Proceedings of the National Conference on
Research and Development in Hardware & Systems
(CSI-RDHS 2008)
June 20-21, 2008, Kolkata, India

AOMP: An Agent Based OpenMP Programming

Mostafa Ghazizadeh[1],Hossein Deldari[2] , Mohammad Hadi Zahedi[3]
[1]Ferdowsi University of Mashhad ,Mostafaa350@yahoo.com,
[2]Ferdowsi University of Mashhad ,hdeldari@yahoo.com
[3]Ferdowsi University of Mashhad ,mhadi_zahedi@yahoo.com

Abstract

This paper discusses some of the salient issues involved in implementing the illusion of a shared-memory programming model across a group of distributed memory processors on a cluster of computers. This illusion can be provided by a Software Distributed Shared Memory (SDSM) system implemented by using autonomous agents.

Mechanisms that have the potential to increase the performance by omitting consistency latency intra site messages & data transfers are highlighted.

In this paper we describe the overall design/architecture of a prototype system, AOMP which integrates DSM and Agent paradigms and may be the target of an OpenMP compiler. Our initial goal is to apply this to Cluster Computing Applications.

1. Introduction

One of the main objectives of parallel software research has been the development of a standard programming methodology for the development of efficient programs for parallel machines. Such standardization would reduce the effort needed to train programmers, facilitate the porting of programs, and, in general, would reduce the burden of adopting parallel computing.

So far the most popular way of programming parallel machines, especially clusters and distributed memory machines in general, is to write SPMD (Single Program Multiple Data) programs and use Message-Passing Interface (MPI) library routines [4] for communication and synchronization. The second approach, which dominates when the target machine is a Symmetric Multiprocessor (SMP) with a few processors, is to use thread libraries or OpenMP [3] to write parallel programs assuming a shared memory model.

Shared memory is a simpler paradigm for constructing parallel applications, as it offers uniform access methods to memory for all user threads of execution. Therefore it offers an easier way to construct applications when compared to a corresponding message passing implementation.

Through the use of compiler directives, serial code can be easily parallelized by explicitly identifying the areas of code that can be executed concurrently.

We believe it is possible to use OpenMP to generate efficient programs for distributed memory clusters. Clearly, to achieve this goal the appropriate runtime systems, and compiler techniques should be developed.

A possible approach to implement OpenMP is to use a Software Distributed Shared Memory (SDSM) system such a TreadMarks [1] to create a shared memory view on top of the target system. The drawback is that the overhead typical of SDSMs can affect speedup significantly.

A way to reduce the overhead is to translate OpenMP programs so that the SDSM system is implemented by agents.

This can be achieved by applying compiler techniques similar to those developed by NavP [7]. This approach does not suffer from the same overhead problems as the SDSM approach in the case of faulting pages and moving pages from one node to another.

Providing use of agents through extensions to OpenMP will make it possible for the programmers to take advantage of the compiler in order to optimize OpenMP and also avoid the complexities of message-passing programming. The main goal in this work is gaining good performance while we provide easy programming environment without changes in programming syntax .So we can execute any program written with OpenMP directives on Cluster environment without changes in program.

The rest of this paper is organized as follows. Section 2 and 3 introduce related works and some of optimization techniques implemented by using agents. Section 4 details the proposed idea in using agents with combination of Clusters. Section 5 discusses our implementation of some OpenMP directives. Performance evaluation for some directives has shown in section 6 and in section 7 we have conclusion.

2. Related Works

Many commercial compilers for modern hardware architectures can compile OpenMP programs. There are also various open-source implementations of the OpenMP standard for SMPs. OdinMP/CCp [8], OmniOpenMP[16], and OpenUH[14]are source-to-source compilers that preprocess source code with OpenMP directives and create a source program that uses a threading library (OdinMP uses pthreads; Omni OpenMP can use different thread packages ;OpenUH can also compile to native Itanium code.) The upcoming GCC version4.2 is expected to also compile OpenMP (C/C++ and Fortran) code to native.

Two OpenMP specifications for Java are available. The JOMP [9] source-to-source compiler transforms a subset of the OpenMP standard to regular Java and uses the Java Threading API for parallelism. In contrast to JOMP, JaMP[10] compiler benefits from translating rather than rewriting the OpenMP directives, because the Jackal[11] compiler is aware of the parallelization applied. This enables various compiler optimizations, e.g., data race analysis, use of explicit send/receive operations instead of the DSM protocol, and the like.

There is little OpenMP-related work on clusters like JaMP. Intel Cluster OMP[12]extends the OpenMP specification by a special clause to share data between different cluster nodes. It is based on an extended version of the TreadMarks DSM [14]. Omni/SCASH [13] transparently executes OpenMP-enriched programs in the SCASH-DSM [15].

3. Agent based optimizations

Some optimizations previously used for SDSM are as follows:

3.1 Privatization optimization

In this kind of optimization , the focus is on read-only access to data.The data that have read-only accesses is privatized In general, two kinds of shared data can be treated as private data [5,6]. The shared data with read-only accesses in certain program sections can be made "private with copy-in" during these sections. Similarly, the shared data that are exclusively accessed by the same thread can be privatized during such a program phase.

Our system provides this kind of optimization by using agents. Firstly private data are agent's variable which is private to that agent. Secondly agents go toward data and locally access data they need. So shared data is also accessed locally and do not need any privatization.

3.2 Page Placement and data Distribution on the nodes

In this optimization, all shared variables of a program are allocated after all the threads are created and before all the slaves are suspended for the first time.

The first step makes all the pages of an allocation unit distribute across all the execution threads averagely because the allocation is done at the beginning of the execution. Here "threads" are used instead of "nodes", which means if several threads are running on one node, the pages associated with these threads are all located in this node.

The second step is to implement the first-touch placement based on home migration provided by JIAJIA [2]. If the page never migrated is referenced only by one thread in a parallel region, it will be migrated into the node on which the thread is running.

We implemented autonomous agents that migrate toward data, so we do not need to use this technique. Communication cost is almost reduced to migrating agents.

We also try to distribute computation at a coarse granularity level and uniformly at the start of execution so that agents do not need to migrate very soon. Fortunately, many algorithms exhibit some degree of locality of access and are coarse grained.

3.3 Overlapping data communication with computation

One of the other optimizations done in OpenMP is to overlap communication and computation. This optimization is used to reduce all spent time (communication time + computation time) for that process.

At runtime when an access does not have its data available on the same node (locally), the runtime optimizer tries to bring its data before finishing the computation. Here computation and communication overlapping is done.

Since in the agent based system, agents migrate toward data, it is not possible to overlap communication with computation unless we break the agent into two agents. Breaking the agent into two agents should be done at the point of where the agent needs a data not available on the same node. But here we should consider other circumstances such as dependencies of data.

According to NavP, programmer should distribute data. Then programmer with respect to the distribution write a program. One thing that programmer uses is Hop statement which is used by programmer to verify where the destination of migration is and when should an agent migrate.

An important disadvantage is caused by this kind of programming:the structure of the program should be changed if the distribution of data is changed.

In this system no remote data accessing is allowed and all accesses to data is done locally. And so is synchronization.

4. An Agent based OpenMP

As we discussed in the previous session, agent has advantages to reduce communication cost and result in good performance.

Our final goal is scaling OpenMP for distributed machines. To achieve this goal we need some changes in recent OpenMP. With respect to this, we decided to use agents to resolve scalability of OpenMP and increase its performance.

In AOMP, the concept of a mobile agent is used as a programming model. This is like many Java mobile agent systems, where the emphasis is on strong code mobility. Here, strong mobility means that computation migrates through the network. Here agents take the role of threads in usual OpenMP with the autonomous migration option. A preprocessor is used to create agents in the code instead of OpenMP directives.

We introduce a Master Agent which has the role of master thread and also has information of the distribution of data. It is also used to synchronize other agents at the barrier synchronization points.

The distribution of data has an important impact on performance. Therefore, at first user should distribute data between nodes. One of ways to distribute data among nodes is to distribute data statically by user as said [16].

Our DSM implementation is as follows. When Master Agent arrives at a worksharing block, it creates agents and distributes tasks among agents. Each agent need to know about the location of data it will require. So at first and before any migration, each agent asks Master Agent for the location of all data it needs. Agents get the data location addresses and save them in their local agent variables. Thereafter, each agent starts computations allocated to it and migrates where necessary. This can be seen in Fig.1 (a) and (b).

Here DSM is a local shared memory from the view of each node and it is global in the view of each agent because each agent knows where data are and can migrate to access that data. Note that as we said before, agents will migrate toward data but not the inverse due to the management complexity.

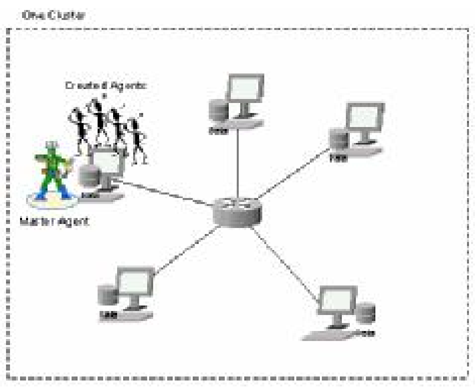
5. AOMP Directives

Since AOMP directives follow the OpenMP standard, its programming model is as expressive as the OpenMP programming model. An OpenMP programmer can use AOMP without learning a new syntax for directives.

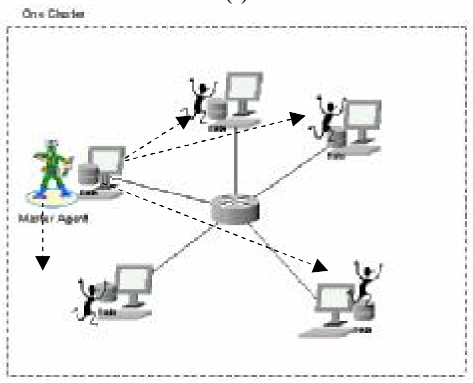
Since they are missing in the Java specification, AOMP provides its own implementation of pragmas. Moreover, we have provided a preprocessor to translate directives and add agents .

The parallel directive marks a section of a program as parallel. When an agent reaches a parallel region, it conceptually creates a team of agents that execute the region's code in parallel. At the end of each parallel region, there is an implicit barrier. Only when all agents executing the region reach the barrier, the Master Agent continues.

AOMP supports data-access types defined by OpenMP. For variables marked as shared, the same memory location in the DSM is used by all agents that are put to work on the parallel region. This means that if an agent wants to access a shared variable, it should migrate to the node where data is placed, so we do not have any false sharing and we are not worry about inconsistency. Private variables are really agent's variables which are local to that agent and other agents can not access them.



(a)



(b)

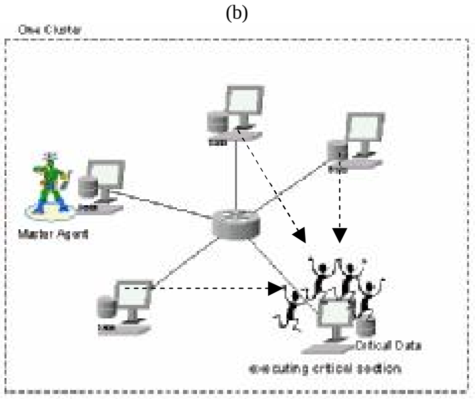


Figure 1. Migration of agents to other nodes in cluster:

(a) Agents ask Master Agent to know data locations
(b) Migration of agents to access data
(c) Migrating all agents to one node to execute a critical section

The iteration space of a loop can be distributed among a set of created agents by means of the Do directive. In a *for* statement, *init* value is the initialization expression of the loop, *cond* is a loop-invariant termination condition, and the *increment* value specifies how to increment the loop variable by some loop-invariant value. According to the OpenMP standard, the loop variable is privatized to each agent:

```
for (<init >; <cond>; <increament>) {
        // some code
}
```

AOMP supports multiplication and summation types of arithmetic reduction operations defined by the OpenMP standard. This is done by communication among Master Agent and the other agents.

With the single directive it is also possible to have code that is executed by only one agent. single directive has also an implicit barrier at the end of the construct. User-defined barriers can be created by means of the barrier directive to create program locations at which all agents wait for each other. When an agent arrives at a barrier point, it will send a brrier message to Master Agent. Master Agent collects these messages and increases a counter 1 by each message. When all of the agents arrive at the barrier point , Master Agent will aware them with a message .The critical directive can be used to mark critical sections that may be executed by only one agent at a time. To ensure the implementation of DSM as we said before, we were forced to implement this directive such that if agents are to execute a critical section, all of them should migrate to one node, the node which has the critical data (data that agents want to access in critical section). This is what is shown in Fig.1(c). Critical directive has also an implicit barrier at the end of the critical section.

6. Performance Evaluation

We have evaluated the performance of the AOMP implementation with a set of microbenchmarks. The microbenchmarks were run on a commodity cluster of Intel machines (2.53GHz) with 700MB of main memory per machine. The nodes are connected by Ethernet network.

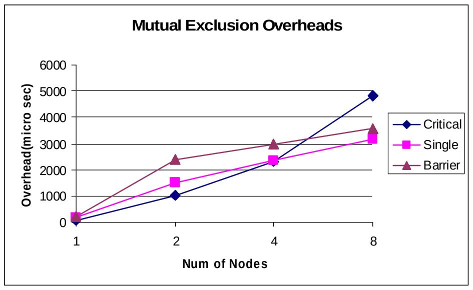
To determine the speed of the basic AOMP operations, we use the same set of micro-benchmarks that has been used to assess the JOMP implementation [17]. As suggested in [18], the microbenchmarks compute the overhead of a particular directive by measuring the runtime of the execution of an empty loop and the runtime of the same loop with the directive added. Fig. 2 shows the execution times of the individual AOMP directives.

The overhead of the barrier statement (see Fig.2(a)) is due to barrier implementation, for which the master node maintains the barrier's counter. Whenever an agent reaches the barrier, it communicates with the master agent and waits until a reply is received. The master node sends reply messages for all agents only after all agents have reached the barrier. Since for large numbers of agents, this kind of barrier algorithm become a bottleneck, a hierarchical implementation will be better.

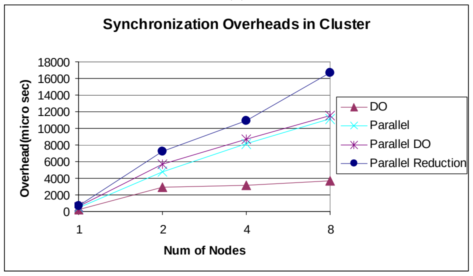
The time needed for a barrier consists of the time required to send 2 communication messages per node. So communications with Master Agent has a big latencies.

The single directive takes roughly the same time, as it is currently implemented as a check of the thread ID plus a barrier at the end of the construct (which is required by the OpenMP specification).

In comparison with JaMP our critical directive has a very high overhead, and this is because all agents should migrate to where the critical data exists. So we have a very high overhead in executing a critical section. And also whenever a agent encounters a critical region, it sends a request to the master node. If the region is currently not owned by any agent, the master node immediately replies. Otherwise, the grant message is deferred until the current owner leaves the critical region.



(a)



(b)

Figure 2.Overhead of AOMP directives

(a) Mutual exclusion overheads
(b) Synchronization overheads

The overhead caused by a parallel region is as shown in Fig.2(b). The overhead consists of (1) creation and initialization of the shared and private objects and also the agents, (2) a sequence of communications between each agent and Master Agent to get location addresses of the data they require, (3) migrating agents toward data, and (4) the final barrier.

The overhead of a for directive, mainly consists of a barrier at the start of for loop to wait for the initialization of the chunks. The second barrier at the end of the for region which is required to synchronize agents.

The overhead of a parallel for region approximately consists of the overhead needed to execute both parallel and for region.

For the parallel reduction, the overhead consists of the time needed for the parallel region and the time needed to combine the partial results of the worker agents. In Fig.2(b) reduction overhead is + reduction for a variable of type long.

One of the most important overhead reductions that we gain in this system is the overhead caused by consistency model.

7. Conclusion

In this paper we introduced a new OpenMP environment for programming on clusters. We have shown an implementation of OpenMP for Java with adding agent capabilities to it. A programmer can write a sequential Java program and enrich it with parallelization directives to make it a parallel AOMP program. We have also omitted consistency overheads exist in previous DSM models and have suggested extensions to the OpenMP specification to make it fit better with Java programs. The overheads of the individual AOMP directives are also small.

At the end we can say if this work is extended it can be a good alternative for many proposed models and also can be a good candidate to be used as a global programming environment for GRID.

8. References

[1] P. Keleher, A.L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In Proc. of the Winter 1994 Usenix Conf., pages 115–131,San Francisco, CA, January 1994.

[2] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol, in Proc. of the High Performance Computing and Networking (HPCN'99), LNCS 1593, pp. 463-472, Springer, Apr. 1999.

[3] OpenMP Application Program Interface, 2008 .http://www.openmp.org/.

[4] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, July 1997.

[5] A. Basumallik, S.-J. Min, and R. 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.

[6] R. Eigenmann, J. Hoeflinger, R. Kuhn, D. Padua, A. Basumallik, S.-J. Min and J. Zhu, Is OpenMP for GRIDs? Workshop on Next-Generation Systems, Int'l Parallel and Distributed Processing Symposium (IPDPS'02), May, 2002.

[7] LeiPan, Ming Kin Lai, KojiNoguchi, Javid J.Huseynov, Lubomir Bic, and Michael B. Dillencourt. Distributed parallel computing using navigational programming. International Journal of Parallel Programming, 32(1):1–37, February 2004.

[8] C. Brunschen and M. Brorsson. OdinMP/CCp - a Portable Implementation of OpenMP for C.
Concurrency and Computation: Practice and Experience, 12(12):1193–1203, 2000.

[9] J.M. Bull and M.E. Kambites , JOMP — an OpenMP-like Interface for Java. In Proc. of the ACM 2000 Java Grande Conf., pages 44–53, San Francisco, CA, USA, June 2000.

[10] M. Klemm, M. Bezold, R. Veldema, and M. Philippsen. JaMP: An Implementation of OpenMP for a Java DSM. In M. Arenaz, R. Doallo, B.Fraguela, and J. Tourino, editors, Proceedings of the 12th Workshop on Compilers for Parallel Computers, pages 242–255, A Coruna, Spain, January 2006.

[11] R. Veldema, R. Bhoedjang, and H. Bal. Jackal, A Compiler Based Implementation of Java for Clusters of Workstations. Technical report, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, Netherlands.

[12] J.P. Hoeflinger. Extending OpenMP to Clusters.http://www.intel.com/cd/software/products/ asmona/eng/compilers/285865.htm, 2006.

[13] Y. Ojima and M. Sato. Performance of Cluster-enabled OpenMP for the SCASH Software Distributed Shared Memory System. In Proc. of the 3rd Intl. Symp. on Cluster Computing and the Grid, pages 450–456, Tokyo, Japan, May 2003.

[14] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. Concurrency and Computation: Practice and Experience (this issue).

[15] H. Harada, Y. Ishikawa, A. Hori, H. Tezuka, S. Sumimoto, and T. Takahashi. Dynamic Home Node Reallocation on Software Distributed Shared Memory. In Proc. of the 4th Intl. Conf. on High-Performance Computing in the Asia-Pacific Region, pages 158–163, Bejing, China, May 2000.

[16] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In Proc. of the 1st European Workshop on OpenMP, pages 32–39, Lund, Sweden, September 1999.

[17] J.M. Bull, M.D. Westbed, M.E Kambites, and J. Obdrzealek. Towards OpenMP for Java. In Proc. Of the 2nd European Workshop on OpenMP, pages 98–105, Edinburgh, Scotland, U.K., September 2000.

[18] J.M. Bull. Measuring Synchronization and Scheduling Overheads in OpenMP. In Proc. of 1st European Workshop on OpenMP, pages 99–105, Lund, Sweden, October 1999.