Hierarchical Programming Models for Exascale Computing—Potential and Challenges

Dinesh Kaushik, David Keyes, Nicholas Allsopp, Satish Balay, and Barry Smith

Citation: AIP Conference Proceedings **1281**, 1783 (2010); doi: 10.1063/1.3498226
View online: http://dx.doi.org/10.1063/1.3498226
View Table of Contents: http://scitation.aip.org/content/aip/proceeding/aipcp/1281?ver=pdfcov
Published by the AIP Publishing

Hierarchical Programming Models for Exascale Computing – Potential and Challenges

Dinesh Kaushik*, David Keyes*, Nicholas Allsopp*, Satish Balay† and Barry Smith†

*King Abdullah University of Science and Technology, Saudi Arabia,
{dinesh.kaushik,david.keyes,nicholas.allsopp}@kaust.edu.sa
†Argonne National Laboratory, Argonne, IL 60439 USA,
{balay,bsmith}@mcs.anl.gov

With the availability of large-scale multicore processor based clusters, the different software models for parallel programming require a fresh assessment. For physically distributed memory machines, the message passing interface (MPI) has been a natural and very successful software model. For another category of machines with distributed shared memory and nonuniform memory access, both MPI and OpenMP have been used with respectable parallel scalability. However, for clusters with several multicore processors on a single node, the hybrid programming model with threads within a node (OpenMP being a special case of threads because of the potential for highly efficient handling of the threads and memory by the compiler) and MPI among the nodes seems natural [1].

Two extremes of execution on hybrid architectures are often employed, due to their programming simplicity. At one extreme is the scenario in which the user explicitly manages the memory updates among different processes by making explicit calls to update the values in the ghost regions. This is typically done by using MPI, but can also be implemented with OpenMP. The advantage of this approach is good performance and excellent scalability since network transactions can be performed at large granularity. When the user explicitly manages the memory updates, OpenMP can potentially offer the benefit of lower communication latencies by avoiding some extraneous copies and synchronizations introduced by the MPI implementation. The other extreme is the case in which the system manages updates among different threads (or processes), e.g., the shared memory model with OpenMP. Here the term "system" refers to the hardware or the operating system, but most commonly a combination of the two. The advantages are the ease of programming, possibly lower communication overhead, and no unnecessary copies since ghost regions are never explicitly used. However, performance and scalability are open issues. For example, the user may have to employ a technique like coloring to create nonoverlapping units of work to get reasonable performance. In the hybrid programming model, some updates are managed by the user (e.g., via MPI or OpenMP) and the rest by the system (e.g., via OpenMP).

In this paper, we evaluate the hybrid programming model using memory performance as a metric in the context of an unstructured implicit CFD code, PETSc-FUN3D [2]. The performance of many scientific computing codes is dependent on the performance of the memory subsystem, including the available memory bandwidth, memory latency, number and sizes of caches, etc. In addition, scheduling of memory transactions can also play a large role in the performance of a code. Ideally, the load/store instructions should be issued as early as possible. However, because of hardware (number of load/store units) or software (poor quality assembly code) limitations, these instructions may be issued significantly late, when it is not possible to cover their high latency, resulting in poor overall performance. OpenMP has the potential of better memory subsystem performance since it can schedule the threads for better cache locality or hide the latency of a cache miss. However, if memory bandwidth is the critical resource, extra threads may only compete with each other, actually degrading performance relative to one thread.

To achieve high performance, a parallel algorithm needs to effectively utilize the memory subsystem and minimize the communication volume and the number of network transactions. These issues gain further importance on modern architectures, where the peak CPU performance is increasing much more rapidly than the memory or network performance.

In a typical PDE computation, four basic groups of tasks can be identified, based on the criteria of arithmetic concurrency, communication patterns, and the ratio of operation complexity to data size within the task. These four distinct groups, present in most implicit codes, are vertex-based loops, edge-based loops, recurrences, and global reductions. Each of these groups of tasks stresses a different subsystem of contemporary high-performance computers.

TABLE 1. Execution time on IBM BlueGene/P (four 850 MHz cores per node) for function evaluations only, comparing the performance of distributed memory (MPI alone) and hybrid (MPI/OpenMP) programming models.

	MPI Processes per Node			Threads per Node in Hybrid Mode		
Nodes	1	2	4	1	2	4
128	162	92	50	162	84	44
256	92	50	30	92	48	26
512	50	16	17	50	26	14

TABLE 2. Total number of linear iterations for the case in Table 1.

	MPI Processes per Node		
Nodes	1	2	4
128	1217	1358	1439
256	1358	1439	1706
512	1439	1706	1906

After tuning, linear algebraic recurrences run at close to the aggregate memory-bandwidth limit on performance, flux computation loops over edges are bounded either by memory bandwidth or instruction scheduling, and parallel efficiency is bounded primarily by slight load imbalances at synchronization points [2].

While implementing the hybrid model, the following three issues should be considered.

- Cache locality
- Work Division Among Threads
- Update Management

There are many implementations possible that strike a different balance of these factors. In Table 1, we present one such implementation for hybrid model where work is divided among threads in a manual way (as is done in the pure MPI case). Here, each MPI process calls MeTiS to further subdivide the work among threads, ghost region data is replicated for each thread, and "owner computes" rule is applied for every thread. We expect this implementation to give much better performance than when work is divided by the compiler. The performance data in Table 1 on up to 512 nodes (2048 cores) appear promising for the hybrid model. However, the performance advantage primarily stems from algorithmic reasons (see Table 2). It is well known in the domain decomposition literature that the convergence rate of single level additive Schwarz method (parallel preconditioner in PETSc-FUN3D code [2]) degrades with the number of subdomains. Therefore, the preconditioner is stronger in the hybrid case since it uses fewer subdomains as compared to pure MPI case. We believe this to be one of the most important advantages of the hybrid model. In our full paper, we will have more details on this issue, compare three different implementations of the hybrid model, and document performance data on more machines.

REFERENCES

1. E. Lusk, and A. Chan, "Early Experiments with the OpenMP/MPI Hybrid Programming Model," in *Proceedings of the Fourth International Workshop on OpenMP (IWOMP 2008)*, 2008, pp. 36–47, west Lafayette, Indiana.
2. W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, *Journal of Parallel Computing* **27**, 337–362 (2001).