A Multithreaded Communication Engine for Distributed Adaptive Applications *

Sivapriya Ramanathan & Manish Parashar
Department of Electrical and Computer Engineering
Rutgers University
Piscataway, NJ, U.S.A.
{sivapriy,parashar}@caip.rutgers.edu

Abstract : ***This paper presents the design, implementation of a multi-threaded communication engine to enable scalable implementations of distributed adaptive mesh-refinement (AMR) applications. The primary objective is to manage the computational heterogeneity inherent in this class of applications and to exploit the multiple granularities and levels of parallelism offered by the applications. The engine uses the MPI communication library and implements the capability for registering message handlers at the application level. These handlers enable each computational thread to define how messages are to be processed. Communications threads can now independently manage all communications and maximize their overlap with computations. An experimental evaluation of the multithreaded engine on the Sun E10K using an AMR kernel from numerical relativity demonstrates that it significantly improves application performance.***
Keywords: Parallel/Distributed Adaptive Applications, Structured Adaptive Mesh-Refinement, Multithreaded Communication, Sun E10K

1 Introduction

Dynamically adaptive methods for the solution of partial differential equations that employ locally optimal approximations can yield highly advantageous ratios for cost/accuracy when compared to methods based upon static uniform approximations. These techniques seek to improve the accuracy of the solution by dynamically refining the computational grid in regions of high local solution error. Distributed implementations of these methods offer the potential for accurate solution of physically realistic models of important physical systems. These implementations however lead to interesting challenges in dynamic resource allocation, data-distribution and load balancing, communications and coordination, and resource management. The overall efficiency of the adaptive algorithms is limited by the ability to manage the underlying data-structures at run-time so as to expose all inherent parallelism, minimize communication/synchronization overheads, and balance load.

AMR grids are inherently heterogeneous, varying in both resolution and extent, and can be created, moved and deleted on the fly. Furthermore, AMR applications offer multiple levels and granularities of parallelism. Grids at the same level of refinement can be operated on in parallel. Similarly composite slices across all refinement levels (i.e. a parent grid and all its children) can also be operated on in parallel. Finally, each grid can itself be operated on in a data-parallel fashion. AMR algorithms require that each grid be periodically synchronized with its parents and its neighboring siblings, thus requiring communications at regular

*The work presented here was supported by the National Science Foundation via grants numbers WU-HT-99-19 P029786F (KDI) and ACI 9984357 (CAREERS) awarded to Manish Parashar

intervals. Clearly, there is a need for a runtime engine that can exploit these many levels and granularities of parallelism and effectively manage the computational heterogeneity, and efficiently overlap the synchronizations and communications with computations.

In this paper we present the design, implementation, and evaluation of such a multithreaded communication engine that addresses the issues listed above and enables scalable implementations of parallel/distributed AMR applications. The engine uses the MPI[2] communication library (to ensure portability) and provides the capability for registering message handlers at the application level (similar in principal to Active Messages [3]). These handlers enable each computational thread to provide a handler function that defines how a particular class of messages is to be processed and how the thread is to be informed about message arrivals. Communications threads can now independently manage all communications and maximize their overlap with computations. The multithreaded communication engine has been built on the GrACE SAMR[4, 5] library.

The rest of the paper is organized as follows: Section 2 describes the adaptive mesh-refinement algorithm and the underlying grid structures. Section 3 presents the design of the multithreaded communication engine. Section 4 describes the implementation the engine on the SUN Enterprise 10000 (E10K) system, and presents an experimental evaluation of the communication engine. Section 5 present conclusions and outlines future research directions.

2 Problem Description and Related Work

2.1 Adaptive Mesh Refinement

Dynamically adaptive numerical techniques for solving differential equations provide a means for concentrating computational effort to appropriate regions in the computational domain. In the case of hierarchical adaptive mesh refinement (AMR) methods, this is achieved by tracking regions in the domain that require additional resolution and dynamically overlaying finer grids over these regions. AMR-based techniques start with a base coarse grid with minimum acceptable resolution that covers the entire computational domain. As the solution progresses, regions in the domain requiring additional resolution are tagged and finer grids are overlaid on the tagged regions of the coarse grid. Refinement proceeds recursively so that regions on the finer grid requiring more resolution are similarly tagged and even finer grids are overlaid on these regions. The resulting grid structure is a dynamic adaptive grid hierarchy. The adaptive grid hierarchy corresponding to the AMR formulation by Marsha Berger and Joseph Oliger[1] is shown in Figure 1.

Distribution of adaptive applications based on hierarchical AMR consists of appropriately partitioning the adaptive grid hierarchy across available computing nodes, and concurrently operating on the local portions of this domain. Parallel AMR applications require two primary types of communication: (a) Inter-grid Communications: Inter-grid Communications are defined between component grids at different levels of the grid-hierarchy and consist of prolongations (coarse to fine transfers) and restrictions (fine to coarse transfers). These communications typically require a gather/scatter type operations based on an interpolation or averaging stencil. Inter-grid communications can lead to serialization bottlenecks for naïve decompositions of the grid hierarchy. (b) Intra-grid Communications: Intra-grid Communications are required to update the grid-elements along the boundaries of local portions of a distributed grid. These communications consist of near-neighbor exchanges on the stencil defined by the difference operator. Intra-grid communications are regular and can be scheduled so as to overlap with computations on the interior region of the local portions of a distributed grid.

2.2 Related Work

There exists a several infrastructures that support parallel and distributed implementations of AMR applications. These include

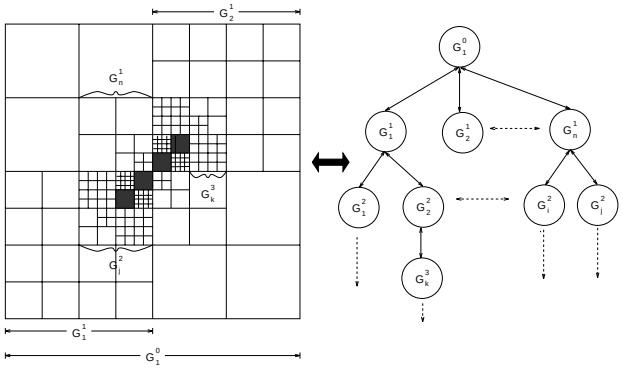

Figure 1: Adaptive Grid Hierarchy - 2D (Berger-Oliger AMR Scheme)

SAMRAI[6], PARAMESH[7] and GrACE[4]. The existing infrastructures however do not support multithreading. Multithreaded runtime for AMR infrastructures has been researched by Chrisochoides[8], and Felten[9]. Chrisochoides' work on multithreading employs threads for load balancing. All processors start with a pool of threads. Threads can be interior threads or interface (boundary) threads. The thread scheduler schedules these threads in so as to minimize the overheads of communication. "New threads" is a thread library developed by Felten et al. that can be used to improve the performance of general message passing applications. The library provides communication support between threads on different processors by using globally unique port numbers.

3 A Multithreaded Engine for AMR Applications

Parallel/distributed implementations of adaptive mesh refinement techniques for solving PDEs typically consist of three phases – (a) computation phase, (b) load balancing phase and (c) data-migration phase. The computation phase is again sub-divided into a pure computation phase and the ghost synchronization phase. The ghost synchronization phase involves the exchange of ghost or boundary regions that are shared between processors. These message exchanges are required frequently during each integration step on a level, and can significantly effect application performance. One solution is to use an overlap to alleviate the cost of communication. This is achieved by the multithreaded communication engine.

The multithreaded communication engine presented in this section has been designed for the GrACE SAMR framework. GrACE is an object-oriented toolkit for the development of parallel and distributed applications based on a family of adaptive mesh-refinement and multigrid techniques. It is built on a "semantically specialized" distributed shared memory substrate that implements a hierarchical distributed dynamic array (HDDA) [10, 11]. HDDA provides uniform array access to heterogeneous dynamic objects spanning distributed address spaces and multiple storage types. The array is hierarchical in that each element of the array can be an array; it is dynamic in that the array can grow and shrink at run-time. Communication, synchronization and consistency of HDDA objects are transparently managed for the user. Distribution of the HDDA is achieved by partitioning its array index space across the processors. The index-space is directly derived from the application domain using locality-preserving space filling mappings [12] that efficiently map N-dimensional space to 1-dimensional space.

The GrACE communication engine operates as follows. At the beginning of the computation phase, each processor computes the boundary regions shared with other processes, anticipates the messages to be received from the neighboring processes, and register a handler for the expected messages. During the ghost synchronization phase, messages are shipped in the form of HDDA objects or buckets consisting of a header and a payload consisting of the actual data. The header contains information describing how the message is to be handled at the receiver. When the message arrives at the receiving end, the information from the header is extracted and the message processed accordingly. The received

data is then held in the message buffers until it is required by the application. While, this process attempts to reduce overheads by eliminating the need for the application to poll for messages to arrive, it can suffer from increased latencies for AMR grid hierarchies. The multithreaded communication engine attempts to reduce these latencies by overlapping computations with communications.

3.1 Multithreaded Communication Engine: Architecture

In GrACE, the adaptive grid hierarchy is partitioned so that each processor owns a number of grid blocks per level. The fact that computations on a processor proceed on a block-by-block basis can be exploited to overlap computation with communication. As soon as computations have been completed on a blocks, ghost synchronization messages from the block to remote blocks can be sent out. Similarly, the block can be updated using incoming ghost messages from remote blocks. Furthermore, this can be done in parallel with computations on the other blocks at the processor. The multithreaded engine is designed to transparently exploit this scenario to improve the performance of GrACE-based AMR applications. The engine provides a modified messaging substrate consisting of three threads per process - viz. the main computation thread, the send thread and the receive thread, as shown in the Figure 2. The 3 threads share a pair of queues (send queue and receive queue). These queues are used to synchronize the operations of the threads. A key objective of the design was to minimize modifications at the application level. The three threads are briefly described below.

Computation thread: The computation thread spawns the send and receive threads at the beginning of the computation phase. It then proceeds with computations on the local grid blocks at the processor. When the computations on a block are completed, its block number is entered into the send and receive queues. This is a signal to the communication (send/receive) threads that they can start

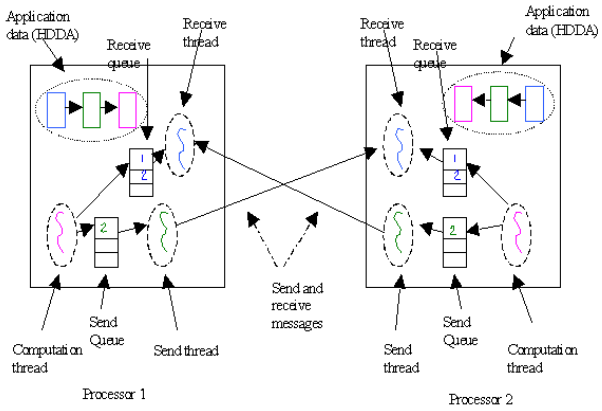

Figure 2: Architecture of the Multithreaded Communication Engine

the ghost synchronizations on the block. After computations are completed, the compute thread waits for the send and received threads to finish, before beginning the next computational cycle. When all computations are completed, the threads join and exit. The computation (or main) thread has the following functions:

- Setting up the grid hierarchy and the necessary grid functions as required by the application.
- Initialize the data structures, calculate and create message handlers for messages.
- Perform computation, load balancing, load distribution.
- Spawn send and receive threads and initialize their data structures.

Send Thread: The send thread maintains the send queue. When computed blocks are entered into this queue by the computation thread, it processes them by sending ghost synchronization messages to remote blocks. The send queue is a FIFO queue and requests are processed in FIFO order. The send thread performs the following functions:

- Wait for a signal from the computation thread signaling the start of the synchronization phase.

- Process the requests or blocks in the order entered in the send queue.
- Send out the blocks to processes.
- Signal the computation thread when the sends are complete.

Receive Thread: The receive thread maintains the receive queue. When computed blocks are entered into this queue by the computation thread, it processes them by updating the block with incoming ghost synchronization messages. The receive queue is also a FIFO queue. The receive thread performs the following functions:

- Wait for signal from the computation thread signaling the start of the synchronization phase.
- Receive messages that have arrived and copy them into the appropriate message buffers.
- Copy the data from message buffers to the grid blocks on which computation has completed.
- Signal the computation thread when all the messages have been received and have been copied into the appropriate grid blocks.

3.2 Multithreaded Communication Engine: Operation

As described earlier, AMR simulations consist of 2 phases - the computation phase and the ghost synchronization phase. Ghost synchronizations involve exchanging the boundary regions of the grid contained with neighboring processors. Thus ghost sends involve copying the data from application buffers into message buffers, packing these buffers and sending them out. On the receiving end, the ghost receives consist of unpacking the incoming message and copying it into the application data structure. The overall operation of the multithreaded communication engine is show in Figure 3 and described below.

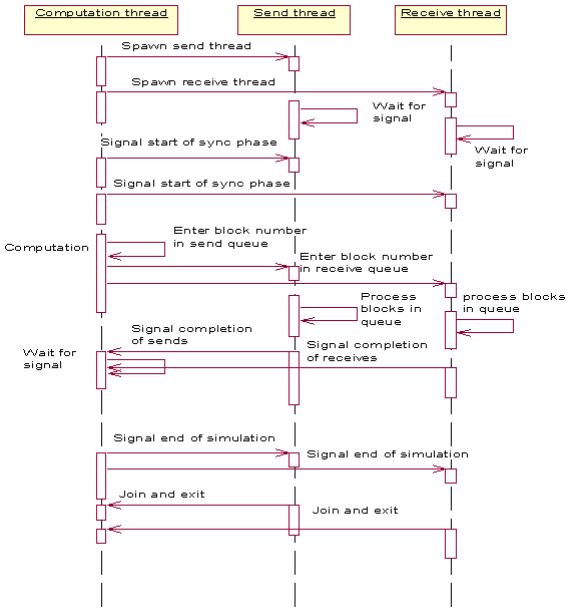

Figure 3: Sequence of events in the multithreaded engine on a single process

At the start of the simulation, the main computation thread performs the initializations on the grid hierarchy and then spawns off the send and receive threads. The queues and data structures are then set up for communication between the threads. When the computation phase starts, the computational thread signals the start of the synchronization phase to both the send and receive threads. As soon as computation on a block is done, its block number is added to the end of the send and receive queues. The communication threads remove the entry from the head of the queue and process ghost synchronization for the blocks. As soon as a synchronization phase is complete, the two threads signal the computational thread so that it can start the next cycle. When the simulation is complete, the threads join the main computational thread and exit.

4 Implementation and Experimental Evaluation

The multithreaded engine was developed and tested on the Sun Enterprise 10000 (E10K) cluster. Each system in the cluster consists of sixty-four 400 MHz SPARC processors, 32GB of RAM, and approximately a terabyte of disk storage. The processors are configured as 16 processor boards with 4 processors per board. Each processor has 4 Mbytes of L2 cache. The communication engine uses the POSIX [13, 14] thread library for creating and scheduling threads and is built using the thread safe MPI [2] implementation available on the Sun E10K.

The application used in these experiments is the 3-dimension numerical relativity kernel (Wave3D) and belongs to the general class of AMR applications. Wave3d solves a coupled set of partial differential equations: Elliptic (Laplace equation-like) constraint equations which must be satisfied at each timestep, and Hyperbolic (Wave equation-like) equations describing time evolution. This kernel is part of the Cactus numerical relativity toolkit [1].

The experiments conducted measure the total execution time with and without the multithreaded communication engine. The applications used different grid sizes and refinement levels for the experiments. Figures 4 and 5 plot the total execution time in seconds for the two cases. As seen in the these plots, the overall application performance increases with the multithreaded engine. Furthermore, performance gaines are larger for the 4 and 8 processors runs as compared to the runs on larger number of processors. This is because, for a given fixed domain size, as the number of processors is increased, the number of grid blocks owned per processor is decreased. The threaded engine exploits the parallelism across multiple blocks on a processor to overlap computations and communications and improve application performance. As the number of blocks decreases, the available parallelism and consequently the opportunity for overlap and corresponding performance improvement also decreases.

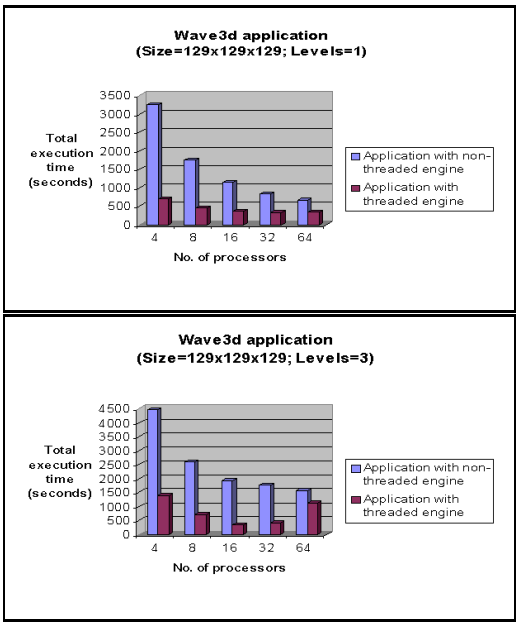

Figure 4: Plots of WaveAMR3D application with 129x129x129 grid size

5 Conclusions

In this paper, we presented the design, implementation and evaluation of a multithreaded communication engine for the GrACE SAMR library. The multi-threaded engine overlaps communications with computations over the set of grid blocks assigned to a processor. The experiments conducted show that the use of the multithreaded engine for communication can improve the performance by more than fifty percent. We are currently evaluating the runtime on large system and exploring combining the multithreaded engine with advanced dynamic partitioning/load-balancing systems. We believe that as domain size and number of levels increase, the number of grid block per processor will also increase and this in turn will increase the opportunity for overlap that is exploited by the multithreaded engine. We are currently evaluating the engine for larger applications (with more levels of refinements)

[1] Cactus Computation Toolkit - http://www.cactuscode.org

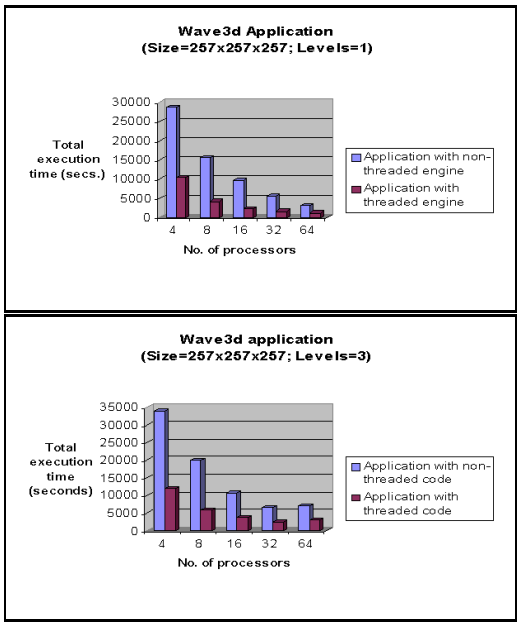

Figure 5: Plots of WaveAMR3D application with a 257x257x257 grid size

and larger system sizes.

References

[1] M. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations", *Journal of Computational Physics, 53, 1983.*

[2] MPI, http://www.mpi-forum.org, MPI Forum homepage.

[3] S.S. Lumetta, A.M. Mainwaring, D.E. Culler, "Multi-Protocol Active Messages on a Cluster of SMP's", *Technical paper at Supercomputing, 1997.*

[4] M. Parashar and J. C. Browne, "System Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement", *in IMA Volume 117: Structured Adaptive Mesh Refinement Grid Methods, IMA Volumes in Mathematics and its Applications.* Springer-Verlag, pp. 1-18, 2000.

[5] M. Parashar, J.C. Browne, C. Edwards, and K. Klimkowski, "A common data management infrastructure for adaptive algorithms for PDE solutions", *Technical paper at Supercomputing, 1997.*

[6] R.D. Hornung, and S. Kohn, "SAMRAI: A Software Framework for Structured Adaptive Mesh Refinement," DOE Conference on High Speed Computing, Salishan Lodge, Gleneden Beach, OR, April 19-22, 1999. Also available as Lawrence Livermore National Laboratory technical report UCRL-MI-133555.

[7] P. MacNiece, K.M. Olson, C. Mobarry, R. deFainchtein and C. Packer, "PARAMESH : A parallel adaptive mesh refinement community toolkit", *Computer Physics Communications, vol.* 126, p.330-354, (2000).

[8] N. Chrisochoides, "Multithreaded model for dynamic load balancing parallel adaptive PDE computations", *Technical Report CTC95TR221, Cornell University, 1995.*

[9] E. W. Felten and D. McNamee, "Improving the performance of Message-Passing Application by Multithreading", *Proceedings of the Scalable High Performance Computing Conference, April 1992.*

[10] M. Parashar and J. C. Browne, "Distributed Dynamic Data Structures for Parallel Adaptive Mesh Refinement", *Proceedings of the International Conference for High Performance Computing*, Dec. 1995.

[11] M. Parashar and J.C. Browne, "On Partitioning Dynamic Adaptive Grid Hierarchies", *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, Jan. 1996.

[12] H. Sagan Space-filling curves *Springer-Verlag, 1994.*

[13] Bradford Nichols, et al, Pthreads Programming: A POSIX Standard for Better Multiprocessing, *O'Reilly*, September 1996.

[14] D.R. Butenhof, *Programming with POSIX threads,* Addison-Wesley, May 1997.