"Green" Multi-layered "Smart" Memory Management System[1]

Andriy Kot [1], Nikos Chrisochoides [2]
1) The College of William and Mary, PO Box 8795 Williamsburg VA 23187 USA,
kot@cs.wm.edu, http://www.cs.wm.edu/~kot
2) The College of William and Mary, PO Box 8795 Williamsburg VA 23187 USA,
nikos@cs.wm.edu, http://www.cs.wm.edu/~nikos

Abstract: *In this project, we investigate the feasibility of using outdated machines with slow processors for tolerating disk latencies for computation and data intensive parallel adaptive and irregular applications.*

Keywords: - Runtime, Software, Petaflops, Architectures, Out-of-Core, Parallel and High Performance Computing

1. INTRODUCTION

Memory speed increases rather slow as it is compared to the processor speed. Outdated machines while slower at the processor speed have about the same memory performance. That creates an opportunity to recycle older machines as a "smart" memory for newer ones. Reusing of mechanically non-recyclable computers also helps to make computing to be environmentally aware or "green".

We propose the design of "Green" Multi-layered "Smart" (GMS) memory management system based on a simplified version of the percolation model originally proposed for the HTMT Petaflops design [1]. Our approach is application-centric. The problem with explicit memory management for adaptive and irregular application is that their computation and communication patterns are variable and unpredictable at runtime. This results in using valuable memory space even if this means the system will remain idle.

Our parallel execution model treats parallel applications as a set of "small" executable fragments (in the HTMT design are called parcels). Parcel consists of a chunk of code and a number of input parameters. An input parameter can hold an actual data or data dependency. In this project, we assume SPMD model so we reduce the parcels to objects with a number of user-defined handlers attached to them. The type, number, and arguments of the handlers are determined at runtime and they are input dependent.

Our preliminary data indicate the maximum GMS overhead is 0.28% on the very fast machines, at the cost of 23.90% and 7.35% at the outdated machines of data servers and control unit respectively. In addition, we observe the same speedup as the traditional object-oriented, distributed computing approach. These are very encouraging data, considering the fact that the size of our test case is not sufficient to show the full potential of the GMS system.

2. ARCHITECTURE

The GMS system consist of three layers of hardware and software respectively: (1) the Data Servers (DS), (2) Control Unit (CU) and (3) Computing Engine (CE). Fig. 1 depicts the GMS architecture:

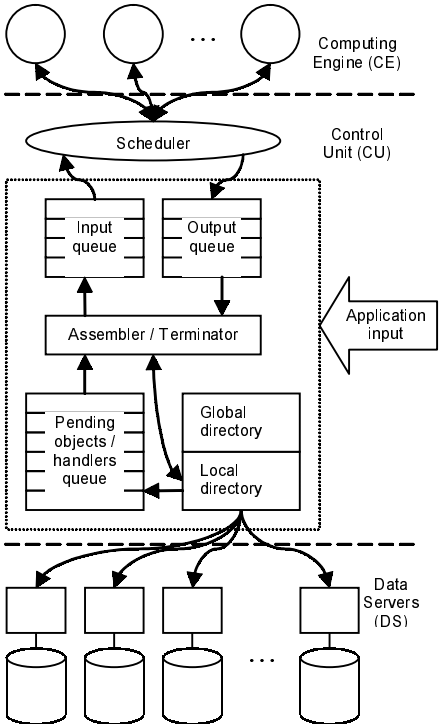

Fig. 1 – Subcluster organization.

The DS runs on a number of relatively slow (i.e., outdated in terms of their processors speed), but still useful in terms of memory speed machines. The DS plays the role of a "smart" storage subsystem. It is independent from the rest of the subsystems. This will allow the plug-and-play with different implementations in hardware and software. In the DS layer, the sub-system holds the application data (i.e. objects) until all of the dependencies of their handlers are resolved. Moreover, the DS layer stores all processed objects and the results of the application. For the current implementation of the DS, we use out-of-core scheme presented by Salmon et al [2]. In his paper, Salmon describes out-of-core way to do a parallel N-body simulation. He stores the data structures (arranged in the octrees) using the algorithm specific mapping between data objects and memory pages. Then, he uses the most recently used paging scheme with prioritized control. We adopted the most recently used paging scheme with prioritized control from Salmon's work. In our future implementations, we will also consider introducing the support detecting the effective

[1] This research is supported in part by NSF Career Award #CCR-0049086 and #EIA-9972853

mapping between the application data structures and memory pages.

The CU runs on big memory, fast network and faster than DS processors SMP machine (CU node should have very fast link, such as shared memory, to the application node) and controls the percolation flow and places the right application-defined handlers with the right data/objects in the right CE node at the right time. The CU is the most complex block of the system. It consists of a single k-way shared memory machine. We design the CU as a multithreaded unit. A directory in CU contains the information about the locations of all objects in the DS. The pending objects/handlers queue contains references to the objects that have pending handlers, so it is a queue of ready-to-execute active messages. The Assembler prepares objects for execution and is responsible for delivering them into the input queue. The Terminator destroys the objects that are finished, and then it frees resources and stores the results (for a subsequent use or iteration) to slower but larger memory (DS). The scheduler is not a separate block but its functionality is distributed between the Assembler, the Terminator, and the CE nodes.

Finally, the CE runs on number of very fast, but "low" in memory nodes (independent workstations or processors) relative to aggregate memory one can put together by using older and slower machines with disks for the DS subsystem. The CE schedules and executes the application-defined handlers to completion. An open research issue is the optimization of resources (cycles and bandwidth) of the CE nodes.

An important design issue is the scalability of our system using a large number of the semi-independent sub-clusters whose workload is balanced automatically (see

Fig. 2). However, in this paper we focus on the design of a single sub-cluster so that scalability will be possible with additional but minimal effort. A single sub-cluster can use two different networks: a relatively outdated and slow network for the DS layer and a faster network to connect the CU with the CE layers.

3. SOFTWARE IMPLEMENTATION

We use the DMCS (Data Movement and Control Substrate) [3] and MOL (Mobile Object Layer) [4] as a low-level communication systems which support AMs (Active Messages) [5] in the context of object/data movement (i.e., up and down movement from a subsystem to a subsystem) during the percolation cycle.

The DMCS provides single-sided communication, as get/put communication operations and remote procedure invocation or remote service requests (RSRs) DMCS's RSRs and communication operations invoke user-defined handler functions like AMs on target processors. DMCS forms the basis for both data migration and computation invocation in the GMS system.

The MOL extends the DMCS by providing a global namespace in the context of object mobility. Mobile objects are application-defined data objects and are not restricted to exist in contiguous memory. A mobile object may be referenced by any processor in the parallel system by using its associated mobile pointer, which is a system-wide unique identifier.

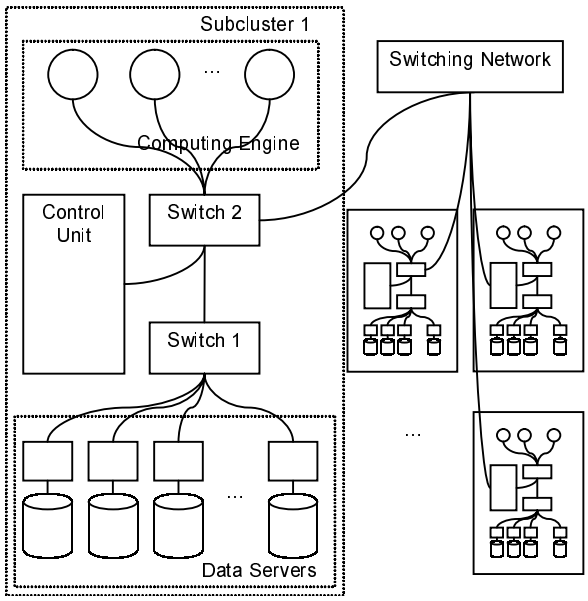

Fig. 2 – Hardware organization of the GMS cluster.

The MOL's communication operations extend the DMCS RSRs by allowing applications to invoke transparently computation handlers at the location of a mobile object, regardless of where it is in the parallel system. In this way, applications can deal directly with data objects without the tedious bookkeeping associated with maintaining up-to-date knowledge of each data object's current location.

4. PERCOLATION CYCLE

At bootstrap, the GMS specifies the roles to the nodes of the parallel machine. It assigns: (i) exactly one CU node and one node to be the front-end of the application, which is used by the application to interact with the rest of the system, (ii) N CE nodes and *M* DS nodes, depending on the user preferences and total number of available nodes.

The application node initially creates user objects and feeds the computation requests to the system. It is also responsible in resolving the object data-dependencies (at the user level in this version of the GMS implementation).

The percolation cycle has several stages:

1. The application injects objects into the system for execution. At this point, depending on the size of the objects and the load of the DS nodes, the system stores the object at the appropriate DS nodes. Their pending handlers are stored into the pending objects/handlers queue and the local directory is updated;

2. Assembler picks the objects (in some order) from the pending objects/handlers queue. It analyzes (in current implementation it just checks for the location of the object and any pending handlers ready for execution) the object and it queries the DS layer for the necessary data (e.g., an argument to object's handler might be another object), and then it assembles the necessary parts and puts them into the input queue;

3. Next the scheduler picks the now ready to execute objects from the input queue and assigns them to the CE nodes where they run to completion all their pending

handlers; after completion the objects and all of their associated data are sent to the output queue;

4. Finally, the Terminator picks the objects from the output queue and it stores them in the DS layer. If, in the mean time, there are new Active Messages with pending handlers it stores all of them into the pending objects/handlers queue and updates the local directory in CU.

5. PROGRAM EXECUTION

Next, we describe the execution and the percolation for application objects within the GMS system.

When an object is registered with the system, the corresponding GMS object consists of the two parts: the object itself as the user created it (to the system it is just a pointer to some data) and the meta data. Meta data contains object specific information (e.g., user functions for moving the object from one node to another) and the mobile pointer to the user data. After the object being created it is "released", which is the object data are transferred to some of the DS node and the meta data are transferred to the CU node. The mobile pointer that user gets after the object's creation points to the meta data rather than to the object's data itself.

After user created all needed (at present) objects, he/she calls objects functions. The call request (from now on, we will call it a message as in MOL) will be delivered by the underlying communication layer to the node where the meta data is residing. Meta data should be located on the CU node, since system transferred it there upon the "release" of the object.

Upon receiving the message, the CU checking whether the targeted object was involved in other computation already. If the object is not involved the CU issues an order for the data of that object to migrate from the DS node to the CE node (system picks the CE depending on scheduling policy). Then the CU stores the handler in a queue. Scheduler can also delay the message depending on its specific policy and parameters (it will not order a migration of the object in such case).

Upon receiving the order for migration from the CU, the DS node packs an object data (using user-registered packing routines) and sends it to the selected CE node. It also sends an ack to the CU to acknowledge that the object has left the DS and has moved (or is still moving) to the CE.

Upon receiving the ack, the CU extracts the delayed handler (or handlers) from the queue and sends them to the data object on the CE node. CU also picks a new storage node for the object and issue a request for migration to this object. Because the MOL messages are causal, this request will not reach the object until all previously issued handlers were executed.

Upon receiving the migration request, the CE node sends an ack to the CU after it uninstalls the object, then sends packed object to the DS node. Upon receiving the ack, the CU may try to schedule any delayed messages.

6. PERFORMANCE EVALUATION

In this section, we present preliminary performance evaluation data using dense matrix-matrix multiplication (MMM) algorithm. We have implemented the MMM using both the GMS and MOL in order to compare the performance of the GMS percolation based approach with the traditional message passing approach. The algorithm we use is not the most efficient MMM algorithm. GMS performs not as good as implicit implementations of matrix-matrix multiplication; it works reasonably well, with overhead much smaller than actual computation. For the testing purposes, we use object-oriented implementation using the MOL. However, the MOL implementation is also far from the best parallel MMM implementation, MOL and the object-oriented model it implies showed to be very good at solving adaptive problem we are most interested. Since we do not have the GMS implementation of an adaptive problem, we want to compare GMS with the system that uses the same programming model.

Our experimental set up consists of the following hardware:

- 1 Dell PowerEdge 6600 with 4 Hyperthreaded Pentium III Xeon 1.4GGz processors and 16GB of RAM (seen as 8 processors under MPI) for the CE layer;
- 2 Dell PowerEdge 2450(2 processors per node) Pentium III 933MGz processors with 1GB of RAM for the DS layer;
- 1 Dell PowerEdge 6450 with 4 (only 2 used) Pentium III 733 MHz processors and 2GB of RAM for CU and application node;
- the accumulative secondary storage of the DS nodes is 18Gb RAID;
- 1Gb Gigabit Ethernet network connection, single switch.

In our implementation, we use $O(n^3)$ matrix-matrix multiplication algorithm:

$$C_{i,j} = \sum_k \left[A_{kj} \times B_{ik} \right] \quad (1)$$

where A and B are the multipliers, C is the product and i, j and k indexes from 0 to n where n is the number of rows/columns in the matrix.

Because of the object-oriented nature of the system, we rearrange the multiplications that though do not affect the time (actual computing time) or correctness of the execution. For every $A_{i,j}$,

$$C_{i,k} += A_{i,j} \times B_{j,k} \quad (2)$$

where A and B are the multipliers, C is the product and i, j and k indexes from 0 to n where n is the number of rows/columns in the matrix.

In our implementation, we store matrix blocks within objects, $A_{i,j}$, $B_{i,j}$ and $C_{i,j}$ are stored within single object. The implementation contains several steps as following:

1) for every block A, compute a list of pairs of pointer to the objects that contain appropriate B and C (as in equation 2);

2) for every block A, call a process handler on the object where that block is stored in with the list as an argument;

3) on a call to process handler, go thru the list and call multiply handlers on the objects that contain appropriate *B* 's, giving the content of object's *A* and appropriate *C* pointer from the list as the arguments;

4) on a call to multiply handler, multiply the incoming *A* block with the *B* block of the object, call append handler on the object, pointer to which comes as the second argument with the result as the argument;

5) on a call to append handler add the argument to the *C* block, increase counter of updates, if counter becomes equal to the number of the blocks in row/column send a notification to the node 0 that the *C* block of the object is ready;

6) on receiving of the confirmations for all the *C* blocks save the resulting *C* matrix and terminate the application.

We used this very implementation to test performance of both MOL and GMS (with few system specific changes).

In

Fig. **3** we show the MOL timing for multiplying matrices of size 6250000 doubles (50000000 bytes) that divided in 25 (5 by 5) blocks.

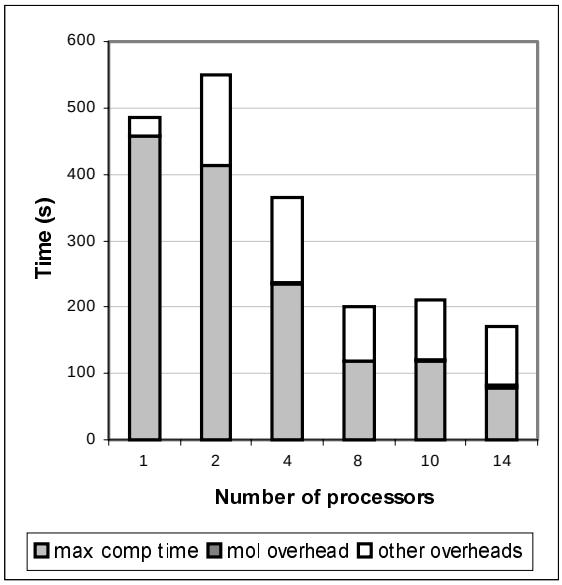

Fig. 3 – MOL timing.

Every bar shows the time that it takes to execute the test with some number of processors (1 through 14), it is wall clock time, the longest that it takes to execute among all the nodes. The bottom part of a bar is the actual computation time – the time that processors actually spend in computing. The middle part is the MOL overhead. The top part is the other overheads, such as communication, not directly related to the MMM computations, lower level communication library overhead, etc., that time processors performs tasks that are not directly related to computing.

We can see that the overhead (mean all the additional computation and communication) is almost constant except for the single processor where no data movement is performed. The computing time is changing proportionally to the number of processors for the first eight and then we see some slow down. It is because first eight processors are fast processors we will later use in CE layer and the later four are the slower ones we will use for supporting tasks (DS, CU and application node).

In Fig. 4 we show the GMS CE timing for multiplying matrices of size 6250000 doubles (50000000 bytes) that divided in 25 (5 by 5) blocks. There are 14 logical processors in the system however only eight of them are used for the computation (1 though 8). The bottom bar is the actual computing time (max among all nodes). The middle bar is the overhead in the CE node; these are the computations that are not directly related to the application's computations. The bottom part is the idle time, it include the time CE receiving data from the network, send data to the network and just stays idle waiting for data.

In Fig. 5 we show the GMS all timing: total time versus the average time DS nodes spend computing, versus the time CU node spends computing. From this data, we can see that CU spends very few cycles comparing to the others, this gives us further flexibility to enhance and improve the control mechanisms for the system in the future versions.

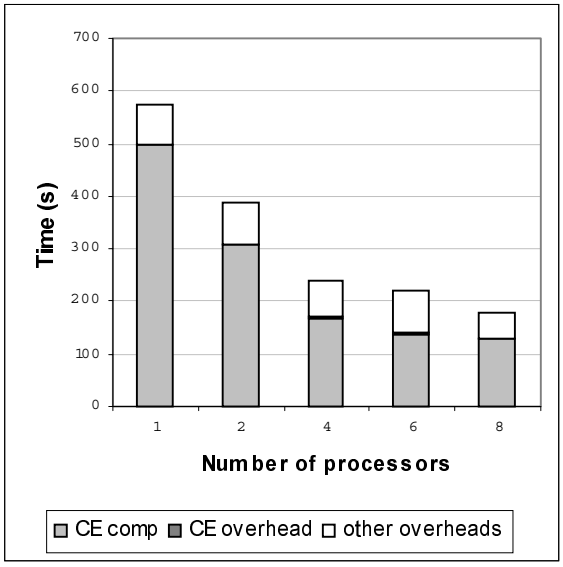

Fig. 4 – GMS CE timing.

Fig. 6 shows the speedup curves for MOL and GMS. We can see the perfect speedup as the straight line. The MOL speedup present for all 14 processors. Both GMS speedups are the same, the difference is that GMS (CE) only considers CE processors and GMS (all) considers all processors.

We can see that with eight computing processors GMS beats MOL with all processors and comes very close to MOL with fourteen processors. It shows that the supporting processors can indeed decrease the time we have to spend in the computing processors.

As one of the goal in this research we plan to improve the performance of the GMS by controlling the percolation depending on the execution flow, which includes the order of the percolation, the postponement of the promotion (percolation to the CE) or the retirement (percolation to the DS), grouping the objects for percolation etc. We do not know yet how exactly we will implement each of the features, but we can try to "fake" the support of the system for some of them. From the description of our implementation of matrix-matrix

multiplication reader can see that we send n (which is number of blocks in the row/column) messages with A block and n messages with update for C to every object. According to the description of the GMS, every time there is a message for execution, the object must promote to execute it (of course if there are more than one message they all will be executed in one promotion). So far, we have very simple control over the promotion/retirement policy thus the object will promote as soon as the first message is available. This means that in the worst case every object have to percolate $2n+1$ times instead of only 3 in the best case.

In Table 1 we present the timing results on which the Fig. 3, Fig. 4 and Fig. 5 are build upon (there are no results for GMS for 10 and 14 processors as only up to 8 processors can be involved into the computation).

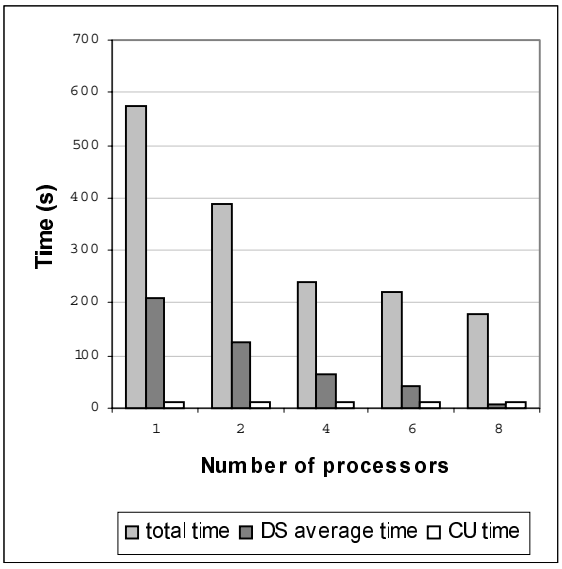

Fig. 5 – GMS all timing.

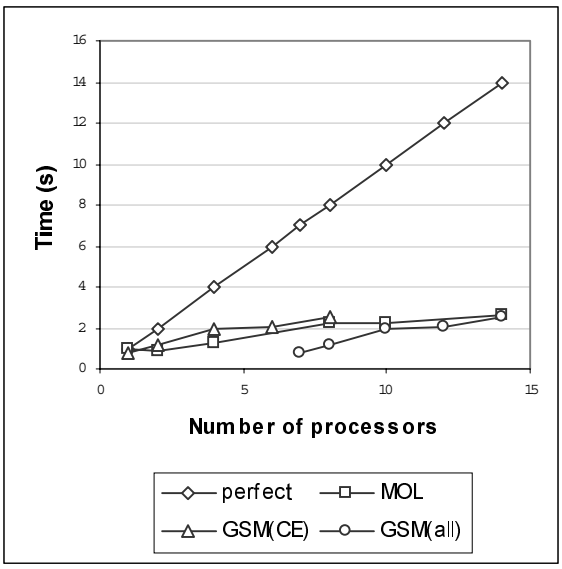

Fig. 6 – Speedup.

We changed the GMS code in order for it to "know" the matrix block object and be aware of number of messages it still needs to receive before it can promote. Of course, this approach is very application specific and we cannot use it in a general case, the only reason for it is to see whether we will get any improvement out of this.

Here is the timing for both GMS and "tweaked" GMS system (Fig. 7).

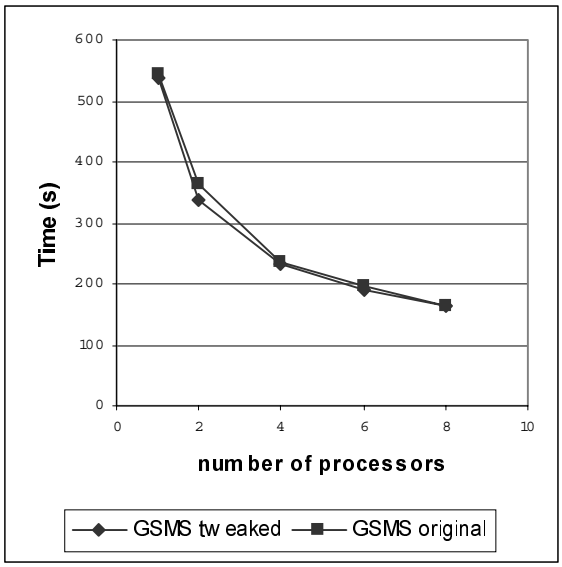

Fig. 7 – GMS vs. GMS tweaked.

We can see that, though the tweaked version is faster, the difference is very small. It is easy to explain. We do decrease the number of percolations; however, the number of messages is the same. In our problem (matrix-matrix multiplication), the messages are roughly of the same size as the object and the total size of the all messages is order of magnitude bigger than the total size of all objects. Thus, we do not get much improvement here. Still, we believe that for problems with bigger size of objects such optimization (we do not now how we will do it yet though) might be very beneficial.

7. CONCLUSION

The Green "Smart" Memory Management System (GMS) system handles and processes requests for handlers' execution at least as effective as the conventional systems (like MOL). Additional knowledge of data dependencies and the ability to change the execution flow based on that data allow that the GMS system can exploit execution patterns that programmer by himself might not be able to discover.

As the results, the speedup for 8 processor is 2.56 versus 2.29 in the traditional implementation (though the implementation we used is object-oriented and thus not the most optimal for test problem) implementation with 8 processors (2.67 for traditional with 14 processors; there are 8 computing processors in our GMS test system, though 6 additional ones are allocated for the serving purposes, which makes total of 14).

The GMS does its job at least as effective as MOL, for in-core problems. Though the GMS system and its variation of the percolation model were design for very big out-of-core problems, the size of our benchmark is much better suited for the traditional in-core computations. Despite this GMS shows comparable results and we expect much better for large out-of-core problems.

Table 1. Computation time, traditional (MOL) and GMS overheads

Number of processors	1	2	4	8	10	14
Max computing time (traditional)	458.304272	412.9528054	234.1910059	117.5811551	117.5398148	79.17232673
MOL overhead (traditional)	0	0.68387587	2.0874521	1.70181793	1.96689483	2.10051155
Other overheads (traditional)	28.17318115	137.0629688	129.3840101	81.86200218	91.13373348	90.26528372
Max CE computing time (GMS)	498.2381	307.7264	169.2926	130.6417		
Max CE overhead (GMS)	1.3685	0.8267	0.4690	0.3693		
Other CE overheads (GMS)	74.1713	78.6442	68.7073	48.1216		

8. FUTURE WORK

We will focus on the applications with variable and unpredictable data access pattern and/or the applications that require support for out-of-core execution. Our challenge is to minimize the overhead introduced by the percolation execution model and GMS in order to realize the benefit of: (1) lower overhead for memory reads compared to overheads of disk reads, and (2) the utilization of slow but additional free nodes that perform the memory management (including disk I/O and caching).

9. REFERENCES

[1] G. Gao, K. Theobald, A. Marquez, and T. Sterling. The HTMT program execution model, *CAPSL Technical Memo 09*, University of Delaware, July 1997.

[2] K. Barker, N. Chrisochoides, J. Dobbelaere, D. Nave, and K. Pingali. Data Movement and Control Substrate, *Concurrency and Computation Practice and Experience*, Vol 14, pp 77-101, 2002.

[3] N. Chrisochoides, K. Barker, D. Nave, and C. Hawblitzel. Mobile Object Layer: A Runtime Substrate for Parallel Adaptive and Irregular Computations, *Advances in Engineering Software*, Vol 31 (8-9), pp. 621-637, August 2000.

[4] J. Salmon, M. Warren. Parallel Out-of-core Methods for N-body Simulation, *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[5] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein and Klaus Erik Schauser. Active Messages: A Mechanism for Integrated Communication and Computation, *19th International Symposium on Computer Architecture*, pp. 256-266, 1992.