Bi-modal MPI and MPI+threads Computing on Scalable Multicore Systems

Michael A. Heroux, Ronald B. Brightwell [1], Michael M. Wolf [2]

[1] Scalable Algorithms Department, Sandia National Laboratories
P.O. Box 5800, MS 1320, Albuquerque, NM 87185-1320, USA
{maherou, rbbrigh}@sandia.gov
[2] Embedded and High Performance Computing Group, MIT Lincoln Laboratory
244 Wood Street Lexington, MA 02420-9185
michael.wolf@ll.mit.edu

***Abstract*—There is ample evidence that many single level MPI-only applications will not perform optimally on scalable multicore systems as core counts per node increase. However, within a given application, certain phases of computation are impacted more than others. For example, finite element applications tend to scale very well with MPI-only when performing stiffness matrix computations, but scale poorly in the solvers.**

We present and demonstrate use of bi-modal parallel programming using proposed new MPI features. Specifically, for MPI processes on the same node, we exploit the underlying architecture to create shared memory regions visible to some or all processes on that node. Using these regions we can reduce memory usage for some algorithms with minimal effort. We can also initialize buffers in MPI-only mode and then switch to MPI+threads mode to use threaded algorithms.

We demonstrate this bi-modal approach on several problems and show how it provides essential value for current and future computer systems. Specifically we illustrate how the bi-modal features enable tunable shared memory thread counts that can be changed to match the needs of various phases within a single application.

I. Introduction

Modern multi-node computers have more than one core per node, permitting two principle ways to program the system. Figure 1 shows that we can view all cores through a single level MPI-only interface, call it Mode 1, ignoring the underlying shared memory architecture (which is still typically exploited by MPI runtime layer), or, in Mode 2, we can explicitly manage the shared memory node via threads, leaving MPI to manage internode parallelism[1].

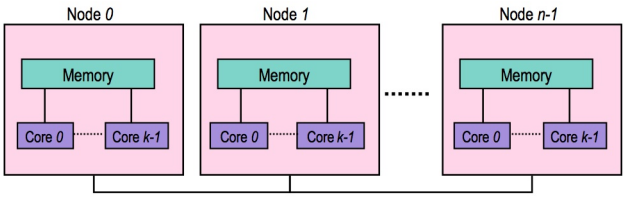

Fig. 1. Simple block diagram of a multinode, multicore machine, typically programmed using $p = n * k$ MPI processes, or n MPI processes with k threads per process.

There are many situations where Mode 2 is advantageous. Replicated data needed for initialization, property tables and related global information can be stored per node instead of per process. Mode 2 also permits the use of more sophisticated fine-grain data parallelism that can lead to more robust and higher performance computations. In particular, there is a large collection of thread-based parallel libraries that offer excellent performance for computations such as sparse solvers and basic linear algebra computations. All of these libraries become accessible for Mode 2 applications. Finally, threading has less memory and data copying overhead than MPI, which can be important for small data sets.

The challenge for an existing Mode 1 application is that converting to Mode 2 requires ubiquitous change, even for phases of computation that do not see a performance improvement from Mode 2. What would be very attractive is to selectively switch to Mode 2 for only some phases of computation.

In this paper we introduce a small set of proposed features for MPI that enable bi-modal execution: Mode 1 in some (most phases) of computation and Mode 2 in others. These features are simple and incremental and inspired by the LIBSM library [1]. They enable all of the advantages listed above. Furthermore, they allow us to dynamically change the scope of threaded parallelism, effectively allowing us to choose the MPI process vs. thread count factoring of the p cores at will during the execution of an application to best match the requirements of the given phase of computation.

II. Proposed MPI Shared Memory Features

In this section we present a set of proposed new features for MPI [2]. These features exploit the underlying shared memory that is present on multicore nodes, by allocating a region of memory that is visible to all MPI ranks that participate in the allocation request. The new feature set is composed of two new functions and mechanisms for defining communicators whose

[1]These two ways of viewing the machine have many variations. In fact, on nodes with non-uniform access memory, many applications find optimal performance by assigning one MPI process per set of cores with uniform access to memory, so that on a quad-socket, quad-core node we would have 4 MPI processes per node with 4 threads per MPI process.

ranks have a common shared memory resource, as shown in Figures 2-3.

MPI_COMM_ALLOC_MEM(comm, size, info, baseptr)

- IN comm - input communicator.
- IN size - size of memory segment in bytes.
- IN info - info argument.
- OUT baseptr - pointer to beginning of memory segment allocated.

- Collective call.
- Allocates region of shared memory accessible by ranks in input communicator.
- No guarantee of identical baseptr across ranks.
- Otherwise, semantics are same as MPI_ALLOC_MEM().
- Returns MPI_ERR_COMM if no shared memory is possible.
- Return MPI_ERR_NO_MEM if memory is exhausted.

MPI_COMM_FREE_MEM(comm, base)

- IN comm - input communicator.
- IN base - initial address of memory segment allocated by MPI_COMM_ALLOC_MEM

- Collective call
- Same semantics as MPI_FREE_MEM()

Fig. 2. Proposed MPI shared memory allocation functions and semantics.

New pre-defined MPI communicators:

- MPI_COMM_NODE - All ranks on the same shared memory node.
- MPI_COMM_CACHE - All ranks that share a cache.
- MPI_COMM_NETWORK - Inter node (one rank per node).

Alternative approach:

Function to produce a communicator with specific properties.

MPI_GROUP_CREATE(comm, properties, newgroup)

- IN comm - input communicator.
- IN properties - bit string of desired properties, such as ''all MPI processes that share uniform memory access''.
- OUT baseptr - pointer to a new MPI_Group that can be passed to MPI_Comm_Create to build a new communicator.

Fig. 3. Proposed pre-defined MPI communicators (actual names may change) or an alternative approach of a function call to create the desired communicator.

Upon successful return from calling MPI_Comm_alloc_mem, each rank in the participating communicator will be pointing to a shared buffer with the global specified size. Except when running on specialized light-weight operating systems, the physical pages will not be mapped until the buffer values are initialized. This fact is good since the threaded algorithms use case discussed in Section III will typically have each MPI process initialize a portion of the buffer before switching to threaded computation, so that data will be properly placed on systems with non-uniform memory access.

Given this feature set, a basic usage is as follows:

```
int n = ;
double *vals;
MPI_Comm_alloc_mem(
  MPI_COMM_NODE, // comm
  n*sizeof(double), // size in bytes
  MPI_INFO_NULL, // placeholder for now
  &vals); // Pointer to shared array (out)

// At this point:
// - All ranks in MPI_COMM_NODE have
//   pointer to a shared buffer (vals).
// - Computation will continue in
//   MPI-only mode with each MPI rank
//   initializing a portion of vals,
//   as described in the following
//   section on use cases.
// After vals buffer is filled:
// - Can use the vals buffer as a common
//   data table or
// - Can continue in MPI mode (using
//    shared memory algorithms) or
// - Implement one of the other use cases,
//   such as using threads from one rank:
int rank;
MPI_Comm_rank(MPI_COMM_NODE, &rank);

// Start threaded code segment
if (rank%cores_per_node==0) {
// Rank 0 executed threaded code.
// Other ranks wait.

}

MPI_Comm_free_mem(MPI_COMM_NODE, values);
```

The shared memory MPI features can be used in many ways, but one of the most attractive properties is that threading can be introduced within an existing MPI-only application in an isolated fashion. It is even possible for a library to use shared memory without changing the application interface.

An MPI-only application can use the shared memory MPI features to incrementally introduce threaded or hybrid MPI/threaded kernels into the code, targeting the most performance critical sections first. Figures 4-6 demonstrate how threaded/hybrid codes could be introduced into a MPI-only code in a painless fashion using the shared memory MPI extensions. Figure 4 is a very simple MPI program with two MPI-only kernels that use the arrays x and y. In the Figure 5 example, x and y are allocated using the MPI shared memory

function and now are shared memory segments (shared across the node) of size k times their sizes (on each MPI task) in the first example, where k is the number of MPI tasks per node. These shared memory variables are used in the hybrid version of the second kernel in some threaded fashion. The MPI kernel (**MPIkernel1**) has not changed. After this shared memory infrastructure is in place, it is simple to replace the **MPIkernel1** with a hybrid kernel as shown in Figure 6. In this fashion, multithreading can be introduced into an application incrementally.

```
double *x = new double[n];
double *y = new double[n];

MPIkernel1(x,y);
MPIkernel2(x,y);

delete [] x;
delete [] y;
```

Fig. 4. Simple MPI program.

```
MPI_Comm_size(MPI_COMM_NODE, &nodeSize);
MPI_Comm_rank(MPI_COMM_NODE, &nodeRank);

double *x, *y;

MPI_Comm_alloc_mem(MPI_COMM_NODE,
        n*nodeSize*sizeof(double),
        MPI_INFO_NULL, &x);

MPI_Comm_alloc_mem(MPI_COMM_NODE,
        n*nodeSize*sizeof(double),
        MPI_INFO_NULL, &y);


MPIkernel1(&(x[nodeRank * n]),
           &(y[nodeRank * n]));

if(nodeRank==0)
{
   hybridKernel2(x,y);
}

MPI_Comm_free_mem(MPI_COMM_NODE, &x);
MPI_Comm_free_mem(MPI_COMM_NODE, &y);
```

Fig. 5. One MPI kernel and one MPI+X hybrid kernel.

```
MPI_Comm_size(MPI_COMM_NODE, &nodeSize);
MPI_Comm_rank(MPI_COMM_NODE, &nodeRank);

double *x, *y;

MPI_Comm_alloc_mem(MPI_COMM_NODE,
        n*nodeSize*sizeof(double),
        MPI_INFO_NULL, &x);

MPI_Comm_alloc_mem(MPI_COMM_NODE,
        n*nodeSize*sizeof(double),
        MPI_INFO_NULL, &y);


MPIkernel1(&(x[nodeRank * n]),
           &(y[nodeRank * n]));

if(nodeRank==0)
{
   hybridKernel1(x,y);
   hybridKernel2(x,y);
}

MPI_Comm_free_mem(MPI_COMM_NODE, &x);
MPI_Comm_free_mem(MPI_COMM_NODE, &y);
```

Fig. 6. Two MPI+X hybrid kernels.

III. Bi-Modal Use Cases

In this section we discuss several possible uses for shared memory features presented in the previous section. These use cases are meant to illustrate a few of many possibilities.

A. Access to read-only data

Read-only data is commonly used for initialization steps in applications, such as defining the geometry of a domain, storing physical properties such as chemical reaction rates and more. If the data set size is large, it must be distributed across MPI processes, and data access must be coordinated. In many cases, supporting distributed access to this kind of data requires sophisticated programming strategies, since discovering ownership of the data portions can be complicated. Fortunately, the data set size is often small enough to be replicated across MPI processes, bypassing the need for complicated programming, but this approach does introduce an overhead that is nontrivial as processor counts increase.

Shared memory allocation reduces the impact of replicated data by creating a shared buffer for all processes on a node and loading the read-only data into that buffer. The only performance issue to consider is on non-uniform memory architecture (NUMA) nodes, where it might make sense to have a shared buffer for each uniform access memory region.

B. Connectivity structure setup

Applications with unstructured data must execute a discovery phase where each process determines who it must communicate with for distributed computations. Although there are sophisticated rendezvous algorithms that can reduce storage complexity, the most straight-forward algorithms for discovery computations use $O(p^2)$ storage, where p is the number of MPI tasks involved in the collective operation. For small values of p, this is not a large cost, but as p increases, this cost can be substantial. Shared memory allows us to reduce the storage cost to $O(n^2)$, where n is the number of nodes on the system. As core counts per node increase, this reduced complexity can be important.

C. Bi-modal MPI-only and Accelerators transition

Many emerging computing systems contain both multicore CPUs and one or more accelerator devices, such as GPUs. For example, the Cray XK6 [3] will combine 16 AMD cores and two Nvidia GPUs on a single node. For these systems, an existing MPI-capable application can execute 16 MPI processes per node. However, if these processes create data that will then be used as input for the GPUs or use output data from the GPUs, it would be very challenging for all 16 MPI processes to coordinate with each other in transferring data to the GPUs. Instead, the MPI_Comm_alloc_memfunction could be used to create shared data objects and allow the MPI processes to store and retrieve data from these objects. Then only one MPI rank per GPU would be responsible for transferring data between the host and device.

D. Threaded Algorithms

Although the above uses are important, the most important opportunities for using shared memory lie in the ability to introduce threaded algorithms and libraries into existing MPI applications. MPI applications have traditionally been very portable across many different architectures, including multicore nodes using one MPI rank per core. MPI enforces an unambiguous placement and ownership of data that naturally match cache-based memory systems within a multicore node, in addition to discrete memory images across multiple nodes. As a result, even though there are unnecessary overheads using MPI to manage parallelism on the node, MPI on multicore nodes tends to perform as well as threading approaches, as long as there is enough work to minimize the impact of MPI overheads [4].

Having said this, use of MPI for parallelism on the node complicates use of fine-grain data parallel algorithms and does not perform as well as threading when data sizes per core are small. Furthermore, the performance of some phases of computation within an MPI-only application tend to degrade more quickly than others, reducing overall program efficiency.

In many applications, the computations that suffer first and most are the solvers, since solver performance is often dictated by memory system performance. Multicore processor memory designs allow one or a few cores to use all available bandwidth if the computation requests it. This is a good thing, but it means that memory system bound computations cannot effectively use additional cores on a node. Threading can mitigate the impact of this performance bottleneck by allowing multiple cores to share data retrieved from main memory via caches, thereby reducing the overall bandwidth requirements.

Threaded algorithms are also available from many high-quality libraries. Threaded BLAS [5] and LAPACK [6] libraries are available from commercial vendors. Threaded sparse direct solvers have been around for decades and are presently represented by packages such as PARDISO [7], MUMPS [8] and WSMP [9]. Many other threaded libraries exist, as do excellent threaded programming environments such as OpenMP [10], Intel Threading Building Blocks (TBB) [11], CILK [12] or Pthreads [13]. Furthermore, there is a tremendous amount of work going on this area, since the entire computing community is focused on addressing multicore performance.

IV. Case Study: Finite Volume Miniapplication

A. Preconditioned iterative methods

Preconditioned iterative methods for solving linear systems are good examples of numerical algorithms that can benefit from a hybrid MPI/threaded approach. Solver implementations based on a flat MPI programming model (where subcommunicators are not utilized) often suffer from poor scalability for large numbers of tasks. One difficulty with these approaches is that with domain decomposition based preconditioners, the number of iterations per linear solve step increase significantly as the number of MPI tasks (and thus the number of subdomains) becomes particularly large. Figures 7 and 8 shows an example of this difficulty for Charon, a semiconductor device simulation code [14], [15], [16], with a three level multigrid preconditioner. As the number of MPI tasks increases, the number of linear solver iterations increases (Figure 8). Figure 7 shows that these extra iterations require an increasingly higher percentage of the total runtime as the number of MPI tasks increase, resulting in a degradation in the parallel performance. This kind of phenomenon is well-known in the solver community and is one reason why solvers dominate performance costs on large systems.

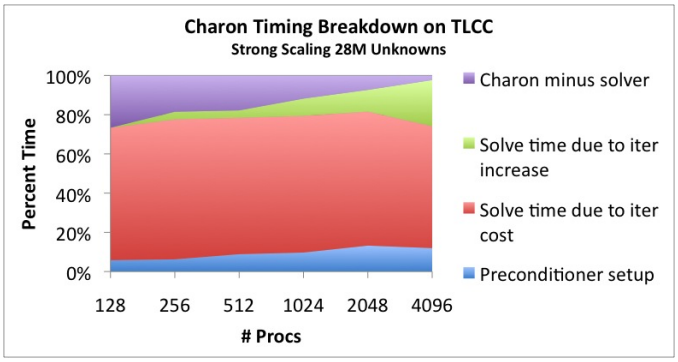

Fig. 7. Timing profile for Charon using strong scaling of 28M unknowns on Sandia Tri-Lab Linux Capability Cluster.

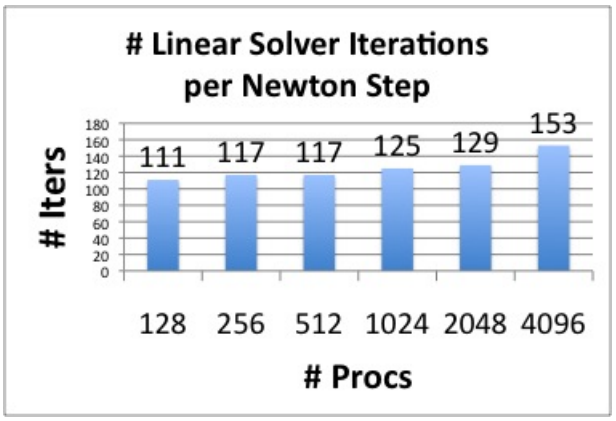

Fig. 8. Average iteration counts as a function of processors used.

One approach to addressing this issue is to use shared memory to reduce the number of subdomains in the solver. Although the application would continue to operate with a single subdomain per core, solver data would be organized to reduce the number of subdomains so that multiple cores would work together. Mathematically, this improves the convergence rate and robustness of the solver. Thus, as long as we can retain or improve parallel performance by using threads on a subdomain, we will see better performance from fewer iterations.

B. Bi-modal HPCPCG

To illustrate bi-modal MPI and MPI+threads programming, we use a miniapplication from the Mantevo project [17], [18] called HPCPCG (HPC preconditioned conjugate gradient). This miniapplication mimics some of the performance characteristics of an unstructured finite volume application. It partitions a 3-dimensional domain in the z dimension and stored data in an unstructured fashion. It implements a conjugate gradient iterative method preconditioned by a symmetric Gauss-Seidel sweep. The basic kernels of HPCPCG are sparse matrix-vector multiplication, vector dot product, vector update and the forward/back sweep of the Gauss-Seidel preconditioner. The forward/back sweep of the symmetric Gauss-Seidel preconditioner can be implemented as one lower triangular solve and one upper triangular solve [19], which allows us to use a previously developed triangular solver implementation [20]. These kernels are representative of a whole class of preconditioned iterative methods, including those used in Charon.

The original implementation of HPCPCG was an MPI-only code. MPI does fairly well on all kernels when compared to threads, except for the forward/back sweep. HPCPCG works like most domain decomposition codes by restricting the sweep to each subdomain assigned to an MPI process. In this way, there is no inter-processor communication and the sweep is done independently on each process. However, this original implementation suffers from iteration inflation, similar to that seen in Figure 8.

Using the shared memory MPI features (described in Section II), we can change the behavior of the preconditioner so that it operates on fewer but larger domains. This can accomplished by having one of the MPI tasks belonging to a given MPI_COMM_NODE communicator be responsible for the preconditioning using a larger domain corresponding to the combined domains of all the MPI tasks on the MPI_COMM_NODE communicator. This MPI task can spawn threads to utilize all the cores on a given node for the preconditioning step. Decreasing the number of domains for the preconditioner should reduce the number of iterations in the PCG algorithm. Assuming the multithreaded preconditioning steps scale, this should result in a reduction in the runtime over the MPI-only application. By using the shared memory MPI extensions, the rest of the application, including the solver, can continue to be implemented in MPI-only mode, with only minor changes to the code.

The preconditioned conjugate gradient method is shown in Figure 9. This figure allows us to outline our new bi-modal HPCPCG implementation, which differs from the original MPI-only HPCPCG implementation in the preconditioning steps. The preconditioning steps, which are implemented using the threaded triangular solve kernels, are encapsulated in the red boxes. Thus, the vectors z_i, the vectors r_i, and the preconditioning matrix M (in reality the factors of M) are allocated using the MPI shared memory extensions since these are the entities involved in the preconditioning. As before, the operations not encapsulated in the red boxes are computing in an MPI-only mode.

$$
\begin{aligned}
& r_0 = b - Ax_0 \\
& z_0 = M^{-1} r_0 \\
& p_0 = z_0 \\
& \text{for } (k = 0;\ k < maxit,\ ||r_k|| < tol) \\
& \{ \\
& \quad \alpha_k = \frac{r_k^T z_k}{p_k^T A p_k} \\
& \quad x_{k+1} = x_k + \alpha_k p_k \\
& \quad r_{k+1} = r_k - \alpha_k A p_k \\
& \quad z_{k+1} = M^{-1} r_{k+1} \\
& \quad \beta_k = \frac{r_{k+1}^T z_{k+1}}{r_k^T z_k} \\
& \quad p_{k+1} = z_{k+1} + \beta_k p_k \\
& \}
\end{aligned}
$$

Fig. 9. Outline of bi-modal MPI-only and MPI+threads HPCPCG implementation for solving $Ax = b$. M represents the symmetric Gauss-Seidel preconditioner. Multithreaded preconditioning steps are encapsulated by red boxes. The remaining lines of the algorithm are computed in MPI-only mode.

C. Level-set triangular solver

An important part of this bi-modal MPI and MPI+threads miniapplication HPCPCG is the multithreaded triangular solve kernel. We use a level-set triangular solve that we previously

implemented [20] and was previously described in [21]. After expressing the data dependencies of the triangular solve for a triangular system as a directed acyclic graph (DAG), a level-set is calculated for this DAG. The level-sets of this DAG represent sets of row operations in the triangular solve operation that can be performed independently. Threads are used to work on these independent sets of row operations with synchronization of the threads (barriers) occurring after each level. This approach is most beneficial for solving triangular systems resulting from incomplete factorizations, where the resulting matrix factors are sufficiently sparse to yield sufficiently large levels. For matrices that do not result in sufficiently large levels, this approach to parallelism will not be particularly effective (as we will see in the subsequent section). We showed that for matrices where the resulting levels are sufficiently large, the synchronization costs in our multithreaded algorithm are small enough to allow for good parallel performance [20].

V. Results

We made the modifications to HPCPCG to obtain the bimodal MPI-only and MPI+threads implementation as described in the previous section. The HPCPCG implementation calls the level-set multithreaded triangular solver (described in [20]), which uses pthreads for multithreaded parallelism. HPCPCG was built with a current development branch of OpenMPI that has implemented the proposed MPI shared memory extensions and architecture aware communicators. We ran our experiments on the Tri-Laboratory Linux Capacity Clusters computing resource glory, which has four quad-core 2.2 GHz AMD processors per node.

In our initial set of numerical experiments, we ran HPCPCG for two different size problems, a 16x16x16 grid and a 32x32x32 grid. We varied the number of threads in the experiments, using one (MPI-only), two, four, and eight threads in the preconditioning step. All runs utilized eight MPI tasks in the rest of the HPCPCG algorithm.

Figures 10 and 11 show the results of our numerical experiments[2]. Figure 10 reports the number of iterations needed for convergence to a specified tolerance for the different versions of the HPCPCG algorithm and the different size problems. As expected the MPI-only version of HPCPCG (magenta bars) requires more iterations to converge than the multithreaded HPCPCG variants that use larger subdomains in their preconditioning.

Figure 11 shows the runtimes of the multithreaded HPCPCG variants relative to the MPI-only version of HPCPCG. A relative runtime less than one corresponds to an improvement over the MPI-only HPCPCG while a relative runtime greater than one corresponds to a variant being more costly than the MPI-only HPCPCG. Although the multithreaded HPCPCG variants had significantly fewer linear iterations in the PCG solver than the MPI-only version, the runtimes of these multithreaded HPCPCG are only slightly improved over than that of the MPI-only version for the two thread variant and worse for the other variants. This increased runtime is due primarily to an increase in the cost of the preconditioning step for the multithreaded HPCPCG versions. The preconditioning cost is growing as the number of threads increases because the level-set multithreaded triangular solver is not scaling well for these triangular systems. The triangular solver implementation does not scale well since the average number of rows per level resulting from the triangular systems is typically small for these small regular grids. For systems with a larger average number of rows per level, we would expect better scalability for the triangular solver and improved runtime for the overall HPCPCG algorithm.

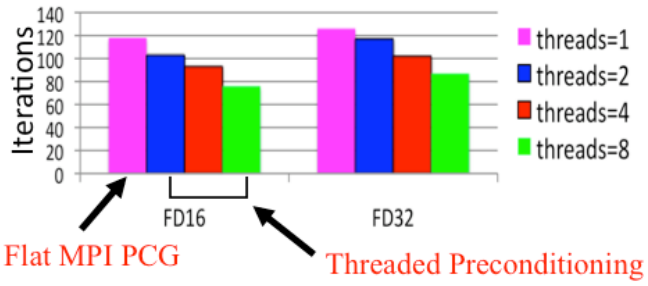

Fig. 10. Iteration counts for two different size (FD16: 16x16x16 grid, and FD32: 32x32x32 grid) HPCPCG experiments as a function of number of threads used.

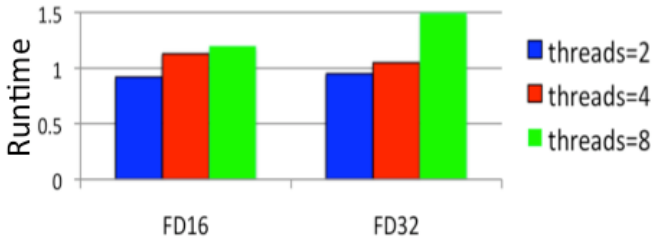

Fig. 11. Runtime (relative to MPI-only HPCPCG) for two different size (FD16: 16x16x16 and FD32: 32x32x32) HPCPCG experiments as a function of number of threads used.

VI. Summary and Conclusions

As the number of cores per node increase in multicore systems, it will become increasingly difficult for many MPI-only applications to scale. At the same time, migrating an entire MPI-only application to MPI+threads is unnecessary at this time, since MPI-only is sufficient for many phases of computation. Thus, we advocate that these applications, moving forward, adopt a bi-modal MPI and MPI+X programming model. The appeal of this approach results from the ease in which threading/hybrid parallelism can be integrated into targeted kernels of an MPI-only application with few changes to the rest of the MPI-only application. For instance, the programmer can replace the MPI-only kernels that are not

[2]These results are preliminary and not indicative of the full potential of this approach due to a bug in our MPI library that severely limits the size of problem we can solve. Based on previous work, we expect our timings to improve markedly before the final version of this paper is due.

scaling with MPI+threaded kernels. In this paper, we focused on bi-modal MPI and MPI+threads as a reasonable approach. We discussed how MPI can naturally provide shared memory extensions and architecture aware communicators to facilitate bi-modal MPI and MPI+threads parallelism. MPI is the natural place for these extensions because the MPI runtime layer is aware of the processor topology and can partition core sets as requested by the user. We described three use cases where using these MPI shared memory extensions can be particularly beneficial: accessing read-only data, setting up connectivity structures, and introducing threading programming into an MPI-only application.

One promising use of this bi-modal MPI and MPI+threads approach is in scalable linear solvers. Such hybrid MPI/threaded algorithms can lower iteration counts by reducing the number of MPI tasks (and subdomains), hopefully allowing the solvers to scale to hundreds of thousands of computational cores on multi-core architectures. We gave a specific example of a bi-modal MPI and MPI+threads approach to linear solvers with the miniapplication HPCPCG. The new bi-modal implementation of HPCPCG uses MPI-only kernels for most of the computation and MPI+threads for the preconditioning in order to mitigate the rising iteration counts due to the number of subdomains in the preconditioner. We used the MPI shared memory extensions to interface this hybrid kernel with the rest of the MPI-only kernels in the miniapplication. We were successful in using the threaded kernels to reduce the iteration count but less successful in reducing the overall runtime of the algorithm. The major problem we faced in reducing the runtime was the performance of the multithreaded triangular solve for this problem. In order to gain more scalability in this linear solver, we need a more scalable multithreaded triangular solve implementation, which we will address in future work.

Acknowledgments

We thank Brian Barrett, Greg Koenig, Geoffroy Vallee for their efforts to define and provide the shared memory MPI capabilities, and Paul Lin and Carter Edwards for the use of their performance data. We thank Erik Boman for his input and help with the preconditioning. This work was funded as part of the Extreme-scale Algorithms and Software Institute (EASI) by the Department of Energy, Office of Science. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration, under contract DE-AC-94AL85000.

REFERENCES

[1] David Shirley, "LIBSM: A Shared Memory Enhancement to MPI programming," http://www.siam.org/meetings/cse00/ms31.htm, Washington, DC, USA, 2000, Presentation at the FIrst SIAM Conference a Computational Science and Engineering.

[2] (2010) Shared memory extensions for mpi. [Online]. Available: http://meetings.mpi-forum.org/secretary/2010/09/slides/brightwell-mpi-shared-memory.pdf

[3] (2011) Cray Inc., The Supercomputer Company - Cray XK6 System. [Online]. Available: http://www.cray.com/Products/XK6/KX6.aspx

[4] H. C. Edwards, "Trilinos threadpool library v1.1," Sandia National Laboratories, Tech. Rep. SAND2009-8196, 2009.

[5] "An updated set of basic linear algebra subprograms (blas)," *ACM Trans. Math. Softw.*, vol. 28, no. 2, pp. 135–151, 2002.

[6] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "Lapack: a portable linear algebra library for high-performance computers," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 2–11.

[7] O. Schenk, K. Gärtner, W. Fichtner, and A. Stricker, "Pardiso: a high-performance serial and parallel sparse linear solver in semiconductor device simulation," *Future Gener. Comput. Syst.*, vol. 18, no. 1, pp. 69–78, 2001.

[8] (2010) Mumps solver. [Online]. Available: http://www.enseeiht.fr/lima/apo/MUMPS

[9] A. Gupta, "Recent advances in direct methods for solving unsymmetric sparse systems of linear equations," *ACM Trans. Math. Softw.*, vol. 28, no. 3, pp. 301–324, 2002.

[10] (2010) Openmp.org. [Online]. Available: http://openmp.org/

[11] (2009) Intel thread building blocks homepage. [Online]. Available: http://www.threadingbuildingblocks.org/

[12] (2010) Multicore Programming Software. [Online]. Available: http://www.cilk.com

[13] (1997) pthread.h. [Online]. Available: http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html

[14] P. Lin, J. Shadid, M. Sala, R. Tuminaro, G. Hennigan, and R. Hoekstra, "Performance of a parallel algebraic multilevel preconditioner for stabilized finite element semiconductor device modeling," *Journal of Computational Physics*, vol. 228, no. 17, pp. 6250–6267, 2009.

[15] G. Hennigan, R. Hoekstra, J. Castro, D. Fixel, and J. Shadid, "Simulation of neutron radiation damage in silicon semiconductor devices," Sandia National Laboratories, Tech. Rep. SAND2007-7157, 2007.

[16] P. T. Lin and J. N. Shadid, "Performance of an MPI-only semiconductor device simulator on a quad socket/quad core InfiniBand platform," Sandia National Laboratories, Tech. Rep. SAND2009-0179, 2009.

[17] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving Performance via Mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[18] Michael A. Heroux, "Mantevo Home Page," http://software.sandia.gov/mantevo, 2008.

[19] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2003.

[20] M. M. Wolf, M. A. Heroux, and E. G. Boman, "Factors Impacting Performance of Multithreaded Sparse Triangular Solve," Sandia National Laboratories, Tech. Rep. SAND2010-0331, 2010, presented at VECPAR'10.

[21] J. H. Saltz, "Aggregation methods for solving sparse triangular systems on multiprocessors," *SIAM Journal on Scientific and Statistical Computing*, vol. 11, no. 1, pp. 123–144, 1990.