Programming Options for "Hybrid" Architectures

- **Pure MPI** – each core runs an MPI process
  - new MPI-3 support for shared memory makes MPI+MPI "hybrid" programming a viable option*
- **Pure OpenMP**
  - single process, fully multi-threaded
  - virtual distributed shared address space
- **MPI and OpenMP**
  - non-overlapped ("Masteronly") – only a master thread makes MPI calls, while no other threads are active
  - overlapped - many interesting approaches here

* T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, R. Thakur: MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory. *Computing*, 95(12):1121– 1136, December 2013.

Reasons to Add OpenMP

- OpenMP can be a more efficient solution for *intra-node* parallelism
  - uses less memory than MPI
  - more efficient for *fine-grained* parallelism
  - may require use within **NUMA** nodes

- Constraint on total number of MPI processes that can be used for application
  - per-node memory limits
  - system limits on number of processes that can be spawned
  - application doesn’t scale past a certain number of MPI processes

- Application exhibits hierarchical parallelization pattern
  - natural to use MPI for top-level, and OpenMP for second level

- Unbalanced MPI workloads – can assign more threads to heavily-loaded MPI processes

Reasons to be Cautious

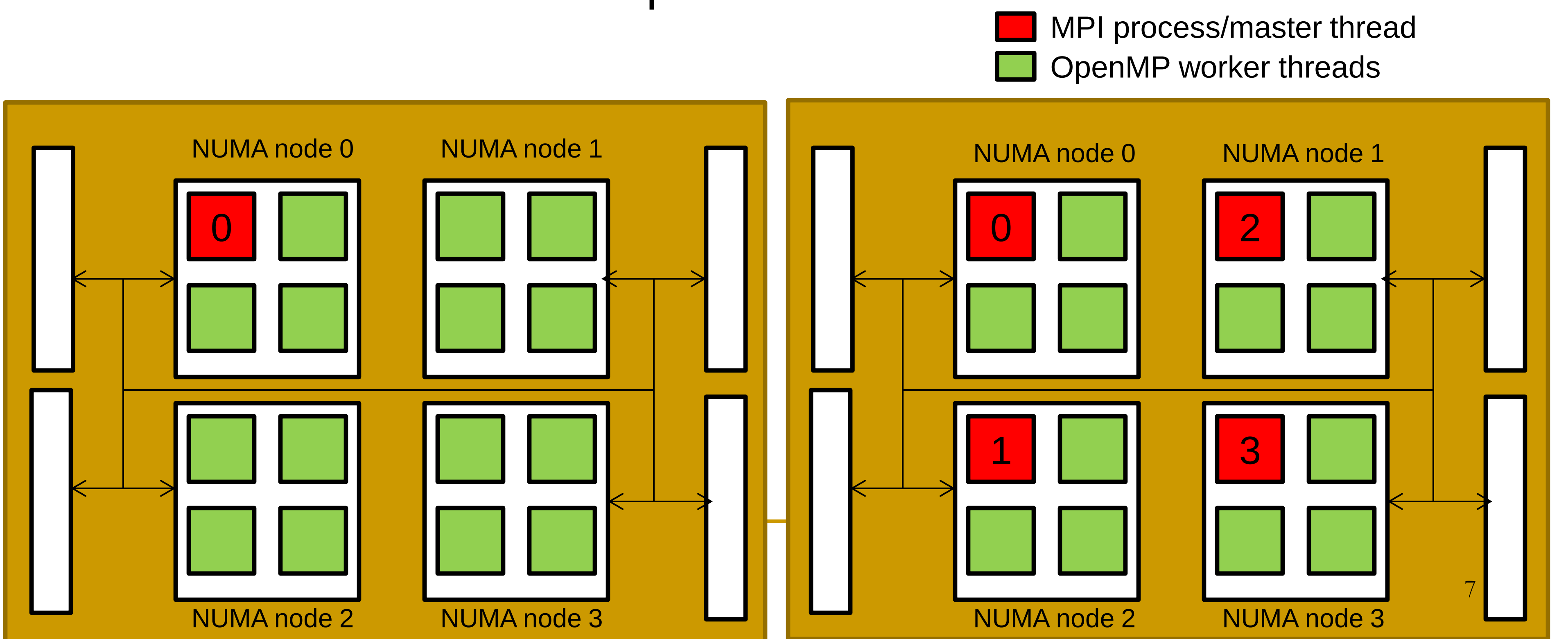
- Interoperability issues between MPI and OpenMP implementations
  - is MPI library thread-safe?
  - how might presence of additional threads impact MPI's performance?

- Added complexity in program - beware of shared memory programming pitfalls such as data races or false sharing

- If limiting communication to a single thread, are we still able to saturate the network?

NUMA considerations

- NUMA, Non-Uniform Memory Access
  - this is a common case for your compute nodes
  - Nodes -> (NUMA nodes) Sockets -> Cores -> H/W Threads
  - consideration of process/thread assignment to cores is critical for performance



Resource Utilization Considerations

- Network Utilization
  - if only one MPI process per node, can we still saturate the network port?
  - usually yes, but maybe not if multiple network ports become commonplace in the near future
- Core Utilization
  - Threads can help overlap computation with communication
  - Can also help balance workloads through worksharing constructs
  - However: sleeping threads (“Masteronly” mode) will limit core utilization

Hybrid Programming in Practice

- Typically start with an MPI program, and you use OpenMP to parallelize it
  - loop parallelism
  - task parallelism
  - SIMD and Accelerators (next talk: OpenMP 4.0)
- Strategies
  - vary number of threads based on workload in each process
  - find best mapping of threads to cores
  - use threads to overlap computation with MPI calls for more asynchronous progress
  - generally requires experimentation to find best combination (e.g. # processes, # threads/process, thread affinity)

MPI Thread Support Modes (Recap)

- Request/get thread support mode using call to MPI_Init_thread instead of MPI_Init

- MPI_THREAD_SINGLE (default with MPI_Init)
  - assume MPI process is not multi-threaded

- MPI_THREAD_FUNNELED
  - multi-threaded processes allowed
  - only one designated thread is making MPI calls

- MPI_THREAD_SERIALIZED
  - multi-threaded, and multiple threads may make MPI calls
  - calls must be serialized

- MPI_THREAD_MULTIPLE
  - multi-threaded, no restrictions
  - requires *fully* thread-safe MPI implementation

Example: MPI_THREAD_FUNNELED

```
#include <mpi.h>

int main(int argc, char **argv)
{
    int rank, size, ierr, i, provided;
    MPI_Init_thread(&argc,&argv,
                    MPI_THREAD_FUNNELED,
                    &provided);
  ...
  #pragma omp parallel
{
  #pragma omp master
   { ... MPI calls ... }
  #pragma barrier
  #pragma omp for
   for (i = 0; i < N; i++) {
        do_something( i );
    }
    ...
```

call MPI_Init_thread to request MPI_THREAD_FUNNELED

now we can do MPI in parallel region
(NOTE: master construct ensures its the same thread which does it)

REMEMBER: if using master, we may also need a barrier

Example: MPI_THREAD_SERIALIZED

```
...

   MPI_Init_thread(&argc,&argv,
                   MPI_THREAD_SERIALIZED,
                   &provided);
  ...
  #pragma omp parallel
{
   ...
   #pragma omp single
    { ... MPI calls ... }

   #pragma omp for
    for (i = 0; i < N; i++) {
          do_something( i );
    }
    ...
```

With SERIALIZED, we can now use a SINGLE construct for more flexibility.

NOTE: Use nowait clause if you wish to avoid implicit barrier at the end and obtain overlap

Example: MPI_THREAD_MULTIPLE

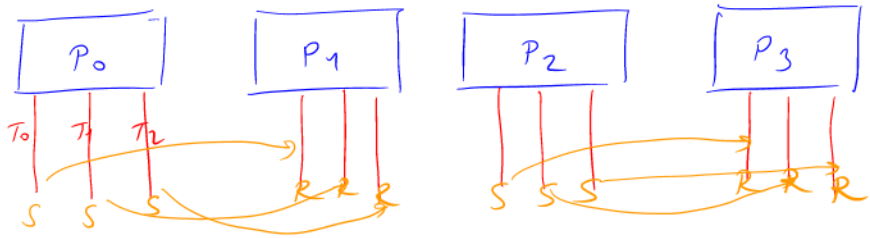
```
...

  MPI_Init_thread(&argc,&argv,
                  MPI_THREAD_MULTIPLE,
                  &provided);
 ...
  #pragma omp parallel
{
    tid = omp_get_thread_num();
    ...
    if (mpi_rank % 2) {
        MPI_Send(data, N, MPI_INT, mpi_rank-1, tid, ... );
    } else {
        MPI_Recv(data, N, MPI_INT, mpi_rank+1, tid, ... );
    }
     ...
}
```

With MULTIPLE, no restrictions on using MPI calls in a parallel region.

P = 4 processes

N = 3 threads for each processes



6 send's

6 recv's

12 p2p comm. routines.

MPI_THREAD_MULTIPLE

MPI is thread-safe ✓

Hiding Communication Latency using OpenMP

- MPI communication is often blocking
  - even non-blocking calls may require MPI calls to achieve progress
  - hardware support and/or helper threads might help, but often not available
- Strategies using OpenMP
  - use an “explicit” SPMD approach
  - use nested parallel region
  - use tasks

Achieving Overlap using a SPMD approach

```
...
  MPI_Init_thread(...);
 ...
 #pragma omp parallel
{
    tid = omp_get_thread_num();
    ...
    if (tid == 0) {
       /* first thread does MPI stuff */
    } else {
       /* remaining threads carry on with independent
          computation */
    }
    #pragma omp barrier
}
```

Here we divide thread team into two "subteams" using thread ID.

Main Issue:

- work-sharing constructs in "else" block are unavailable to us
- requires explicit coding of work-sharing, cumbersome and inflexible

Achieving Overlap using Nested Parallelism

```
...
  omp_set_nested(true);
  ...
 #pragma omp parallel num_threads(2)
{
    tid = omp_get_thread_num();
    ...
    if (tid == 0) {
      /* do MPI stuff */
    } else {
      /* thread 1 spawns a new parallel region to do work */
      #pragma omp parallel
      { ... }
    }
    ...
}
```

nested parallel region here can perform all work-sharing constructs independent of the MPI communication by thread 0

Achieving Overlap using nowait clause

```
...
   MPI_Init_thread(...);
  ...
 #pragma omp parallel
{
   #pragma omp master
   { /* first thread does MPI stuff */ }

    /* remaining threads continue with other work */
   #pragma omp for schedule (...) nowait
    for(...) { ... }
   #pragma omp for schedule(...) nowait
    for(...) { ... }
    ...
}
```

This approach allows us to utilize all threads (including, eventually, the MPI-designated thread(s)) for doing computation

Achieving Overlap using explicit tasks

```
...
  MPI_Init_thread(...);
 ...
 #pragma omp parallel
{
   ...
   #pragma omp master
   {
     for (...) {
     #pragma omp task
       { /* create tasks for other threads to work on */ }
     }
     /* after task creation, master does MPI stuff*/
   }

   #pragma omp barrier
   ...
}
```

Here, the master creates tasks which may be picked up by the other threads.

Recall: barriers are task scheduling points.

Summary

- Technological trends makes hybrid programming all the more important
  - “fatter” nodes with cc-NUMA characteristics
  - reduced memory available per core
  - extreme-scale computing will require dynamic, load balancing strategies
- With OpenMP, you can
  - develop more memory-efficient algorithms for within the node
  - “workshare” among threads using various scheduling policies, to curtail load imbalance
  - hide communication latency using a variety of strategies
- As always, choose the best programming system for your problem.

Common Sources of Errors

- Wrong “spelling” of sentinel
- Wrongly declared data attributes (shared vs. private, firstprivate, etc.)
- Incorrect use of synchronization constructs
  - Less likely if user sticks to directives
  - Erroneous use of locks can lead to deadlock
  - Erroneous use of NOWAIT can lead to race conditions.
- Race conditions (true sharing)
  - Can be very hard to find

It can be very hard to track race conditions. Tools may help check for these, but they may fail if your OpenMP code does not rely on directives to distribute work. Moreover, they can be quite slow.

Care with Synchronization

- Recall that a thread's temporary view of memory may vary from shared memory
  - Value of shared objects updated at synchronization points
  - User must be aware of the point at which modified values are (guaranteed to be) accessible
- Compilers routinely reorder instructions that implement a program
  - Helps exploit the functional units, keep machine busy
- Compiler cannot move instructions past a barrier
  - Also not past a flush on all variables
  - But it can move them past a flush on a set of variables so long as those variables are not accessed

Race Condition

- Several threads access and update shared data concurrently
  - One thread writes and one or more threads read or write same memory location at about the same time
  - Outcome depends on relative ordering of operations and may differ between runs
- User is expected to avoid race conditions
  - insert synchronization constructs as appropraite, or
  - privatize data
- Some tools exist to detect data races at runtime
  - e.g. Intel Thread Checker, Oracle Solaris Studio Thread Analyzer