Hybrid Parallel Overview

Steve Lantz
Center for Advanced Computing
Cornell University

Based on materials developed by CAC and TACC
RAM Arrangement on Stampede

- **Many nodes → distributed memory**
  - each node has its own local memory
  - not directly addressable from other nodes
- **Multiple sockets per node**
  - each node has 2 sockets (chips)
- **Multiple cores per socket**
  - each socket (chip) has 8 cores
- **Memory spans all 16 cores → shared memory**
  - node’s full local memory is addressable from any core in any socket
- **Memory is attached to sockets**
  - 8 cores sharing the socket have fastest access to attached memory
  - we are ignoring any attached MIC coprocessors for the moment…
Dealing with NUMA

How do we deal with NUMA (Non-Uniform Memory Access)?

Standard models for parallel programs assume a uniform architecture –

• Threads for shared memory
  – parent process uses pthreads or OpenMP to fork multiple threads
  – threads share the same virtual address space
  – also known as SMP = Symmetric MultiProcessing

• Message passing for distributed memory
  – processes use MPI to pass messages (data) between each other
  – each process has its own virtual address space

If we attempt to combine both types of models –

• Hybrid programming
  – try to exploit the whole shared/distributed memory hierarchy
Why Hybrid? Or Why Not?

Why hybrid?
• Eliminates domain decomposition at node level
• Automatic memory coherency at node level
• Lower (memory) latency and data movement within node
• Can synchronize on memory instead of barrier
• Can be easier to incorporate coprocessors

Why not hybrid?
• An SMP algorithm created by aggregating MPI parallel components on a node (or on a socket) may actually run slower
• Possible waste of effort
Motivation for Hybrid

- Balance the computational load
- Reduce memory traffic, especially for memory-bound applications
Two Views of a Node

OpenMP

MPI
Two Views = Two Ways to Write Parallel Programs

• OpenMP (or pthreads) only
  – launch one process per node
  – have each process fork one thread (or maybe more) per core
  – share data using shared memory
  – can’t share data with a different process (except maybe via file I/O)

• MPI only
  – launch one process per core, on one node or on many
  – pass messages among processes without concern for location
  – (maybe create different communicators intra-node vs. inter-node)
  – ignore the potential for any memory to be shared

• With hybrid OpenMP/MPI programming, we want each MPI process to launch multiple OpenMP threads that can share local memory
Some Possible MPI + Thread Configurations

• Treat each *node* as an SMP
  – launch a single MPI process per node
  – create parallel threads sharing full-node memory
  – typically want 16 threads/node on Stampede, e.g.

• Treat each *socket* as an SMP
  – launch one MPI process on each socket
  – create parallel threads sharing same-socket memory
  – typically want 8 threads/socket on Stampede, e.g.

• No SMP, ignore shared memory (all MPI)
  – assign an MPI process to each core
  – in a master/worker paradigm, one process per node may be master
  – not really hybrid, may at least make a distinction between nodes
Creating Hybrid Configurations

To achieve configurations like these, we must be able to:

- Assign to each process/thread an *affinity* for some set of cores
- Make sure the *allocation* of memory is appropriately matched
NUMA Operations

Where do processes, threads, and memory allocations get assigned?

- If memory were completely uniform, there would be no need to worry about questions like, “where do processes go?”
- Only for NUMA is the placement of processes/threads and allocated memory (NUMA control) of any importance

The default NUMA control is set through policy

- The policy is applied whenever a process is executed, or a thread is forked, or memory is allocated
- These are all events that are directed from within the kernel

NUMA control is managed by the kernel.
NUMA control can be changed with numactl.
Process Affinity and Memory Policy

• One would like to set the *affinity* of a process for a certain socket or core, and the *allocation* of data in memory relative to a socket or core.

• Individual users can alter kernel policies (setting Process Affinity and Memory Policy == PAMPPer):
  – users can PAMPPer their own processes
  – root can PAMPPer any process
  – careful, libraries may PAMPPer, too!

• Means by which Process Affinity and Memory Policy can be changed:
  1. dynamically on a running process (knowing process id)
  2. at start of process execution (with wrapper command)
  3. within program through F90/C API
Using numactl, at the Process Level

```
numactl <option socket(s)/core(s)> ./a.out
```

<table>
<thead>
<tr>
<th>For a Process: <strong>Socket</strong> Control</th>
<th>For a Process’s Memory: <strong>Socket</strong> Control</th>
<th>For a Process: <strong>Core</strong> Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket assignment -N</td>
<td>memory allocation -l -i --preferred -m</td>
<td>core assignment -C</td>
</tr>
<tr>
<td>(local, interleaved, preferred, mandatory)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Quick Guide to numactl

<table>
<thead>
<tr>
<th>Socket Affinity</th>
<th>Affinity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-N</td>
<td>{0,1}</td>
<td>Execute process on cores of this (these) socket(s) only.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Policy</th>
<th>Affinity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-l</td>
<td>no argument</td>
<td>Allocate on current socket; fallback to any other if full.</td>
</tr>
<tr>
<td>-i</td>
<td>{0,1}</td>
<td>Allocate round robin (interleave) on these sockets. No fallback.</td>
</tr>
<tr>
<td>--preferred=</td>
<td>{0,1}</td>
<td>Allocate on this socket; fallback to any other if full.</td>
</tr>
<tr>
<td>-m</td>
<td>{0,1}</td>
<td>Allocate only on this (these) socket(s). No fallback.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core Affinity</th>
<th>Affinity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-c</td>
<td>{0,1,2,3,4,5,6, 7,8,9,10,11,12, 13,14,15}</td>
<td>Execute process on this (these) core(s) only.</td>
</tr>
</tbody>
</table>
SMP Nodes

Hybrid batch script for 16 threads/node

- Specify **total MPI tasks** to be started by batch
- Specify **total nodes equal to tasks**
- Set number of **threads for each process**
- PAMPPering at **job level**
  - controls behavior (e.g., process-core affinity) for ALL processes
  - no simple/standard way to control *thread*-core affinity with `numactl`

<table>
<thead>
<tr>
<th>job script (Bourne shell)</th>
<th>job script (C shell)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#SBATCH -n 2 -N 2</code></td>
<td><code>#SBATCH -n 2 -N 2</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>export OMP_NUM_THREADS=16</code></td>
<td><code>setenv OMP_NUM_THREADS 16</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>ibrun numactl -i all ./a.out</code></td>
<td><code>ibrun numactl -i all ./a.out</code></td>
</tr>
</tbody>
</table>
SMP Sockets

Hybrid batch script for 2 tasks/node, 8 threads/task

- Specify **total MPI tasks** to be started by batch
- Specify **total nodes equal to tasks/2** (so 2 tasks/node)
- Set number of **threads for each process**
- PAMPPering at **process level**, must create script to manage affinity
  - `tacc_affinity` script pins tasks to sockets, ensures local memory allocation
  - use it as a `numactl` starting point if it’s not quite right for your application

<table>
<thead>
<tr>
<th>job script (Bourne shell)</th>
<th>job script (C shell)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#SBATCH -n 4 -N 2</code></td>
<td><code>#SBATCH -n 4 -N 2</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>export OMP_NUM_THREADS=8</code></td>
<td><code>setenv OMP_NUM_THREADS 8</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td><code>ibrun tacc_affinity ./a.out</code></td>
<td><code>ibrun tacc_affinity ./a.out</code></td>
</tr>
</tbody>
</table>
Basic Hybrid Program Template

Start with MPI initialization

(Serial regions are executed by the master thread of the MPI process)

Create OMP parallel regions within each MPI process

– MPI calls may be allowed here too
– MPI rank is same for all threads

Call MPI in single-threaded regions

Finalize MPI
Types of MPI Calls Among Threads

**Single-threaded messaging**
- Call MPI from a serial region
- Call MPI from a single thread within a parallel region

**Multi-threaded messaging**
- Call MPI from multiple threads within a parallel region
- Requires an implementation of MPI that is thread-safe
Multiple Threads Calling MPI

- Thread ID as well as rank can be used in communication
- Technique is illustrated in multi-thread “ping” (send/receive) example
Example: Multiple Threads Calling MPI

call mpi_init_thread( MPI_THREAD_MULTIPLE, iprovided, ierr)
call mpi_comm_rank(MPI_COMM_WORLD, irank, ierr)
call mpi_comm_size( MPI_COMM_WORLD, nranks, ierr)
if (iprovided >= MPI_THREAD_MULTIPLE) then  ! All threads can call MPI
!$OMP parallel private(j, ithread, nthreads)
  nthreads=OMP_GET_NUM_THREADS()
  ithread =OMP_GET_THREAD_NUM()
call pwork(ithread, irank, nthreads, nranks...)
if( irank == 0) then
  call mpi_send(ithread,1,MPI_INTEGER, 1, ithread, MPI_COMM_WORLD, ierr)
else
  call mpi_recv( j,1,MPI_INTEGER, 0, ithread, MPI_COMM_WORLD, istat, ierr)
  print*, "Yep, this is ", irank, " thread ", ithread," I received from ", j
endif
!$OMP end parallel
endif

Communicate between ranks.

Threads use tags to differentiate.
NUMA Control in Code, at the Thread Level

• Within a code, **Scheduling Affinity** and **Memory Policy** (SCAMPer?) can be examined and changed using libnuma routines:
  - sched_getaffinity, sched_setaffinity
  - get_mempolicy, set_mempolicy

  } see man pages

• This is the *only* way to set affinities and policies that differ per **thread**

• To make scheduling assignments, set bits in a mask:

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1  
Assignment to Core 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
Assignment to Core 15
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1  
Assignment to Core 0 or 15
```
Code Example for Scheduling Affinity

```c
#include <spawn.h> // C API parameters and prototypes

int icore=3; // Set core number
cpu_set_t cpu_mask; // Allocate mask

CPU_ZERO(&cpu_mask); // Set mask to zero
CPU_SET(icore,&cpu_mask); // Set mask with core #

err = sched_setaffinity( (pid_t)0 , // Set the affinity
    sizeof(cpu_mask),
    &cpu_mask);
```
Programming for MIC: Hybrid and Heterogeneous

• Each Stampede node currently has 2 processors + 1 MIC card
• MIC = Many Integrated Cores = a “coprocessor” on a PCIe card that features >60 cores; released as Xeon Phi™
  – Represents Intel’s response to GPGPU, especially NVIDIA’s CUDA
  – Answers the question: if 8 modern Xeon cores fit on a die, how many early Pentiums would fit?
• MIC answers CUDA’s API problem: just compile like any normal code
  – Instruction set is x86 with support for 64-bit addressing
  – Recent x86 extensions may not be available
  – Developers use familiar Intel compilers, libraries, and tools
• However, MIC adds yet another level of programming complexity
  – Stampede is a multi-core machine where not all the cores are the same
MIC Strategies for HPC Codes

Initial MPI code, could be hybrid with OpenMP

No change – run on CPUs, MICs, or both

Expand existing hybrids; or, add OpenMP offload

Build on libraries like Intel MKL, PETSc, etc.
OpenMP Offload Constructs: Base Program

```c
#include <omp.h>
#define N 10000

void foo(double *, double *, double *, int );
int main(){
    int i; double a[N], b[N], c[N];
    for(i=0;i<N;i++){ a[i]=i; b[i]=N-1-i;}

    ...

    foo(a,b,c,N);
}

void foo(double *a, double *b, double *c, int n){
    int i;

    for(i=0;i<n;i++) { c[i]=a[i]*2.0e0 + b[i]; } }
```

- Objective: offload foo to a device
- Use OpenMP to do the offload
OpenMP Offload Constructs: Requirements

• Direct (Intel) compiler to offload function or block
• “Decorate” function and prototype
• Ideally, familiar-looking OpenMP directives work on device

```c
#include <omp.h>
define N 10000
#pragma offload_function_spec
void foo(double *, double *, double *, int );
int main(){
    int i; double a[N], b[N], c[N];
    for(i=0;i<N;i++) { a[i]=i; b[i]=N-1-i;}

    ...
    #pragma offload_this
    foo(a,b,c,N);
}
#pragma offload_function_spec
void foo(double *a, double *b, double *c, int n){
    int i;
    #pragma omp parallel for
    for(i=0;i<n;i++) { c[i]=a[i]*2.0e0 + b[i]; } }
```
Pros and Cons of MIC Programming Models

• Offload engine: MIC serves as coprocessor for the host
  – Pros: distinct hardware gets distinct role; programmable via simple calls to a library such as MKL, or via directives (we’ll go into depth on this)
  – Cons: PCIe is the only path for most work; must try to retain data on card

• “Symmetric” #1: Everything is just an MPI core
  – Pros: MPI works for all cores (though 1 MIC core < 1 server core)
  – Cons: memory may be insufficient to support a μOS plus lots of data; fails to take good advantage of shared memory; PCIe is a bottleneck

• “Symmetric” #2: MIC and host are just different SMPs
  – Pros: MPI/OpenMP works for both host and MIC; more efficient use of limited PCIe bandwidth and limited MIC memory
  – Cons: hybrid programming is already tough on homogeneous SMPs, but OpenMP-based hybrids may be the best path for scaling to 60 cores/MIC
Quick Guide to KMP_AFFINITY

- Set this environment variable to influence thread affinity generally
- Useful for CPU and/or MIC models based on OpenMP (SMP, offload)

```
export KMP_AFFINITY=<type>  (for SMP)
export MIC_KMP_AFFINITY=<type>  (for offload)
```

<table>
<thead>
<tr>
<th>Type</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>compact</td>
<td>Pack threads close to each other.</td>
</tr>
<tr>
<td>explicit</td>
<td>Use the proclist modifier to pin threads.</td>
</tr>
<tr>
<td>none</td>
<td>Does not pin threads.</td>
</tr>
<tr>
<td>scatter</td>
<td>Round-robin threads to cores.</td>
</tr>
<tr>
<td>balanced (Phi only)</td>
<td>Use scatter, but keep OMP thread ids consecutive.</td>
</tr>
</tbody>
</table>
KMP_AFFINITY Types and Thread Placement

- Imagine a system with 4 cores and 4 hardware threads/core
- Placement of 8 threads is illustrated for the 3 types
- Compact type does not fully utilize all cores; not recommended
Roadmap: What Comes Next?

- Expect many of the upcoming large systems to be accelerated
- MPI + OpenMP will be the main HPC programming model
  - If you are not using Intel TBBs or Cilk
  - If you are not spending all your time in libraries (MKL, etc.)
- Many HPC applications are pure-MPI codes
  - Start thinking about upgrading to a hybrid scheme
  - Adding OpenMP is a larger effort than adding the extra MIC directives
- Special MIC/OpenMP considerations
  - Many more threads will be needed:
    - 60+ cores on production Xeon Phi™ → 60+/120+/240+ threads
  - Good OpenMP scaling (and vectorization) are much more important
Conclusions

• On heterogeneous NUMA systems like Stampede, placement and binding of processes and their associated memory are important performance considerations.
• Process Affinity and Memory Policy have a significant effect on pure MPI, pure OpenMP, and Hybrid codes—both on host and on MIC.
• Simple numactl commands and APIs allow users to control affinity of processes and threads and memory assignments.
• Future prospects for hybrid programming:
  – Core counts will increase on both processors and coprocessors.
  – Even more effort will be focused on process scheduling and data locality.
  – Expect to see more multi-threaded libraries; be alert for their potential interaction with your own multithreading strategy.
References

• Yun (Helen) He and Chris Ding, Lawrence Berkeley National Laboratory, June 24, 2004: Hybrid OpenMP and MPI Programming and Tuning (NUG2004).
  www.nersc.gov/nusers/services/training/classes/NUG/Jun04/NUG2004_yhe_hybrid.ppt

• Texas Advanced Computing Center: Stampede User Guide, see numa section.  www.tacc.utexas.edu/services/userguides/stampede

• Message Passing Interface Forum: MPI-2: MPI and Threads (specific section of the MPI-2 report).

• Intel Corp.: Thread Affinity Interface (Linux and Windows), from the Intel Fortran Compiler User and Reference Guides.
Extra Slides: MPI-2 and Multithreading
MPI-2 and Thread Safety

• **Consider thread safety when calling MPI from threads**
  • Use MPI_Init_thread to select/determine the level of thread support
    – Supported in MPI-2, substitute for the usual MPI_Init
  • Thread safety is identified/controlled by MPI’s provided types
    – *Single* means no multi-threading
    – *Funneled* means only the master thread can call MPI
    – *Serialized* means multiple threads can call MPI, but only 1 call can be in progress at a time
    – *Multiple* means MPI is thread safe
  • Monotonic values are assigned to parameters
    
    ```
    MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED
    < MPI_THREAD_SERIALIZE < MPI_THREAD_MULTIPLE
    ```
MPI-2’s MPI_Init_thread

Syntax:

```c
int MPI_Init_thread(int *argc, char ***argv, int rqd, int *pvd)
int MPI::Init_thread(int& argc, char**& argv, int rqd)
```

- **Input:** \texttt{rqd}, or “required” (integer)
  - Indicates the desired level of thread support
- **Output:** \texttt{pvd}, or “provided” (integer)
  - Indicates the available level of thread support
- If thread level \texttt{rqd} is supported, the call returns \texttt{pvd = rqd}
- Otherwise, \texttt{pvd} returns the highest provided level of support
## MPI-2 Thread Support Levels

<table>
<thead>
<tr>
<th>Support Levels</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_THREAD_SINGLE</td>
<td>Only one thread will execute.</td>
</tr>
<tr>
<td>MPI_THREAD_FUNNELED</td>
<td>Process may be multi-threaded, but only the main thread will make MPI calls (calls are “funneled” to main thread). <em>Default</em></td>
</tr>
<tr>
<td>MPI_THREAD_SERIALIZE</td>
<td>Process may be multi-threaded, and any thread can make MPI calls, but threads cannot execute MPI calls concurrently; they must take turns (calls are “serialized”).</td>
</tr>
<tr>
<td>MPI_THREAD_MULTIPLE</td>
<td>Multiple threads may call MPI, with no restriction.</td>
</tr>
</tbody>
</table>
Example: Single-Threaded MPI Calls

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>include 'mpif.h'</td>
<td>#include &lt;mpi.h&gt;</td>
</tr>
<tr>
<td>program hybsimp</td>
<td>int main(int argc,</td>
</tr>
<tr>
<td>call MPI_Init(ie)</td>
<td>char **argv) {</td>
</tr>
<tr>
<td>call MPI_Comm_rank(...irk,ie)</td>
<td>int rank, size, ie, i;</td>
</tr>
<tr>
<td>call MPI_Comm_size(...isz,ie)</td>
<td>ie= MPI_Init(&amp;argc,&amp;argv[]);</td>
</tr>
<tr>
<td>!Setup shared mem, comp/comm</td>
<td>ie= MPI_Comm_rank(...&amp;rank);</td>
</tr>
<tr>
<td>!$OMP parallel do</td>
<td>ie= MPI_Comm_size(...&amp;size);</td>
</tr>
<tr>
<td>do i=1,n</td>
<td>//Setup shared mem, comp/comm</td>
</tr>
<tr>
<td>&lt;work&gt;</td>
<td>#pragma omp parallel for</td>
</tr>
<tr>
<td>enddo</td>
<td>for(i=0; i&lt;n; i++){</td>
</tr>
<tr>
<td>!Compute &amp; communicate</td>
<td>&lt;work&gt;</td>
</tr>
<tr>
<td>call MPI_Finalize(ierr)</td>
<td>}</td>
</tr>
<tr>
<td>end</td>
<td>// compute &amp; communicate</td>
</tr>
<tr>
<td></td>
<td>ie= MPI_Finalize();</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
</tbody>
</table>
Funneled MPI Calls via Master

• Must have support for **MPI_THREAD_FUNNELEDED** or higher
• Best to **use OMP_BARRIER**
  – there is no implicit barrier in the master workshare construct, **OMP_MASTER**
  – in the example, the master thread will execute a single MPI call within the **OMP_MASTER** construct
  – all other threads will be sleeping
## Example: Funneled MPI Calls via Master

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>include 'mpif.h'</code></td>
<td><code>#include &lt;mpi.h&gt;</code></td>
</tr>
<tr>
<td><code>program hybmas</code></td>
<td><code>int main(int argc,</code></td>
</tr>
<tr>
<td></td>
<td><code>    char **argv) {</code></td>
</tr>
<tr>
<td></td>
<td><code>    int rank, size, ie, i;</code></td>
</tr>
<tr>
<td>!$OMP parallel</td>
<td><code>#pragma omp parallel</code></td>
</tr>
<tr>
<td></td>
<td><code>{</code></td>
</tr>
<tr>
<td>!$OMP barrier</td>
<td><code>    #pragma omp barrier</code></td>
</tr>
<tr>
<td>!$OMP master</td>
<td><code>    #pragma omp master</code></td>
</tr>
<tr>
<td></td>
<td><code>    {</code></td>
</tr>
<tr>
<td>call MPI_&lt;Whatever&gt;(...,ie)</td>
<td><code>        ie= MPI_&lt;Whatever&gt;(...);</code></td>
</tr>
<tr>
<td>!$OMP end master</td>
<td><code>    }</code></td>
</tr>
<tr>
<td>!$OMP barrier</td>
<td><code>    #pragma omp barrier</code></td>
</tr>
<tr>
<td></td>
<td><code>}</code></td>
</tr>
<tr>
<td>!$OMP end parallel</td>
<td><code>}</code></td>
</tr>
<tr>
<td>end</td>
<td><code>}</code></td>
</tr>
</tbody>
</table>
Serialized MPI Calls and OpenMP

- Must have support for **MPI_THREAD_SERIALIZED** or higher
- Best to use **OMP_BARRIER** only at beginning, since there is an implicit barrier in the SINGLE workshare construct, **OMP_SINGLE**
  - Example is the simplest one: any thread (not necessarily master) will execute a single MPI call within the OMP_SINGLE construct
  - All other threads will be sleeping
<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>include 'mpif.h'</td>
<td>#include &lt;mpi.h&gt;</td>
</tr>
<tr>
<td>program hybsing</td>
<td>int main(int argc,</td>
</tr>
<tr>
<td></td>
<td>char **argv) {</td>
</tr>
<tr>
<td>call MPI_Init_thread( &amp;</td>
<td>int rank, size, ie, i;</td>
</tr>
<tr>
<td>MPI_THREAD_SERIALIZED,ipvd,ie)</td>
<td>ie= MPI_Init_thread(</td>
</tr>
<tr>
<td>!$OMP parallel</td>
<td>MPI_THREAD_SERIALIZED,ipvd);</td>
</tr>
<tr>
<td></td>
<td>#pragma omp parallel</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>#pragma omp barrier</td>
</tr>
<tr>
<td></td>
<td>#pragma omp single</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>ie= MPI_&lt;Whatever&gt;(...);</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>//Don't need omp barrier</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td>call MPI_&lt;Whatever&gt;(...,ie)</td>
<td></td>
</tr>
<tr>
<td>!$OMP end single</td>
<td></td>
</tr>
<tr>
<td>!Don't need OMP barrier</td>
<td></td>
</tr>
<tr>
<td>!$OMP end parallel</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td>}</td>
</tr>
</tbody>
</table>
Overlapping Work & MPI Calls

• One core is capable of saturating the lanes of the PCIe network link...
  – Why use all cores to communicate?
  – Instead, communicate using just one or several cores
  – Can do work with the rest during communication
• Must have support for MPI_THREAD_FUNNELED or higher to do this
• Can be difficult to manage and load-balance!
Example: Overlapping Work & MPI Calls

<table>
<thead>
<tr>
<th>Fortran</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>include 'mpif.h'</td>
<td>#include &lt;mpi.h&gt;</td>
</tr>
<tr>
<td>program hybsing</td>
<td>int main(int argc,</td>
</tr>
<tr>
<td>!$OMP parallel</td>
<td>char **argv) {</td>
</tr>
<tr>
<td>if (ithread .eq. 0) then</td>
<td>int rank, size, ie, i;</td>
</tr>
<tr>
<td>call MPI_&lt;Whatever&gt;(...,ie)</td>
<td>#pragma omp parallel</td>
</tr>
<tr>
<td>else</td>
<td>{</td>
</tr>
<tr>
<td>&lt;work&gt;</td>
<td>if (thread == 0){</td>
</tr>
<tr>
<td>endif</td>
<td>ie= MPI_&lt;Whatever&gt;(...);</td>
</tr>
<tr>
<td>!$OMP end parallel</td>
<td>}</td>
</tr>
<tr>
<td>end</td>
<td>}</td>
</tr>
</tbody>
</table>

C:
```c
#include <mpi.h>
int main(int argc,
   char **argv) {
   int rank, size, ie, i;
   #pragma omp parallel
   {
      if (thread == 0){
         ie= MPI_<Whatever>(...);
      }
      if(thread != 0){
         <work>
      }
   }
}
```