object as a parameter. Hence this method provides similar functionality as *readFromSwap*.

All the above functions are combined into a structure of type *CooPrefetchManager*, definition of which is shown in Figure 6.1.

```
typedef struct _CooPrefetchManager {
    int (* msg2ObjId) (void *msg);
#if CMK_OUT_OF_CORE_TOUCH
    void (* touchObject) (void *objptr);
#endif
#if CMK_OUT_OF_CORE_PTHREADS
    void (* writeToSwap) (FILE *swapfile,void *objptr);
    void (* readFromSwap) (FILE *swapfile,void *objptr);
#endif
} CooPrefetchManager;
```

Figure 6.1: Structure Definition of CooPrefetchManager

This CooPrefetchManager structure, with its function pointers, is registered with Converse along with the *_charmHandlerIdx*, using the function

*void CooRegisterManager(CooPrefetchManager *pf,int handlerIdx);*

provided by Converse . The _charmHandlerIdx is specific to Charm++ . Some other language implemented on top of Converse can register a similar CooPrefetchManager structure with converse under a different hanlder number. Converse maintains a list of handler numbers and their corresponding CooPrefetchManager structures.

As discussed earlier in this section, Converse provides a set of functions for the languages to register and de-register objects with it. These functions are summarized below:

- **int CooRegisterObject(CooPrefetchManager *pf,int objsize,void *objptr);**

  Charm++ calls this method to register a new prefetchable out-of-core object into the converse's prefetchable objects table.

  Returns : The object's new id to Charm++

  Parameters:

  - pf: The new object's prefetch manager. This must previously have been passed to CooRegisterManager.

  - objsize: The new object's (current) memory size, in bytes.

  - objptr: The new object's location. This pointer is only used to pass back to writeToSwap or readFromSwap.

- **void CooDeregisterObject(int objid);**

  Charm++ calls this method to delete the object previously registered with Converse with the object ID, objid. This can happen when the object gets destroyed, or when it migrates away.

  Parameters:

  - objid: The *prefetchObjId* previously obtained when registering this object.

- **void CooSetSize(int objid,int newsize);**

  This method is used to inform Converse the change in the object's size so that it can update its object table entry.

  Parameters:

  - objid: The *prefetchObjId* previously obtained when registering this object.
  - newsize: The new size of the object, in bytes.

- **void CooBringIn(int objid);**

  Normally when the object is swapped out, it is brought into memory only when there is a message for the object. But there can be circumstances inside the charm++ language runtime system, when the object need to be in memory but there are no messages for it. In these cases, the Charm++ can explicitly ask Converse to bring the object into memory by using this function. One scenario is during the load balancing phase, if the object needs to be migrated to a remote processor and the swap file is on the local disk only, then before transferring the object, the object's data from the local disk need to be retrieved. In this case, the load balancer module of Charm++ can use this routine to direct converse to bring the object into memory, before migrating away the object. Another potential scenario is the case of *CcdCallBacks*. For these call backs to work the object need to be in memory, but since these are just local call backs but not messages which can be observed by Converse , unless the charm++ language runtime system explicitly asks for it, the object will not be brought into memory.

Parameters:

- objid: The *prefetchObjId* previously obtained when registering this object.

Thus the Converse-Charm++ interface is exteneded in both directions to support automatic Out-of-Core support. Any language other than Charm++ implemented on Converse should be able to implement a similar interface.

6.2 Charm++ Side Implementation

This section will describes how the new Out-of-Core interface on the Charm++ side is implemented. To implement the true Out-of-Core support, when a object needs to be swapped out, the whole object's memory (including stack space and heap space) need to be deallocated and simialarly when the object is brought in, a new object has to be created with the old object's data. As described in section 5, the heap allocated memory by the object can be taken care of through the object's own *pup* virtual method. But to deallocate the stack memory of the object's data the whole object needs to be deleted. In order to recalim all the memory occupied by a object, the object needs to be deleted using *delete [] objPtr*. In the current charm runtime system, the objects of a *Chare Array* are maintained by *Array Manager* module, which deals with creating the objects and passing the messages to the appropriate Array Object. The *Location Manager* component assists the array manager, in identifying the location of each object across the set of processors available to the system. If an object gets deallocated and reallocated for the purpose of Out-of-Core support, the array manager and location manager need to be modified significantly. To keep the number of changes to these complex modules minimal, we do not deallocate the object to recalim the memory occupied by the object. We only recalim the memory occupied by the object's heap allocated data. This will greatly simplify the task of the array manager for Out-of-Core support. Since in many of the target applications, the memory occupied by the object's heap allocated data is significantly higher than the memory occupied by the rest of the object's contents, for the purpose of this thesis, the results presented here does reflect the original idea of Out-of-Core support.

With the above limitation, we proceed to explain the modifications done in charm++ language runtime system. The *Object Touching* and *Object Management* schemes are distinguished in the code base using

#if CMK_OUT_OF_CORE_TOUCH and #if CMK_OUT_OF_CORE_PTHREADS .

The *_CkMigratable_prefetchInit* function (shown in Figure 6.2) is used to initialize the CooPrefetchManager structure and register it with Converse along with the charm handler (_charmHandlerIdx). The purpose of *CkSaveRestorePrefetch* will be explained soon.

```
#if CMK_OUT_OF_CORE_TOUCH
static void _CkMigratable_prefetchInit(void)
{
   CkSaveRestorePrefetch=0;
   CkArrayElementPrefetcher.msg2ObjId=CkArrayPrefetch_msg2ObjId;
   CkArrayElementPrefetcher.touchObject=CkArrayPrefetch_touchObject;
   CooRegisterManager(&CkArrayElementPrefetcher, _charmHandlerIdx);
}
#elif
static void _CkMigratable_prefetchInit(void)
{
   CkSaveRestorePrefetch=0;
   CkArrayElementPrefetcher.msg2ObjId=CkArrayPrefetch_msg2ObjId;
   CkArrayElementPrefetcher.writeToSwap=CkArrayPrefetch_writeToSwap;
   CkArrayElementPrefetcher.readFromSwap=CkArrayPrefetch_readFromSwap;
   CooRegisterManager(&CkArrayElementPrefetcher, _charmHandlerIdx);
}
#endif
```

Figure 6.2: Code for Prefetch Manager Inititialization in Charm++

The implementation details of the services provided by the *Prefetch Manager* to Converse are explained below:

- **TouchObject**

  To touch all the pages of memory occupied by the data members of the object pointed by the object pointer (*objptr*), this routine uses a PUP::er object. The class definition of this PUP::er is given in the Figure 6.4. This class overloads the *bytes* virtual method to read one byte per page for each data member of the object. So when the user's *Chare Array* object's *pup* method is called with this new PUP::er as parameter, each data member of the object is passed to the *bytes* method of *prefetcher*. The *bytes* method accesses a byte per page to force page faults on those pages (if any required). In Figure 6.3, the object pointer (objptr) is casted to *CkMigratable ** to call the *pup* routine on it. This is needed because the charm++ language runtime system is not aware of the user's *Chare Array* class. Since the class *CkMigratable* is one of the parent classes for any of the user's chare array class, the virtual nature of the pup method is exploited here to call the pup method on the object pointer. Figure 6.5 shows this class hierarchy diagram.

```
void CkArrayPrefetch_touchObject(void *objptr)
{
   CkMigratable *elt = (CkMigratable *)objptr;
   Prefetcher p;
   elt->pup(p);
}
```

Figure 6.3: Implementation of *touchObject* service in the Prefetch Manager

- **WriteToSwap**

  As shown in figure 6.6, this method creates a PUP::toDisk object with the given Swapfile as the argument and passes the PUP::er object to the object's pup method. Once the object's data is saved to disk, the object's destructor is explicitly called to reclaim the memory occupied by the data memebers of the object. Since the destructor is vir-

```
class Prefetcher : public PUP::er {
  protected:
     int sum; //Used only to trick compiler into actually doing the read...

     virtual void bytes(void *p,int n,size_t itemSize,PUP::dataType t) {
       int nBytes=itemSize*n;
       // Prefetch: read one byte per page.
       char *cp=(char *)p;
       for (int i=0;i<nBytes;i+=4096) {
          sum+=cp[i];
       }
     }
  public:
     Prefetcher() :PUP::er(IS_SIZING) {}
};
```

Figure 6.4: Implementatiion of the *Prefetcher* PUP::er

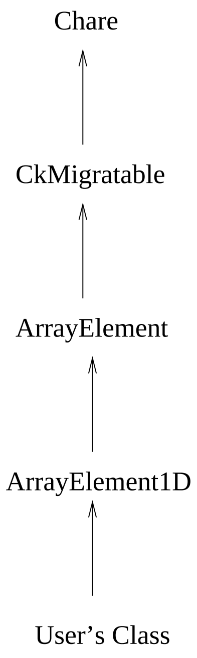

Figure 6.5: Class Hierarchy of the user's Chare Array class

tual in the class hierarchy, in order not to destroy other data in the object the global flag *CkSaveRestorePrefetch* is set before calling the destructor. The destructors other than the user's destructor, observe this global flag being set and immediately return without destroying any data in the object. This ensures that only the user's data in the object is destroyed without deleting any other data of the object required by the language runtime system.

```
void CkArrayPrefetch_writeToSwap(FILE *swapfile,void *objptr)
{
   CkMigratable *elt=(CkMigratable *)objptr;

   //Save the element's data to disk:
   PUP::toDisk p(swapfile);
   elt->pup(p);

   //Call the element's destructor in-place (so pointer doesn't change)
   CkSaveRestorePrefetch=1;
         //because destuctor is virtual, destroys user class too.
   elt->~CkMigratable();
   CkSaveRestorePrefetch=0;
}
```

Figure 6.6: Implementation of *writeToSwap* service in the Prefetch Manager

- **ReadFromSwap**

  As shown in figure 6.7, this method first calls the element's migration constructor to create a new uninitialized instance of the class. Then the elements's *pup* routine is called with the PUP::fromDisk object as a parameter, to retrieve the object's data from disk.

6.3 Converse Side Implementation

This section explains the modifications made to the Converse scheduler, the memory manager and the communication module. We also explain the nitty gritty details of synchronization of data structures, multi-thread safety of the library calls and the signal handlers in the context of converse's scheduler and the machine layer.

```
void CkArrayPrefetch_readFromSwap(FILE *swapfile,void *objptr) i
{
   CkMigratable *elt=(CkMigratable *)objptr;
   //Call the element's migration constructor in-place
   CkSaveRestorePrefetch=1;
   int ctorIdx=_chareTable[elt->thisChareType]->migCtor;
   elt->myRec->invokeEntry(elt,(CkMigrateMessage *)0,ctorIdx,CmiTrue);
   CkSaveRestorePrefetch=0;

   //Restore the element's data from disk:
   PUP::fromDisk p(swapfile);
   elt->pup(p);
}
```

Figure 6.7: Implementation of *readFromSwap* service in the Prefetch Manager

6.3.1 Converse Scheduler

Converse's scheduler is the function *CsdScheduleForever*, which continously picks messages and schedules them. These scheduled messages can in turn create new messages. As explained in section 3.1, there are two kinds of messages - network messages and local messages. A Producer-Consumer queue is maintained for the network messages. When the messages arrive out of network at the interface card, the messages will be enqueued and the scheduler deques to pick up a message. Similarly, for the local messages another producer-consumer queue is maintained. In addition to the above two queues, there is a priority queue with messages for charm. When a network or local message's handler function is executed, the charm handler strips off some part of the message and enques it back into this priority queue based on the priority of the message. The two producer-consumer queues are thread safe without obtaining a big lock around the whole data structure. But the priority queue is not multi-thread safe, if two different threads do enqueing and dequeing. To avoid the performance penalty of grabbing a big lock around the whole priority queue, we change our scheduler's design slightly from that of Figures 4.1 and 4.7.

Our initial design of the scheduler with Out-of-Core support is shown in Figure 6.8. In the modified design (as shown in tFigure 6.9), the scheduler thread picks a message from

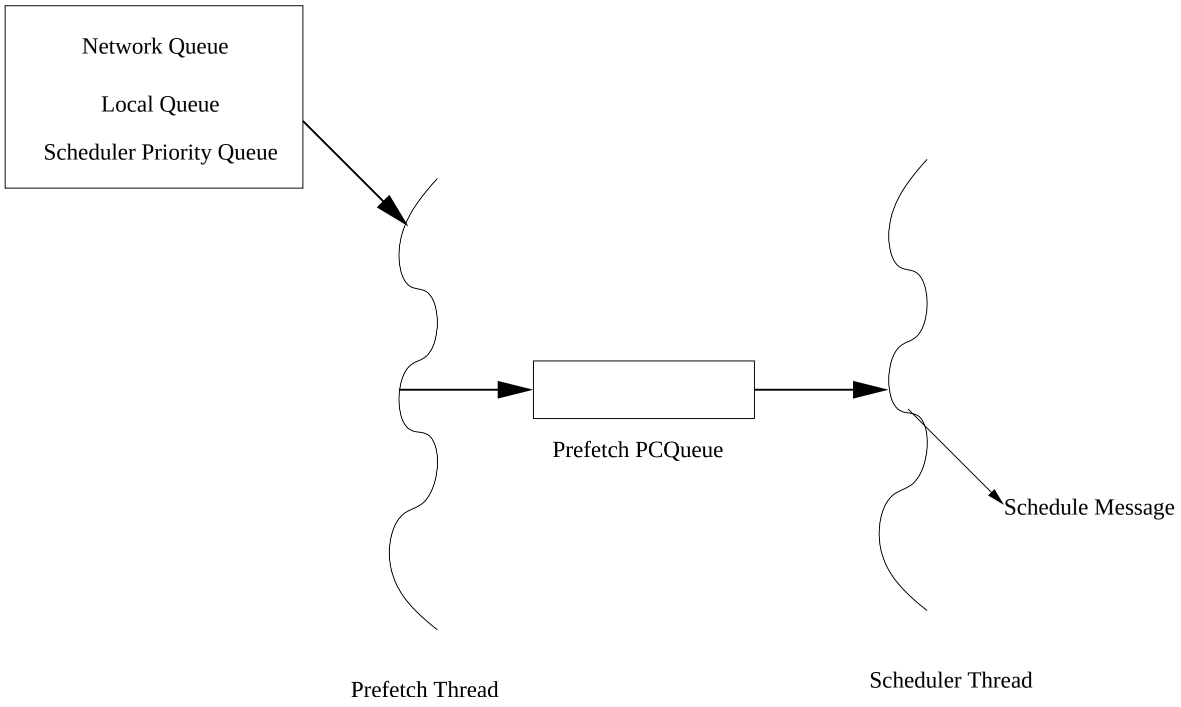

Figure 6.8: Initial Design of the Scheduler

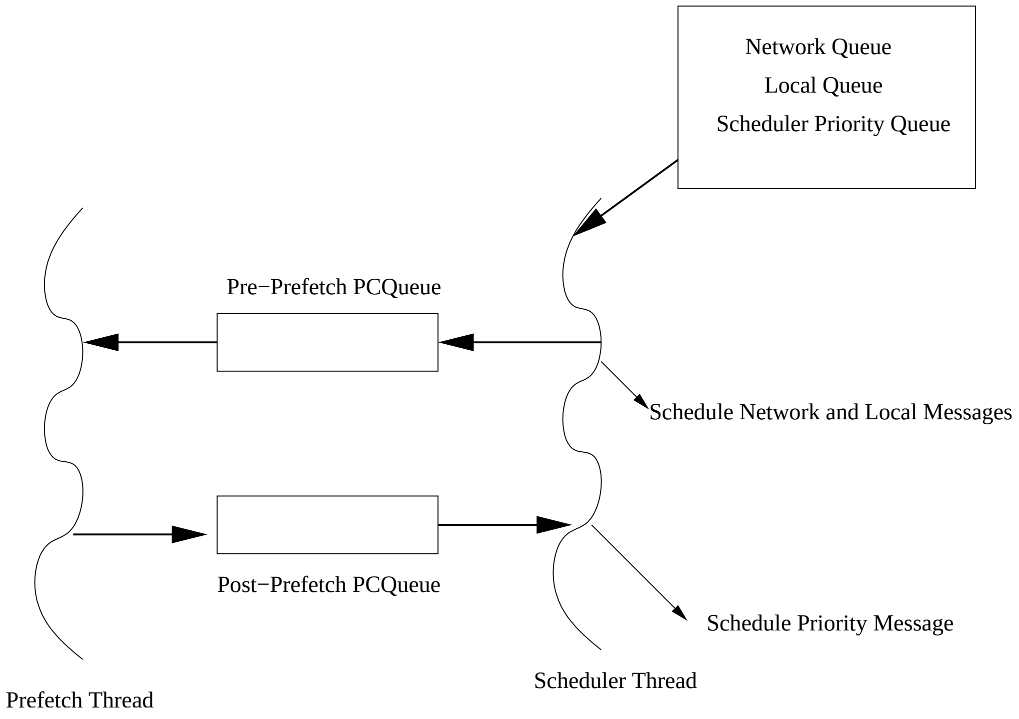

Figure 6.9: Revised Design of the Scheduler for Implementation

any of the three queues, and if it is a network message or a local message it schedules the message immediately. This is because there is no prefetching necessary, as these messages do not access any objects. But if the message is from the priority queue, since the message will access an object it is enqueued into a Pre-Prefetch Queue. When the Prefetch thread dequeues a message from Pre-Prefetch Queue, it performs the prefetching mechanism and then enqueus the message into another Producer-Consumer queue, Post-PrefetchQueue. Then the scheduler thread picks the message from Post-PrefetchQueue and schedules it.

The Pre-Prefetch queue and the Post-Prefetch queue are of size equal to the LEASH size. The scheduler thread has to wait if the Pre-Prefetch queue is full and the prefetch thread will signal the scheduler thread when some slots in the queue becomes empty, through the use of *pthread_cond_wait* and *pthread_cond_signal*. Similarly the scheduler thread could have waited if the Post-Prefetch queue is empty. But if there were no messages in the system (and hecne with the Prefetch thread) the whole system will block forever. Ideally, in this scenario, whenever a message is available in one of the network,local or priority queues the sleeping scheduler therad should be woken up. In case of the *net-linux* version where the network I/O takes place through the SIGIO signal handler, the signal can not wake the thread sleeping on a conditional variable. Hence we need a third thread which can poll the three queues and can signal the scheduler, which essentially sleeps on the emptiness of the both set of queues. For simplicitly of the overall system, we instead opted to use *pthread_cond_timedwait* to wait when the Post-Prefetch queue is empty. In essenece, it becomes a periodic polling of the network,local queues in the scheduler thread itself. In case of the *smp* version, the communication thread can do the polling of the network, local queues and do the signaling to the scheduler thread. Similarly the prefetch thread waits on the conditions of Pre-Prefetch queue being empty and the Post-Prefetch queue being full.

6.3.2 Memory Management

Since the scheduler now has two threads and the malloc/free library calls are not multi-thread safe, we needed to lock all the malloc,free,realloc,valloc,calloc and memalign calls. All the calls to these functions are trapped in the Charm++ library and they are appropriately locked and unlocked with *pthread_mutex_lock* and *pthread_mutex_unlock*. In Charm++'s *net-linnux* version the SIGIO signal handler is used to recieve messages. To avoid deadlock in this version due to locking malloc, at the very beginning of the signal handler we test if the *memory lock* is locked or not using *pthread_mutex_trylock* and if it is locked return immediatelty doing nothing. But if we use the *net-poll* option for *net-linux* version, where the network is polled instead of using a signal handler, we dont have a similar problem.

6.4 Limitations

In this sectioin we discuss the limitations of the current implementation.

- Broadcast Messages

  In the current Out-of-Core system broadcast messages do not work. This is because the converse scheduler relies on the $msg \rightarrow objID$ mapping to manage the objects, but the broadcast messages does not belong to any one object. Hence the object manager can not bring the objects into memory at the righ time for the broadcast messages to be executed on all the objects and thus the system crashes. One way to rectify the situation would be for the msg2ObjId to return the set of objects belonging to a message. But then the Converse schduler has to figure out some way of either bringing all the objects into memory (if memory limits permits) or generate a message for each of the objects and execute the correpsonding message on each object one after another.

- Immediate Messages

  Since the *Immediate Messages* do not go through the converse scheduler and can not be scheduled (and hence the objects can not be brough into memory at the right time), this mechanism of communication with remote objects is not supported in the current Out-of-Core system.

- Load Balancing

  In the present system automatic dynamic load balancing does not work. When the load balancer decides to transfer an object to another processor, the particular object may already been swapped out and in that case the object has to be brought into memory immediately and the contents need to transferred. Due to time constraints this is not yet implemented in the current system.

Chapter 7

Performance Evaluation

The performance of the *Automatic Out-of-Core Execution Support* schemes have been measured with respect to a Charm++ one dimensional Jacobi program. Comparisons were made between native Charm++, Object Touching and Object Management schemes. The Jacobi program was chosen as a benchmark, because each Jacobi1D object contains huge amount of dynamically allocated memory which is ideal for the Out-of-Core scenario. The Jacobi program is memory-intensive and less compute-bound.

We obtain plots for effective time per object (t_o) vs number of objects (N), by keeping the size of each object as constant. The number of columns of the matrix is kept constant. And also the number of rows and the number of chare array objects is chosen equal to make the size of each object constant. By varying the number of rows of the matrix (and hence the number of objects), the Jacobi program is run until the available physical memory is exhausted. The benchmark is also run when the required memory is beyond the physical memory by increasing the number of objects. The same Jacobi program is run under the three different version of charm system - Native, Object-Touching, Object-Management. The only requirements on the applications are to implement the *pup* routine and to implement a destructor to free up the heap-allocated data.

Chapter 8

Summary and Future Work

For applications with large memory requirements, reducing or hiding paging costs is crucial. We proposed two multi-threaded approaches that access objects that are needed in near future, while computation thread is working on the objects available in memory. Data-driven object paradigms, such as CHARM++, facilitates this approach easily, since they can *predict* the objects needed in future. The benchmark Jacobi application demonstrates the results of the prefetching schemes and confirm that prefetching improves the performance of an application by overlapping paging and computation time.

In the *Object Management* scheme, to avoid the overheads of thread creation, context switching and heap contention, we plan to experiment with an approach using asynchronous I/O mechanisms. When an object need to be written to a disk or read from the disk, asynchrnous write,read calls can be used so that these asynchronous operations can be overlapped with the computation. For this approache a new PUP::er with asynchronous I/O calls need to be implemented.

The dynamic load balancer module need to be modified and made aware of the Out-of-Core module. Careful design is needed for the interaction of these two modules. Other optimizations possible are *Reordering Method Invocations.* Since the converse scheduler is aware of which objects are in memory, it can reorder the messages to minimize paging cost. To implement this, we may need a seperate queue of messages for each object to appropriately queue messages for each object until the object is brought into memory. Once the object read into memory, we can execute all the pending messages at once. This scheme improves cache performs also, since all the related memory accesses are clustered. Extending the above idea, we can integrate *Structred Dagger* [3] with the converse scheduler and the array manager. If an object is waiting for bunch of messages (similar to the jacobi object waiting for two messages before doing any computation) before the object's data is needed, then we can exploit this nature to bring in the object only after all the required messages have arrived and execute them all in one go.

References

[1] Lewis Bill and Berg Daniel. *Multithreaded programming with PTHREADS*. Prentice Hall, 1998.

[2] PPL Research Group. Charm++ language manual. See `http://charm.cs.uiuc.edu/ppl_manuals/html/charm++/manual.html`.

[3] L. V. Kale and Milind Bhandarkar. Structured Dagger: A Coordination Language for Message-Driven Programming. In *Proceedings of Second International Euro-Par Conference*, volume 1123-1124 of *Lecture Notes in Computer Science*, pages 646–653, September 1996.

[4] L.V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.

[5] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[6] Neelam Saboo and L. V. Kalé. Improving paging performace with object prefetching. Technical Report 01-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, July 2001.