Extending OpenMP for Agent Based DSM on GRID

Mahdi S. Haghighi, M. Hadi Zahedi, Mustafa Ghazizadeh A and Farnad Ahangary
Ferdowsi University of Mashhad, Mashhad, Iran
Haghighi@ieee.org, Zahedi@ieee.org, Mo_gh@mail.um.ac.ir and ahangary@um.ac.ir

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 from a cluster through to an entire GRID. This illusion can be provided by a distributed shared memory (DSM) system implemented by using autonomous agents. Mechanisms that have the potential to increase the performance by omitting consistency latency intra site messages and data transfers are highlighted.

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

Keywords—Distributed shared memory; AOMPG; Agent; GRID.

I. INTRODUCTION

Development of a standard programming methodology to write efficient programs for all classes of parallel machines is one of the main purpose of parallel software research. In this way training programmers, porting of programs would be easy, and, in general, it 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. OpenMP resembles HPF because of its reliance on directives.

However, the OpenMP standard differs from HPF in that it deals almost exclusively with control flow and synchronization and has practically no mechanism to control data placement and alignment.

Of the two approaches, the former is seen as a low level programming model to the point that MPI has been called the assembly language of parallel programming. Clearly, message-passing programming has the advantage that it gives the programmer direct and explicit control of the communication between threads and provides simple mechanisms to transfer data structures between distributed memory machines to enable the construction of high performance and highly scalable parallel applications.

However, the complexity of subroutines that arise when arrays are distributed manually and the difficulty of changing distributions and, in general, modifying message-passing parallel program makes the message-passing programming model inconvenient and costly. So there is considerable burden placed on the programmer whereby send/receive message pairs must be explicitly declared and used, and this is often the source of errors. Implementations of message passing paradigms exist for GRID too [19].

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.

The disadvantage is limited scalability. But nonetheless, vast quantities of parallel codes have been written in this manner. OpenMP, promoted by multiple vendors in the high performance computing sector, has emerged as a standard for developing these shared memory applications.

Through the use of compiler directives, serial code can be easily parallelized by explicitly identifying the areas of code that can be executed concurrently. This parallelization of an existing serial application can be done in an incremental fashion. This has been an important feature in promoting the adoption of this standard among parallel application developers.

Both OpenMP and thread libraries bring the programming advantages of the shared memory model, but OpenMP has the additional advantage of enforcing a nested structure in the parallel program. This last consideration gives OpenMP an advantage over thread libraries.

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

A possible approach to implement OpenMP is to use a Software Distributed Shared Memory (SDSM) system such as TreadMarks [1] to create a shared memory view on top of the target system. By following this approach the implementation of OpenMP on distributed memory systems becomes equivalent in difficulty to implementing OpenMP on an SMP machine. 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.

To this end, we propose to extend OpenMP to allow users indirect use of agents. This is important because the compiler, especially the earlier versions of it, is not expected to adequately handle all conceivable situations. 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 GRID environment without changes in program.

Our main goal in this paper is introducing our system and OpenMP directive implementations.

The rest of this paper is organized as follows. Section II provides related works and section III introduces some of optimization techniques implemented by using agents. Section IV details the proposed idea in using agents with combination of GRID. Section V discusses our implementation of some OpenMP directives. Performance evaluation for directives has shown in section VI and section VII is our conclusion.

II. 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 version 4.2 is expected to also compile OpenMP (C/C++ and Fortran) code to native.

We are aware of no OpenMP specification in Java for GRID. 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 Jaclak [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 [1]. Omni/SCASH [13] transparently executes OpenMP-enriched programs in the SCASH-DSM [15].

III. AGENT BASED OPTIMIZATIONS

Some of optimizations previously used for SDSM are as follows:

A. 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.

B. 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 JAJIA [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. By using agents 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.

C. 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. For doing this, inspectors at compile time and runtime are needed to do the work of restructuring code and reordering the accesses to arrays. Then with respect to reordered access to arrays, program accesses data. 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 the agent where it needs a data not available on the same node. But here we
should consider other circumstances such as dependencies of data.
As proposed in NavP agents can be a good alternative for page faults and migrating data towards code. That is agents (code) migrate to the nodes having data and locally execute there.
NavP says that programmer should distribute data him/herself; Then programmer with respect to the distribution of data 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.
So the first disadvantage of their system is that programmer should think exactly with respect to distribution of data. The other and also 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.

IV. AN AGENT BASED OPENMP PROGRAMMING FOR GRID COMPUTING

As we discussed in the previous session, agent has advantages to reduce communication cost and result in good performance and also has the effect of some optimizations. Here we show how we use agents for our purpose.

A. Agents as the main concept

Scaling OpenMP from distributed machines to GRID is our final goal. Some other models have developed so far, but their performance is low in comparison with MPICH-G2 [19] (which have the best performance until now). MPICH-G2 is based on MPI and message passing. The biggest problem here is the programming complexity especially on GRID.

To Scale OpenMP from distributed shared memory to GRID computing environment we need some changes in recent OpenMP. With respect to this, we decided to use agents to increase performance and scalability.

One debate that remains is how to apply this to GRID. At first we need a suitable distribution of data over clusters and then migrating agents between nodes of clusters.

To achieve this purpose, we used the three-layered architecture proposed in [21] which is shown in Fig.1.

In first layer (transmission layer) message transmission (sending and receiving messages) between nodes in GRID has been implemented.

Communication layer, propose methods that agents can communicate with each other. The method we used in our system is a hierarchical communication architecture shown in Fig.2.

Agent Directors (ADs) are the points of communication between agents with each other and with Agent Directors. Each cluster in GRID only has one AD that can communicate with other ADs in other clusters. Agent Management System (AMS), which has the responsibility of providing information about where agents and other ADs exist, are associated with ADs. Each AMS also registers all data location information in a computation. This information is used by agents to access the data they need. In a collection of clusters a Master AD manages all ADs.

When an agent wants to access a remote data (by migration) it should have known the location of the data. So at first step after creating agents, they ask AMS where the data they need are placed. Here to reduce the number of communications, each agent gets its required data placement information from AMS only at the time it is created and carries that information with itself everywhere it navigates. In this way, agents move autonomously.

In this way shared data on all nodes are global for all agents and they can locate data required. So DSM system is provided here.

Top layer is the application layer and has the code that agents want to execute. An OpenMP program creates agents and gives each a block of code to execute.

![Fig.1 Three-layered architecture for agent communication](image)

![Fig.2 Communication between agents using hierarchical communication architecture](image)

B. Transparency of data access

We have provided transparency of data access as follows.

As we said in previous session each

Master AMS registers all data location information in a computation. This information is used by agents to access the data they need. As Fig.3 (a) depicts, at first program is divided to some parallel agents. Then they ask AMS where the data they need, are placed. This is done locally at the master node. So information about data locations is also given to that agent at the time of creating agents. Now if an agent wants to access a data, it looks up data location in its local information and with those information moves to where the data is placed.
Migrated agents can be seen in Fig. 3 (b). If data is located at
the node where agent is currently executing, no migration is
done.

![Diagram](image)

**Fig. 3 (a)** Program is divided to some parallel agents
(b) Agents migrating to where data is placed, using local information

**C. Cost of data-consistency maintenance**

The communication costs incurred in maintaining data
consistency can significantly degrade the performance of
DSM programs. When the DSM programs are considered over
a wide area network, this is more serious since the cost of
propagating updates over such networks is greater than that
over a local network. Therefore, if the performance of the user
application is to be optimized, minimizing the number of
messages which must be transferred over the GRID network is
essential.

Traditional DSM systems generally adopt a flat or
hierarchical architecture to perform data consistency
maintenance. In these architectures, each processor propagates
its data updates to all other processors holding copies of the
same data. Therefore, many update messages must be
transferred over the network when these processors are
distributed across different network domains (like GRID
environment), and hence the application performance is
seriously degraded. To address this problem, we can say that
our system has eliminated this data-consistency cost and its
alternative cost is migrating agents. To degrade this cost as we
said previously, distribution of data and computation should
be done 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 course grained.

**V. 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 (we call this
agent, Master Agent), 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. We will describe implementation of barrier
later.

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. As we said before agent migration
destination can be found by asking AMSs in each cluster.
Private variables are really agent’s variables which are local to
that agent and other agents can not access them.

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:

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

AOMP supports multiplication and summation types of
arithmetic reduction operations defined by the OpenMP
standard. This is done hierarchically by some communication
between agents in a cluster and the AD of that cluster and at
the end a communication between ADs can gather all the
reduction information. This means that firstly a reduction is
done in each cluster, and finally a reduction among cluster
ADs.

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
barrier message to AMS of that cluster. AMS of each cluster
collects these messages and increases a counter by 1 for each
message. When all agents in a cluster arrive at the barrier
point, AMS will aware Master AMS which is the AMS
associated with Master AD, and when MAD found that all
agents has reached the barrier point, it will send a message to
AMs in each cluster and then those AMSs will aware agents
to continue their computation. 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). Critical directive has also an implicit barrier at the end of the critical section.

VI. PERFORMANCE EVALUATION

To simulate our work we used gridsim [20] toolkit as [21] has used and has been described in section IV.A. To evaluate performance we used a two-cluster GRID which nodes are distributed uniformly between them.

We have evaluated the AOMPQ directives and simulated a set of algorithms to show the performance. The simulation results are depicted in section VI.B.

A. Gridsim toolkit

GRID introduces a number of resource management and application scheduling challenges in the domain of security, resource and policy heterogeneity, fault tolerance, continuously changing resource conditions, and politics. The resource management and scheduling systems for GRID computing need to manage resources and application execution depending on either resource consumers’ or owners’ requirements, and continuously adapt to changes in resource availability.

In a GRID environment, it is hard and even impossible to perform a performance evaluation in a repeatable and controllable manner as resources and users are distributed across multiple organizations with their own policies. To overcome these limitations, GridSim have been developed which is a Java-based discrete-event GRID simulation toolkit [20] and can be programmed to simulate a GRID system, thus it offers support, in terms of classes, for simulating computing elements, storage elements, Virtual Organizations, etc. The toolkit supports modeling and simulation of heterogeneous GRID resources (both time- and space-shared), users and application models. In a GridSim GRID environment there can be multiple users executing applications concurrently in the simulated GRID so that contention for resources can be modeled. The network speed between resources can be specified so that transfer of data is realistic. Static and Dynamic schedulers can be modeled and here is support for statistical analysis of any operations performed in the system. It can be used to simulate application schedulers for single or multiple administrative domains distributed computing systems such as clusters and GRIDs. It provides primitives for creation of application tasks, mapping of tasks to resources, and their management.

The GridSim toolkit provides a comprehensive facility for simulation of different classes of heterogeneous resources, users, applications, resource brokers, and schedulers. These facilities of GridSim with our packages added to it, made it suitable for our goal of using agents to simulate our SDSM model in GRID simulation environment.

Building an ideal GRID-enabled software DSM system is hugely challenging since many problems must first be overcome, including those of heterogeneity, dynamics, resource allocation and data-consistency costs, etc. Since the problems of heterogeneity and dynamics, this study focuses specifically on the issue of communication costs incurred in accessing data, without consistency problem over the network.

B. Simulation results

We have evaluated the performance of the AOMPQ with a set of benchmarks.

To determine the speed of the basic AOMPQ operations, we use the same set of microbenchmarks 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.4 shows the execution times of the individual AOMPQ directives.

The overhead of the barrier statement (see Fig.4 (a)) is due to barrier implementation, for which the Master AMS maintains the barrier’s counter. Whenever an agent reaches the barrier, it communicates with the cluster AMS and waits until a reply is received. When all agents in a cluster arrive at the barrier point the AMS of that cluster sends its counter to Master AMS, and waits for reply. Master AMS reply messages only after all agents of the team have reached the barrier. Since this is a hierarchical communication model its cost is decreased.

The time needed for a barrier consists of the time required to send 2 communication messages per node in each cluster and 2 communication messages per AMS between each AMS and Master AMS intra cluster. Although in this hierarchical system number of communications between each agent and cluster AMS is high, but the expensive communications which is intra cluster is low and it is cost effective.

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).

Critical directive has a high overhead, and this is because all agents migrate to where the critical data exists. So we have a high overhead in executing a critical section. Whenever a agent encounters a critical region, first it migrates, and after arriving it sends a request to the AMS. If the region is currently not owned by any agent, the AMS immediately replies. Otherwise, the grant message is deferred until the current owner leaves the critical region.

The overhead caused by a parallel region is as shown in Fig.4 (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 AMS to get location addresses of the data they require, (3) migrating agents toward data, and (4) the final barrier.

The overhead of a Do(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 (Do) for region approximately consist 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.4 (b) reduction overhead is + reduction for a variable of type long.

One of the most important overhead-reduction that we gain in this system is the overhead caused by consistency model that we have omitted it with our DSM model.

We also used the parallel Java Grande Forum (JGF) benchmarks [22] section 2 to show speed-up gained.

Sparse Matrix Multiply computes (200 times) the product of two sparse N x N matrixes in compressed-row format. The main loop is simply divided between agents. The speed-up in Fig.5 (a) is the result. As the number of nodes increases, the relative data size per node decreases and so does the speed-up.

Series computes the first Fourier coefficients of the function f(x) = (x+1)^x. The most important component of the program is the loop over the Fourier coefficients. Each iteration of the loop is independent of other loops and the work can be distributed simply between processors. It mainly uses transcendental and trigonometric functions to compute the coefficients. The main loop is divided between agents by means of the parallel for directive. The computation of Series
is inherently parallel. After the chunks of computations have been divided between agents, only a migration would happen and no remote memory accesses are necessary. Approximately half of agents need to migrate between two clusters. Fig.5 (b) shows the Series benchmark speed-up on the nodes.

Crypt performs IDEA (International Data Encryption Algorithm) encryption and decryption an array with the length of N bytes. Crypt strongly depends on bit and byte operations. The main loop is divided among agents simply, because iterations are independent of each other and each agent receives only parts of the array which is independent of the others. The result has shown in Fig.5 (c).

SOR is a simple over-relaxation with 100 iterations on an N x N grid. As Fig.5 (d) shows, SOR has a reasonable performance, because the amount of shared data is smaller compared to the amount of computations. Since only two rows of the matrix are shared between neighboring agents, the overhead of migration to access remote data is not small compared to the other algorithms. But this is caused without the consistency maintenance cost.

Take this note into consideration: when agents located on different nodes want to access the same data variables, data consistency is maintained automatically by migrating code toward data using agents. Also, in this way consistency is preserved automatically.

VII. CONCLUSIONS

In this paper we introduce a new environment for programming in GRID. We have used OpenMP directives in Java and added agent capabilities to it. In this way we integrates DSM simulated with agents to provide a personal shared memory multiprocessor on computational GRIDs and migrating computations toward data. In this system, programmers use the concept of shared memory rather than message passing or even function calls to develop parallel applications on a computational GRID. This allows all OpenMP programs to be applied in a GRID environment.

A programmer can write a sequential Java program and enrich it with parallelization directives to make it a parallel AOMP program. The directives are expressed as OpenMP standard. We have also omitted consistency overheads exist in previous DSM models. We also show the overheads of the individual AOMP directives and the speedup of some JGF benchmark algorithms.

The simulations have shown that the proposed consistency maintenance method is effective in minimizing the number of data movement over the network. However, there are many other problems must also be considered, including data distribution, load balancing, and so on.

REFERENCES


