MPI and Hybrid Programming Models

William Gropp

www.cs.illinois.edu/~wgropp

What is a Hybrid Model?

- Combination of several *parallel* programming models in the same program
  - May be mixed in the same source
  - May be combinations of components or routines, each of which is in a single parallel programming model
- MPI + Threads or MPI + OpenMP is the most familiar hybrid model (that involves MPI)
  - There are other interesting choices for which we should prepare

Why a Hybrid Model?

- Note that in some ways MPI is already a hybrid programming model (MPI + C; MPI + Fortran)
  - Adding a third programming model is not a major change…
- Also note that many *applications* are multilingual, built from pieces in C, C++, Python, Matlab, …
  - Developers use the best tool for each part of their program
- Scale of machines to come encourage the use of different programming models to address issues such as
  - Declining memory per core
  - Multiple threads/core
  - Load balance
  - Algorithmic issues

MPI and Threads

- MPI describes parallelism between processes (with separate address spaces)
- Thread parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.

Programming for Multicore

- Almost all chips are multicore these days
- Today's clusters often comprise multiple CPUs per node sharing memory, and the nodes themselves are connected by a network
- Common options for programming such clusters
  - All MPI
    - Use MPI to communicate between processes both within a node and across nodes
    - MPI implementation internally uses shared memory to communicate within a node
  - MPI + OpenMP (or MPI + OpenACC)
    - Use OpenMP within a node and MPI across nodes
  - MPI + Pthreads
    - Use Pthreads within a node and MPI across nodes
- The latter two approaches are known as "hybrid programming"

Myths About the MPI + OpenMP Hybrid Model

1. Never works
   - Examples from FEM assembly, others show benefit
2. Always works
   - Examples from NAS, EarthSim, others show MPI everywhere often as fast (or faster!) as hybrid models
3. Requires a special thread-safe MPI
   - In many cases does not; in others, requires a level defined in MPI-2
4. Harder to program
   - Harder than what?
   - Really the classic solution to complexity - divide problem into separate problems
     - 10000-fold coarse-grain parallelism + 100-fold fine-grain parallelism gives 1,000,000-fold total parallelism

Special Note

- Because neither 1 nor 2 are true, and 4 isn't entirely false, it is important for applications to engineer codes for the hybrid model. Applications must determine their:
  - Memory bandwidth requirements
  - Memory hierarchy requirements
  - Load Balance
- Don't confuse problems with getting good performance out of OpenMP with problems with the Hybrid programming model
- See *Using OpenMP* by Barbara Chapman, Gabriele Jost and Ruud van der Pas, Chapters 5 and 6, for programming OpenMP for performance
  - See pages 207-211 where they discuss the hybrid model

MPI's Four Levels of Thread Safety

- MPI defines four levels of thread safety. These are in the form of commitments the application makes to the MPI implementation.
  - MPI_THREAD_SINGLE: only one thread exists in the application
  - MPI_THREAD_FUNNELED: multithreaded, but only the main thread makes MPI calls (the one that called MPI_Init or MPI_Init_thread)
  - MPI_THREAD_SERIALIZED: multithreaded, but only one thread at a time makes MPI calls
  - MPI_THREAD_MULTIPLE: multithreaded and any thread can make MPI calls at any time (with some restrictions to avoid races – see next slide)

Specifying the Level of Thread Safety

- MPI defines an alternative to MPI_Init
  - MPI_Init_thread(argc, argv, requested, provided)
    - Application indicates what level it needs; MPI implementation returns the level it supports
- Many (not all) builds of MPICH exploit this runtime control
  - If you don’t need thread safety, there is little extra cost

Specification of MPI_THREAD_MULTIPLE

- When multiple threads make MPI calls concurrently, the outcome will be as if the calls executed sequentially in some (any) order
- Blocking MPI calls will block only the calling thread and will not prevent other threads from running or executing MPI functions
- It is the user's responsibility to prevent races when threads in the same application post conflicting MPI calls
  - e.g., accessing an info object from one thread and freeing it from another thread
- User must ensure that collective operations on the same communicator, window, or file handle are correctly ordered among threads
  - e.g., cannot call a broadcast on one thread and a reduce on another thread on the same communicator

PARALLEL@ILLINOIS

Threads and MPI in MPI-2 (and MPI-3)

- An implementation is not required to support levels higher than MPI_THREAD_SINGLE; that is, an implementation is not required to be thread safe
- A fully thread-safe implementation will support MPI_THREAD_MULTIPLE
- A program that calls MPI_Init (instead of MPI_Init_thread) should assume that only MPI_THREAD_SINGLE is supported

The Current Situation

- All MPI implementations support MPI_THREAD_SINGLE (duh).
- They probably support MPI_THREAD_FUNNELED even if they don’t admit it.
    - Does require thread-safe malloc
    - Probably OK in simple OpenMP programs
- Many (but not all) implementations support THREAD_MULTIPLE
    - Hard to implement efficiently though (lock granularity issue)
- “Easy” OpenMP programs (loops parallelized with OpenMP, communication in between loops) only need FUNNELED
    - So don’t need “thread-safe” MPI for many hybrid programs
    - But watch out for Amdahl’s Law!

What MPI's Thread Safety Means in the Hybrid MPI+OpenMP Context

- MPI_THREAD_SINGLE
  - There is no OpenMP multithreading in the program.
- MPI_THREAD_FUNNELED
  - All of the MPI calls are made by the master thread. i.e. all MPI calls are
    - Outside OpenMP parallel regions, or
    - Inside OpenMP master regions, or
    - Guarded by call to MPI_Is_thread_main MPI call.
      - (same thread that called MPI_Init_thread)
- MPI_THREAD_SERIALIZED

```
#pragma omp parallel
…
#pragma omp single
{
  …MPI calls allowed here…
}
```

- MPI_THREAD_MULTIPLE
  - Any thread may make an MPI call at any time

Some Things to Watch for in OpenMP

- No standard way to manage memory affinity
  - "First touch" (have intended "owning" thread perform first access) provides initial static mapping of memory
    - Next touch (move ownership to most recent thread) could help
  - No portable way to reassign affinity – reduces the effectiveness of OpenMP when used to improve load balancing.
- Memory model can require explicit "memory flush" operations
  - Defaults allow race conditions
  - Humans notoriously poor at recognizing all races
    - It only takes one mistake to create a hard-to-find bug

Some Things to Watch for in MPI + OpenMP

- No interface for apportioning resources between MPI and OpenMP
  - ♦ On an SMP node, how many MPI processes and how many OpenMP Threads?
    - Note the static nature assumed by this question
  - ♦ Note that having more threads than cores can be important for hiding latency
    - Requires very lightweight threads
- Competition for resources
  - ♦ Particularly memory bandwidth and network access
  - ♦ Apportionment of network access between threads and processes is also a problem, as we've already seen.

Where Does the MPI + OpenMP Hybrid Model Work Well?

- Compute-bound loops
  - Many operations per memory load
- Fine-grain parallelism
  - Algorithms that are latency-sensitive
- Load balancing
  - Similar to fine-grain parallelism; ease of
- Memory bound loops

Compute-Bound Loops

- Loops that involve many operations per load from memory
  - This can happen in some kinds of matrix assembly, for example.
  - "Life" update partially compute bound (all of those branches)
  - Jacobi update not compute bound

Fine-Grain Parallelism

- Algorithms that require frequent exchanges of small amounts of data
- E.g., in blocked preconditioners, where fewer, larger blocks, each managed with OpenMP, as opposed to more, smaller, single-threaded blocks in the all-MPI version, gives you an algorithmic advantage (e.g., fewer iterations in a preconditioned linear solution algorithm).
- Even if memory bound

Load Balancing

- Where the computational load isn't exactly the same in all threads/ processes; this can be viewed as a variation on fine-grained access.
- More on this later (OpenMP currently not well-suited, unfortunately. An option is to use Pthreads directly.)

PARALLEL@ILLINOIS

Memory-Bound Loops

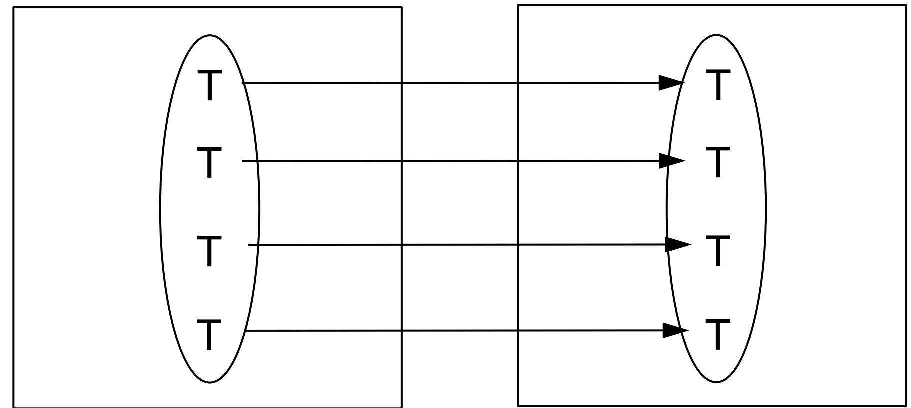
- Where read data is shared, so that cache memory can be used more efficiently.
- Example: Table lookup for evaluating equations of state
  - Table can be shared
  - If table evaluated as necessary, evaluations can be shared

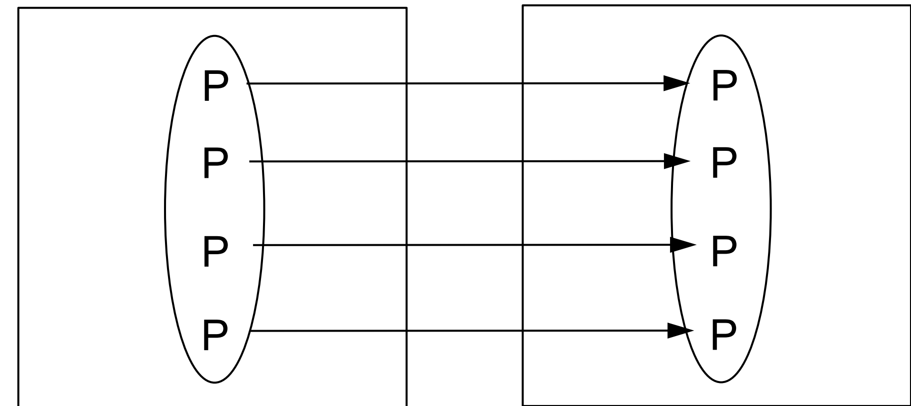
Where is Pure MPI Better?

- Trying to use OpenMP + MPI on very regular, memory-bandwidth-bound computations is likely to lose because of the better, programmer-enforced memory locality management in the pure MPI version.
- Another reason to use more than one MPI process - if a single process (or thread) can't saturate the interconnect - then use multiple communicating processes or threads.
  - Note that threads and processes are not equal - see next slides

Tests with Multiple Threads versus Processes

- Consider these two cases:
  - Nodes with 4 cores
  - 1 process with four threads sends to 1 process with four threads, each thread sending, or
  - 4 processes, each with one thread, sending to a corresponding thread
- User expectation is that the performance is the same
- Results are joint work with Rajeev Thakur (Argonne)

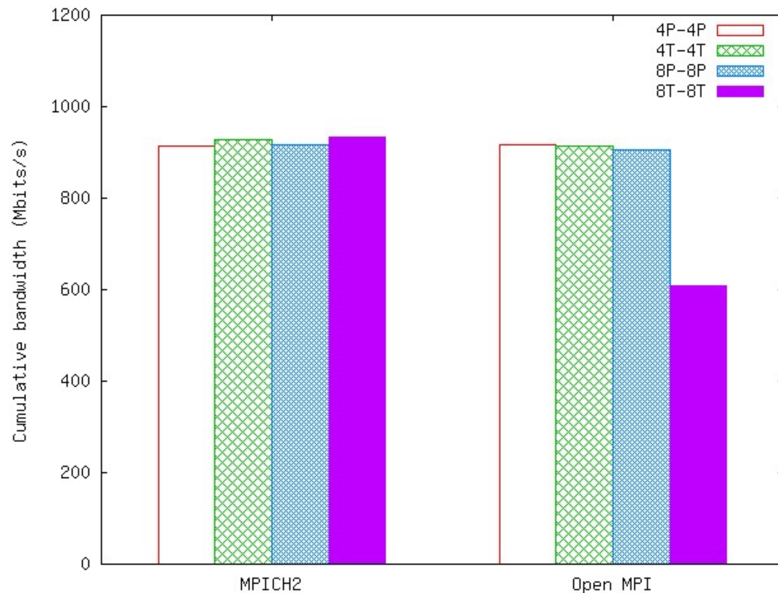


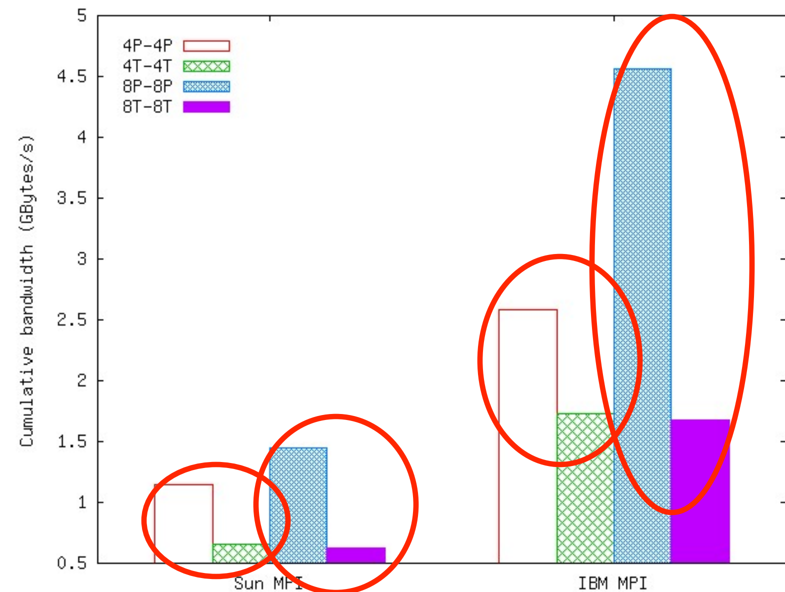


1867

PARALLEL@ILLINOIS

Concurrent Bandwidth Test





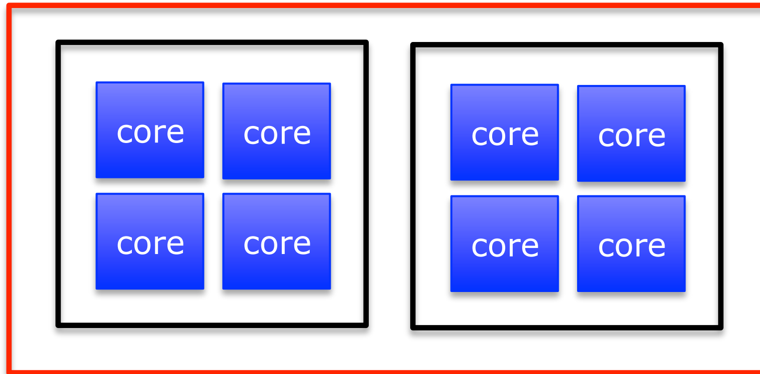
Lesson: Its hard to provide full performance from threads

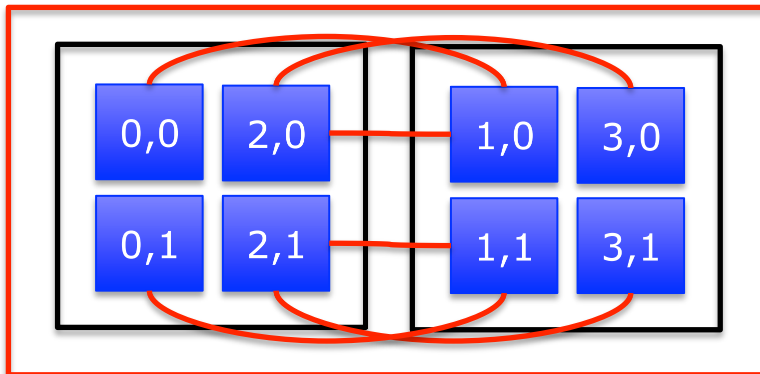
(Recent results on current platforms show similar behavior)

Locality is Critical

- Placement of processes and threads is critical for performance
  - ♦ Placement of processes impacts use of communication links; poor placement creates more communication
  - ♦ Placement of threads within a process on cores impacts both memory and intranode performance
    - Threads must bind to preserve cache
    - In multi-chip nodes, some cores closer than others – same issue as processes
- MPI has limited, but useful, features for placement

Importance of ordering processes/ threads within a multichip node





- 2x4 processes in a mesh
- How should they be mapped onto this single node?
- Round robin (by chip)?
    - Labels are coordinates of process in logical computational mesh
    - Results in 3x interchip communication than the natural order
    - Same issue results if there is 1 process with 4 threads on each chip, or 1 process with 8 threads on the node

Hybrid Model Options: Fine

- Fine grain model:
  - Program is single threaded except when actively using multiple threads, e.g., for loop processing
  - Pro:
    - Easily added to existing MPI program
  - Con:
    - Adds overhead in creating and/or managing threads
    - Locality and affinity may be an issue (no guarantees)
    - Amdahl's Law problem – serial sections limit speedup

PARALLEL@ILLINOIS

Hybrid Model Options: Coarse

- Coarse grain model
  - ♦ Majority of program runs within "omp parallel"
  - ♦ Pro:
    - Lowers overhead of using threads, including creation, locality, and affinity
    - Promotes a more parallel coding style
  - ♦ Con:
    - More complex coding, easier to introduce race condition errors

Challenges for Programming Models

- Parallel programming models need to provide ways to <u>coordinate</u> resource allocation
  - ♦ Numbers of cores/threads
  - ♦ Assignment (affinity) of cores/threads
  - ♦ Intranode memory bandwidth
  - ♦ Internode memory bandwidth
- They must also provide clean ways to share data
  - ♦ Consistent memory models
  - ♦ Decide whether its best to make it easy and transparent for the programmer (but slow) or fast but hard (or impossible, which is often the current state)
- Remember, parallel programming is about performance
  - ♦ You will always get higher programmer productivity with a single threaded code

Challenges for Implementations

- Sharing of communication infrastructure
  - ♦ For example, the Berkeley UPC implementation makes use of GASNET, an efficient, portable communication layer
  - ♦ But GASNET does not provide all of the features required by an efficient, full MPI implementation
  - ♦ Similarly, the communication layer used by MPI implementations may not provide all of the features needed by UPC (see "Problems with using MPI 1.1 and 2.0 as compilation targets for parallel language implementations", Dan Bonachea, Jason Duell)
- It is possible to build such infrastructures
  - ♦ But current examples only address some of the issues.
  - ♦ Resource allocation and sharing not covered

Conclusions

- Hybrid programming models exploit complementary strengths
  - In many cases, can replace OpenMP with OpenACC or other accelerator programming system
- Evolutionary Path to Hybrid Models
  - Short term - better support for resource sharing
    - We need to experiment with specifying additional information, e.g., through mpiexec
  - Medium term - better support for interoperating components
    - We need to ensure that communication infrastructures can cooperate
    - Consider extensions to make implementations aware that they are in a hybrid model program
  - Long term - Generalized model, efficient sharing of communication and computation infrastructure