Hybrid MPI-thread parallelization of adaptive mesh operations

Dan Ibanez, Ian Dunn, Mark S. Shephard

Parallel Computing 52 (February 2016)

Talk Overview

- → Introduction
- → Problem
- → Solution
- → Results
- → Conclusion

Introduction

- Leading supercomputer architectures are a hybrid of shared memory and network-distributed memory
- Using a hybrid model for adaptive unstructured mesh requires a flexible interface for efficient load balancing
- The programmer needs a way to take advantage for hybrid memory models without major refactoring

Talk Overview

- → Introduction
- → Problem
- → Solution
- → Results
- → Conclusion

Problem

- Two parallel programming models
  - Distributed memory (MPI)
  - Shared Memory (OpenMP)

- Hard to combine these two models from a software engineering perspective
  - Most algorithms are based on the MPI model due to simplicity

- Want to provide a way to maintain MPI model with hybrid applications

Talk Overview

- → Introduction
- → Problem
- → Solution
- → Results
- → Conclusion

Solution

- A new C library was developed called the Parallel Control Utility (PCU)
  - Open Source BSD-3, available at https://github.com/SCOREC/core/tree/master/pcu

- Implements hybrid model using MPI and pthreads
  - Pthreads are more flexible than OpenMP batch forks and joins
  - Implements all the same primitive operations as MPI but for threads
    - Non-blocking synchronous send
    - Non-blocking send request completion test
    - Non-blocking probe
    - Blocking receive

Thread-Safe MPI

- Instead of using MPI_Init, use MPI_Init_Thread

int MPI_Init_thread(int *argc, char *((*argv)[]), int required, int *provided)

- Provided will return what level of thread support will be provided (depending on configuration)
- We will be using MPI_THREAD_MULTIPLE
    - Multiple threads can call MPI with no restrictions
- Can also work with MPI_THREAD_FUNNELED without much difficulty

Encoding of source and destination thread IDs

- Make use of the standard MPI_TAG metadata integer, 32-bit signed integer
  - Use the first 10 bits of this integer to encode each of the local thread IDs
  - This way we can differentiate each thread in a process

- Use MPI_IPROBE to inspect the tag before using MPI_RECV

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
```

Multithreaded Collective Operations

- Implements advanced non-blocking collectives for threads. These are used to overlap communication and computation. Benefits of non-blocking:
  - The overlap allows PCU to check any communication progress
  - Also non-blocking implementation is convenient, hides latency

- Builts 3 fundamental collective operations (based on message passing primitives):
  1. Broadcast
  2. Reduce
  3. Scan

Phased Message Passing

Termination detection problem:

Difficult to determine when to stop receiving without a priori knowledge of the extent of information to be received.

Solution: Phased Message Passing Algorithm

Phased Message Passing

```
Function PHASE(outgoing messages M) {
    Let R <- the requests from synchronous non-blocking sends of M
    While (there are incomplete requests in R do) {
        receive and process messages }

    Begin non-blocking barrier

    While non-blocking barrier is not done do {
        receive and process messages }
}
```

Notes:

- A prior optimization: gather all the data travelling to the same thread into a buffer.
- Mesh Migration is a primary motivator of the phased communication algorithm (in the next slide).

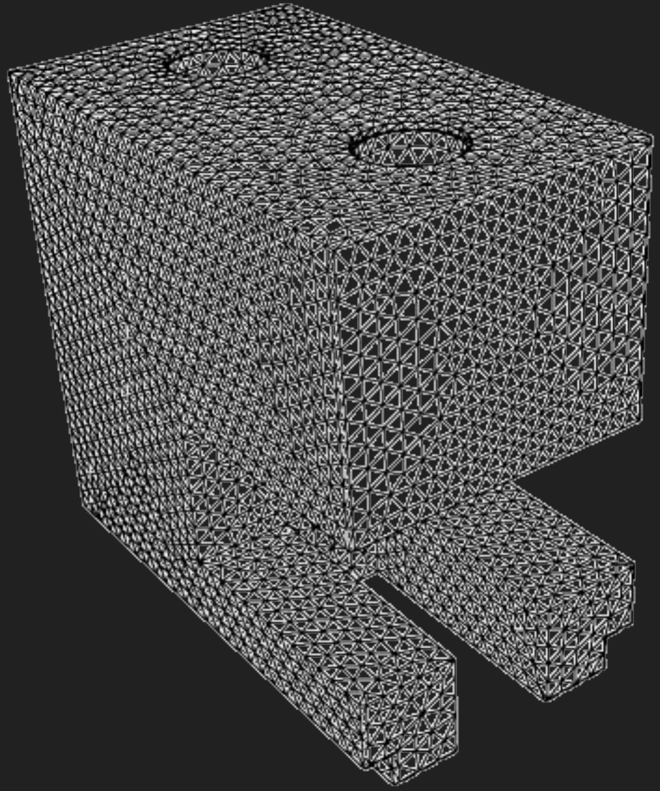
Unstructured Mesh Operation

- Mesh Migration: An operation that changes the partitioning during a mesh simulation. i.e. move elements of the mesh from one thread to another.

- The following operations may lead to a mesh migration:
  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

Explained in later slides...

Mesh Migration Use Case



Algorithm 3 Send entities.

function SEND_ENTITIES(mesh M)
    for dimension d from 0 to 3 **do**
        for original copy M_o^d **do**
            let $A \leftarrow \{M_o^d\{M^{d-1}\}\}$
            for thread t to which M_o^d must be sent **do**
                let A_R be the copies of A on t
                pack (M_o^d, A_R) in message to t
            end for
        end for
        begin communication phase 1
        for (M_o^d, A_R) received from t **do**
            construct M_n^d from A_R
            pack (M_o^d, M_n^d) in message to t
        end for
        begin communication phase 2
        for (M_o^d, M_n^d) received from t **do**
            record M_n^d as a remote copy of M_o^d
        end for
    end for
end function

Tetrahedral Mesh generated with the TetMesh scheme [Digital image]. (n.d.). Retrieved November 24, 2016, from https://cubit.sandia.gov/public/13.2/help_manual/WebHelp/mesh_generation/meshing_schemes/traditional/tetmesh.htm

Threaded Repartitioning

- Problem size is changing as the simulation continues. This motivates an incremental approach:
  - An initial coarse mesh is generated with N parts with total of e elements.
  - The mesh is successively refined in a series of T steps using multiplication factor m.
    - At first step, em elements and Nm parts.
  - After T steps, mesh has (em^T) elements and (Nm^T) parts.
- Up-scaling Workflow:
  - N-part mesh loaded on N of the Nm processors.
  - Mesh refinement multiples the element count by m.
  - A local partitioner separates each part into m parts.
  - Migration distributes these parts onto the Nm processors.
  - A global diffusive repartitioner further improves balance.
  - The resulting Nm parts are output.

T times

Threaded Repartitioning

- Parallel mesh algorithm prefers a system that changes #threads from N to Nm.
- PCU uses pthreads
  - Allows to creates/destroy threads within a process.
  - PCU creates m initial threads for each of the N parts, such that there is one thread per part.

Advantage:
  - Increases parallelism, performance
  - Decreases size of source code
  - Maintains hardware restrictions (one time user-triggered process creation)

Talk Overview

Results

- Created a 1.6 billion element mesh on a 16k core IBM Blue Gene/Q
- Several stress tests were run to test efficiency of PCU hybrid parallelism
- On Blue Gene, we start with 2 processes per node, with 16 threads per node, or 1 thread per core.

Results

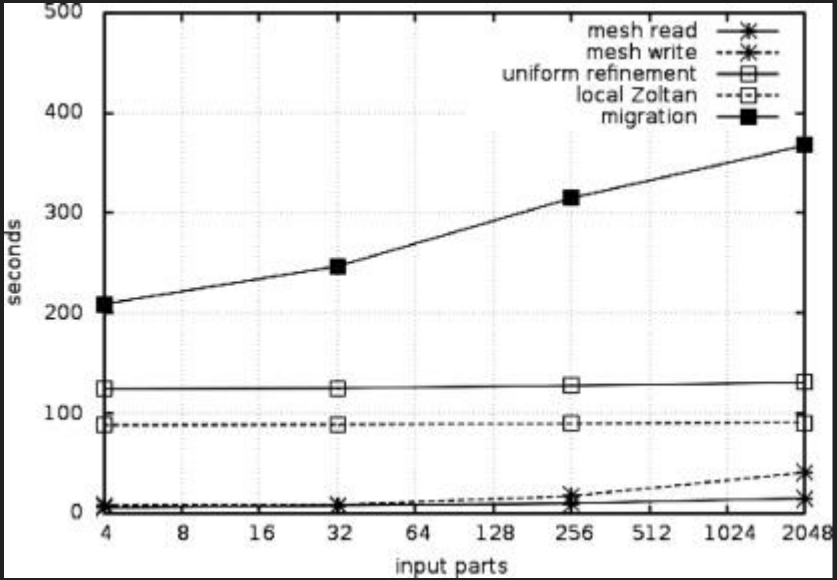


Fig. 1.
Times for hybrid up-scaling.

- Migration and file writing happening with 8 threads per process
- File reading is done without multithreading, yet times are very comparable to writes
- Thread parallelism is achieved
- Large increase in runtime as parts increase.
- Can be explained by number of neighbours

Results

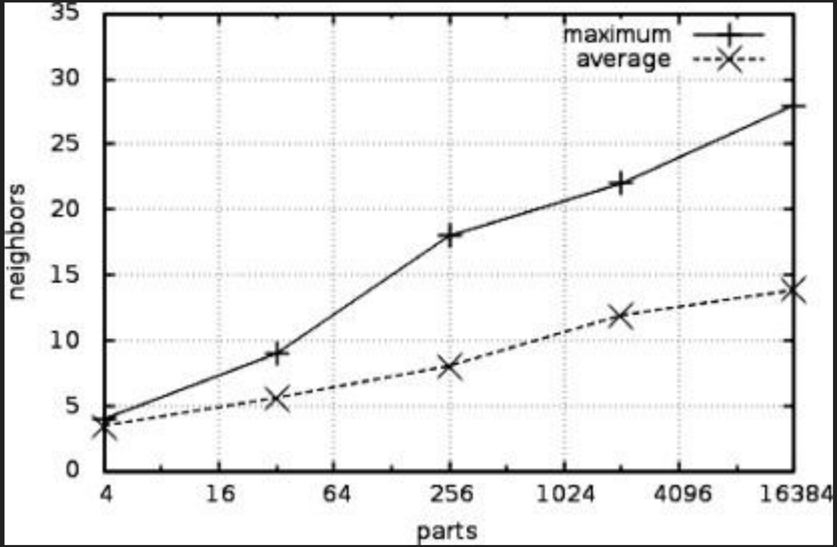


Fig. 2.
Neighborhood increase during up-scaling.

- Migration and file writing happening with 8 threads per process
- File reading is done without multithreading, yet times are very comparable to writes
- Thread parallelism is achieved
- Large increase in runtime as parts increase.
- Can be explained by number of neighbours

Results

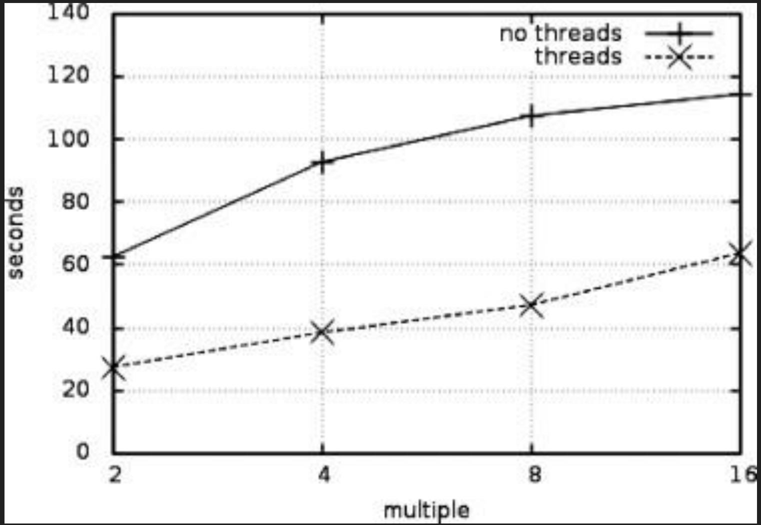


Fig. 3.
Threading speedup for repartitioning.

- Measure of migration time to divide 2k part, 200 million element mesh by different factors
- Results in 32k part, 200 million element mesh
- Paper predicted a speedup of 2 with threads, which was achieved
- Inter-thread message passing is faster than non-threaded memory copy, otherwise we will lose speedup

Results

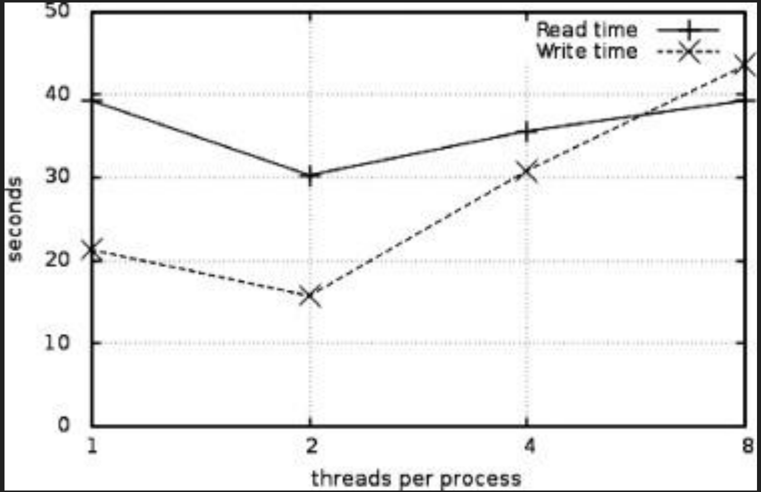


Fig. 4.
Hybrid File IO performance.

- This graph studies the overhead of threading with 16k part, 1.6 B mesh
- Using 16k cores, 1024 nodes, one rack
- Every thread migrates 10k nodes to their neighbours, the resulting mesh is written out.
- File IO performance remains fairly constant, but highly dependent on system use.

Results

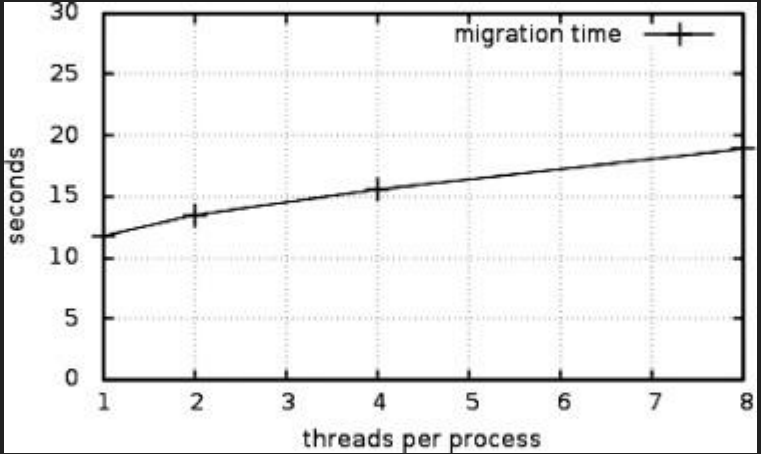


Fig. 5.
Hybrid migration performance.

- Migration time has a logarithmic overhead
- MPI still works by process!
- Single MPI process has to handle mutually exclusive calls by threads
- Intention to explore a custom inter-thread message passing implementation to reduce overhead

Results

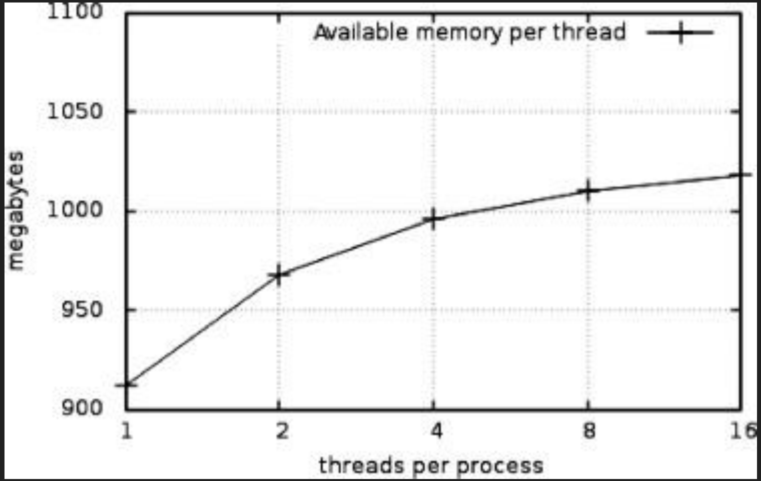


Fig. 6.
Available memory with threads.

- Test of available memory per thread at each number of threads per process
- Executable is approx. 90 MB, copied for each process
- When maxed out at 16 threads per process (1 per node), we get about 100 MB more memory per node then with 1 thread per process
- On the Blue Gene, which allocates 1 GB to each core, that’s 10% of core’s total memory!

Talk Overview

- → Introduction
- → Problem
- → Solution
- → Results
- → Conclusion

Conclusions

- Presented an implementation of non-blocking inter-thread message passing system (PCU).
- Uses phased message passing algorithm for inter-thread communication rounds.
- These message passing capabilities are used to implement adaptive mesh simulation operations.
  - Show good speedup over threads per process.
  - Overcome hardware limitations.
  - Lead to greater memory performance (than using processes alone)