ParADE: An OpenMP Programming Environment for SMP Cluster Systems

Yang-Suk Kee
Institute of Computer Technology
Seoul National University
Seoul 151-742, Korea
+82-2-880-7292
yskee@iris.snu.ac.kr

Jin-Soo Kim
Division of Computer Science
KAIST
Daejeon 305-701, Korea
+82-42-869-3545
jinsoo@cs.kaist.ac.kr

Soonhoi Ha
School of Computer Science and Engineering
Seoul National University
Seoul 151-742, Korea
+82-2-880-8382
sha@iris.snu.ac.kr

ABSTRACT

Demand for programming environments to exploit clusters of symmetric multiprocessors (SMPs) is increasing. In this paper, we present a new programming environment, called ParADE, to enable easy, portable, and high-performance programming on SMP clusters. It is an OpenMP programming environment on top of a multi-threaded software distributed shared memory (SDSM) system with a variant of home-based lazy release consistency protocol. To boost performance, the runtime system provides explicit message-passing primitives to make it a hybrid-programming environment. Collective communication primitives are used for the synchronization and work-sharing directives associated with small data structures, lessening the synchronization overhead and avoiding the implicit barriers of work-sharing directives. The OpenMP translator bridges the gap between the OpenMP abstraction and the hybrid programming interfaces of the runtime system. The experiments with several NAS benchmarks and applications on a Linux-based cluster show promising results that ParADE overcomes the performance problem of the conventional SDSM-based OpenMP environment.

Keywords

programming environment, SMP cluster, software distributed shared memory, hybrid programming, OpenMP, MPI

1. INTRODUCTION

Currently, commodity off-the-shelf microprocessors and network components are widely used as building blocks for parallel computers. This trend has made cluster systems consisting of symmetric multiprocessors (SMPs) attractive platforms for high-performance computing. Even though it is easy to configure cluster systems, specifically of small scale, it is challenging to utilize them easily and fully.

Message-passing (MP) and shared address space (SAS) [1] are two leading parallel programming models. An MP model assumes a distributed memory system in which a processor is an independent processing unit with private memory. In this model, inter-processor communication is accomplished by explicit message passing. MPI [2] is the most popular MP model and it provides both functional and performance portability. MPI uses consistent MP interfaces not only for inter-node communication but also for intra-node communication. Despite high performance, however, the application programmers have hard time in developing parallel programs in MPI; they should handle too many details about communication and synchronization. This makes SAS more appealing to application programmers.

An SAS model provides a simple abstraction of parallel computers in which all processors share the same address space. As an SAS model, OpenMP [3] is gaining its popularity. OpenMP consists of a small set of compiler directives for shared memory parallelism and it provides high-level interfaces to thread programming. The directives define how to share workloads among threads, how to synchronize threads, and how to determine the scope of variables. The application programmers can easily implement a parallel program just by inserting directives into time critical sequential codes incrementally. Furthermore, OpenMP anticipates high performance in scientific applications on a shared memory system.

Even though OpenMP is originally designed for shared memory multiprocessor system, this model can be applicable to cluster systems with a middleware support, e.g. software distributed shared memory (SDSM) [4]. However, there are two obstacles to integrating the OpenMP model with an SDSM system. The first is to support multi-threading for intra-node parallelism. Even though most conventional SDSM systems are single-threaded, an SDSM system for OpenMP should be multi-threaded. The second issue is to overcome the poor performance of SDSM. An SDSM program moves much larger amount of data between nodes than an equivalent MPI program. Moreover, SDSM has the poor performance of synchronization operations. This means that an inefficient integration of OpenMP and SDSM will lead to disappointing results. The previous studies of OpenMP on SMP clusters [5-7] have mainly focused on how to extend the OpenMP model to clusters but they did not deal with the performance issues being left for future work.

From the viewpoint of programming easiness, the application programmers prefer SAS to MP. In contrast, MP or hybrid models are desirable for high performance. To escape from this dilemma, we separate the programming model and execution model. Our

SC '03, November 15-21, 2003, Phoenix, Arizona, USA.
Copyright 2003 ACM 1-58113-695-1/03/0011...$5.00.

approach is to use an SAS model for programming and to use a hybrid model for execution.

In this paper, we propose an efficient OpenMP-based programming environment for SMP clusters, called ParADE (Parallel Application Development Environment). ParADE consists of an OpenMP translator and a runtime system. The runtime system provides a single system image through a multi-threaded SDSM. However, it is augmented with explicit message-passing primitives to reduce the synchronization overhead of conventional SDSM systems. Hence, the ParADE runtime system provides a hybrid execution model of MP and SAS. Those message-passing primitives are invisible to the application programmers who write a standard OpenMP program. The OpenMP translator automatically replaces several synchronization and work-sharing directives associated with small data structures in OpenMP programs with explicit collective communication primitives. This explicit use of message-passing primitives avoids the conventional lock-based synchronization processes and barrier operations imposed implicitly by the OpenMP standard.

The rest of this paper is organized as follows. First, we review related work on programming models and methodologies for SMP cluster and discuss our motivation of this study in section 2. We overview the ParADE architecture in section 3 and discuss the details of two main components, the OpenMP translator and the ParADE runtime system, in sections 4 and 5. We give experimental results of several microbenchmarks, NAS benchmarks, and real applications in section 6. In section 7, we introduce several programming guidelines to achieve better performance. Section 8 concludes this paper discussing our ongoing and future work.

2. RELATED WORK AND MOTIVATION

2.1 Related Work

Programming models for SMP clusters can be categorized into a unified programming model and a hybrid programming model. In a unified programming model, the programmers use a single set of programming interfaces to describe the inter-node and intra-node communication. For example, a pure MPI model regards an SMP cluster as a shared nothing architecture while a pure SDSM model regards an SMP cluster as a shared everything architecture. Shan et al. [8] compared the performance of the pure MPI model with the pure SDSM model for several applications. They found that the SDSM versions achieve only half the performance of the corresponding MPI versions for many applications.

In a hybrid model, the programmers mix an MP model and an SAS model. It is intuitive to use an SAS model for intra-node communication and an MP model for inter-node communication. Especially, the mixture of OpenMP and MPI draws attraction and the performance comparison with the pure MPI model has been studied [9,10]. Cappello et al. [9] extensively compared the performance of the hybrid model and the pure MPI model on an IBM SP machine. In the paper, they found that the pure MPI model is better for most of the NAS benchmarks. On the other hand, Shan concluded in [8] that the hybrid of SDSM and MPI has small advantage over pure MPI even though the programming complexity increases very highly.

As another unified programming model, there have been several studies of OpenMP on SMP clusters. The main focus of the studies is how to provide a multi-threading environment by modifying existing SDSM. Hu et al. [7] extended TreadMarks for OpenMP and compared the performance of the NAS benchmarks under three programming environments: original TreadMarks, OpenMP on multi-threaded TreadMarks, and MPI. A key lesson from the study is that developing OpenMP applications requires intensive cares. Compared to the MPI versions, the OpenMP programs experience more barriers and suffer from higher network traffic. In OpenMP, the fork-join execution model and work-sharing directives include the implicit barriers at the end of execution. Moreover, the memory consistency protocol using virtual memory management techniques incurs unnecessary page transfers. Hence, we should optimize mappings of the OpenMP directives to the SDSM interfaces. Their main contribution is to first develop a multi-threaded SDSM system for OpenMP; they extended the original TreadMarks system to a multi-threaded one, and implemented a new OpenMP translator for the target system.

Since the main focus of prior SDSM systems was how to efficiently emulate a shared address space, they did not consider any specific programming model like OpenMP. This ignorance of programming model introduces extra overhead in integrating them with OpenMP. Basumallik et al. [6] discussed the performance issues of OpenMP on SDSM. In the paper, they presented several ideas on optimization techniques to improve performance. They pointed out frequent synchronizations, specifically barriers, as the main obstacle to high performance but they have not presented any real system to demonstrate their ideas.

2.2 Motivation

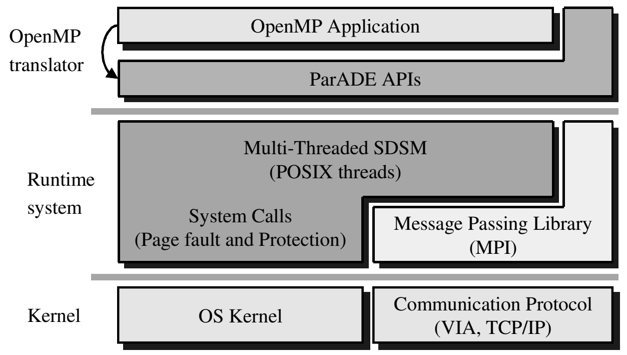
The previous studies of OpenMP on SDSM raise a demand for a new runtime system to achieve the goal of easy and high-performance programming. As addressed in [6], efficient implementation of synchronization directives is crucial to overall performance since the directives themselves are time consuming and suppress concurrency in applications. Therefore, we need to reduce the cost and the number of synchronization operations.

At first, we have observed that many code blocks protected by synchronization directives are statically analyzable at compile time and they can be replaced with cheap message-passing primitives. In conventional SDSM systems, a process is allowed to enter a critical section only after it has acquired the lock for the critical section. To acquire the lock, the process requests a lock of the lock home and the home grants the lock piggybacking with consistency information. This lock mechanism is expensive and the process experiences long latency. However, exploiting message-passing primitives has several benefits. In an MP model, message-passing primitives can replace the costly locks since mutual exclusion between processes is implicit. They also eliminate the memory consistency mechanism in the critical path of memory access; *twin* and *diff* creation are not required. Moreover, collective communications contribute to reducing the number of barriers imposed by the work-sharing directives since they perform a kind of global synchronization implicitly.

These observations motivate us to design a new multi-threaded SDSM system augmented with message-passing primitives. To convert the OpenMP directives to a hybrid style of MP and SAS, we also need a new OpenMP translator. ParADE successfully covers these issues overcoming the poor performance of the conventional SDSM-based approaches.

3. PARADE ARCHITECTURE

Figure 1 depicts the architecture of the ParADE programming environment. Two key components are the OpenMP translator and the ParADE runtime system.



OpenMP translator
OpenMP Application
ParADE APIs
Runtime system
Multi-Threaded SDSM (POSIX threads)
System Calls (Page fault and Protection)
Message Passing Library (MPI)
Kernel
OS Kernel
Communication Protocol (VIA, TCP/IP)

Figure 1. Architecture of the ParADE parallel programming environment

A multi-threaded SDSM and a message-passing library comprise the runtime system. To provide thread-safe communication, we implemented a subset of MPI library for Virtual Interface Architecture (VIA) [11] while we use MPI/Pro by MPI Software Technology for TCP/IP protocols [12]. We also developed our own SDSM system, which provides a home-based lazy release consistency (HLRC) [13] with migratory home to exploit data locality. The lock mechanism is eliminated from the SDSM system in the critical path to shared memory access by utilizing explicit message-passing primitives. ParADE classifies data structures according to their size and applies different protocols: update protocol for small data structures by using message-passing and invalidate protocol for large data structures by using HLRC.

The OpenMP translator converts an OpenMP program to a multi-threaded program by the hybrid communication interfaces of the ParADE runtime library and enables the program to be executable on the SMP cluster. Especially, the translator focuses on how to exploit the message-passing operations in converting synchronization and work-sharing directives. In the following sections, we will discuss the details of two main components.

4. PARADE OPENMP TRANSLATOR

The basic role of OpenMP translator is to bridge the gap between the OpenMP specification and the programming interfaces of the underlying runtime system. The ParADE OpenMP translator converts an OpenMP program into the C codes using POSIX threads and MPI libraries: the ParADE APIs encapsulate the details of POSIX threads and MPI. Our OpenMP translator is based on the Omni OpenMP compiler [5]. We modified the C-front program in the Omni package. The original use of C-front is to analyze a preprocessed C program and then to make a parse tree. Instead, we modified it to reconstruct the original C program from the parse tree in reverse. Similar to the Omni compiler, the OpenMP translator consists of three steps. In the first step, the translator invokes the installed preprocessor and expands macros and header files. Then, C-front reads these preprocessed codes and builds a parse tree, which contains the information about the OpenMP directives. In the last step, C-front reconstructs another C program from the parse tree replacing the directives with the corresponding ParADE APIs. The current implementation of the OpenMP translator follows the OpenMP standard version 1.0 for C and C++ application programming interface [3]. In the following subsections, we explain how some important OpenMP directives are translated by using the ParADE APIs.

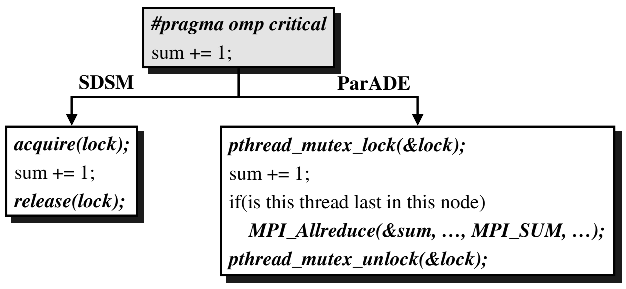
4.1 `Parallel` Directive

A `parallel` directive is the basic directive that starts parallel execution. A code block annotated with the `parallel` directive is encapsulated into a thread function and the directive is replaced with the ParADE runtime interfaces to realize the fork-join execution model. The pointers to the variables declared as `shared`, `firstprivate`, `lastprivate`, and `reduction` are passed to the thread function while the `private` variables are declared automatic inside the thread function.

According to the OpenMP standard, the default scope of variables in a `parallel` block is `shared` since threads in a process share all data structures except thread stacks. However, this assumption is inappropriate to MP architectures: the variables on different nodes cannot be shared for free. For better performance, it is highly recommended to explicitly annotate all the variables used in `parallel` blocks to avoid unnecessary network traffic between nodes.

4.2 Synchronization Directives

There are several directives for synchronization. A `critical` directive provides mutual exclusion between threads and commonly used to reduce non-scalar variables. When a thread enters a critical section, it is guaranteed to see all previous modifications. The directive can be translated to a collective communication operation together with a pthread lock. Figure 2 illustrates how a `critical` directive is translated for ParADE and for a conventional SDSM system, respectively. In ParADE, the mutual exclusion is hierarchically performed: the pthread lock ensures mutual exclusion between threads in a process and the collective communication operation performs synchronization between processes. For the SDSM system, however, the lock primitives are used for the intra-node and inter-node mutual exclusion. The collective communication operations update modified variables immediately avoiding the overhead due to creating memory consistency information and perform mutual exclusion implicitly eliminating the lock for inter-process mutual exclusion.



```
#pragma omp critical
sum += 1;
```

```
acquire(lock);
sum += 1;
release(lock);
```

```
pthread_mutex_lock(&lock);
sum += 1;
if(is this thread last in this node)
    MPI_Allreduce(&sum, …, MPI_SUM, …);
pthread_mutex_unlock(&lock);
```

Figure 2. Translated codes of the `critical` directive for a conventional SDSM system and for ParADE

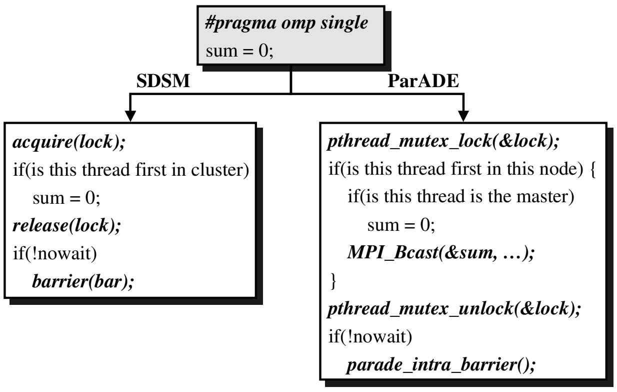
An `atomic` directive ensures the atomic update of a specific memory location and the `atomic` code block must be one of the predefined expression statements. We regard the `atomic` directive as a special case of the `critical` directive. The code block can be exactly mapped to a collective communication primitive.

There is a clause to exploit message-passing primitives. The `reduction` clause performs a reduction on the scalar variables. Similar to the `atomic` directive, a `reduction` variable is exactly mapped to a collective communication primitive. If there are more than one `reduction` variables, they are merged into a structure-type variable and reduced at once with a user-defined reduction operation. To exploit the benefits of message-passing as much as possible, the programmers are guided to use the `reduction` clause or the `atomic` directive instead of the `critical` directive. In the case of the `critical` directive, it is highly recommended to write a lexically analyzable code block.

4.3 Work-Sharing Directives

Work-sharing directives distribute workloads among threads. A `for` directive divides the iterations of a loop into smaller slices and assigns them to threads. The translator extracts the range and the increment information of the loop and the loop scheduler in the runtime system determines the range of iterations for threads at run time. Current implementation supports only `static` scheduling: the iterations are evenly distributed to threads. As of now, nested `for` directives are ignored.

A `single` directive identifies a code block to be executed by the earliest thread. This directive is mainly used to initialize shared variables. Figure 3 illustrates the translated codes of a `single` directive for a conventional SDSM system and for ParADE. Similar to the `critical` directive, synchronization within a node is replaced with a pthread lock and synchronization between nodes is replaced with an explicit message-passing operation; this contributes to reducing the number of inter-process barriers and to eliminating the inter-node lock in the critical access path to the variable.



```
#pragma omp single
sum = 0;
```

SDSM

```
acquire(lock);
if(is this thread first in cluster)
    sum = 0;
release(lock);
if(!nowait)
    barrier(bar);
```

ParADE

```
pthread_mutex_lock(&lock);
if(is this thread first in this node) {
    if(is this thread is the master)
        sum = 0;
    MPI_Bcast(&sum, ...);
}
pthread_mutex_unlock(&lock);
if(!nowait)
    parade_intra_barrier();
```

Figure 3. Translated codes of the `single` directive for a conventional SDSM system and for ParADE

5. PARADE RUNTIME SYSTEM

The ParADE runtime system provides users a single, unified abstraction of clusters. ParADE is a kind of multi-threaded SDSM. A major issue of SDSM systems is how to preserve memory consistency in distributed memory systems. In this section, we address two issues on memory management in the ParADE runtime system: thread-safe page update and memory consistency protocol.

5.1 Atomic Page Update

Conventional SDSM systems are implemented at the user-level by using a page-based virtual memory protection mechanism. This kind of SDSM system detects an application's access to an invalid shared memory region by catching a SIGSEGV signal generated by the operating system when the application violates memory access privilege. Then, the SDSM system carries out a series of operations to fetch the most up-to-date page from a remote node in a user-defined SIGSEGV signal handler. Since the series of operations are performed sequentially, the SDSM system can update the invalid page atomically from the application point of view: program control is returned to the application only after the signal handler completes the service on the protection fault.

However, this mechanism will not work any more in a multi-threaded environment because other threads may try to access the same page during the page update period. Figure 4 illustrates this situation. On the first access to an invalid page, the system should set the access permission of the page writable in order to replace it with the most up-to-date page. Unfortunately, this change of access permission also allows another application thread, T1 in Figure 4, to access the same page without raising any protection fault. This phenomenon is known as atomic page update and change right problem [14] or mmap() race condition [15]. Simply, we call this *the atomic page update problem*.

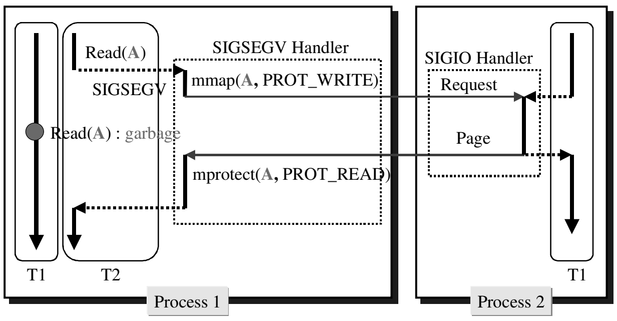


Figure 4. The atomic page update problem in a conventional page-based SDSM system

This problem is caused by the fact that the system and the application share the same address space. When the system changes the access permission of a page, the application also experiences the same change. A general solution to this problem is to separate the application address space from the system address space for the same physical memory, and to assign different access permission to each of the address spaces. Since the virtual memory protection mechanism is implemented in the per-process page table, different virtual addresses (pages) can have different access permissions even though they refer to the same physical page. It means that the system address space can be writable while the application address space is write-protected. Therefore, the system can guarantee the atomic page update by changing the access permission of a virtual page in the application address space only after it completes the page update through the system address space.

The conventional method to realize this solution is to use file mapping. The mmap() system call allows a file to be mapped multiple times to a process or several processes. When a file is mapped multiple times, multiple independent access paths to a physical page are provided because the same file is referred to by multiple virtual addresses.

We have developed three other methods to solve the same problem: System V shared memory, a new mdup() system call for page table duplication, and child process creation. Since full explanation of these methods [16] is beyond the scope of this paper, we briefly summarize them. In the System V shared memory solution, a process creates a shared memory object in the kernel by the shmget() system call and attaches it to the address space by the shmat() system call. A process can attach the shared memory to its address space more than once and a different virtual address is assigned to each attachment. In the second solution, we have implemented a new system call, mdup(), which creates a detour to an anonymous memory region. The basic operation is to allocate new page table entries for the detour and to copy the page table entries of the anonymous memory region to new ones. In the child process creation method, we use the fact that the child process inherits the execution image of the parent process when a process forks a child process. Especially, the content of the child process page table is copied from that of the parent process. Since the Copy-On-Write policy is not applied to the shared memory area, we can create two different access paths to a shared physical page by making two processes access the same page.

Through extensive experiments, we observe that all the methods achieve comparable performance on an SMP Linux cluster system while it is not always possible to implement them in a certain operating system due to various constraints of the operating system. In particular, the conventional file mapping method shows poor performance on IBM SP Night Hawk system with an AIX 4.3.3 PSSP 3.2 version.

5.2 Memory Consistency Protocol

A key feature of ParADE is to utilize message-passing primitives explicitly for some synchronization and work-sharing directives associated with small data structures. The OpenMP translator identifies these directives and replaces them with collective communication primitives while the runtime system takes care of the other shared variables by using a variant of the conventional HLRC protocol.

5.2.1 Message-passing for small data structures

Most page-based SDSM systems use locks to exclusively access shared variables. The lock mechanism must guarantee that the lock acquirer views the most up-to-date values of the variables in the critical section. Instead, ParADE ensures mutual exclusion implicitly by using message-passing operations in accessing small shared variables in a critical section. ParADE preserves memory consistency of the small data structures guarded by synchronization and work-sharing directives on the basis of object, which is similar to the entry consistency protocol [17].

ParADE adopts a kind of update protocol and the values modified in a critical section are propagated to other processes immediately. The ParADE runtime system does not have to create a *twin* and to calculate the *diffs* for the modified page where the small size data structures reside. Moreover, the explicit use of message-passing primitives reduces the number of barriers: some collective communication operations imply barrier at the end of the operations.

Consistency mechanism switches from the HLRC mode to the message-passing mode when the code block enclosed by a synchronization or work-sharing directive is lexically analyzable and the total size of the shared data structures in the code block is smaller than a certain threshold. The threshold is dependent on the startup cost of message-passing operations and the overhead of creating a *twin* and *diffs* for a page. For example, we set the threshold to 256 bytes for our Linux cluster.

5.2.2 HLRC with migratory home

Except small data structures, ParADE ensures memory consistency by using a variant of conventional HLRC protocol. When a thread tries to access a page not in its local memory, the runtime system fetches the page from its home. Once the page is brought to the local memory, the subsequent accesses to the page are localized until the next barrier. Home-based protocols like HLRC and scope consistency [18] have a fixed home that has the most up-to-date page. Home-based protocols are preferable to homeless protocols in that they reduce the number of control messages and the page fetch latency because every node knows where to fetch the most up-to-date pages. In addition, the home can avoid creating a *twin* for the modified page because all *diffs* are merged into its page.

However, the fixed home approach may incur unnecessary network traffic when the modifier of page does not coincide with the home. The main target of OpenMP is scientific applications with regular computing patterns. Typically, these applications consist of several loops with numerous iterations accessing huge arrays. When the iterations are distributed over the processes on different nodes, locality of arrays is crucial to overall performance. We try to increase locality by dynamically designating the modifier as home. Home migration occurs only at barrier time. If there is only one modifier of a page between two consecutive barriers, the node becomes the new home. Otherwise, the node with the highest priority becomes home. The current home node has the highest priority and the node with the smallest node id has priority. This is another feature different from the original HLRC.

A potential overhead of home migration is to notify new homes to all nodes. To reduce the number of control messages, all the write-notices are combined into a single message, and the message is piggybacked with a barrier message; the master node gathers write-notices piggybacked with barrier arrival messages and notifies the new homes with barrier departure messages.

5.2.3 Page management

Each node maintains a page table for the pages in the shared memory pool. Illustrated in Figure 5, there are five possible states for a page: INVALID, TRANSIENT, BLOCKED, READ_ONLY, DIRTY. When a page is not in the local memory, the state of the page is INVALID. The access to this page violates page protection and the runtime system should prepare a valid page. The two states, TRANSIENT and BLOCKED, are introduced because ParADE is a multi-threaded system. The TRANSIENT state tells other threads that a thread is trying to update the page but the update is not completed. The BLOCKED state tells the runtime system that there are threads waiting for the completion of the page update and the runtime system should wake them up after the page update is completed. The page state is

READ_ONLY when a page is valid and clean, and the page state is DIRTY when a page is valid and modified.

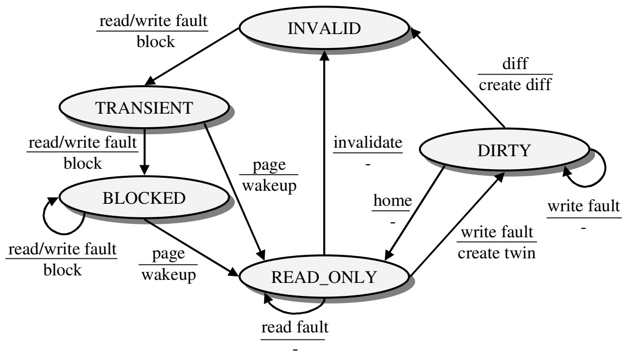


Figure 5. State transition diagram for the pages in the shared memory pool

Similar to conventional page-based SDSM systems, the unique entry point of application to the system is the page fault handler, which is a user-defined reliable SIGSEGV handler. The signal handler supervises shared memory. The handler determines the address where the protection violation occurs and the fault type. When the current state of the page is INVALID, the handler fetches the most up-to-date page from the home. Otherwise, it changes the state and the access permission of the page.

In the beginning, all shared pages in the non-master nodes are initialized as INVALID and the home of all pages is set as the master node while the page states of the master node are READ_ONLY. Coherence misses occur when incoming write-notices invalidate a page. Invalidation involves changing the page state from READ_ONLY to INVALID, and removing the access right from the page. A coherent miss indicates that more than one node has modified the page and the modification should be reflected in the local copy before the page is accessed again. In response to a write fault, the handler prepares a valid page, changes the page state to DIRTY, and creates a *twin* for the page.

5.3 Parallel Library

In ParADE, each node has a thread dedicated to communication to handle asynchronous incoming control messages. Since application threads and the communication thread can issue communication requests simultaneously, a communication library should be thread-safe. However, most MPI libraries in public domains are not thread-safe. Moreover, a high-performance communication library with respect to latency and bandwidth is desirable. We adopt the TCP/IP version of MPI/Pro by MPI Software Technology and implement a minimal set of MPI routines on top of VIA to exploit the full performance of VIA. ParADE uses only send/receive point-to-point communication and two collective communication primitives: MPI_Bcast() and MPI_Allreduce().

6. EXPERIMENTS

In this section, we present the preliminary experimental results of the ParADE system. Our experimental platform is a Linux cluster consisting of four dual-Pentium III 550Mhz SMP nodes and four dual-Pentium III 600Mhz SMP nodes. Each node has 512 MB main memory, connected to a 3Com Fast Ethernet switch and a Giganet's cLAN VIA switch. Redhat 8.0 of a 2.4.18-14 SMP kernel runs on each node. We used a GNU gcc compiler with the –O2 option. Even though we made our best efforts to compare ParADE with OpenMP programs on other SDSM systems like Score/Scash, we could not get such systems working. Therefore, we measured the performance of the ParADE system with respect to synchronization latency and scalability.

6.1 OpenMP Microbenchmarks

To evaluate the performance benefit of using explicit message-passing operations, we compared the performance of the `critical` and the `single` directives on ParADE and on an HLRC-based SDSM system. The translated codes for both the systems are shown in Figure 2 and 3 and the experimental results are shown in Figure 6 and 7. We took the average execution time of the micro-benchmark program [19] after running over 100 times varying the number of nodes. We used an HLRC-based SDSM system called KDSM [20] and MPI/Pro TCP/IP version.

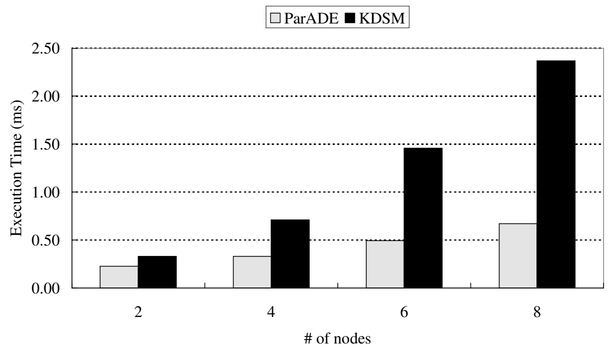


Figure 6. Performance comparison of the `critical` directive between ParADE and KDSM

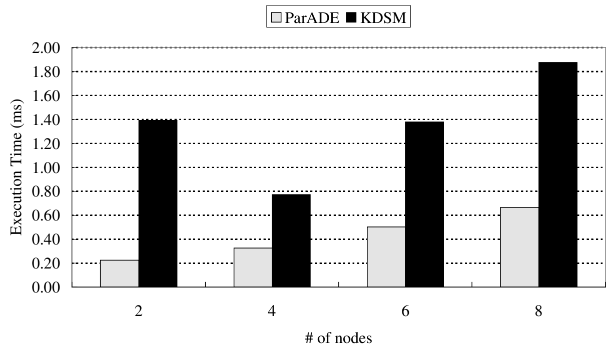


Figure 7. Performance comparison of the `single` directive between ParADE and KDSM

The ParADE versions of the `critical` and `single` directives outperform the SDSM versions and the gap becomes wider as the number of nodes increases; the number of control messages to get locks and the amount of data moving around increases with the number of nodes. The abnormal result of the `single` directive for KDSM with 2 nodes results from busy waiting to get the lock.

6.2 Benchmark Performance

In this section, we present the performance of the ParADE system by measuring the execution times of two NAS benchmarks and two real applications. The CG and EP kernels of class-A are adopted from the NAS 2.3 benchmarks [21]. The CG kernel solves an unstructured sparse linear system by the conjugate gradient method and the EP kernel measures the capability of floating-point operations. Meanwhile, two real applications are adopted from OpenMP sample programs [22,23]. The Helmholtz program solves a wave equation on a regular mesh using an iterative Jacobi method with over-relaxation and the MD program implements a simple molecular dynamics simulation in continuous real space. We use the following three configurations to measure the programs.

- **1Thread-1CPU** Start the operating system using the uniprocessor kernel: a single processor should handle both computation and communication.
- **1Thread-2CPU** Start the operating system using the SMP kernel but create only one computational thread per node: one processor is dedicated to the computational thread and the other to the communication thread.
- **2Thread-2CPU** Start the operating system using the SMP kernel and create two computational threads per node: two computational threads and a communication thread share two processors.

Figure 8 and figure 9 show the execution times of the NAS benchmarks on our VIA-based Linux cluster. The CG program experiences relatively larger page migration with 64-megabyte shared memory than other programs. The configuration of 1Thread-1CPU suffers from high communication delay because a single processor is responsible for both computation and communication. There is little overlapped communication, which serializes the execution of processes on other nodes. The performance gap between 1Thread-1CPU and 1Thread-2CPU becomes wider as the number of nodes increases. These results demonstrate that overlapped communication is a crucial factor in high-performance of cluster system. In the case of EP, there is little shared memory and communication between nodes occurs at the end of the program just once. Hence, it is natural that ParADE is highly scalable.

The experimental results of the Helmholtz and MD programs are shown in figure 10 and in figure 11. The Helmholtz program repeats about one thousand iterations until the calculated value become smaller than a certain threshold. Nodes communicate with only the adjacent nodes. However, each node updates a shared variable competitively to check the value satisfying the threshold. In the ParADE system, the OpenMP translator replaces this code with a reduction operation and the overall performance is nearly linear. Meanwhile, the MD program repeats one thousand iterations and determines the dynamics of molecules. With respect to communication pattern, MD is similar to Helmholtz but the amount of shared memory and inter-node communication of MD is less than that of Helmholtz. Hence, ParADE is scaled well for all the configurations.

7. PROGRAMMING GUIDELINES

OpenMP is originally proposed for shared memory multi-processors but we have extended the use of OpenMP to SMP clusters. The main difference between OpenMP on cluster and OpenMP on multiprocessors is communication cost. Accessing shared variables in a cluster may cause page migration between nodes. Since the inter-node communication cost is very expensive, the programmers should make efforts to avoid unnecessary page migration.

The main feature of ParADE is to utilize message-passing primitives for synchronization and work-sharing directives. An `atomic` directive or a `reduction` clause is directly mapped to a collective communication primitive. In addition, the `critical` directive that does not enclose any function call is mapped to a collective communication primitive. Therefore, using these directives or clause for mutual exclusion eliminates page migration and simplifies the conventional locking mechanism. Especially, applications like equation solver repeating iterations until satisfying a certain termination condition take significant advantage of explicit message-passing primitives.

Since the default scope of variables in a `parallel` block is `shared`, careless development of applications increases network traffic due to inter-node communication. Therefore, programmers are recommended to annotate the scope information of small data structures explicitly. For example, we can annotate local variables as `private`, read-only shared variables as `firstprivate`, writable shared variables associated with a `critical` directive as `reduction`, and so on.

The last consideration is to reduce page migration due to large arrays. In ParADE, the consistency unit of large arrays is page. An idea is to reduce the number of shared pages at the expense of additional memory. For example, we can reduce the number of shared page by declaring the arrays used temporarily to store intermediate values as local variables within a `parallel` block.

8. CONCLUSION AND FUTURE WORK

In this paper, we present a new programming environment called ParADE to enable easy, portable, and high-performance programming for SMP clusters. It is an OpenMP programming environment on top of a multi-threaded SDSM system. A variant of HLRC protocol is adopted for memory consistency. To boost performance, the ParADE runtime system provides MPI-like explicit message-passing primitives to make it a hybrid-programming environment at this level of abstraction. However, the OpenMP abstraction hides the details of hybrid programming and the OpenMP translator bridges the gap between the application specification and the underlying runtime system.

The experimental results with two NAS kernels and two real applications on a Linux cluster demonstrate the benefits of the proposed environment, easy and portable programming, and high-performance execution. Even when applications are designed without application specific optimization, the ParADE system shows the performance between those of an SDSM application and a pure MPI application.

However, we have several issues to improve our system. In many cases, processes wait a long time at barrier due to load-imbalance in executing the `for` blocks since the current version of ParADE supports only the `static` loop scheduling. Even though the OpenMP standard describes various loop-scheduling policies, they are not all appropriate for SMP cluster systems. Further studies on loop scheduling for SMP cluster systems will promise significant improvement in system performance.

Another issue is to adapt the system configuration during runtime. As the experimental results show, more processors do not always give better performance. For a given problem, we want

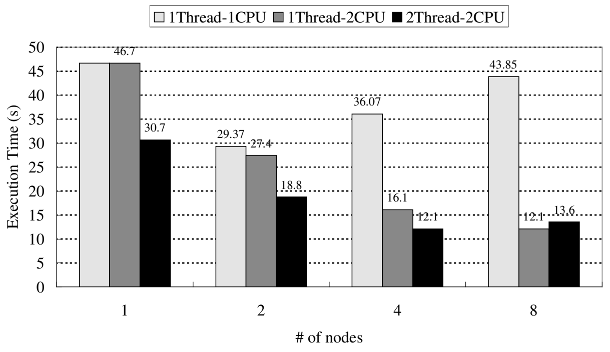


Figure 8. Execution time of the CG kernel on cLAN (A class)

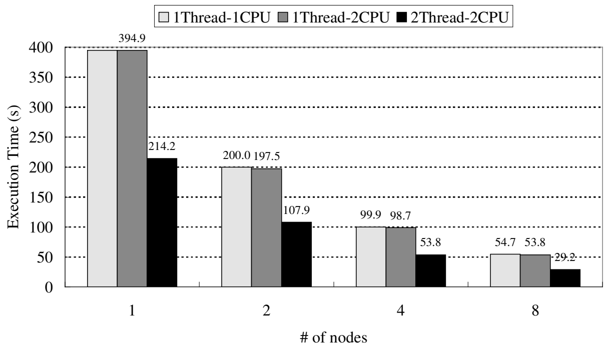


Figure 9. Execution time of the EP kernel on cLAN (A class)

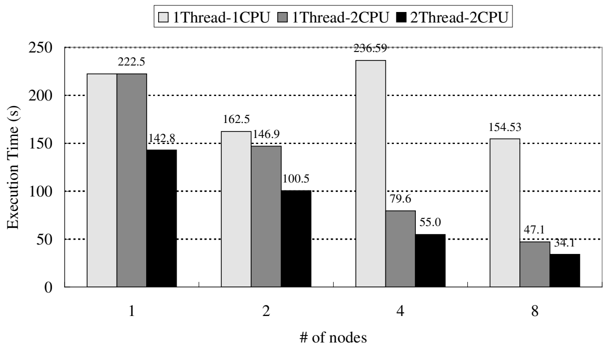


Figure 10. Execution time of the Helmholtz program on cLAN

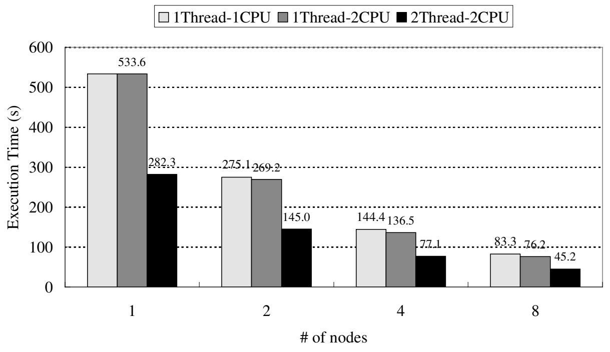


Figure 11. Execution time of the MD program on cLAN

to find the best configuration to extract the best performance. The current ParADE system creates a fixed number of threads according to the number of processors available in the system. We may dynamically determine a proper number of processors and threads by measuring the idle time of threads between subsequent iterations.

The last problem is related to the OpenMP translator. More intelligent translator may accelerate ParADE significantly. For example, the translator can analyze locality of arrays. If arrays are partitioned across nodes, then the synchronization for the arrays is not required. Moreover, if the arrays are used temporarily as buffers, then we can skip memory consistency process for the arrays.

9. ACKNOWLEDGMENTS

This work was supported by National Research Laboratory Program (No. M1-0104-00-0015) and Brain Korea 21 Project. The ICT at Seoul National University provides research facilities for this study.

10. REFERENCES

[1] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta, *Parallel Computer Architecture*, Morgan Kaufmann, San Francisco, CA, 1999.

[2] Message-passing Interface Forum, "MPI: A Message-Passing Interface Standard," *International Journal of Supercomputer Applications and High Performance Computing*, vol. 8, no. 3/4, Fall/Winter 1994, pp. 159-416.

[3] OpenMP C and C++ Application Programming Interface, Version 1.0, http://www.openmp.org, Oct. 1998

[4] Kai Li and Paul Hudak, "Memory coherence in shared virtual memory systems," *ACM Transactions on Computer Systems*, vol. 7, no. 4, Nov. 1989, pp. 321-359.

[5] Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka, Design of OpenMP Compiler for an SMP Cluster, In Proceedings of *European Workshop on OpenMP (EWOMP'99)*, Sep. 1999.

[6] Ayon Basumallik, Seung-Jai Min, and Rudolf Eigenmann, "Towards OpenMP execution on software distributed shared memory systems," *Int'l Workshop on OpenMP: Experiences and Implementations (WOMPEI'02)*, Lecture Notes in Computer Science, #2327, Springer Verlag, May, 2002, pp. 457-468.

[7] Y. Charlie Hu, Honghui Lu, Alan L. Cox, and Willy Zwaenepoel, "OpenMP for Networks of SMPs," *Journal of Parallel and Distributed Computing*, vol. 60, no.12, Dec. 2000, pp. 1512-1530.

[8] Hongzhang Shan, Jaswinder P. Singh, Leonid Oliker, and Rupak Biswas, "Message Passing and Shared Address Space Parallelism on an SMP Cluster," *Parallel Computing*, vol. 29, no. 2, Feb. 2003, pp. 167-186.

[9] Franck Cappello and Daniel Etiemble, "MPI versus MPI + OpenMP on IBM SP for the NAS benchmarks," In proceedings of ACM/IEEE Conference on *Supercomputing*, Nov. 2000.

[10] Lorna Smith and Paul Kent, “Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code,” *Concurrency: Practice and Experience*, vol. 12, no. 12, Dec. 2000, pp. 1121-1129.

[11] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, Chris Dodd, “The Virtual Interface Architecture,” *IEEE Micro*, vol. 18, no. 2, Mar./Apr. 1998, pp. 66-76.

[12] http://www.mpi-softtech.com

[13] L. Iftode. ”Home-based Shared Virtual Memory”. Ph.D. thesis, Princeton Univ., Aug. 1998.

[14] Frank Mueller, "Distributed Shared-Memory Threads: DSM-Threads," *Workshop on RunTime systems for Parallel Programming*, Apr. 1997, pp. 31-40.

[15] Markus Pizka and Christian Rehn, “Murks-A POSIX Threads Based DSM System,” In Proceedings of *The International Conference on Parallel and Distributed Computing Systems*, Aug. 2001. pp. 642-648.

[16] Yang-Suk Kee, Jin-Soo Kim, and Soonhoi Ha, “Atomic Page Update Methods for OpenMP-Aware Software DSM,” submitted for publication.

[17] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon, *The Midway Distributed Shared Memory System*, CMU Technical Report CMU-CS-93-119, School of Computer Science, Carnegie Mellon Univ., 1993.

[18] Liviu Iftode, Jaswinder Pal Singh, and Kai Li, “Scope Consistency: A Bridge Between Release Consistency and Entry Consistency,” *ACM Symposium on Parallel Algorithms and Architectures (SPAA’96)*, Jun. 1996, pp. 277-287.

[19] J. M. Bull, “Measuring Synchronization and Scheduling Overheads in OpenMP,” In Proceedings of *European Workshop on OpenMP (EWOMP’99)*, Sep. 1999.

[20] Hee-Chul Yun, Sang-Kwon Lee, Joonwon Lee, Seungryoul Maeng, “An Efficient Lock Protocol for Home-based Lazy Release Consistency,” *International Workshop on Software Distributed Shared Memory System*, May 2001, pp.527-532.

[21] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow, ”*The NAS Parallel Benchmarks*”. Technical Report, NAS-95-020, 1995.

[22] Joseph Robicheaux, http://www.openmp.org/samples/jacobi.f, 1998.

[23] Bill Magro, Kuck, and Associates, http://www.openmp.org/samples/md.f, 1998.