White Paper

Jay P. Hoeflinger
Senior Staff Software Engineer
Intel Corporation

Extending OpenMP* to Clusters

Table of Contents

Executive Summary

OpenMP* is a well-known parallel programming paradigm for shared-memory multiprocessors. In the past, OpenMP has been confined to Symmetric Multi-Processing (SMP) machines and teamed with Message Passing Interface (MPI) technology to make use of multiple SMP systems. A new system, Cluster OpenMP*, is an implementation of OpenMP that can make use of multiple SMP machines without resorting to MPI. This advance has the advantage of eliminating the need to write explicit messaging code, as well as not mixing programming paradigms. The shared memory in Cluster OpenMP is maintained across all machines through a distributed shared-memory subsystem. Cluster OpenMP is based on the relaxed memory consistency of OpenMP, allowing shared variables to be made consistent only when absolutely necessary.

Introduction

It is generally agreed that shared-memory parallel programming leads to lower program-development costs than parallel programming by message-passing. The advantage of being able to access data without thinking about whether it must be fetched first is a significant one, relieving the programmer of a level of complexity. Further, complicated data structures in modern algorithms often lead to irregular and rapidly changing patterns of data access, making it even more difficult to write an MPI program using them.

The OpenMP parallel programming language[1] has a further advantage over other shared-memory programming languages, because it is a directive language embedded in a serial program written in a base language (currently the base languages are C, C++, and Fortran), giving the programmer a clear way to retain the serial program intact. It is possible to build an OpenMP program that can be run and debugged serially, nicely partitioning serial program issues from parallel ones.

One problem with OpenMP and other shared-memory programming languages is that programs using these languages have been confined to run on a single multiprocessor machine, and if a large number of processors is desired, that machine can be very expensive. Bus-based multiprocessors don't scale well beyond about four processors. Machines supporting more than four processors typically have employed more sophisticated (and expensive) interconnects. The need for an expensive machine has limited the penetration of OpenMP in many markets. Message-passing programs have been successful on large clusters, because cheap interconnects have made them much less expensive than a shared-memory machine of the same processor count, and the largest clusters simply have no shared-memory counterpart.

A new software system from Intel has the potential to change the dynamics of this situation. Cluster OpenMP has extended the OpenMP programming language to make it usable on clusters. This change combines the advantages of the easier programming model with the advantage of usability on cheaper hardware.

Cluster OpenMP currently runs on Itanium®-based platforms and on systems based on processors that support Intel® EM64T running Linux*, with support for sockets and the uDAPL verb API.7 Cluster OpenMP has been tested on Ethernet and Infiniband* interconnects, although other interconnects will be supported in the future.

In this paper, we point out key aspects of OpenMP, and then describe how Cluster OpenMP takes advantage of the OpenMP relaxed-memory model to hide communication latency. We describe how various OpenMP operations are implemented by Cluster OpenMP, as well as discuss performance considerations in a Cluster OpenMP program and show some performance results. Finally, we touch upon future plans for the Cluster OpenMP software.

OpenMP Programming

The OpenMP Execution Model

An OpenMP program executes according to a fork-join model. This means that an OpenMP program begins execution as a single thread, called the *master thread*, and when a `parallel` directive is encountered by that thread, execution forks and the parallel region is executed by a team of threads. When the `parallel` region is finished, the threads in the team join again at an implicit barrier, and the master thread is the only thread that continues execution.

This fork-join structure continues in a nested way, if necessary. If a parallel region is currently active and any thread in the team that is executing the parallel region encounters another parallel directive, execution forks again, another team of threads is formed, and those threads execute the new parallel region, then join at its end.

OpenMP provides a menu of synchronization constructs: critical sections for mutual exclusion, barriers to hold threads until all in a given team have arrived, atomic constructs to confer atomicity on individual operations, reduction operations, and a way of ordering code sections within a parallel loop. These synchronization operations make it possible to implement most of the desired forms of cooperation between threads.

The OpenMP Memory Model

OpenMP provides memory that is shared by all threads. Each thread reads and writes the shared memory directly, but the memory consistency is relaxed, in that writes to memory are allowed to overlap other computation. Similarly, under certain circumstances, reads from memory are allowed to be satisfied from a local copy of memory, referred to in the OpenMP 2.5 specification as the thread's *temporary view*. Each thread can also create *thread-private* variables that may not be accessed by any other thread.

Every variable used in a parallel region has an *original variable* by the same name outside the parallel region. A reference inside the parallel construct to a variable, determined to be shared by the `parallel` directive, is a reference to that variable's original variable. A reference inside the parallel construct to a variable, determined to be private by the parallel directive, is a reference to a variable of the same type and size as the original variable that is private to the thread.

The relaxed consistency of OpenMP memory is similar to *weak ordering*[2,3,4] memory consistency, with the OpenMP *flush* operation serving as a memory-synchronization operation. All reads and writes are unordered with respect to each other (except those that must be ordered due to data-dependence relations or the semantics of the base language). However, a read or write of a variable **is** ordered with respect to an OpenMP *flush* operation for that variable.

```
1:  A = 1

   . . .

2: flush(A)
```

Figure 1. A write to A in memory may complete as soon as point 1, and as late as point 2

The ordering of memory operations with respect to the flush operation must be obeyed by both the compiler and the machine. In Figure 1 above, the compiler is prohibited from reversing the order of the write to `A` and the `flush(A)`. Similarly, in the execution of the program fragment in Figure 1, the value written by the assignment statement at point 1 is required to be firmly lodged in memory, available to all other threads, before execution is allowed to continue past point 2. Future memory operations involving A would not be allowed to start until after the flush `(A)` completes.

The fact that OpenMP allows execution to continue while the write to `A` is still not complete allows the overlap of a memory write with computation (the "…" in Figure 1). Further, if a read of `A` occurs within the "…" computation, the OpenMP implementation is allowed to satisfy that read out of some cheap local storage, the temporary view, without going all the way to memory. The asynchronous writes, along with the ability to do cheap reads in certain circumstances, makes it possible to hide memory latency within an OpenMP program.

A flush operation must be executed on both the sending and receiving threads. Operations must be done in exactly the following order:

1. The value is written to a variable in memory by the sending thread.
2. The variable is flushed by the sending thread.
3. The variable is flushed by the receiving thread.
4. The value is read from the variable by the receiving thread.

A flush operation for all visible variables is implied at all barriers, all lock operations, and at entry to and exit from all parallel regions, within an OpenMP program. An atomic construct implies a flush operation for the location involved, both before and after the atomic operation.

DSM-based OpenMP Programming

The relaxed consistency of the OpenMP memory model makes it possible to implement a Distributed Shared-Memory (DSM) version of OpenMP efficiently. Implementations can hide the latency of accesses to remote nodes of a cluster by overlapping writes with other computation and by fulfilling reads from local memory instead of remote memory in certain circumstances.

Implementing OpenMP with Distributed Shared Memory

We have implemented a DSM system, called Cluster OpenMP, for supporting OpenMP programs that run on a cluster. The system is based on a licensed derivative of the TreadMarks*[5] software from Rice University. Beginning with version 9.1, the Intel® C++ Compiler for Linux and the Intel® Fortran Compiler for Linux are available with Cluster OpenMP.

Cluster OpenMP extends OpenMP with one additional directive – the `sharable` directive. The `sharable` directive identifies variables that are referenced by more than one thread. These variables are the ones that are managed by the DSM system.

Certain variables are automatically made `sharable` by the compiler, without the need for a sharable directive. Any variable that is allocated memory space on the stack in a routine, and then is used as a shared variable in a parallel region in that same routine, is automatically made sharable by the compiler. In C or C++, the same applies to any pass-by-value formal parameter used in a shared way in some parallel region inside the routine.

File-scope variables in C and C++ must be declared explicitly sharable. Global variables (Fortran) may be made sharable with a `sharable` directive or by a compiler option. Compiler options exist to make all COMMON variables, all module variables, and all SAVE variables sharable. Call-by-reference in Fortran presents a slight complication. If an expression is used as an actual argument at a call site, and then the value of the expression is used in a shared way somewhere in the call graph, the value of the expression must be made sharable. Doing this by hand would involve creating a new temporary variable at the call site, assigning the expression to the variable, declaring the temporary variable sharable, and passing the temporary variable as the actual parameter of the call. A Fortran compiler option exists to do all of this automatically for every expression used in a call site, regardless of whether the variable is actually used in a shared way.

In addition to identifying all sharable variables, all memory allocations from the heap (`malloc, calloc, etc.`), need to be scrutinized to determine whether the memory being allocated should be sharable or not. If it should be made sharable, the memory allocation call must be replaced by a call to the sharable memory equivalent (for example, `kmp_sharable_malloc` instead of `malloc`).

The DSM Mechanism

The task of keeping shared variables consistent across distributed nodes is handled by the Cluster OpenMP run-time library. Sharable variables are grouped together on certain pages in memory. The basic mechanism relies on protecting memory pages with an `mprotect` system call. When a particular page is not fully up-to-date, the page is protected against reading and writing. Then, when the program reads from the page, a segmentation fault occurs, and a SIGSEGV is delivered to the Cluster OpenMP software, which requests updates from all nodes that have modified the page since it was last brought up-to-date. The updates from each modifying node are applied to the page, the page protection against reading is removed, and the instruction is restarted. This time, the instruction finds the memory accessible, and the read completes successfully.

The page is still protected against writing, so that the Cluster OpenMP software can trap any modification to the page. When the next write to the page happens, a segmentation fault occurs again and is delivered to Cluster OpenMP. Since the page is being modified, Cluster OpenMP first makes a copy of the page (called a "twin"), and then removes all protection from the page. The twin makes it possible to determine all changes that were made to the page after removing all protection. At the next synchronization operation, the node will receive notice of all the pages that other nodes have modified, causing all of those pages to be protected against reading and writing again.

After the twin is made, further reads and writes on the page, prior to the next synchronization, happen without engaging the Cluster OpenMP software at all. This makes all of these reads and writes extremely cheap compared to the cost of those that cause the page to be brought up-to-date and cause the twin to be made. The higher the ratio of cheap memory accesses to expensive memory accesses, the better the program will perform.

The use of the cheap memory accesses is allowed by the relaxed memory consistency of the OpenMP memory model. The local memory of a node serves as the temporary view for each thread. At each synchronization operation, nodes receive notification about which pages have been modified on other nodes, invalidating those pages, and causing the next access to those pages to be an expensive operation.

Mapping OpenMP to the DSM Mechanism

First, we must define terminology for identifying the threads in a Cluster OpenMP program. Each node has a designated master thread (the node master thread), with the rest of the threads on that node designated as node worker threads. The nodes themselves are divided into the home node (where the program was launched) and the rest of the nodes (the remote nodes).

OpenMP barriers are implemented in Cluster OpenMP using a two-level structure. A barrier between threads is done within each node, then across nodes. The nodes exchange the lists of pages each has modified since the last synchronization, causing each node to protect the pages modified by other nodes.

OpenMP locks are likewise implemented in a two-level structure. Threads compete for a lock both within a node and across nodes. It is faster to grant a lock to a thread on the same node than it is to grant the lock to a thread on a remote node. However, starvation can occur if the lock is always granted on-node first. Therefore, the system must strike a balance, by attempting to hand off a lock within a node first for a limited number of times, then to a thread on another node.

Reductions are implemented as part of the barrier code, and are therefore done in two stages – within a node and across nodes. An up-call to a routine containing the reduction code is executed on each thread as part of the barrier arrival process.

The flush operation is a critical part of OpenMP. Flush provides the synchronization with memory that the OpenMP memory consistency mechanism depends upon to make access to shared memory possible. Flushes of all visible variables are implied as a part of an OpenMP barrier, OpenMP locks, on entry to and exit from a critical section, and on entry to and exit from an atomic operation. The purpose of the flush is to make modifications to memory by any given thread visible to all other threads. For Cluster OpenMP, this is equivalent to sending a write notice from a thread that did a modification to a thread that needs to see the modification.

Therefore, write notices from all threads are sent to all other threads at barriers. Write notices are also sent from a node granting a lock to a node receiving the lock, because modifications made to data protected by the lock need to be visible to the next thread acquiring the lock. Locks are used to implement critical sections and even atomic operations, to guarantee the appropriate visibility of sharable data modifications when these operations are performed.

The explicit flush directive is implemented similarly by acquiring a "flush lock." This makes the flushed data visible to the other threads that do a flush. Cluster OpenMP makes no attempt to implement the flush of specific variables, although it would be possible to do so by only passing write notices of pages that cover the list of variables to be flushed. Therefore, with Cluster OpenMP, every flush is a flush of all variables.

Performance Considerations for Cluster OpenMP

The Cost of Memory Operations

As described above, for Cluster OpenMP, some memory operations are much more expensive than others. To achieve good performance with Cluster OpenMP, the number of accesses to unprotected pages must be as high as possible, relative to the number of accesses to protected pages. This means that once a page is brought up-to-date on a given node, a large number of accesses should be made to it before the next synchronization.

In order to accomplish this, a program should have as little synchronization as possible, and re-use the data on a given page as much as possible. This translates to **avoiding fine-grained synchronization**, such as atomic constructs or locks, and **having high data locality**.

The OpenMP memory model allows individual reads and writes to memory to be done in any order, as long as the synchronization operations (flushes) are done in a strict order – the same order in which they appear in the original user's program. The lack of ordering between reads and writes to memory makes possible their concurrent execution, but all flushes in a program must be serialized, adding overhead to the program.

Latency to L1: 1-2 cycles
Latency to L2: 5 - 7 cycles
Latency to L3: 12 - 21 cycles
Latency to memory: 180 – 225 cycles
Gigabit Ethernet latency to remote node: ~28000 cycles
InfiniBand* latency to remote node: ~23000 cycles

Figure 2. Itanium® processor latency to cache and memory compared with messaging latency to remote nodes

Figure 2 shows the number of processor cycles required for an access to different levels of cache and the latency to access a value in the memory of a remote node. This shows that access to the memory of a remote node is approximately 100 times slower than access to the local memory, and thousands of times slower than access to a value in cache. This comparison should drive home the point that local, rather than remote, memory should be used as much as possible.

RMS Workloads

Obviously, not all applications are suitable for Cluster OpenMP. An application with fine-grained synchronization is forced to spend a large percentage of its time fetching data from remote nodes and paying thousands of times more for memory accesses than it would on an SMP. An important consideration for a potential user of Cluster OpenMP is whether the prospective application is suitable for Cluster OpenMP.

It turns out that an important class of applications is well-suited to Cluster OpenMP – the class recently referred to as RMS workloads[6]. The "RMS" stands for "recognition, mining, and synthesis." This class of application is typified by large volumes of data that must be processed by searching, pattern matching, rendering, synthesizing meaning, etc. The volume of data to be processed is large and getting larger all the time. This can be video data, scientific data, commercial data, etc. The raw data itself is essentially read-only. The conclusions being drawn from the data are usually much smaller in volume, and would be read/write. Access to the data is often unpredictable and irregular.

This set of characteristics works well with Cluster OpenMP. The read-only data can flow to the nodes where it is needed, and remain there, accessed from the local memory of each node. Any given piece of the read-only data may move to multiple nodes and be used in parallel. The small volume of read/write data may cause expensive memory accesses, but since there are so few compared to the number of cheap read accesses, performance can be quite good overall. The irregular accesses make it difficult to use an explicit messaging system, such as MPI, to program the application.

The performance of RMS workload applications should be reasonably good, the applications are relatively easy to write, and the application can be run on relatively cheap hardware (a cluster). This makes RMS workloads an excellent fit for a system such as Cluster OpenMP.

Performance Results

We have run some preliminary performance experiments with a set of applications on Cluster OpenMP. The applications came from prospective customers, at various companies and academic institutions. These applications include:

1. a particle-simulation code
2. a magneto-hydro-dynamics code
3. a computational fluid dynamics code
4. a structure-simulation code
5. a graph-processing code
6. a linear solver code
7. an x-ray crystallography code

We ran all of these experiments on two systems: with OpenMP on a 32-processor hardware shared memory machine using Itanium 2 processors, and with Cluster OpenMP on an Itanium 2 processor-based cluster. The processors of all the machines ran at 1.5 GHz with a 6-MB L3 cache. We then calculated speedup versus the best serial time for each machine and calculated the ratio of the speedups.

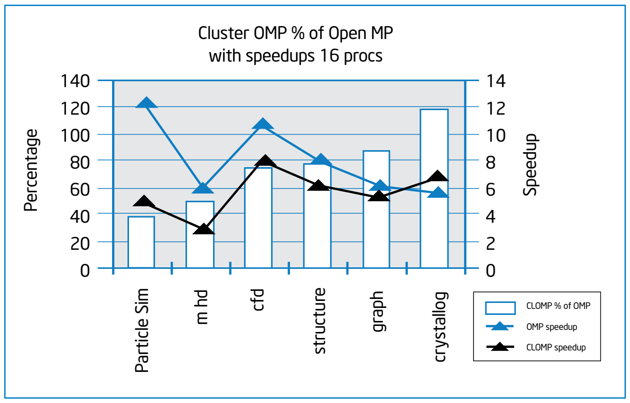

Figure 3. Raw speedup of Cluster OpenMP on a cluster and OpenMP on a hardware shared memory machine, plus speedup percentage of Cluster OpenMP versus OpenMP for a set of codes.

Figure 3 shows the raw performance of each, as well as the resulting speedup ratios. The significance of these results is that certain applications can achieve a large percentage of the performance of a hardware shared memory machine on a cluster, by using Cluster OpenMP. Five of the seven applications achieved at least 70 percent of the performance of the same application on a hardware shared memory machine.

Future Work

Future work on the Cluster OpenMP system will involve the use of advanced interconnection fabrics. As the DAPL interface is implemented on more interconnection fabrics, they will become available for use with Cluster OpenMP. The more important impact of advanced fabrics, however, is how they will influence the algorithms that are used in Cluster OpenMP.

Performance improvements should become possible by taking advantage of some new features of the advanced fabrics, such as RDMA (remote direct memory access) and remote atomic operations. RDMA should make it possible to directly load DSM data structures during messaging, eliminating or greatly reducing processor overhead. Remote atomic memory operations should make it possible to greatly reduce the performance impact of lock acquisition. Today, locks are implemented by sending and receiving messages between nodes. Some messages are necessary just to find the current owner of a given lock. Remote atomic memory operations should make it possible to implement cross-node locks with similar algorithms to those used in SMP systems today.

We anticipate redesigning some of the basic DSM algorithms with advanced fabrics in mind. We believe that these changes will serve to make Cluster OpenMP appropriate for even more applications.

Conclusion

Cluster OpenMP provides shared memory across a cluster for an OpenMP program. It takes advantage of the relaxed memory model of OpenMP to optimize the memory accesses in an OpenMP program. Cluster OpenMP does not perform well for all types of programs, but programs with certain characteristics can achieve reasonably good performance on a cluster, compared with attainable performance on a hardware shared memory machine. We showed performance results for a set of applications, showing that most were able to achieve greater than 70 percent of the performance of OpenMP programs run on a shared memory machine.

References

1. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 2.5. OpenMP Architecture Review Board (2005)
2. Adve, S.V., Gharachorloo, K. Shared Memory Consistency Models: A Tutorial, *IEEE Computer,* Vol. 29, No. 12, pp 66-76, 1996 (Also: WRL Research Report 95/7, Digital Western Research Laboratory, Palo Alto, California, 1995).
3. Hennessy, J.L., Patterson, D.A. Computer Architecture A Quantitative Approach, Second Edition, Morgan Kaufman Publishers, Inc, San Francisco, California, 1996.
4. Adve, S.V., Hill, M.D. Weak Ordering – *A New Definition. In Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2-14, May 1990.
5. P. Keleher, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems, Proceedings of the Winter 94 Usenix Conference, pp. 115-131, January 1994.
6. http://www.intel.com/technology/computing/archinnov/teraera/
7. http://www.datcollaborative.org/udapl.html#spec uDAPL:User Direct Access Programming Library.

For product and purchase information visit:
www.intel.com/software/products

0506/DAM/ITF/PP/500 312568-001