OpenMP Extension to SMP Clusters

Yang-Suk Kee

Computer Science & Engineering, University of California San Diego
9500 Gilman Drive, La Jolla, CA 92093-0404
yskee@csag.ucsd.edu

Abstract

Easy-to-use programming paradigm for high performance computing has been a challenging issue in the parallel computing community. This paper discusses on the approaches to apply the OpenMP programming model to SMP (Symmetric Multi-Processor) clusters using SDSM (Software Distributed Shared Memory). Especially, the major focus of this paper is on the challenges that the prior studies faced and on their solution techniques. The major obstacles are thread-unsafe memory access, slow inter-process synchronization, and excessive remote page accesses, which stem from the page-based memory consistency mechanisms of the traditional SDSM systems. Exploiting message passing primitives explicitly for the OpenMP synchronization and work-sharing directives enables light inter-process synchronizations while the techniques such as variable privatization to reduce the shared address space and selective data touch and migratory home to exploit data locality avoid unnecessary page migrations significantly. Through the evaluation of an OpenMP micro-benchmark program and a real application, an exemplary system, ParADE that leverages these techniques achieved scalable performance on small-scale SMP clusters.

A programming model provides a high-level abstraction of underlying hardware system to application programmers. For cluster systems, message passing models have been dominant since the models go with the distributed memory architecture of clusters, in which each processor has its private memory and communicates each other by explicit message exchange. MPI (Message Passing Interface) is a standard message passing programming model, which provides functional and performance portability. Despite high performance, a major challenge in MPI programming is programming complexity; every detail about programming logic, communication, and synchronization is up to the application programmers. This programming complexity is critical to the application programmers who are not aware of details of underlying hardware systems.

In contrast, a Shared Address Space (SAS) programming model enables easy programming in that processors see a same address space and communicate each other using the conventional read/write memory operations. OpenMP has been widely accepted as a standard SAS programming model. OpenMP provides a high-level interface to thread programming and enables the application programmers with an easy, simple way to develop parallel programs. OpenMP consists of the compiler directives that declare work-sharing algorithms, synchronization methods, and scope rules and the runtime library that provides auxiliary primitives for synchronization and information query. The application programmers can parallelize time critical sequential codes inserting the directives. Even though OpenMP is originally proposed for single shared memory multiprocessor systems, as cluster systems have become a viable platform for high-performance computing, extending the model to clusters is demanding.

Challenges in OpenMP for Clusters

An intuitive, straightforward approach to extending OpenMP to SMP clusters is exploiting SDSM (Software Distributed Shared Memory) systems. An SDSM system provides a shared address space to users transparently across distributed memory. The previous studies on SDSM have revealed that high synchronization overhead, long memory access latency due to remote pages, and heavy network traffic due to page migrations are the major performance bottlenecks. These bottlenecks would be critical when an SDSM system is used for the OpenMP extension.

A research group at Rice University conducted the first study on OpenMP for SMPs, using a multi-threaded version of the TreadMark SDSM system. They expanded the original system for multi-threading and implemented an OpenMP compiler for the new system. Through the intensive evaluation of the system against MPI, they confirmed that the performance bottlenecks in the SDSM system were still the major sources of poor performance.

The common challenges that the approaches based on SDSM face when extending OpenMP to SMPs can be summarized as follows.

- Thread-safe memory access: Since most conventional SDSM systems are single-threaded, an SDSM system for OpenMP should support multi-threads efficiently. Specifically, thread-safe memory access is a key requirement of the new SDSM system.
- Synchronization overhead: Synchronization operations across cluster are very expensive. Moreover, OpenMP programs experience more synchronizations than the corresponding MPI programs because many work-sharing and synchronization directives enforce implicit barriers at the end of execution.
- Consistency protocol overhead: Most SDSM systems preserve memory consistency in the unit of page. An access to shared address space includes resolving the location of up-to-date page, fetching the page from its owner, bookkeeping for the updates, etc, which increase memory access latency and network traffic due to page migrations.

Solution techniques

The focus of this paper is on the techniques to support OpenMP efficiently on top of SDSM systems and new SDSM requirements, not on those to efficiently implement an SDSM system itself.

● Thread-safe memory access

Let me discuss a new requirement of SDSM first. Most conventional SDSM systems implement the memory consistency protocols at the user-level using page-based virtual memory protection mechanisms. Initially, the system preserves a segment of application virtual address space for the shared address space and prohibits accesses to it. When an application accesses a page in the shared memory region, the SDSM system detects the access by catching a segmentation fault signal generated by the operating system. Then, a user-defined segmentation fault handler implemented in SDSM system performs a series of operations and fetches the most up-to-date page from the owner(s) of the page. This page fetch operation is atomic from the application perspective because the 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 perfectly in a multi-threaded environment when multiple application threads compete to access a page simultaneously. On the first access to an invalid page, the system sets the access permission of the page writable in order to fetch the most up-to-date page. Unfortunately, this change of access permission also allows other threads to access the same page freely. This constitutes the **atomic page update problem**.

The cause of this problem is that both the system and the application share the same virtual address space. Virtual pages can have different access permissions even if they might reside in the same physical page. Therefore, a general solution can be to partition the virtual address space for the application and the system but to make the virtual pages in the two address spaces reside in the same physical pages. Then, the system can update the physical pages through the virtual pages in the system address space while controlling accesses to the virtual pages in the application address space.

Most SDSM systems use the file mapping mechanism to implement this general solution. A UNIX mmap() system call maps a file to a certain memory region in the virtual address space. Then, the user can access the file using normal memory operations. Moreover, a file can be mapped to different address spaces at the same time. Similarly, three other methods can solve this problem as well. First, the System V shared memory mechanisms can create multiple address spaces. A shmget() system call creates a shared object in the kernel, and a shmat() system call attaches it to the application virtual address space. That object can be attached to different virtual addresses at the same time. Second, the process fork mechanism is another way to create two address spaces. When forked, a child process inherits the runtime image of the parent process including the page tables. Since the Copy-On-Write policy is not applied to the shared memory segments (e.g. mapped files, System V shared objects), the child share the same physical memory with the parent. Lastly, a new system call (e.g., mdup() in ParADE system) can be plugged into the operating system to implement this solution directly.

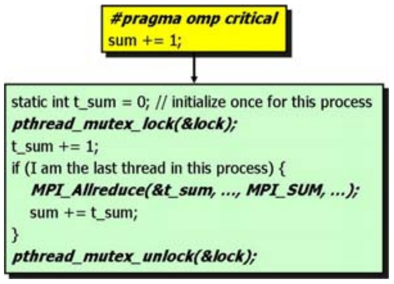
Even though almost all the SDSM systems are based on UNIX, Windows systems also provide similar virtual memory protection and process fork mechanisms and APIs for file mapping. All the methods have comparable performance while it is not always possible to implement them in certain operating systems due to various constraints of the operating systems.

● Efficient synchronization

Since synchronization operations are expensive, and they suppress concurrency in applications, efficient implementation of the operations has been one of the major issues in parallel computer architecture. To safely enter a critical section in conventional SDSM systems, a process must acquire a lock, negotiating with the owner of the lock. This locking mechanism is very expensive and critical to performance especially when an SDSM system uses a slow interconnect network.

An interesting observation from scientific OpenMP programs is that many code blocks protected by synchronization directives have regular forms, and they are statically analyzable at compile time. Another observation is that collective communication primitives perform an implicit global synchronization. These give us new opportunities to reduce the synchronization overhead. First, collective communications bypass the complicated memory consistency mechanisms, but they can simulate the behavior of the code block in the critical section. Second, collective communications can remove the barriers that the synchronization and work-sharing directives impose at the end of execution because it performs a kind of global synchronization implicitly even though its semantic does not enforce any barrier.

For example, a `critical` directive provides mutual exclusion between threads, and in most cases it is used to reduce non-scalar variables. As illustrated in Figure 1, a `critical` directive and its associated code block can be translated to a new code block using a thread lock and a collective communication operation. First, each process initializes the variable with the identity value (0) for the function (+). Next, the threads under the control of a thread lock execute the code block and produce a partial result. Then, a collective communication operation reduces the partial results of all processes, and finally each process adds the reduced value to the original one. Overall, the new codes simulate the behavior of the original code block and produce the same result.



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

```
static int t_sum = 0; // initialize once for this process
pthread_mutex_lock(&lock);
t_sum += 1;
if (I am the last thread in this process) {
    MPI_Allreduce(&t_sum, ..., MPI_SUM, ...);
    sum += t_sum;
}
pthread_mutex_unlock(&lock);
```

Figure 1. Translation of a `critical` code block

● Variable privatization

A `parallel` directive is the basic directive that starts parallel execution. Any variables declared outside of `parallel` regions but accessed within them are shared among threads. An interesting observation from many OpenMP applications is that a significant number of variables within parallel regions are read-only (e.g., variables for array size and problem size). Another observation is that for certain shared arrays, the read and write references access disjoint parts of the array from different nodes, and a node reads from the region of the array that it writes to (e.g., array that stores temporary results of computation). That is, the array is partitioned, and each segment of the array is exclusively accessed by only one node. Even though these variables do not need to preserve consistency, the memory consistency protocols of SDSM system create unnecessary network traffic.

Privatizing variables in these categories can reduce the shared address space for the system to manage. For the read-only variables, redundant local computation against private variables can avoid synchronization; the OpenMP compiler identifies the read-only variables within `parallel` regions and checks if any statements in the preceding serial regions write these variables. If any, execute those statements redundantly on all nodes. These writes in the serial regions do not require any following barrier synchronizations. However, if these writes are associated with other shared variables, a barrier must precede to these references.

In comparison, an exclusively accessed array can be transformed to private arrays on each node. The compiler allocates a private instance of the array for each node that has the same size to the segment accessed by each node. This technique enables fast access to the array avoiding expensive remote page accesses. Furthermore, this eliminates the false sharing between the nodes that partition a page and reduces unnecessary network traffic. Finally, privatization reduces the shared address space and consequently lessens the overall consistency preservation overhead.

● Data locality exploitation

Most SDSM systems preserve consistency by exchanging information about which nodes updated each page between synchronization points. A node that modifies a shared page becomes a temporary owner of the page. These ownership changes significantly affect application performance because they are closely related to page migration.

A general idea to reduce page migration is to exploit data locality. To avoid inefficient access patterns, the program needs to be selective about which nodes touch which portions of the data. For example, SMP machines may prefer executing small parallel loops serially to avoid parallelization overhead. However, cluster systems may experience performance degradation if the data accessed in the loop is distributed across cluster. Instead, the nodes owning the data would perform computation in parallel; the performance gain caused by local access can exceed the penalty due to parallelization overhead.

To the contrary, another approach to exploiting data locality is to use a consistency protocol with migratory home. Most SDSM systems have a fixed home node for a page that has the most up-to-date page. If a frequent writer to a page is not the home, the page moves back and forth between the writer and the home for every cycle of modification and synchronization. The key idea of migratory home approach is to relocate the home of a page to the node that updates the page most frequently. An OpenMP compiler can give information about the access pattern of shared memory and help the runtime system decide right homes. Then, the writers can find the up-to-date pages locally after home migration, which reduces not only page access latency but also outstanding network traffic.

A Case Study: ParADE System

ParADE (Parallel Application Developing Environment) system was initiated to realize easy, high-performance programming for SMP clusters. ParADE exploits most aforementioned solution techniques to overcome the performance bottlenecks. Specifically, the intensive use of message passing primitives reduces synchronization overhead significantly.

● Architecture

As illustrated in Figure 2, the ParADE system consists of an OpenMP compiler and a ParADE runtime system. Further, two subcomponents, a multi-threaded SDSM and a thread-safe MPI library compose the runtime system. The OpenMP translator converts an OpenMP program into a multi-threaded message passing program using the ParADE APIs. The major focus of the translator is on exploiting the message-passing primitives for synchronization and work-sharing directives while the runtime system implements a shared address space across distributed memory and message passing primitives.

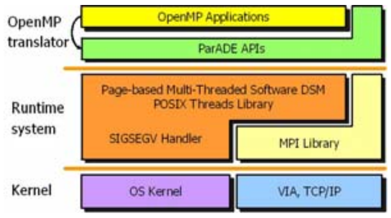


Figure 2. ParADE system architecture

● Key features

The unique feature of the ParADE system is an efficient translation of the synchronization and work-sharing directives. The following highlights how the translator converts some important directives.

◈ `Parallel` directive

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

According to the OpenMP specification, the default scope of variables in a `parallel` block is `shared`. However, this assumption is inappropriate when OpenMP is extended to clusters: the variables on different nodes cannot be shared for free. For better optimization and portability, it is highly recommended to explicitly annotate all the variables used in `parallel` blocks.

◈ Synchronization directives

The OpenMP specification defines several synchronization directives. A `critical` directive provides mutual exclusion between threads, and in most cases it is used to reduce non-scalar variables. As discussed earlier, this directive can be translated to a collective communication operation and a thread lock. ParADE executes a `critical` code block hierarchically: the threads of each process execute the code block under the control of a thread lock, and then the collective communication operation merges the partial results across the processes. In consequence, a collective communication operation together with a thread lock simulates the behavior of the code block without any costly lock over the network while the heavy lock primitives of SDSM result in high synchronization overhead.

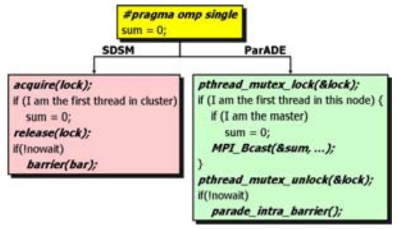
Similarly, an `atomic` directive ensures the atomic update of a specific memory location. An `atomic` code block must be one of the simple arithmetic expressions defined in the OpenMP specification. The ParADE translator regards an `atomic` directive as a special case of `critical` directive with well defined simple form.

Lastly, a `reduction` clause performs a reduction on scalar variables as a part of work-sharing directives. Similar to the `atomic` directive, a `reduction` clause must be one of the predefined expressions, and accordingly the translator converts the code block in the same manner as the `atomic` directive. One difference from the `atomic` directive is that a reduction clause can declare multiple variables. To efficiently handle multiple variables, the translator packs them into a new structure-type variable, defines a user-defined reduction operation for the new type, and applies the operation to the variable.

◈ Work-sharing directives

Work-sharing directives distribute workloads among threads. A `for` directive is used within the scope of or in a combined form of a `parallel` directive, and it defines a parallel execution of loop. Its associated loop scheduler determines and assigns the chunks of the loop to threads. In addition, an implicit barrier is enforced at the end of its execution, and `reduction` variables can be declared for the loop. Since a collective communication operation performs an implicit global synchronization, it combines and replaces the reduction operation and the implicit barrier.

In contrast, a `single` directive let only the first arriving thread execute its code block. A major use of this directive is to initialize shared variables. This directive also enforces an implicit barrier at the end of execution by default. Different from the variable privatization technique, ParADE replaces a `single` code block with a broadcast operation. That is, only one thread of the first process executes the code block, packs the modified variables into a single message, and broadcast the message to all processes. Since the other threads can proceed only after they have received a message from the first thread, a broadcast operation can replace the directive and the implicit barrier.



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

```
acquire(lock);
if (I am the first thread in cluster)
    sum = 0;
release(lock);
if(!nowait)
    barrier(bar);
```

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

Figure 3. Translation of a `single` directive

● Performance

A Linux cluster that consists of four dual-Pentium III 550Mhz SMP nodes and four dual-Pentium III 600Mhz SMP nodes was used to evaluate the performance of the ParADE system. Each node had 512 MB main memory, and they were 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 ran on each node. A GNU gcc compiler was used with the –O2 option.

To evaluate the performance benefit of using explicit message-passing operations, the average execution time of an OpenMP micro-benchmark program over TCP/IP was measured for ParADE and a single-threaded SDSM system. Especially, two directives, `critical` for synchronization and `single` for work-sharing were considered. Figure 4 shows that the intensive use of message passing operations reduce synchronization overhead significantly, and the gain becomes bigger as the number of nodes increases because SDSM needs to exchange control messages over slow network to grant a lock and the accesses are serialized.

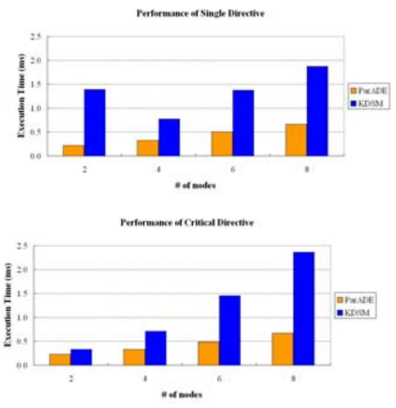


Figure 4. Performance of OpenMP micro-benchmark

Furthermore, a real application was used to evaluate the overall performance. The Helmholtz program solves a wave equation on a regular mesh using an iterative Jacobi method with over-relaxation. It repeats about one thousand iterations until an estimated error becomes smaller than a certain threshold. Each thread communicates with only its direct neighbors while every thread updates a shared variable competitively and checks if the termination condition is satisfied.

To demonstrate the benefits of OpenMP for SMPs, three configurations were used, and the results are shown in Figure 5.

- **1Thread-1CPU** Start the operating system in uniprocessor mode, and create one compute thread and one communication thread per node
- **1Thread-2CPU** Use the same configuration as 1Thread-1CPU except starting the operating system in multiprocessor mode
- **2Thread-2CPU** Use the same configuration as 1Thread-2CPU except creating two compute threads per node

For 1Thread-1CPU, the system spends more time in serving the page requests from remote nodes as the number of nodes increases because a single processor serves both computation and communication. In consequence, the excessive page migration due to large shared address space influences on performance adversely. In comparison, an additional processor lessens the burden on the compute processor due to handling the page requests; it reduces the page fetch latency and spares more CPU time for computation. In consequence, the application achieved scalable performance, which is a merit of a multi-threaded system over single-threaded one. However, assigning a processor to communication can waste resources. In many cases, compute threads are frequently suspended to fetch the non-cached pages. Multiple threads can overlap communication and computation and utilize CPU more efficiently. As illustrated in Figure 5, 2Thread-2CPU achieved better scalable performance than the others.

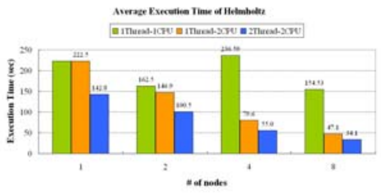


Figure 5. Performance of a wave equation solver

Future of OpenMP

This paper surveyed the challenges and solution techniques to extend OpenMP to SMP clusters. In addition, the ParADE system was overviewed as a case study, which enables easy, high performance programming for SMP cluster systems. Techniques such as variable privatization, message passing exploitation for synchronization directives, and high data locality, overcome the major performance bottlenecks in SDSM systems.

However, still OpenMP based on SDSM does not show comparable performance to pure MPI. The major efforts to improve performance are on reducing the network traffic due to memory consistency protocols. A very challenging approach is to directly translate an OpenMP program to an MPI program without the help of SDSM system. Such approaches as Omni/Scash and ParADE showed promising results that an intelligent compiler could analyze the uses of shared variables precisely and utilize message passing primitives directly.

Moreover, several orthogonal research issues can contribute to improving performance. First, even though the standard OpenMP provides various loop-scheduling algorithms, not all of them are appropriate for SMP clusters. Further studies on loop scheduling for SMP clusters will promise significant improvement in system performance. Another issue is to adapt the system configuration during runtime. As the Helmholtz results show, more processors do not always give better performance. For a given application, users want to find the best configuration to achieve the best performance. A proper number of processors and threads could be determined statically or dynamically by analyzing the behaviors of applications.

Finally, it is exciting that OpenMP is extended to other research fields. Mostly, OpenMP is considered as a promising programming model. Some noticeable examples are the studies on OpenMP for multi-processors on a chip in the embedded system community and OpenMP for computational Grids in the high performance distributed computing community.

Read more about it

- OpenMP Forum, “Openmp: A Proposed Industry Standard API for Shared Memory Programming,” Technical Report, Oct. 1997.
- Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano, and Yoshio Tanaka, Design of OpenMP Compiler for an SMP Cluster, European Workshop on OpenMP, Sep. 1999.
- Y. Charlie Hu, Honghui Lu, Alan L. Cox, and Willy Zwaenepoel, OpenMP for Networks of SMPs, Journal of Parallel and Distributed Computing, 60(12): 1512-1530, Dec. 2000.
- Seung-Jai Min, Ayon Basumallik and Rudolf Eigenmann, Optimizing OpenMP Programs on Software Distributed Shared Memory Systems, International Journal of Parallel Programming, 31(3):225-249, 2003.
- Yang-Suk Kee, Jin-Soo Kim, and Soonhoi Ha, ParADE: An OpenMP Programming Environment for SMP Cluster Systems, International Conference on High Performance Computing and Communication, Nov. 2003.

About the author

Dr. Yang-Suk Kee is a post-doctoral researcher at University of California, San Diego. Dr. Kee received his B.S. and M.S. degrees in Computer Engineering and his Ph.D. degree in Electrical Engineering and Computer Science from Seoul National University, Korea. He is one of the major contributors to the ParADE system. Contact him at <yskee@csag.ucsd.edu>