Run-time Support for Controlling Communication-Induced Memory Fluctuation

Yan Shi, Gengbin Zheng and Laxmikant V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign
{yanshi, gzheng, kale}@cs.uiuc.edu

Abstract

Many parallel applications require a large volume of transient memory to hold data from communication, therefore demonstrating a pattern of communication-induced memory usage fluctuation. Even though these applications' persistent working data might fit in physical memory, the transient peak memory usage could still lead to disk swapping or even out-of-memory error. In this paper, we present a solution to the above problems by runtime support for controlling the communication-induced memory fluctuation. The idea consists of imposing runtime flow control for large data transfers and thus controlling the peak transient memory consumed by communication. We explore the idea with both send-based and fetch-based low level communication primitives. We develop a runtime support based on the Charm++ integrated runtime environment. We test this runtime system with a set of real applications and show considerable performance improvements.

1 Introduction

A large number of parallel applications exhibit a pattern of fluctuating memory usage at runtime. Many of these patterns are established when parallel objects grasp data from others, do computation with the data and ultimately throw the data away. Frequently, the amount of transient memory is proportional to the program's static memory consumption or larger. This wavy pattern is substantially undesirable for several reasons. One is that large memory fluctuation might drive the program into disk swapping zone where performance will be miserable due to the severe overhead of disk swapping. Another is that a large memory footprint might potentially bring poor cache performance. Further, applications might fail to run as a result of insufficient swapping space. An extreme architecture case is the IBM BlueGene/L machine, where there's no virtual memory with only 512MB of physical memory available.

A vital observation here is that many of these transient memory variations are associated with data communications between parallel entities. In Section 3, we will better illustrate the idea with an example of 7-point stencil with 3D decomposition. The same pattern is also shown in a broad range of both structured and unstructured meshed applications. Various commonly used parallel libraries, such as matrix multiplication exhibit similar behaviors. Clearly, these stated applications could benefit from a runtime system that controls the transient memory and reduces the memory fluctuation. The same runtime system should require minimal user code modification and incur negligible overhead in the normal case while improving performance when the memory fluctuation is high.

In search of relevant work, we find that the stated problems have rarely been addressed directly. Many memory related studies focus on optimizing memory hierarchy based on locality. Other works try to solve the memory problem with faster swapping mechanisms.

In this paper, we present an approach where we try to confront the communication-induced memory problem head-on. Given knowledge at the runtime level of the communication-induced memory pattern, we could limit transient memory usage from communication by controlling large data transfers. Various flow control strategies could be applied in the runtime system to facilitate the selection process. We study the applicability of integrating this idea with both send-based and fetch-based communication. A runtime support for this approach is implemented in Charm++ [11] and AMPI [10], an integrated parallel runtime system. Throughout the paper, we demonstrate by drawing examples from a set of scientific applications, where the communication-induced memory fluctuation pattern persists. We believe the generality of our approach is maintained and it could be applied to various parallel systems where a high degree of concurrency is present.

The rest of the paper is organized as follows: Section 2 discusses the background of our work and its related work. Section 3 presents our methodology implemented in a runtime system to handle communication-induced memory fluctuation. Section 4 describes the performance with case studies of several real-world applications. Section 5 concludes with some future plans.

2 Background and Related Work

2.1 Related Work

Explicit Memory Control

Most work in explicit memory control aims at improving application performance by exploiting the memory hierarchy through better memory management using an educated policy for caching data in faster memory [7, 4]. For large data applications, out-of-core [6, 19, 18] methods are designed to overcome the memory capacity limitation. These approaches typically block data sets and use DRAM as a cache for slow bulk media such as hard drive or tape drives. The performance gain largely stems from applying application specific knowledge and replacing the operating system in the role of manipulating data swapping. By keeping the real working data set in-core, thrashing hopefully will be avoided.

Another relevant work is on resource-constrained sandboxing [17, 3], where irrevocable restrictions exists on resource usage, such as memory. It is primarily in the context of real-time systems where fair sharing and no-starvation guarantees are required. Relying largely on kernel support, monitoring resources, code instrumentation and system call interception, resources are enforced in a qualitative way.

Although some of the works above, such as the out-of-core method, explicitly control the application memory footprint, our work addresses the memory-constrained problem from a different perspective. We focus on controlling transient memory fluctuation caused by communication to reduce the memory footprint to fall within the bounds of system availability. In fact, our work can be used as a complement to out-of-core methods to better solve the memory problem. Our work leverages some of the techniques listed above, such as memory monitoring, code instrumentation and system call interception.

Communication Flow Control

In our work, we use the token-based communication flow control, which by itself is not a new idea. The Myrinet GM communication library [16] provides a simple communication flow control via regulating send and receive tokens, representing space allocated to the client in various internal GM queues. A client program may send or receive a message only when it possesses a send or receive token for a myrinet port. However, this mechanism does not provide an effective flow control for eager messages. In the MPICH-GM implementation, eager messages are received as unexpected messages. A fast unexpected sender can easily flood a slow receiver. MPICH-GM therefore implements a rudimentary but somewhat effective throttling mechanism to choke the sender if the unexpected queue is getting big.

ChaMPIon/Pro [15] MPI runtime enforced flow control by imposing some reasonable resource limit, such as, message buffer size to user processes. When the message buffers for un-

expected messages run out, the runtime simply aborted the program and showed to users that there is a resource issue with their application. This usually indicates possible load imbalance of the job because normally a process would get a large number of unexpected messages only when it falls behind the rest. This implementation, however, was not appreciated by users because aborting on a semantically correct MPI program is not desirable.

Communication flow control is effective in controlling the message buffer size used for unexpected messages between a pair of communicating processors. It, however, can not solve the memory fluctuation problem caused by communication. In a parallel application, a process tends to communicate with multiple processors, therefore a per link communication flow control is not sufficient in controlling the total buffer usage in a process. Furthermore, such low level communication flow control does not react to memory usage fluctuation caused by applications.

Safe MPI Program

MPI literature [13] calls a program safe if it can be executed to completion regardless of memory limitations. Non-blocking calls relaxes pressure on memory compared to blocking calls. The k-safe notion [2] relaxes the requirement of the safe program to being safe in an environment with k system buffers available per processor. Our approach of having flow control on large data communications also raises the question of being safe and deadlock free. We discuss these questions in Section 3.4 and argue that under certain assumptions, the program could be guaranteed to avoid deadlocks.

2.2 Parallel Run-time

Controlling communication-induced memory fluctuation often requires flow control of communication. Such a scheme may result in degraded performance due to delays in communication. To alleviate such a performance problem, it is essential for a runtime system to provide dynamic overlapping of computation and communication through a high degree of concurrency to hide the increased communication latency.

The Charm++ and AMPI runtime systems, which our work is based on, provide such techniques. The Charm++ runtime system employs an approach called *processor virtualization* [11, 12]. An application divides a problem into a large number of parallel entities (N), each a virtual processor, that will execute on P physical processors. N is independent of P, and typically $N>>P$ so that there are multiple virtual processors on each physical processor for high concurrency. The user's view of the program consists of these parallel entities and their interactions; the user need not be concerned with how the components map to processors. The underlying run-time system takes care of this (see Figure 1).

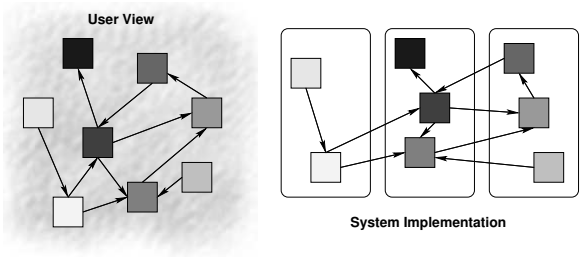


Figure 1. Virtualization in Charm++

In Charm++, these parallel entities are encapsulated in *chares*. Chares are C++ objects with methods that may be invoked asynchronously from other chares. Adaptive MPI (AMPI) [10, 9] is an adaptive implementation and extension of MPI built on top of the Charm++ run-time system. AMPI implements virtualized MPI processes using light-weight migratable user-level threads (analogous to Chares), several of which may be mapped to one physical processor.

Since many parallel entities (chares or threads) can be mapped to a single processor, Charm++ uses *message-driven execution* to determine which chare or thread executes at a given time. If one Chare is blocked on a receive, another Chare on the same physical processor can run if there is an incoming message for it. This provides adaptive overlapping of communication and computation which largely eliminates the need for the programmer to manually specify

some static computation/communication overlapping.

We have also demonstrated that virtualization has minimal performance penalty [12], due to the low scheduling overheads of Chares and user-level threads. In fact, Charm++ and AMPI runtime systems promote better cache performance, which leads to improved performance. A virtual processor handles a smaller set of data than a physical processor, so a virtual processor will have better memory locality. This blocking effect is the same method many *serial* cache optimizations employ, and Charm++ and AMPI programs get this benefit automatically.

In typical Charm++ and AMPI applications with fine grained computation and high degree of concurrency, there are multiple objects or threads running on one processor. These objects or threads tend to act independently regardless of the memory constraint on a node. As a result, a bursty communication pattern may occur which leads to significant amount of transient memory usage for sending and receiving messages in a short period of time. This may either push the application into the swap zone with dramatically degraded performance, or even cause it to run out of memory. In the next section, we will present our effort in making the runtime system memory aware to control such bursty memory fluctuation caused by communication.

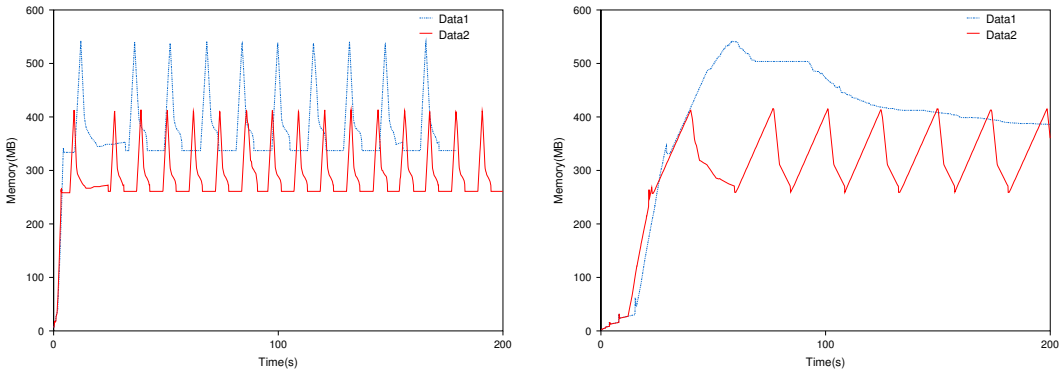
3 Design and Implementation

It is often found in many scientific applications, that cross processor communication, including collective communication, may lead to significant memory problem, MPI_Alltoall is such an example (Section 3.5). Furthermore, as the number of parallel entities that participate the communication increases, memory usage may arise nonlinearly. After communication finishes, the memory usage returns to normal. This paper focuses on such transient memory usage problems caused by communication. A concrete example application is given next.

3.1 A Motivating Example

Let's take a 7-point Jacobi relaxation program as a better illustration. In 3D Jacobi problem, data of a regular rectangular cube are partitioned into equal sized small cubes and distributed evenly over all processors. In every iteration, data in each small cube is updated locally with its own original data and data from its neighbors. With a 7-point centered scheme, each small cube depends on 1 adjacent slab with 1 width from all of its 6 neighbors. These data are usually stored locally and are called ghost cells. In our implementation, for purposes of memory efficiency, we allocate data for ghost cells on the fly, construct them as they come and after computation, free the stale ghost data. In this example, we use two data sets. For *Data1*, the 3D cube is of size $2048 * 512 * 512$; while for *Data2*, $2048*512*384$. Both data sets are partitioned into $8*8*8$ small cubes. Each cube would need six ghost cells of $8 * 8$ rectangles, one from each neighbor. These ghost cell communications comprise the primary communication cost in this program. Figure 2(a) exhibits the iterative wavy pattern of the memory usage over a sample run with these two different data sets. As expected, *Data1* takes about 40% more time per iteration. Figure 2(b) shows the same run but on a different cluster, where *Data1* runs into disk swapping and got slowed down substantially. While *Data2* completes almost 7 iterations in 200 seconds, *Data1* hardly finishes its first!

The amplitude of this memory fluctuation will be multiplied if the ghost cell region widens or the data decomposition grows finer. The former could be result of particular numerical algorithm([5]), and the latter could result from the processor virtualization idea we discussed in section 2.2. In this example in figure 2(b), if the amplitude of the fluctuation could be reduced and controlled within the bounds of physical memory, disk swapping could be effectively reduced or avoided and performance would be greatly improved.



(a) Running on machines with 1GB Memory

(b) Running on machines with 512MB Memory

Figure 2. seven-point 3D stencil jacobi, Total data grid for data1: 2048*512*512; for data2: 2048*512*384. Both are decomposed into subgrid of 8*8*8. 4 nodes of a X86 cluster is used to run.

3.2 Memory-aware Control

The problem of interest here could be formulated as the following: We assume that an application starts with a memory footprint (M_A) in the bounds of physical memory (M) and there is a limit on the memory per processor (M_C) that can be used for holding data from communication. The M_C is determined so that it prevents the application from entering the swap zone, such that:

$$M_A + M_C <= M \tag{1}$$

For simplicity, assuming each message is of size C, therefore the runtime is able to schedule $\lfloor M_C/C \rfloor$ outstanding messages at a given time. The memory-aware runtime we designed schedules communication under such constraint.

It is clear that a runtime system can only achieve this goal when the message size is less than M_C, otherwise having even one message leads to swapping. To enable the runtime system to control memory usage effectively, one important design decision is to allow applications to be decomposed into finer grained computation. Fine grained computation leads to fine grained communication, which gives the runtime more opportunities to schedule communication in a memory efficient manner. We will see later in section 4.2 the advantage of fine grained computation encapsulated in the concept of virtualization.

Note that the inequality (1) is really not a hard limit, which means even if it can't be satisfied, if swapping is supported the program should be able to run. But we show in our study that in order to obtain undegraded performance, it's desirable to provide a best-effort soft guarantee that the inequality be met. In the next subsection, we present a token-based control strategy to provide this best-effort service at runtime.

3.3 Token-based Scheduling

Similar to Myrinet flow control, we apply tokens to represent memory resource allocated to an application. Data communications are posted only tentatively by the application, and the requests are queued by the runtime. The runtime would only schedule the transfer if the application possesses the token. Various interesting questions arise in this scenario such as to whom

and in what order to assign tokens to. An ideal allocation scheme should incur the least delay, yet bring extra benefits such as avoiding communication hotspot, balancing work load , reusing data and etc. For this paper, we are concerned with how to minimize memory usage.

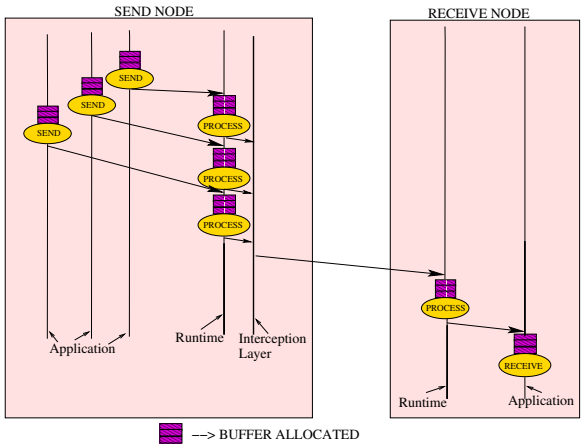
This token-based communication control scheme requires several extensions to the runtime in order to provide memory efficient communication. First, the runtime needs to intercept normal communication phases by injecting token-based control. Second, instead of letting an application pre-allocate a receive buffer, the runtime manages the message buffer as memory resource regulated by tokens. Next, we will examine implementing the token-based scheduling with both fetch-based and send-based schemes.

3.3.1 Fetch-based vs Send-based Scheme

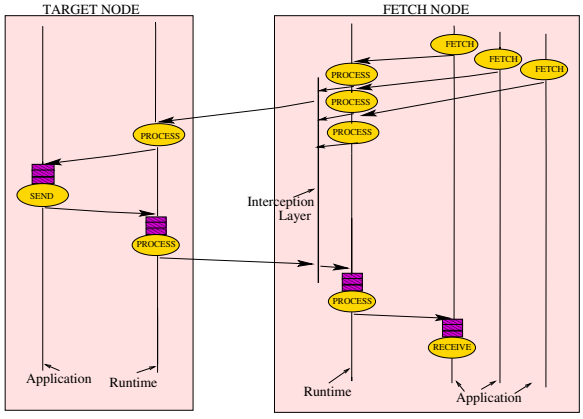
Different communication primitives pose different difficulty levels when trying to apply the token-based runtime control on communication buffer memory. First of all, to control specific large data transfers, we need to define points of interception. Secondly, the system needs to have knowledge about the party that the control will have an effect on. With a send-based model as shown in Figure 3(a), for the purpose of this discussion, we split the send-receive process into four phases:

1. **application requests to send**
2. **runtime processes the requests**
3. **receiving system receives the message**
4. **receiving system delivers the message**

In all four phases, memory is unavoidably consumed. With this model, possible runtime interception could happen during phase two or phase three. Regardless of where it takes place, in phase one, memory should already be allocated by the sender to prepare the send data, and memory is potentially needed to buffer the data at phase two. For the sake of discussion, assume runtime intercepts at phase two. At this stage, the best it can do is to avoid memory explosion caused by sending this data at the receiver side. In order to achieve that, knowledge about the receiver's memory usage is required. Either pre-knowledge exists or new knowledge is acquired on demand. Both approaches, however, run the risk of that knowledge being outdated. Further, the latter approach would bring extra delay for communicating with the receiver in a on-demand fashion.



(a) Send-based Model



(b) Fetch-based Model

Figure 3. Send-based vs Fetch-based

With a fetch-based model, as shown in Figure 3(b), the protocol is described in seven-phases:

1. **application poses tentative fetch**
2. **runtime processes the request**
3. **destination runtime receives request**
4. **destination application returns data**
5. **data passed to destination runtime**
6. **data passed to requesting runtime**
7. **data delivered to requesting application**

Combining the seven-phase model with the idea of runtime allocation of memory, the user doesn't preallocate memory for receiving data at phase one. Instead, the runtime allocates the memory when the fetch data is received at phase six. By decoupling the fetch request and the memory allocation, consumption of memory is pushed back to later phases – four, five, six and seven. Under this model, we intentionally choose runtime interception at phase two, in which the requester side's runtime queues up the tentative fetch requests and selectively schedules the ones within limits of its memory. Thus memory allocation occurring at later phases all falls within control. Moreover, the runtime making the scheduling decision would only require knowledge about itself in order to avoid the burst of memory allocation.

The advantage of the fetch-based model over the send-based model is simplicity of implementation and effectiveness of control. In the latter, interception occurs before all memory allocations take place; while in the former, it happens after data is generated and buffered. However, the side effect of using fetch is losing the explicit synchronization brought by the send-receive pair. Thus more careful synchronization is needed when fetch is being applied in the user program.

3.3.2 Detecting Memory Availability

Having addressed how the runtime regulates communication via token-based scheduling policy with the fetch-based model, the practical question remaining is that how to detect the amount of memory available to the job at the compute node, and hence decide on the number of tokens necessary. Since the memory availability for transient memory usage may change dynamically as a result of the variation of the application memory usage pattern over time, keeping track of memory availability is mandatory in order to adapt the number of tokens during the lifetime of the program.

Specifically, we need to calculate the application memory usage (M_A) and total available memory on a node (M) [1]. Application memory usage (M_A) can be easily instrumented in a memory allocator for each malloc and free. It is the peak memory usage measured in a certain period of time in the execution. Detecting memory availability on a node however is a nontrivial task [14] because most operating systems do not provide accurate free memory information. Often, even though operating systems report that the amount of free memory is close to zero, a large memory request from a process can still be accommodated. This is because many operating systems use as much free memory as it can as buffer memory, which can be adapted for user memory requests.

In this paper, we focus on dedicated parallel environments where there is no time sharing of other user applications. Therefore, the total amount of physical memory available on a compute node stays relatively stable during the execution time of a compute job, so that the application only needs to detect the physical memory availability at start time. A simple way to estimate the available physical memory is to first get the total amount of physical RAM installed, and then to subtract certain amount of memory (say 100MB) from this to leave room for the kernel and daemons. A more reliable way is to try to allocate and use memory as much as it can until a page fault occurs. This is to see how much of the temporarily claimed memory can be maintained in the program's non-swapped physical memory (as often reported in the RSS field of the Unix *top* utility), which can be used to define M.

In our current implementation, we assume that an application's base memory usage stays relative stable, and therefore we only calculate M_C once and use it as the maximum amount of the transient memory that the runtime is allowed to use for communication. In the future we plan

[1] M_C can then be calculated as $M - M_A$

to extend this scheme with token adaptivity ability which will be discussed in Section 5.

3.4 Guaranteed Progress and Deadlock Free

With token-based flow control, parallel threads issue a serial of fetch-data requests and later block waiting on them, which can be represented by a *fetch set*:

$$R_n = \{F_1, F_2, ..., F_n, W\} \quad (2)$$

where F_i is the issued fetch-data request and W is the waitall. The blocking wait introduces chances for deadlocks if there are dependencies between threads. To simplify the task of avoiding deadlocks at runtime level without the knowledge of application dependencies, we made the following two assumptions. One is the atomicity property of fetch requests $(F_1, ..., F_n)$ posted by any single thread. Any thread would execute in a pattern of posting fetch-data requests, doing computation, later waiting on the requests. Atomicity requires that the issuing of fetch requests $(F_1, ..., F_n)$ being atomic and thus guarantees that there's no interleaving of issuing fetch-data requests from different threads in the request queue. Under this assumption, an application can be simply viewed by the runtime as a sequence of *fetch sets* (Rs):

$$\{R_1, R_2, ..., R_k\} \quad (3)$$

where R_i is defined in (2) which is issued by a particular thread in the application. This assumption allows the runtime to execute the fetch requests in the order they are received and fulfill waits in the same order thread by thread. This avoids the detection of thread dependencies and significantly reduces unnecessary complexity of the implementation. The other assumption is that the number of tokens available would satisfy the progress of any single parallel thread, that is any R_i can be satisfied memory-wise. This assumption simply guarantees enough resources to make at least one thread progress. Under these two assumptions, we call the program *f-live* program which indicates it's guaranteed to progress without a deadlock situation on waiting for the fetch-data requests. With the charm++ and AMPI system, an execution of any thread is non-preemptive until it finishes and surrender the control to the scheduler. Thus atomicity is automatically satisfied without extra effort.

3.5 Applications in MPI

The above ideas on runtime control of communication-induced memory allocation can be applied to MPI implementations. We use the MPI_Alltoall as an example in this section to illustrate our implementation in the Adaptive MPI runtime.

In MPICH, the default implementation of MPI_Alltoall uses different algorithms based on the size of messages and communicators. For small messages (less than 256 bytes), MPICH uses a very efficient algorithm by Jehoshua Bruck et al [1]. It is a store-and-forward algorithm that takes $\log p$ steps, where p is the number of processors. Due to the messages being small, there is no memory issue for this algorithm. For medium size messages (less than 32KB), MPICH uses an algorithm that posts all irecvs and isends and then does a waitall, which however, requires significant transient memory for communication and does not scale to a very large number of processors. For example, to send a 16KB messages to 32,000 processors (BlueGene/L for example) requires about 512MB transient memory buffer which barely fits in BlueGene/L's memory. For large messages, MPICH switches to a memory-conservative implementation that uses a pairwise exchange algorithm, which takes $p - 1$ steps for p processors. This pairwise exchange algorithm makes sure the transient memory required between two processors in a step is strictly limited. This algorithm however may not fully utilize the communication bandwidth even though there may be enough memory for transient message buffers. It is clear that without the memory awareness, it is difficult for a runtime to choose the best algorithm that is both memory and speed efficient. The runtime has to pick either the second algorithm which communicates aggressively assuming the memory is sufficient, or the third algorithm which restricts the com-

munication to only a pair of send/recv between two processors at a time assuming the memory is extremely limited.

Our new implementation of MPI_Alltoall treats medium and large size messages in a way that adapts to the available physical memory. MPI runtime issues communication requests aggressively, while the underlying communication runtime serves the requests using tokens. The communication is progressed according to the physical memory availability. When physical memory allows, this scheme can process as many communication requests as possible. When physical memory is limited in serving all requests, it restricts the outstanding communication. In Section 4.2 we demonstrate that the new implementation achieves better performance than the default MPI implementation.

4 Performance Case Studies

We evaluated our token-based memory control scheme on several platforms with several Charm++ and MPI applications and compare with the normal scheme without memory control. For the rest of the paper, *normal* scheme refers to the send-based scheme without any control on communication, while the *controlled* scheme indicates the fetch-based scheme with token-based flow control. And specifically, *controlled-8token* would be a controlled scheme with 8 tokens applied. In our experiment, we use one token to represent one message.

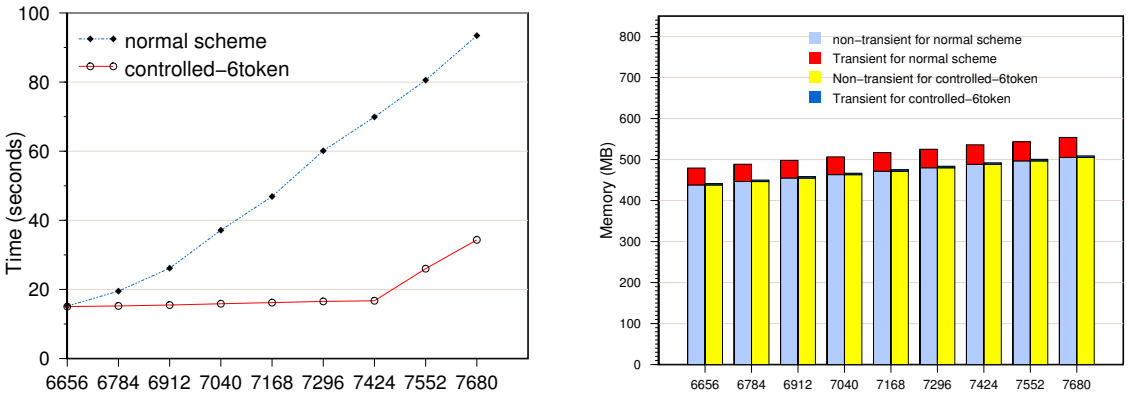
Two different clusters are used as testing platforms. The first one is a x86 Linux cluster with 8 nodes. Each node has 4 PentiumIII processors and 512MB of shared memory. Each processor is of 500MHz frequency and has 512KB cache. 100MB Ethernet is used as the interconnection. The second cluster is a AMD64 Linux cluster, where each node has 2 processors and 1GB of shared memory. Its processors are AMD Opterons of 1.8GHz and 1GB of cache. Nodes are connected with Gigabit Ethernet. From now on, we will refer to them as x86 cluster and AMD64 cluster respectively.

4.1 Jacobi (Charm++)

The first test program we run is the 3D stencil program as described in Section 3.1. Figure 4(a) shows the execution time on the x86 Cluster of the normal method vs the controlled method. A problem of size $X * 1024 * 128$, where X varies from 6656 to 7680, is partitioned into $128 * 128 * 128$ sub-cubes. A 7-point centered stencil is used which leads to ghost cell size of $128 * 128 * 1$. It can be observed that as soon as the total data grid exceeds a certain threshold determined by system memory size, the execution time of the normal method blows up while the controlled one stays relatively flat and curves up much later. From table 1 we see that disk swapping picks up at the third data point to 2469 and drastically increases more than tenfold for the fourth point. This corresponds to the nonlinear increase of execution time in Figure 4(a). The controlled scheme also starts to swap at $X = 7552$ and its performance is degraded. At this point, the non-transient memory of the program has exceeded the amount of available memory of the system. Figure 4(b) draws the memory usage of the same experiment. The height of the bar represents total memory allocated during the lifetime of execution and the top part is the portion of transient memory used for communication for the ghost cell data transfers. While the static memory consumption of the two methods are almost identical, the normal method possesses a much larger transient usage while the controlled one uses so little for transient that it is almost invisible in the graph.

Figure 5 shows a sample run of the program at the AMD64 cluster. The problem being solved is of size $X * 256 * 256$, where X varies from 3840 to 7040, and is decomposed into sub-cubes of $64 * 64 * 64$. A 13-point centered stencil is applied in the computation and the resulting ghost cell, in this case, is of size $64 * 64 * 2$. With the ghost cell widening and decomposition finer, the performance improvement is even more substantial than in Figure 4. Furthermore the undegraded execution zone is greatly extended.

In both cases, we see good performance improvement in execution time with reduced mem-



(a) (b)

Figure 4. Jacobi running on x86 cluster Total data grid is $X * 1024 * 128$, X steps from 6656 to 7680; running on 8 nodes, 1 proc per node, of the X86 cluster

scheme	6656	6784	6912	7040	7168	7296	7424	7552	7680
normal scheme	9	42	2469	45510	32528	54505	42105	73987	90632
controlled-6token	0	0	0	0	0	11	20	1043	7521

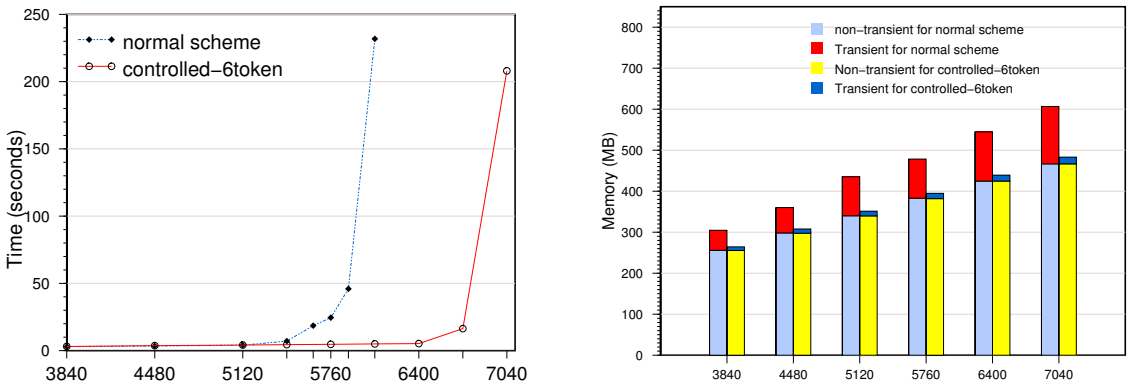
Table 1. Number of page faults during 20-iteration period of the Jacobi program, running on x86 cluster

ory footprint.

4.2 NAS Benchmark FT

Here we test the performance of our MPI_Alltoall implementation with the well-known NAS FT benchmark([8]). FT solves a three dimensional partial differential equation using forward and inverse FFTs. Hence it does several MPI_Alltoall's per iteration with relatively large data size. We run the unmodified FT benchmark with AMPI and compare the performance of the different schemes. Since the data size from class A to B to C multiplies rather than incrementally increases, instead of controlling the problem data size we take control of the amount of system memory available to the program. This is achieved by running a small program that uses a specified amount of memory by pinning every memory page of the region periodically. Since each compute node is 4-way SMP, running the memory-using program on one processor while running the FT program on another processor will not introduce any contention for CPU time between them.

We solve the class B problem on 8 nodes of the x86 cluster. The class B consists of a 3D data grid of $512 * 256 * 256$. The problem is decomposed for 16, 32, 64, 128 and 256 virtual processors respectively, and running on 8 nodes cluster, 1 processor per node. Three methods are compared: normal scheme, controlled scheme with 4 tokens and controlled scheme with 8 tokens. Figure 6(a) shows the execution time of the FT.B.128, which is decomposed for 128 virtual processors. As we can see, the controlled



(a) (b)

Figure 5. Jacobi running on AMD64 cluster Total data grid is $X * 256 * 256$, X steps from 3840 to 7040; running on 8 nodes, 2 proc per node, of the AMD64 cluster. *not in the graph: normal scheme takes 465.9sec at $x = 6400$, and fails to complete within 2 hrs at $x = 7040$*

scheme	3840	4480	5120	5760	6400	6720
normal scheme	0	0	0	21613	740370	**
controlled-6token	0	0	0	0	0	90632

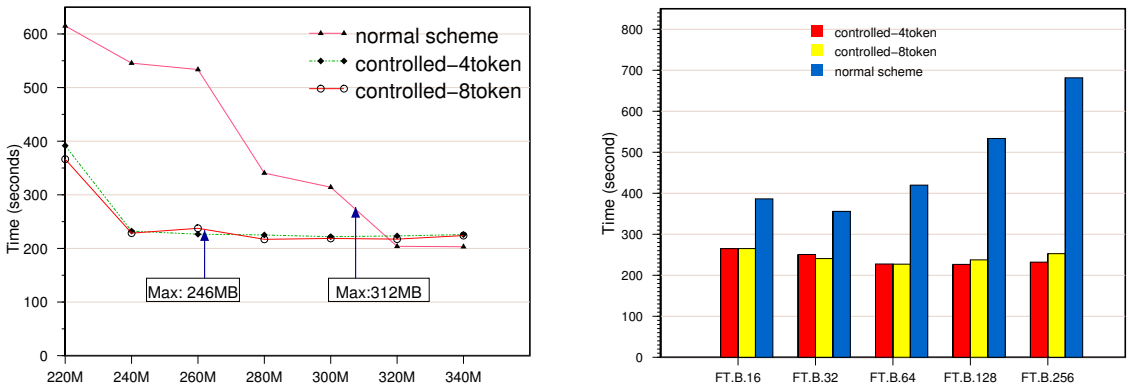
Table 2. Number of page faults during 10-iteration period of the Jacobi program, running on AMD64 cluster. *** indicates execution takes longer than 2 hours, no page fault data obtained*

scheme has improved performance when available memory is less than 320MB. Table 3 shows the number of page faults occuring during the same sample run. Figure 6(b) illustrates execution time of the three methods for different virtual processor numbers when system memory is 260MB. As we observed, in the normal scheme, when the number of virtual processors increases, for the same class B problem, the execution time first decreases and then increases due to the combined effect of cache performance gain and finer grained message overhead. For the controlled schemes, however, larger number of virtual processors gives the runtime more opportunities to schedule communication to overlap with the computation, leading to better performance.

Overall, flow control for large MPI_Alltoall communication improves performance when memory is limited by reducing the memory peak usage. With virtualization, this effect is being aggregated and improvement is greater.

5 Conclusion

We presented a memory-aware runtime system that controls communication-induced memory fluctuation, which helps applications with large memory footprint to keep within the bounds of physical memory and avoid disk swapping. The runtime imposes flow control via communication tokens for large data transfer and thus control the peak transient memory consumed by communication. This runtime support is imple-



(a) FT.B.128

(b) virtualization

Figure 6. NAS FT Benchmark, running on x86 cluster, time taken for 10 iterations

	220MB	240MB	260MB	280MB	300MB	320MB	340MB
normal scheme	65766	55580	32355	24346	8417	15	6
controlled-4token	16516	1143	3	1	0	0	0
controlled-8token	17053	1185	58	0	0	0	0

Table 3. Number of Page Fault during 2-iteration period of FT.B.128 Run, on the x86 cluster

mented in Charm++ and AMPI runtime, which is portable to a variety of platforms and used by a variety of parallel applications. The performance results demonstrate that considerable performance improvements have been achieved for a variety of applications that involve large volume of transient memory for communication such as MPI_Alltoall.

In the future we plan to enhance our token-based memory control scheme to be able to adapt to the availability of the physical memory. This allows our scheme to work efficiently for time-sharing environment where the memory availability is influenced by other applications running on the same node. It also allows our scheme to handle the dramatic variation of the application memory usage. The token adaptivity can be realized by periodically probing both the available physical memory and the current application memory usage, which can be used to adapt the number of tokens during the execution of the program. We also plan to use the runtime we developed with out-of-core methods, which provides an effective way of controlling both the application memory and the transient communication memory to further eliminate the disk swapping overhead.

References

[1] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 08(11):1143–1156, November 1997.

[2] Alan Wagner Chamath Keppitiyagama. Asynchronous mpi messaging on myrinet. In *15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, 2001.

[3] Fangzhe Chang, Ayal Itzkovitz, and Vijay Karamcheti. User-level Resource-Constrained sandboxing. In *4 th USENIX Windows Systems Symposium*, pages 25–36, August 2000.

[4] S. Coleman and K. S. M c Kinley. Tile size selection using cache organization and data layout. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, June 1995.

[5] C.H.Q. Ding and Y. He. A ghost cell expansion method for reducing communications in solving pde problems. In *roceedings of the 2001 ACM/IEEE conference on Supercomputing*, November 2001.

[6] J. Dongarra, S. Hammarling, and D. Walker. Key concepts for parallel out-of-core lu factorization. *Parallel Computing*, 23(1-2):49–70, April 1997.

[7] M.S.Lam et al. The cache performance and optimizations of blocked algorithms. In *Proc. ASPLOS-IV*, 1991.

[8] A.J. Ferrari, A. Filipi-Martin, and S. Viswanathan. The NAS Parallel Benchmark Kernels in MPL. Technical Report CS-95-39, University of Virginia, 1995.

[9] 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.

[10] 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.

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

[12] 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.

[13] Steve Otto Marc Snir and etc. *MPI: The Complete Reference*, volume 1. The MIT Press.

[14] Richard T. Mills, Andreas Stathopoulos, and Dimitrios S. Nikolop oulos. Adapting to memory pressure from within scientific applications on multiprogrammed cows. In *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, NM, April 2004. IEEE Computer Society.

[15] ChaMPIon/Pro MPI. Champion/pro mpi. http://verari.com/champion_pro.asp.

[16] MyriCom. The gm-2 message passing system. http://www.myri.com/scs/GM-2/doc/html.

[17] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. pages 119–126.

[18] E. Rothberg and R. Schreiber. Efficient methods for out-of-core sparse cholesky factorization. *SIAM Journal on Scientific Computing*, 21(1):129–144, January 2000.

[19] Jeffrey Scott Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.