Parallel Computing 52 (2016) 133–143

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Hybrid MPI-thread parallelization of adaptive mesh operations

Dan Ibanez*, Ian Dunn, Mark S. Shephard

Scientific Computation Research Center, Rensselaer Polytechnic Institute, 110 Eighth Street, Troy, NY 12180, USA

ARTICLE INFO

Article history:
Received 19 March 2014
Revised 1 November 2015
Accepted 7 January 2016
Available online 15 January 2016

Keywords:
Hybrid system
Shared memory
Parallel programming models
MPI
Termination detection
Non-blocking

ABSTRACT

Many of the world's leading supercomputer architectures are a hybrid of shared memory and network-distributed memory. Such an architecture lends itself to a hybrid MPI-thread programming model. We first present an implementation of inter-thread message passing based on the MPI and pthread libraries. In addition, we present an efficient implementation of termination detection for communication rounds. We use the term phased message passing to denote the communication interface based on this termination detection. This interface is then used to implement parallel operations for adaptive unstructured meshes, and the performance of resulting applications is compared to pure MPI operation. We also present new workflows enabled by the ability to vary the number of threads during runtime.

1. Introduction

The ability to advance parallel processing in an energy efficient manner requires the continued addition of layers to the hardware hierarchies. A key addition that is being shared by all new HPC hardware is a hybrid model combining the benefits of closely packed cores with the benefits of distributed memory by networking together a set of nodes, each of which contains many cores which share in-node RAM.

Operating systems have the analogous concepts of processes, which have unique address spaces and executions, and threads, which have unique executions but shared address spaces. Each process is a thread since it executes, but more threads may be created within a process. All threads in a process share the process' address space. Taking effective advantage of this hybrid software model is essential to scalability and parallel efficiency. However, doing that while not requiring substantial refactoring of large, complex simulation software requires the insertion of a new interface that directly replaces the inter-process communication currently being used by that software.

It is most natural to take advantage of this hardware configuration by writing programs that map processes to nodes and threads to cores. Of course, performance optimization may dictate that certain ratios of processes to nodes and thread to cores is are more optimal, but the general idea remains the same.

All operating system processes start with at least one thread, sometimes called the master thread. Extra threads may be created within a process. Unless otherwise specified, the term "thread" for the remainder of this paper denotes an execution instance that can communicate with others, even in the case when it is just the master thread of a normal process with no extra threads.

* Corresponding author. Tel.: +1 5182278857.
E-mail addresses: dan.a.ibanez@gmail.com, ibaned@rpi.edu (D. Ibanez), dunni@rpi.edu (I. Dunn), shephard@rpi.edu (M.S. Shephard).
URL: http://scorec.rpi.edu/~dibanez (D. Ibanez)

http://dx.doi.org/10.1016/j.parco.2016.01.003
0167-8191/

We present here a refinement of the above approach with additional motivations from a software engineering standpoint. The MPI programming model has been successful in large part due to its elegant concept of communicating parallel processes, and the fact that the hardware actually is composed of communicating parallel nodes, so the abstraction does not introduce excessive overhead. This programming model has formed the basis not only for source code but also the design of scalable algorithms.

Past work has taken the approach of programming two levels of parallelism, one based on communication and another tailored to shared memory. Given the established body of algorithms and source code based on the MPI model and the inherent complexity of using two distinct programming models, a goal of our work is to preserve this single programming model while enabling operation which is hybrid at an operating system and hardware level.

This means providing inter-thread communication capabilities equivalent to those required by existing MPI programs, such that an MPI-based code can be ported to hybrid operation with essentially no modification. Note that, since the on-node performance of a hybrid system is better than its inter-node performance, developers will want to modify software to take advantage of shared memory and other locality. However, such modification can now happen more gradually after the initial port, and at a higher level than the detailed thread management.

Section 2 introduces the standard support for threading and communication on which this work is built, and Section 3 gives an overview of related hybrid systems also built on this support. Section 4 details the implementation of inter-thread message passing at the heart of our system, and Section 5 covers collective operations. The phased algorithm for termination detection is derived in Section 6, which proves useful to unstructured applications regardless of threading model.

These combined capabilities are packaged into a C library called the Parallel Control Utility. PCU is available under the BSD 3-clause license for researchers interested in its source code: https://github.com/SCOREC/core/tree/master/pcu. This library is used to implement a variety of adaptive unstructured mesh operations, which enable them to execute in a hybrid environment. Two specific operations serve as illustrative examples: mesh migration in Section 7 and repartitioning in Section 8. The performance of these mesh operations is presented and discussed in Section 9.

2. Supporting technology

There are several layers of support built into the system software of HPC architectures on which the parallel control utility presented in this paper relies. Our primary target architectures are HPC systems consisting of a large number of distributed memory nodes where each node has a large number of cores sharing common memory. Current testing is done on the IBM Blue Gene/Q, although the design has been kept general enough that our system runs on other targets such as the Intel Xeon Phi.

There are two dominant thread management interfaces, pthreads and OpenMP. The pthreads library is more general, standard, and can directly create and destroy single threads. OpenMP focuses on HPC-style thread parallelism and ease of use, allowing users to execute tasks in a batched fork-join style, which may be implemented underneath using a thread pool [1]. In our work, we adopt a similar model to OpenMP but maintain control of the threads through pthreads. This should allow us to integrate with OpenMP-based codes in the future, by using OpenMP's C function interface.

The other system library of importance is MPI. The MPI standard has introduced a multi-level system of thread safety. That is to say that the MPI library will perform correctly when interacted with by multiple threads per process, but it continues to think in terms of inter-process communication. For initial simplicity, our code uses the MPI_THREAD_MULTIPLE level of support, but we could adjust down to MPI_THREAD_FUNNELED without much difficulty [2]. There are issues involved with the implementation of a thread-safe MPI library [3], and even a testing framework developed to measure the ability of an MPI implementation to support such use [4]. These are performance concerns that can be addressed by moving more of the synchronization responsibility to our user-level code.

The Blue Gene/Q has a few restrictions which are naturally resolved through the use of threads. First, individual process creation by the user is not possible, so changing the degree of parallelism (number of running threads) is only possible by creating threads in processes. Second, the executable instructions for a user's program and its required libraries are loaded into system memory once for each process. By using less processes with multiple threads per process, one gets an equal level of parallelism with reduced memory use by executable instructions. This principle also applies to program structures which are allocated once per process, such as those of the MPI library.

Finally, the MPI implementation on the Blue Gene/Q is configured to create its own helper threads on unused cores to help improve the performance of user programs which do not create threads. However, one should disable this feature in order to free the cores for user-created threads.

3. Related work

The study of hybrid MPI and threads is receiving serious attention at the time of this writing, as are parallel mesh-based simulation methods.

Mavriplis studied the performance of an unstructured mesh Navier-Stokes solver using MPI and OpenMP, as well as the same scheme for using MPI to send messages between threads that we employ in Section 4. He found significant speedup when using a two-level message passing scheme instead, with 140 processes on an SGI-Origin 2000 [5].

On the Blue Gene/Q, Kumar et al. developed a series of fine-grained threading techniques in concert with Charm++ and achieved good scaling of the NAMD molecular dynamics code using high thread per process counts on several hundred thousand cores [6].

Cramer et al. have studied the use of OpenMP on the Intel Xeon Phi chip and show that CPU-targeted code can be ported to it fairly easily with decent speedup [7]. Dokulil et al. created a C++ system for array-based operations which runs in hybrid on the host and Phi co-processor and distributes work based on a parallel-for paradigm [8].

In general, mesh-based simulation methods have entered the billion element, billion degree of freedom era. With such large meshes, there are only a few machines capable of running these simulations. Komatitsch et al. use a Spectral Element Method for earthquake simulation that reaches 14.6 billion degrees of freedom. Running on the Earth Simulator machine, it simulates earthquake propagation over the entire planet [9]. Stowell et al. simulate radar wave propagation through buildings using a 10 billion element regular Cartesian grid mesh [10], and Burstedde et al. use adaptive mesh refinement to create a 12.4 billion element mesh for mantle convection simulations on TACC's Ranger computer [11]. Rasquin et al. [12] simulate the influence of active flow control devices on airfoil drag reduction with meshes of 92 billion elements running on the Mira Blue Gene/Q at Argonne National Laboratory.

4. Inter-thread message passing

The main unusual design choice of PCU compared to other hybrid programming systems is its focus on inter-thread message passing. Since we must rebuild some of the high-level message passing capabilities, we identify a set of primitive operations as described by the MPI standard [13] which are sufficient for our applications:

1. Non-blocking synchronous send
2. Non-blocking send request completion test
3. Non-blocking probe
4. Blocking receive

These conceptual message-passing primitives are independent of their particular implementation. Note that because we require the send operation to be synchronous, it will complete if and only if the message is completely received at its destination.

All the remaining algorithms rely only on these guaranteed properties of the message passing primitives. To develop inter-thread message passing, we implement inter-thread message passing primitives. These are currently based on calling thread-safe MPI versions of the same primitives, but an area of future work involves implementing more efficient primitives.

In order to use MPI itself to pass messages between threads, we require that the implementation correctly handle self-sends. Then, we need to encode the source and destination thread IDs into the message metadata such that messages can be multiplexed out of a single process and demultiplexed at their destination process. The encoding of thread IDs makes use of the standard MPI_TAG metadata integer, which is typically a 32-bit signed integer. We use 10 bits of this integer to encode each of the local IDs for the source and destination threads. This encoding of source and destination means that threads must inspect messages with more sophisticated checking of the tag than MPI_RECV offers, since messages arriving at the same process may be destined for different threads within that process. We use MPI_IPROBE to inspect the tag before using MPI_RECV to commit to being the receiver. This combined probe and conditional receive procedure is specified in Algorithm 1.

5. Non-blocking collectives

Collective operations are a necessary staple of distributed-memory high-performance computing. Operations such as parallel reduction, broadcast, and other collectives are key to coordinating threads [14]. More details on these collective

Algorithm 1 Non-blocking pattern-match receive.

```
function RECEIVE(pattern P)
    let message M ← non-blocking probe
    if M is null (there is no message) then
        return null
    end if
    if metadata of M does not match P then
        return null
    end if
    allocate buffer b per metadata of M
    blocking receive M into b
    return (M, b)
end function
```

algorithms and tradeoffs in their implementation can be found in [15]. Non-blocking collectives are more advanced implementations which are typically used to overlap communication and computation. By using them, PCU benefits from this overlap. Furthermore, in Section 6, we present an algorithm for which blocking collectives are insufficient, and a non-blocking implementation is not just convenient but necessary.

Since collectives are used by nearly all applications, we place a focus on developing built-in multithreaded collective operations based on the inter-thread message passing primitives.

We consider three fundamental collective operations: broadcast, reduce, and scan. Other operations such as exclusive scan and all-reduce can be built from the first three. These operations were selected as the minimal subset of collectives needed for our unstructured mesh operations.

These three collectives share many common features: they use $O(\log n)$ steps for n threads, and at each step each thread is either idle, sending one message, or receiving one message. These shared characteristics make it easier to implement all the collectives in a general framework which abstracts away their differences, starting with the specific communication pattern used. A thread only needs to know which of the three actions to perform at each step and with which thread it is communicating, if any. This combined information is referred to as the communication pattern.

The second abstraction we can make is that of a merge operator, which is essentially the MPI reduction operator (e.g. MPI_SUM or MPI_MAX). The merge operator modifies the local data based on incoming data. For example, an reduction sum adds incoming values to local values. We do not refer to it as the reduction operator because it is used in all cases, including broadcast. As an interesting corner case, the merge operator for broadcast simply assigns the incoming value as the local value.

Following good software design, the communication pattern, merge operator, and data are each specified separately and are orthogonal from one another. This follows the example of interfaces such as MPI_REDUCE.

With these abstract components specified, we can execute a collective operation using the non-blocking point-to-point message passing primitives developed in Section 4. Although the simplicity of collectives would allow blocking primitives to be used, using non-blocking primitives gives us a great benefit: we obtain a non-blocking collective operation. Such operations have been implemented before [16] and subsequently proposed for the latest MPI standard [17]. Our work implements hybrid threaded versions of such collectives.

Users of this system initiate a collective operation, and can interleave computation with communication progress queries. Communication progress consists of checking for incoming messages in the current step and proceeding to the next step when they are received.

Non-blocking collectives are useful from the perspective of of hiding latency, but they will prove to be indispensable to Algorithm 2.

6. Phased message passing

A common problem that arises when dealing with parallel graphs, and similar structures, such as the adjacency relations of unstructured meshes, has to do with transporting graph, or mesh entities, from one thread to another due to changes in the graph or to other operations which affect load balance. Such transportation is specified in a one-sided, push-driven manner, which means that each thread knows which entities it should send to which other threads, but does not know what it will be receiving.

Without *a priori* knowledge of the extent of information to be received, it is difficult to determine when to stop receiving information. A thread can perform a continuous loop which receives messages, but we must determine when to terminate that loop.

This problem has been solved previously in a slightly less efficient manner [18], and is an important special case of the general termination detection problem. Here we present an optimal solution based on non-blocking barriers. The fundamental idea is this: all threads can stop receiving once all messages have been received. Given an individual message, we can use the synchronous non-blocking send provided by MPI to inform the sending thread of when the message is fully received by the receiving thread. In this way, each thread can be informed upon the successful receipt of each message it sends.

Algorithm 2 Phased message passing.

```
1: function PHASE(outgoing messages M)
2:     let R ← the requests from synchronous non-blocking sends of M
3:     while there are incomplete requests in R do
4:         receive and process messages
5:     end while
6:     begin non-blocking barrier
7:     while non-blocking barrier is not done do
8:         receive and process messages
9:     end while
10: end function
```

If each thread keeps track of these send completions then, at some point, each individual thread is aware that every messages it sent has been received. If all threads could synchronize after that event, then we would know that all messages are received. Synchronizing after an event is usually done with a barrier, however in this case there is a problem with using a blocking barrier. If a thread enters a blocking barrier after messages it sent have been received, it may indefinitely ignore incoming messages which it is still responsible for receiving. In other words, the thread is done sending but it is not done receiving. This means that threads must continue to accept messages until the barrier is over, which necessitates a non-blocking barrier. The operation of sending and receiving messages this way is here referred to as a communication phase. The procedure for a single communication phase is more precisely defined by Algorithm 2.

The runtime properties and bounds of Algorithm 2 are vital to the scalability of user applications. As such, we will derive the important runtime properties and show how they are optimal in terms of the target applications.

Given a set of messages M, the time required to fully transmit them can be approximated as $O(\alpha|M| + \beta\|M\|)$, where $|M|$ is the number messages and $\|M\|$ is the total sum of their content sizes, while α and β are arbitrary constants representing latency and inverse bandwidth, respectively.

Let us examine the runtime of Algorithm 2 line-by-line. The runtime of line 2 is simply $O(|M|)$ since sends are non-blocking. The time for the loop on lines 3–5 is the time it takes for sends of M to be acknowledged, which is on the same order as the time required to transmit M since each message just incurs constant added latency waiting for acknowledgement. The loop on lines 7–9 is bounded by the maximum of barrier and receive times. Our non-blocking barrier is implemented using an $O(\log n)$ algorithm where n is the total number of threads. The worst case for receive time is if all incoming messages get received at line 8. Let I be the set of incoming messages, then given linear-time processing the second loop requires at least time $O(\beta\|I\|)$. We will omit latency costs for I since the thread is only receiving and it will not alter our conclusions. In total, the communication phase requires time on the order of:

$$O(\alpha|M| + \beta\|M\| + \max(\beta\|I\|, \log n))$$

In our use cases, $|M|$, $\|M\|$ and $\|I\|$ are not dependent on the number of threads n so, with respect to n, a communication phase is essentially bound by the barrier's $O(\log n)$ time. Note that we do overlap all incoming message processing with all outgoing message wait operations, so the potential for latency hiding is maximized.

When implementing phased message passing, we also optimize performance by buffering messages. The user interface of phased communication allows us to pack all data traveling between the same pair of threads into a single message. This is done prior to executing Algorithm 2. As such, the number of messages sent $|M|$ is equal to the number of unique destinations and the number of messages received $|I|$ is equal to the number of unique sources.

This relation of message counts to communication neighbors (sources and destinations) is quite useful in the process of determining runtime bounds. This is because, due to mesh partitioning, each thread has a small and constant number of neighbors, about 40 at most [19]. This limits the source and destination counts to 40 as well.

We have shown previously that this buffering can greatly improve performance due to the latency cost α and MPI's own management overhead per message, especially for applications with a tendency to send very small messages between the same pair of threads [18]. In the case of multiple threads per process, buffering also reduces the number of calls to MPI_SEND and MPI_RECV, which reduces contention for the MPI library between the threads in that process. When these calls are protected by a lock, avoiding contention is important [5].

7. Migration use case

We now move to describing the application of PCU to massively parallel adaptive finite element simulations. Such simulations make use of unstructured meshes which are being adapted as the simulation is run on a supercomputer [12,20,21]. The mesh is partitioned, giving a subset of its elements to each thread [12,22,23].

There are many situations in which we would like to change the partitioning, i.e. move elements from one thread to another. This operation, called mesh migration [23–25], arises in many scenarios. When described in terms of threads, this operation includes the following types of operations:

1. Changes in the number of elements per thread that must be rebalanced.
2. Threads requiring data from elements on another thread.
3. New threads being created and existing elements moved onto them

Since only the thread which has a particular element knows the full details of the element, threads performing migration typically know only which of their own elements will be sent to other threads, but not which elements they will receive. This is exactly the communication problem described in Section 6, and thus migration is a primary motivator of the phased communication algorithm.

Although the migration algorithm is fairly complex, we can give a simplified definition to illustrate the use of PCU. A detailed description of migration can be found in [24,25]. Each mesh entity (vertices, edges, triangles, and tetrahedra) is an object with adjacency relations to objects of lower dimension, as well as relations to copies representing the same entity on another thread. A pre-processing stage of migration determines, for each entity, the set of threads to which new copies must be sent. One of the existing copies is designated as the "original" copy from which new copies are made. To reconstruct the

Algorithm 3 Send entities.

```
1:  function SEND_ENTITIES(mesh M)
2:      for dimension d from 0 to 3 do
3:          for original copy M_o^d do
4:              let A ← {M_o^d{M^(d−1)}}
5:              for thread t to which M_o^d must be sent do
6:                  let A_R be the copies of A on t
7:                  pack (M_o^d, A_R) in message to t
8:              end for
9:          end for
10:         begin communication phase 1
11:         for (M_o^d, A_R) received from t do
12:             construct M_n^d from A_R
13:             pack (M_o^d, M_n^d) in message to t
14:         end for
15:         begin communication phase 2
16:         for (M_o^d, M_n^d) received from t do
17:             record M_n^d as a remote copy of M_o^d
18:         end for
19:     end for
20: end function
```

mesh on the receiving end, we send entities in order of increasing dimensionality. This ensures new vertices exist when new edges are being constructed, etc. Finally, the remote copy relations must be updated by echoing newly created copies back to the original to collect the new set of remote copies.

These processes are outlined in Algorithm 3. Per the notation in [25], $\{M_o^d\{M^{d-1}\}\}$ denotes the set of entities of dimension $(d-1)$ adjacent to entity M_o^d of dimension (d).

At the beginning of a communication phase, all messages which have been packed up to that point are sent and cleared. During a phase, messages which were sent at the beginning of that phase are received.

The reason that we do not denote a communication phase with a simple call has to do with the non-blocking nature of the phased message passing algorithm. In the for loop given in lines 11–14 of Algorithm 3, the program is simultaneously receiving and processing messages from phase 1, while the termination detection for phase 1 is running. The loop will terminate when the phased algorithm terminates. Part of the message processing, in this case, involves creating content to send in phase 2. The conditions are similar in lines 16–18 in which the loop will terminate when phased message passing detects that there are no messages to receive. In this case, there are no more phases to pack for.

At the end of this algorithm, all new copies have been created with the mesh adjacency relations fully updated [24,25] and the original copies have a full list of all the newly created copies. This is accomplished with only two non-blocking barriers in the background even though threads could not anticipate the sources or lengths of incoming messages.

This background synchronization sits in contrast to synchronization schemes in which all threads are blocked during a barrier. Using PCU, these threads are constantly doing important user-level work, and will only be blocked by the barrier if the amount of user work to do in one phase is faster to complete than the barrier itself.

8. Threaded repartitioning

Ideally, an adaptive finite element simulation would run once on a large supercomputer, accepting a problem description and automatically generating a mesh in parallel, and entering a parallel loop that that iteratively solves the PDEs on that mesh, estimates the discretization errors and improves the mesh until the desired level of solution accuracy is obtained [12,20,21]. However, the relative scalability of the different steps, the fact that the problem size is changing as the simulation continues coupled with the fact that the queuing and operating systems of massively parallel computers does not support the automatic changing of number of cores assigned during a run, motivates an incremental approach, in which a coarse initial mesh is generated using a small number of cores and is scaled up using successively larger machine allocations as the mesh is adapted [12].

Consider the workflow for the simple case of first generating a “coarse” mesh with N parts and e elements, that is successively refined in a series of T steps using some multiplication factor m such that each step multiplies the number of elements and parts by m. At the end of this process, the mesh has (em^T) elements and (Nm^T) parts. Increasing element counts is done using mesh refinement, and increasing part counts is done using a combination of partitioning tools [12].

Although research is ongoing on choosing combinations of partitioning tools, we will focus on the following concrete approach, which has been used successfully to generate meshes of up to 92 billion elements [12,22]. The local graph

partitioner used is the ParMETIS graph partitioner [26], which is in turn being used through Zoltan [27]. The global diffusive repartitioner is ParMA [12,22]. The up-scaling workflow proceeds as follows:

1. An N-part mesh is loaded in parallel on N processors.
2. Mesh refinement multiplies the element count by m.
3. A local graph partitioner separates each part into m parts.
4. A global diffusive repartitioner further improves balance.
5. The resulting Nm parts are output.

Because N processors are used, each process must handle its m output parts serially. We can improve upon this by using Nm processors as follows:

1. An N-part mesh is loaded in parallel on N of the Nm processors.
2. Mesh refinement multiplies the element count by m.
3. A local graph partitioner separates each part into m parts.
4. Migration distributes these parts onto the Nm processors.
5. A global diffusive repartitioner further improves balance.
6. The resulting Nm parts are output.

Now, every step from migration onwards runs with increased parallelism. Since the parallel mesh algorithms are best written for the case when the number of threads equals the number of mesh parts, we would prefer a system that can change the number of threads from N to Nm.

The MPI standard defines MPI_SPAWN, which is a functionality to create new processes, but machines such as the IBM Blue Gene/Q do not support user-triggered operating system process creation for efficiency reasons. On the other hand, they do support user creation of threads within a process.

We take advantage of this by creating m threads on each of N initial processes, thereby creating Nm total threads. PCU's API provides exactly the same message passing capability to these threads, and the user communication code written using PCU is the same when operating on N processes or Nm threads. The migration, diffusive repartitioning, and result output codes were written using PCU, whose programming model is general enough to support all these algorithms.

Because the transition from N to Nm threads requires the initial N threads to send all mesh entities to the other $N(m-1)$ threads, the maximum speedup for the migration step is $2\times$ since only the receiving half is parallelized. However, the subsequent global diffusive repartitioning and output steps receive full Nm parallelism.

In addition, due to the lack of MPI_SPAWN mechanisms, the old approach required two levels of programming: one to handle inter-processor communication and one to handle activity between the m parts on a single process. This new approach allows all algorithms to be written for the simple model of one thread per part, and achieves the desirable outcome of increasing performance while decreasing the size of the source code and allowing the hardware to maintain its efficient restrictions.

9. Results and discussion

In order to test the capability of the hybrid MPI-thread system, PCU, a 1.6 billion element mesh is created using up to 16K cores of an IBM Blue Gene/Q. We then run several stress tests on the resulting mesh to quantify the efficiency of hybrid parallelism.

Mesh generation begins with a 4-part, 400K element tetrahedral mesh and proceeds so as to maintain a part density of 100K elements per part. Each up-scaling repartitioning uses uniform mesh refinement, which multiplies element counts (and thus part counts) by 8. On the Blue Gene, we use 2 processes per node initially which results in 16 threads per node or one thread per core at the end.

Fig. 1 shows the time consumed by each step of the PCU supported workflow as the number of parts is increased from 4 to 16K. The final step converts 2K parts into 16K parts. In this workflow, migration and mesh file writing is always being done using 8 threads per process. By comparison, file reading is done without multithreading, and the times are very comparable even though each process is writing 8 times more data, so thread parallelism is achieved. Migration shows a large increase in runtime as part count is increased. This can be explained by Fig. 2, which shows how the number of neighbor parts to any given part is increasing at the same rate as the mesh scales up.

Recall from Sections 6 and 7 that phased communication and migration times are directly determined by neighborhood sizes. Research shows that there should eventually be a steady-state neighbor count around 40 [19], at which point this runtime will level off.

In Section 8, we predicted a speedup of 2 for the threaded migration in this workflow compared to its non-threaded equivalent. In Fig. 3, we measure the migration time required to divide the 2K part, 200 million element mesh by different factors. The final case results in a 32K part, 200 million element mesh. As this figure shows, we achieve the maximum predicted speedup. It is interesting to also note that the threaded version uses inter-thread message passing while the non-threaded code directly copies memory. If the message passing mechanism were significantly slower than copying, we would have lost much of this speedup.

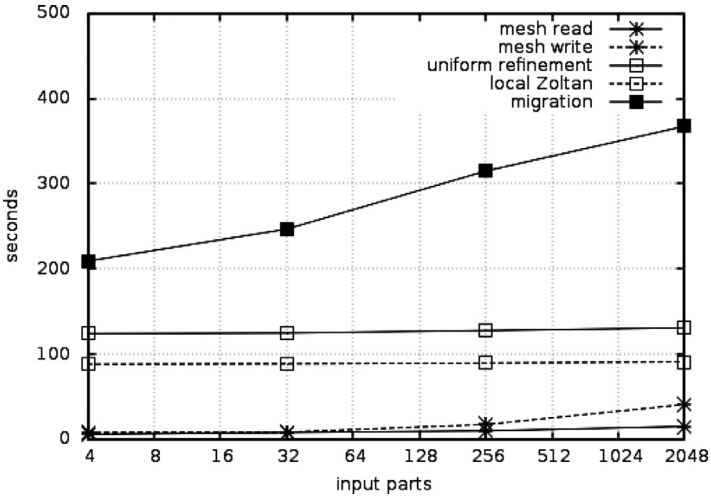

Fig. 1. Times for hybrid up-scaling.

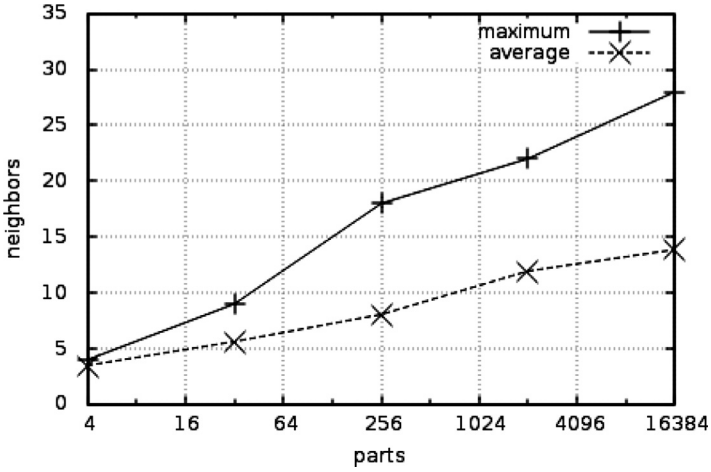

Fig. 2. Neighborhood increase during up-scaling.

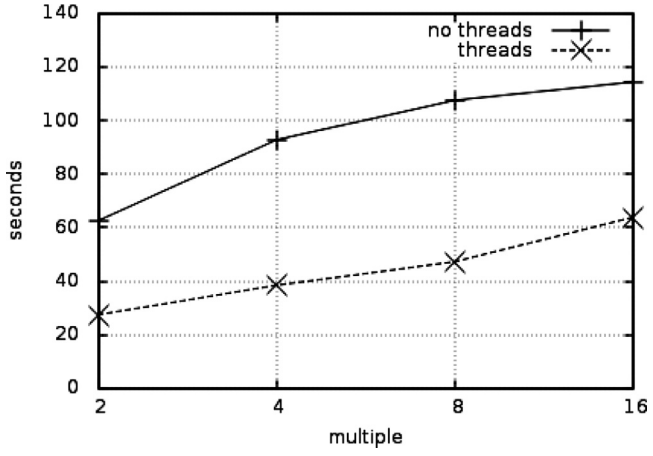

Fig. 3. Threading speedup for repartitioning.

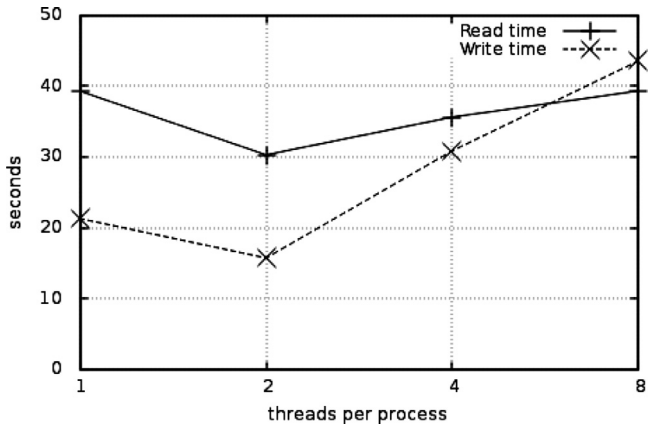

Fig. 4. Hybrid File IO performance.

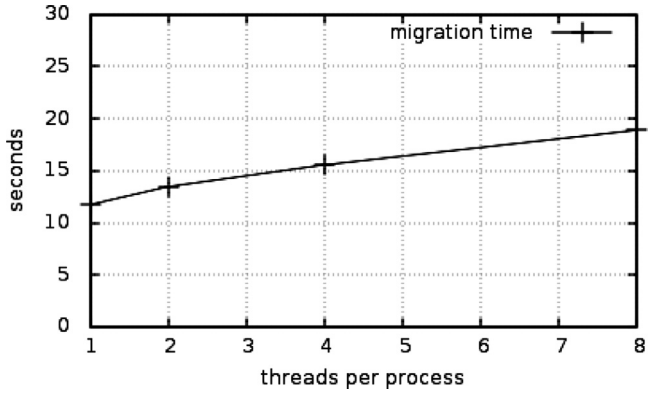

Fig. 5. Hybrid migration performance.

Finally, we study the overhead of threading with a simple workflow. Beginning with N processes and m threads per process such that Nm always equals 16K, we read the 16K-part, 1.6 billion element mesh. In terms of hardware, we are using 16K cores of the Blue Gene/Q, which is 1024 nodes or one full rack. Then every thread migrates 10K elements to one of its neighbors. The resulting mesh is then written out. All operations are done in the hybrid mode, and the number of threads per process m is varied. In all cases, the ideal outcome is that runtime remains the same.

Fig. 4 shows the times for hybrid file IO. File IO is more prone to fluctuation because the filesystem and associated networks are shared by all users. Despite this, we see file IO performance remains fairly constant as we move from process-only to hybrid operation.

Fig. 5 shows migration time over threads per process. Since the ideal result is constant, we see a logarithmic overhead in this part of the workflow. This is most likely attributed to having a single MPI instance per process which must handle the message passing during mutually exclusive calls by threads [5]. We intend to explore a custom implementation of inter-thread message passing in the future to reduce this overhead.

As noted in Section 2, an important benefit of threading is to reduce the amount of static memory used, thus increasing the available memory for user data. Fig. 6 shows the available memory *per thread* at each level of threads per process. The size of the executable used was approximately 90MB, which was copied 16 times initially. Using 16 threads, there is now only one copy of the executable and each thread has about 100MB of additional free memory due to reduced executable copies and other static memory allocated per process. On a machine such as this Blue Gene which allocates about 1GB to each core, this is 10% of the core's total memory.

10. Closing remarks

We have presented an implementation of non-blocking inter-thread message passing from which we build non-blocking collectives and the powerful phased message passing algorithm. These message passing capabilities are used to implement a variety of operations for handling adaptive unstructured meshes, and these operations show good speedup over threads per process. In addition, the use of threads overcomes certain limitations and can lead to greater memory performance than is attainable through processes alone.

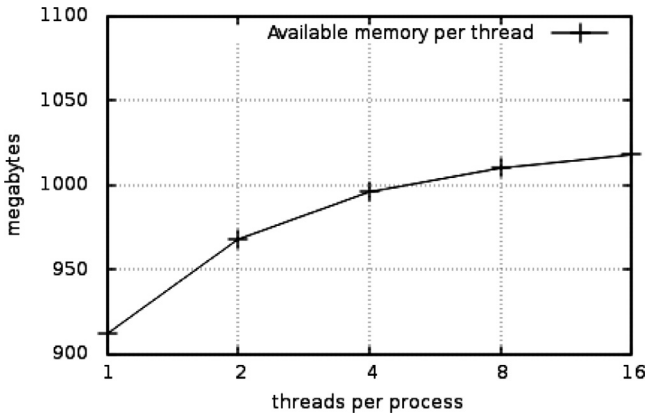

Fig. 6. Available memory with threads.

Acknowledgments

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Grant DE-SC00066117 (FASTMath SciDAC Institute) and in part by the National Science Foundation under Grants OCI-1126125 and IIP-1237555. We gratefully acknowledge the use of the resources of the Rensselaer Center for Computational Innovations and the Leadership Computing Facility at Argonne National Laboratory.

References

[1] L. Dagum, R. Menon, Openmp: an industry standard API for shared-memory programming, IEEE Comput. Sci. Eng. 5 (1) (1998) 46–55.
[2] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, R. Thakur, Toward efficient support for multithreaded MPI communication, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2008, pp. 120–129.
[3] W. Gropp, R. Thakur, Issues in developing a thread-safe MPI implementation, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2006, pp. 12–21.
[4] R. Thakur, W. Gropp, Test suite for evaluating performance of MPI implementations that support mpi_thread_multiple, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2007, pp. 46–55.
[5] D.J. Mavriplis, Parallel performance investigations of an unstructured mesh Navier–Stokes solver, Int. J. High Perform. Comput. Appl. 16 (4) (2002) 395–407.
[6] S. Kumar, Y. Sun, L.V. Kale, Acceleration of an asynchronous message driven programming paradigm on IBM Blue Gene/Q, in: Proceedings of the 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), IEEE, 2013, pp. 689–699.
[7] T. Cramer, D. Schmidl, M. Klemm, D. an Mey, Openmp programming on intel r xeon phi tm coprocessors: an early performance comparison (2012).
[8] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, B. Bachmayer, High-level support for hybrid parallel execution of C++ applications targeting intel xeon phi™ coprocessors, Procedia Comput. Sci., 2013 International Conference on Computational Science 18 (2013) 2508–2511. http://dx.doi.org/10.1016/j.procs.2013.05.430.
[9] D. Komatitsch, S. Tsuboi, C. Ji, J. Tromp, A 14.6 billion degrees of freedom, 5 teraflops, 2.5 terabyte earthquake simulation on the earth simulator, in: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, ACM, 2003, p. 4.
[10] M.L. Stowell, B.J. Fasenfest, D.A. White, Investigation of radar propagation in buildings: a 10-billion element cartesian-mesh fetd simulation, IEEE Trans. Antennas Propag. 56 (8) (2008) 2241–2250.
[11] C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L.C. Wilcox, S. Zhong, Scalable adaptive mantle convection simulation on petascale supercomputers, in: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, IEEE Press, 2008, p. 62.
[12] M. Rasquin, C. Smith, K. Chitale, E.S. Seol, B. Matthews, J. Martin, O. Sahni, R. Loy, M.S. Shephard, K.E. Jansen, Scalable fully implicit finite element flow solver with application to high-fidelity flow control simulations on a realistic wind design, Comput. Sci. Eng. (2014) Submitted to.
[13] D.W. Walker, J.J. Dongarra, MPI: a standard message passing interface, Supercomputer 12 (1996) 56–68.
[14] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G.E. Fagg, E. Gabriel, J.J. Dongarra, Performance analysis of MPI collective operations, Cluster Comput. 10 (2) (2007) 127–143.
[15] R. Thakur, W.D. Gropp, Improving the performance of collective operations in MPICH, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2003, pp. 257–267.
[16] T. Hoefler, A. Lumsdaine, W. Rehm, Implementation and performance analysis of non-blocking collective operations for MPI, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, 2007. SC'07, IEEE, 2007, pp. 1–10.
[17] T. Hoefler, J. Squyres, G. Bosilca, G. Fagg, A. Lumsdaine, W. Rehm, Non-blocking Collective Operations for MPI-3, Technical Report 8, Open Systems Lab, Indiana University, 2006.
[18] A. Ovcharenko, D. Ibanez, F. Delalondre, O. Sahni, K.E. Jansen, C.D. Carothers, M.S. Shephard, Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications, Parallel Comput. 38 (3) (2012) 140–156. http://dx.doi.org/10.1016/j.parco.2011.10.013.
[19] M. Zhou, Petascale adaptive computational fluid dynamics, Rensselaer Polytechnic Institute, 2009 (Ph.D. thesis).
[20] A. Galimov, O.Sahni, R. LaheyJr., M. Shephard, D. Drew, K. Jansen, Parallel adaptive simulation of a plunging liquid jet, Acta Math. Sci. 30B (2010) 522–538.
[21] M. Zhou, O. Sahni, H. Kim, C. Figueroa, C. Taylor, M. Shephard, K. Jansen, Cardiovascular flow simulation at extreme scale, Comput. Mech. 46 (2010) 71–82, doi:10.1007/s00466-009-0450-z.
[22] M. Zhou, O. Sahni, T. Xie, M.S. Shephard, K.E. Jansen, Unstructured mesh partition improvement for implicit finite element at extreme scale, J. Supercomput. 59 (3) (2012) 1218–1228.
[23] M. Zhou, T. Xie, S. Seol, M. Shephard, O. Sahni, K. Jansen, Tools to support mesh adaptation on massively parallel computers, Eng. Comput. 28 (3) (2011) 287–301.
[24] E. Seol, M. Shephard, Efficient distributed mesh data structure for parallel automated adaptive analysis, Eng. Comput. 22 (3-4) (2006) 197–213.

[25] M.S. Shephard, S. Seol, Flexible distributed mesh data structure for parallel adaptive analysis, Adv. Comput. Infrastruct. Parallel Distrib. Adapt. Appl. (2010) 407.
[26] G. Karypis, Metis and parmetis, Encycl. Parallel Comput. 4 (2011) 1117–1124.
[27] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, C. Vaughan, Zoltan data management services for parallel dynamic applications, Comput. Sci. Eng. 4 (2) (2002) 90–96.