Hybrid Programming Model for Implicit PDE Simulations on Multicore Architectures

Dinesh Kaushik[1], David Keyes[1], Satish Balay[2], and Barry Smith[2]

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

Abstract. The complexity of programming modern multicore processor based clusters is rapidly rising, with GPUs adding further demand for fine-grained parallelism. This paper analyzes the performance of the hybrid (MPI+OpenMP) programming model in the context of an implicit unstructured mesh CFD code. At the implementation level, the effects of cache locality, update management, work division, and synchronization frequency are studied. The hybrid model presents interesting algorithmic opportunities as well: the convergence of linear system solver is quicker than the pure MPI case since the parallel preconditioner stays stronger when hybrid model is used. This implies significant savings in the cost of communication and synchronization (explicit and implicit). Even though OpenMP based parallelism is easier to implement (with in a subdomain assigned to one MPI process for simplicity), getting good performance needs attention to data partitioning issues similar to those in the message-passing case.

1 Introduction and Motivation

As the size of multicore processor based clusters is increasing (with decreasing memory available per thread of execution), the different software models for parallel programming require continuous adaptation to match the hierarchical arrangement at the hardware level. For physically distributed memory machines, the message passing interface (MPI) [1] has been a natural and very successful software model [2, 3]. For another category of machines with distributed shared memory and nonuniform memory access, both MPI and OpenMP [4] have been used with respectable parallel scalability [5]. However, for clusters with several multicore processors on a single node, the hybrid programming model with threads within a node and MPI among the nodes seems natural [6–8]. OpenMP provides a portable API for using shared memory programming model with potentially efficient thread scheduling and memory management by the compiler.

Two extremes of execution of a single program multiple data (SPMD) on hybrid multicore 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 graph coloring based on the underlying mesh to create non-overlapping 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 in the context of an unstructured implicit CFD code, PETSc-FUN3D [9]. This code solves the Euler and Navier-Stokes equations of fluid flow in incompressible and compressible forms with second-order flux-limited characteristics-based convection schemes and Galerkin-type diffusion on unstructured meshes. This paper uses the incompressible version of the code to solve the Euler equations over a wing.

The rest of this paper is organized as follows. We discuss the primary performance characteristics of a PDE based code in Section 2. We present three different implementations of the hybrid programming model (using OpenMP within a node) in Section 3. These implementations strike a different balance of data locality, work division among threads, and update management. Finally, the performance of pure message-passing and hybrid models is compared in Section 4.

2 Performance Characteristics for PDE Based Codes

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. For a vertex-centered code (such as PETSc-FUN3D used in this work), where data is stored at cell vertices, these tasks can be summarized as follows (see a sample computational domain in Figure 1):

- Vertex-based loops
  - state vector and auxiliary vector updates (often no communication, pointwise concurrency)
- Edge-based "stencil op" loops
  - residual evaluation, Jacobian evaluation (large ratio of work to datasize, since each vertex is used in many discrete stencil operations)
  - Jacobian-vector product (often replaced with matrix-free form, involving residual evaluation)
  - interpolation between grid levels, in multilevel solvers
- Sparse, narrow-band recurrences
  - (approximate) factorization, back substitution, relaxation/smoothing
- vector inner products and norms
  - orthogonalization/conjugation
  - convergence progress checks and stability heuristics

Each of these groups of tasks stresses a different subsystem of contemporary high-performance computers. 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 [9].

3 Three Implementations of OpenMP

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

- Cache locality
  - Both temporal and spatial are important. For a code using an unstructured mesh (see Figure 1), a good reordering technique for vertex numbering such as the Reverse Cuthill-McKee algorithm (RCM) [10] should be used.
  - TLB cache misses can also become expensive if data references are distributed far apart in memory.

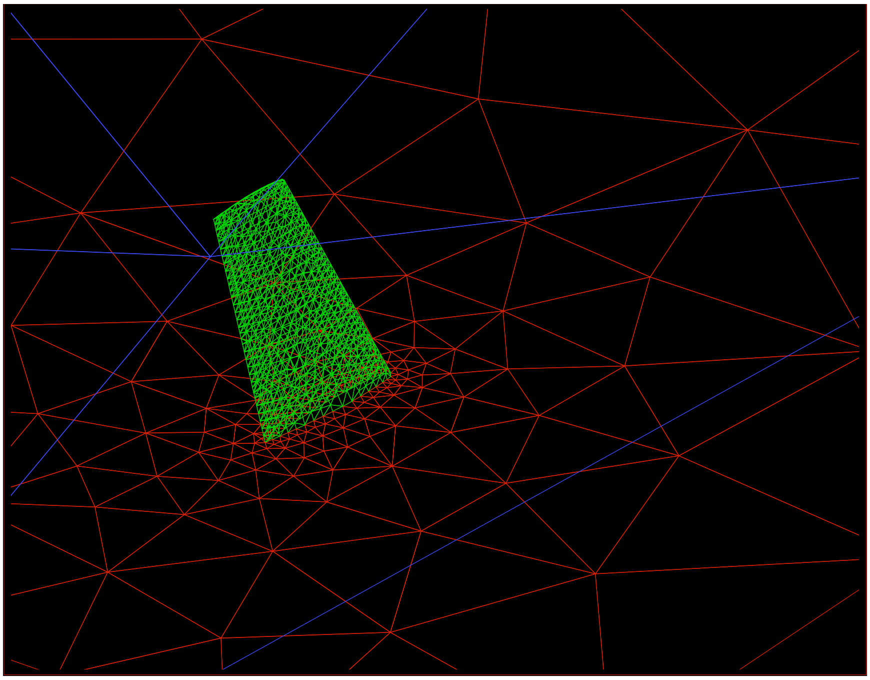

Fig. 1. Surface mesh over ONERA M6 wing. The unstructured mesh used here requires explicit storage of neighborhood information. The navigation over mesh is done by going over edges, which can be represented as a graph. These meshes can be divided among subdomains using any graph partitioner.

- Work Division Among Threads
  - This can be done by the compiler or manually. For unstructured mesh codes, it is very difficult for the compiler to fully understand the data reference patterns and work may not be divided in an efficient and balanced way.
- Update Management
  - The shared arrays need to be updated carefully and several techniques can be used for conflict resolution among threads. The extra memory allocation for private arrays and subsequent gathering of data (usually a memory bandwidth limited step) can be expensive.
  - There is potential for some redundant work for conflict resolution, which should be acceptable in phases that are not CPU-bound.

In an unstructured finite volume code, the residual calculation (or function evaluation) is a significant fraction of the execution time, especially when a Jacobian-free Newton-Krylov approach is used [11]. This calculation traverses all the edges in a subdomain (including those among the local and ghost vertices). The updates to the residual vector are done only for a local vertex while the ghost vertices are needed in the 'read-only' mode.

This work can be divided in several ways among threads with different balance of cache locality, extra memory allocation for private arrays, and update management. We discuss here three implementations we have employed in this paper next.

3.1 Edge Coloring for Vertex Update Independence

In Figure 2, we show an ordering of edges so that no vertex is repeated in a color. All the threads can safely operate within each color (resolving conflicts without any extra memory overhead for private arrays). However, the temporal cache locality can be extremely poor, especially when the subdomain (or per MPI process problem size) is reasonably big (to fully utilize the available memory on the node). In this case, no vertex data can be reused and no gathering of shared data is needed later. This technique was widely used for vector processors but lost appeal on cache-based processors. However, this may become important again for heterogeneous multicore processors where the fine division of work should happen among thousands of threads under tight memory constraints.

3.2 Edge Reordering for Vertex Data Locality

This variant is presented in Figure 3 where edges are reordered to provide reasonable temporal cache locality but residual vector updates are a challenge. There are several ways in which this reordering can be done. We have implemented a simple approach where vertices at the left ends of edges are sorted in an increasing order (with duplicates allowed). A typical edge-based loop will traverse all the neighbors of a given vertex before going to the next vertex. This reordering (when combined with a bandwidth reducing technique like RCM [10]) implies high degree of data cache reuse. However, each thread here needs to allocate its private storage for the shared residual vector. The contribution from each thread needs to be combined at the end of the computation. This update becomes a serialized memory-bandwidth limited operation, presenting limitations on scalability.

3.3 Manual Subdivision Using MeTiS

Here the subdomain is partitioned by a graph partitioner (such as MeTiS [12]) in an appropriately load balanced way. 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. Note that this is the second level of partitioning done in this way (while the first is done for MPI). This creates a hierarchical division of computational domain. This hierarchy can be implemented using a MPI sub-communicator, Pthreads, or OpenMP. We have used OpenMP in this paper. The vertices and edges can be reordered after partitioning for better cache locality. In essence, this case gives the user complete control over the work division, data locality, and update management. We expect this implementation to perform better than the first two cases. However, this is achieved at the expense of the simplicity of OpenMP programming

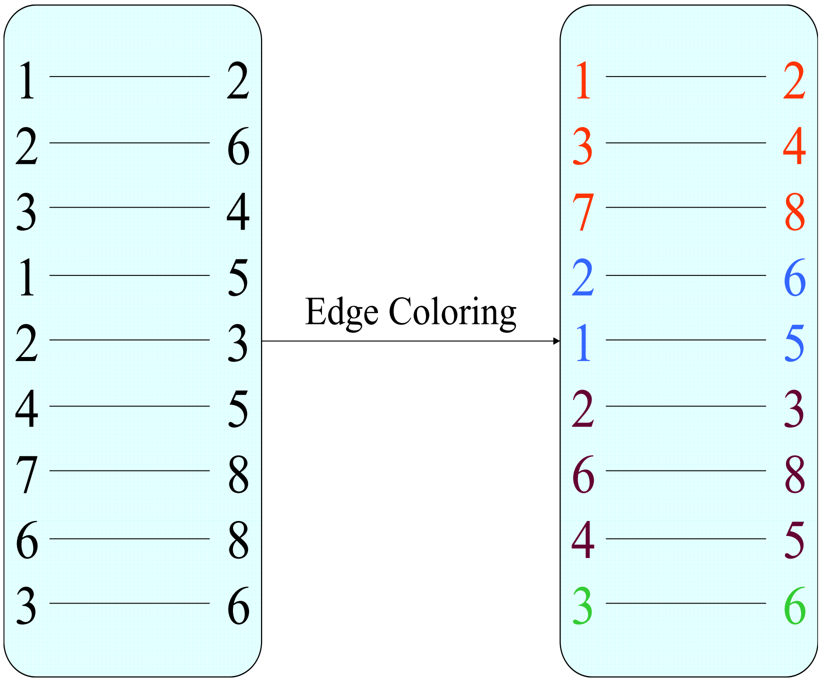

Fig. 2. Edge coloring allows excellent update management among threads but produces poor cache locality since no vertex can repeat within a color

model (such as incremental parallelism). The user must also face the additional complexity at the programming level. However, this implies the same level of discipline (for data division) as was done for the pure MPI case (where several application codes begin anyway).

4 Results and Discussion

There are several implementations of the hybrid programming model possible that strike a different balance of memory and data synchronization overheads. In Table 1, we study the three implementations presented in the previous section on 128 nodes of Blue Gene/P (with 4 cores per node). The mesh consists of 2.8 million vertices and about 19 million edges. As expected, the MeTiS divided case performs the best among the three. We have observed that the differences among the three cases diminish as the subdomain problem size gets smaller since the problem working set size will likely be fitting in the lowest level of cache (L3 in the case of Blue Gene/P).

In Table 2, we compare the MeTiS divided implementation of the hybrid model with the pure MPI model. The performance data in Table 2 on up to 1,024 nodes (4,096 cores) appears promising for the hybrid model. As the problem size

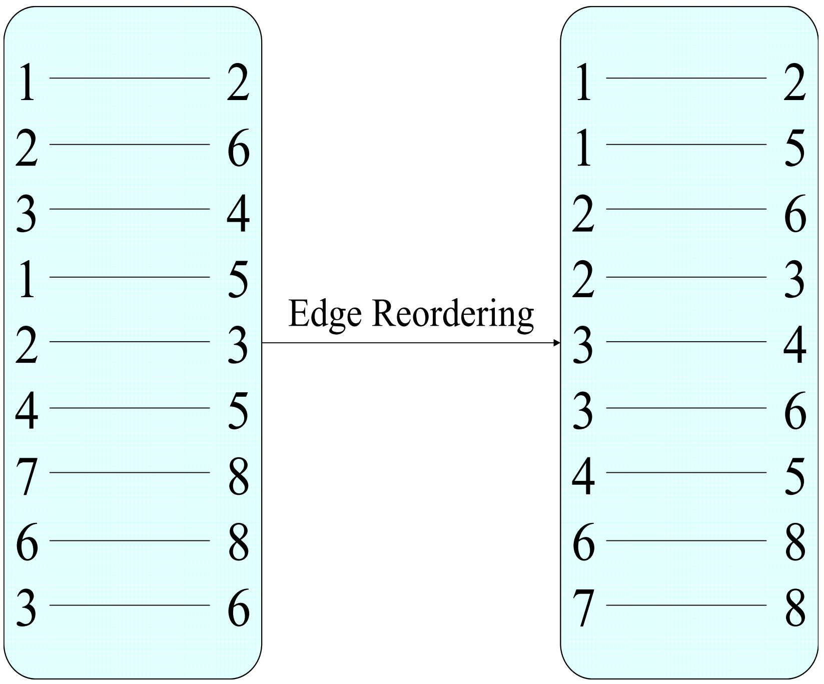

Fig. 3. Vertex localizing edge reordering (based on sorting the vertices at one end) gives good cache locality but allows updates that can overwrite the data in an array shared among threads

per MPI process (from the global 2.8-million vertex mesh) gets smaller (there are only about 674 vertices per MPI process on 4,096 cores), the two models perform close to each other. However, mesh sizes much larger than 2.8 million are needed in real life to resolve the complex flow field features (say, around full configuration airplanes) and we may not be able to afford the partitioning at this fine granularity. For larger mesh sizes, the difference in performance are expected to grow. Nevertheless, the lessons learned from this demonstration comparing these two different programming models are relevant, especially at the small node count level.

We note that the performance advantage in the case of hybrid programming model primarily stems from algorithmic reasons (see Table 3). 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 [9]) degrades with the number of subdomains, weakly or strongly depending on the diagonal dominance of the underlying operator. 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 practical advantages of the hybrid model, on machines with contemporary hardware resource balances.

Table 1. Overall execution time for the different OpenMP implementations on 128 nodes of IBM Blue Gene/P (four 850 MHz cores per node)

	Threads per Node		
Implementation	1	2	4
Edge Coloring for Vertex Update Independence	211	107	54
Edge Reordering Vertex Data Locality	198	103	57
Manual Subdivision Using MeTiS	162	84	44

Table 2. Execution time on IBM Blue Gene/P 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	30	17	50	26	14
1024	30	17	10	30	16	9

Table 3. Total number of linear iterations for the case in Table 2

	MPI Processes per Node		
Nodes	1	2	4
128	1217	1358	1439
256	1358	1439	1706
512	1439	1706	1906
1024	1706	1906	2108

In particular, the number of iterations decreases, and with it the number of halo exchanges and synchronizing inner products that expose even minor load imbalances.

5 Conclusions and Future Work

We have demonstrated the superior performance of hybrid programming model to that of pure MPI case on a modest number of MPI processes and threads. The data partitioning similar to the message-passing model is crucial to get good performance. As the number of processors and threads grows, it is important from execution time and memory standpoints to employ a hierarchy of programming models. This hierarchy can be implemented in several ways such as MPI+OpenMP, MPI+MPI, and MPI/CUDA/OpenCL.

Our past work [3] has demonstrated the scalability of MPI+MPI model (using MPI communicators for a six dimensional problem in particle transport) on the petascale architectures available today. The pure (flat) MPI case will impose too much memory overhead at the extreme scale. As the system size grows, we expect algorithmic advantages (in terms of convergence rate) to be increasingly dominant. The synchronization frequency is another important factor at the extreme scale, especially when hundreds of threads (e.g., in a GPU) are used to accelerate the important kernels of the code, such as function evaluation or sparse matrix vector product. Our future work will focus on parallel algorithms and solvers that will require less synchronization at both fine- and coarse-grain levels on heterogeneous as well as homogeneous architectures.

Acknowledgments

We thank William Gropp of University of Illinois at Urbana Champaign for many helpful discussions. For computer time, this research used resources of the Supercomputing Laboratory at King Abdullah University of Science and Technology (KAUST), and the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory. ALCF is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

References

1. MPI Forum, `http://www.mpi-forum.org`
2. Sahni, O., Zhou, M., Shephard, M.S., Jansen, K.E.: Scalable Implicit Finite Element Solver for Massively Parallel Processing with Demonstration to 160K Cores. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 68:1–68:12. ACM, New York (2009)
3. Kaushik, D., Smith, M., Wollaber, A., Smith, B., Siegel, A., Yang, W.S.: Enabling High-Fidelity Neutron Transport Simulations on Petascale Architectures. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 67:1–67:12. ACM, New York (2009)
4. The OpenMP API specification for parallel programming, `http://www.openmp.org`
5. Mallón, D.A., Taboada, G.L., Teijeiro, C., Touriño, J., Fraguela, B.B., Gómez, A., Doallo, R., Mouriño, J.C.: Performance evaluation of MPI, UPC and openMP on multicore architectures. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) PVM/MPI. LNCS, vol. 5759, pp. 174–184. Springer, Heidelberg (2009)
6. Rabenseifner, R., Hager, G., Jost, G.: Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In: 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 427–436 (Febraury 2009)
7. Lusk, E., Chan, A.: Early Experiments with the OpenMP/MPI Hybrid Programming Model. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 36–47. Springer, Heidelberg (2008)
8. Cappello, F., Etiemble, D.: MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In: ACM/IEEE 2000 Conference on Supercomputing, p. 12 (November 2000)

9. Gropp, W.D., Kaushik, D.K., Keyes, D.E., Smith, B.F.: High Performance Parallel Implicit CFD. Journal of Parallel Computing 27, 337–362 (2001)
10. Cuthill, E., McKee, J.: Reducing the Bandwidth of Sparse Symmetric Matrices. In: Proceedings of the 24th National Conference of the ACM (1969)
11. Knoll, D.A., Keyes, D.E.: Jacobian-free Newton-Krylov Methods: A Survey of Approaches and Application. Journal of Computational Physics 193, 357–397 (2004)
12. Karypis, G., Kumar, V.: A fast and high quality scheme for partitioning irregular graphs. SIAM Journal of Scientific Computing 20, 359–392 (1999)