Cornell University
Center for Advanced Computing

Hybrid Parallel Overview

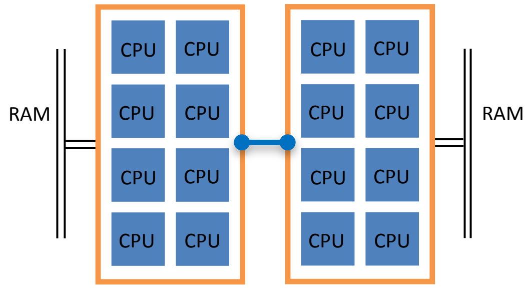
Steve Lantz
Center for Advanced Computing
Cornell University

Workshop: Parallel Computing on Stampede, June 11, 2013

Based on materials developed by CAC and TACC

RAM Arrangement on Stampede

- *Many nodes → <u>distributed memory</u>*
  - 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 → <u>shared memory</u>*
  - 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 the 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?

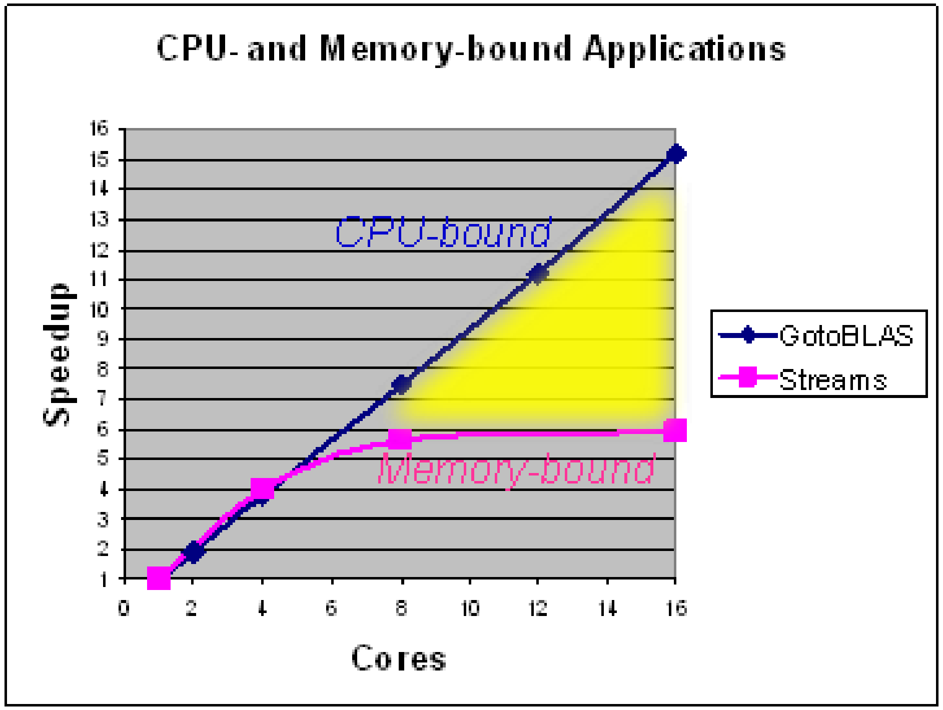
- Eliminates domain decomposition <u>at node level</u>
- Automatic memory coherency <u>at node level</u>
- Lower (memory) latency and data movement <u>within node</u>
- Can synchronize on memory instead of barrier

Why not hybrid?

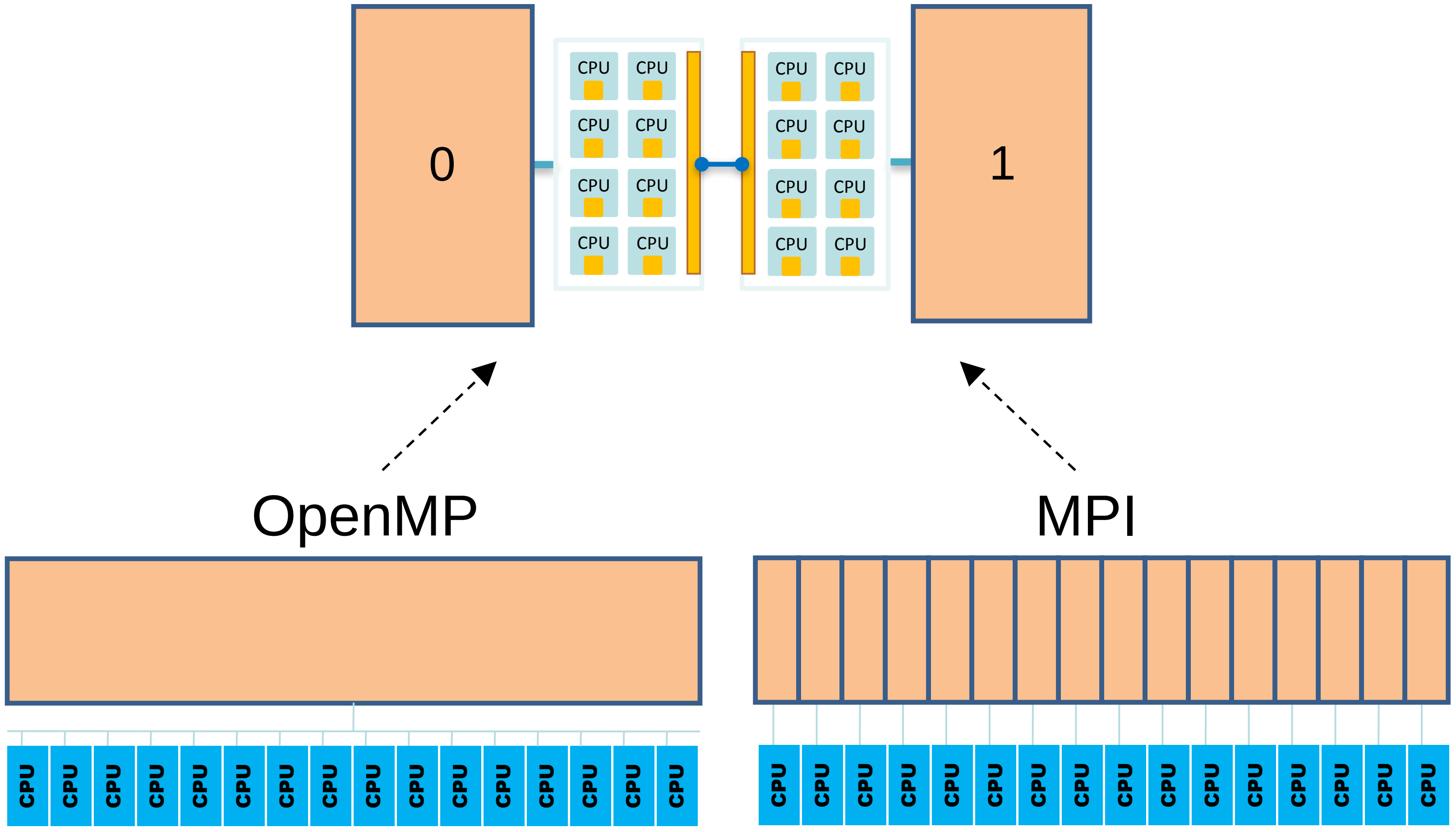
- An SMP algorithm created by aggregating MPI parallel components on a node (or on a socket) may actually run slower
- Possible waste of effort

Motivation for Hybrid

- Balance the computational load
- Reduce memory traffic, especially for memory-bound applications



Two Views of a Node



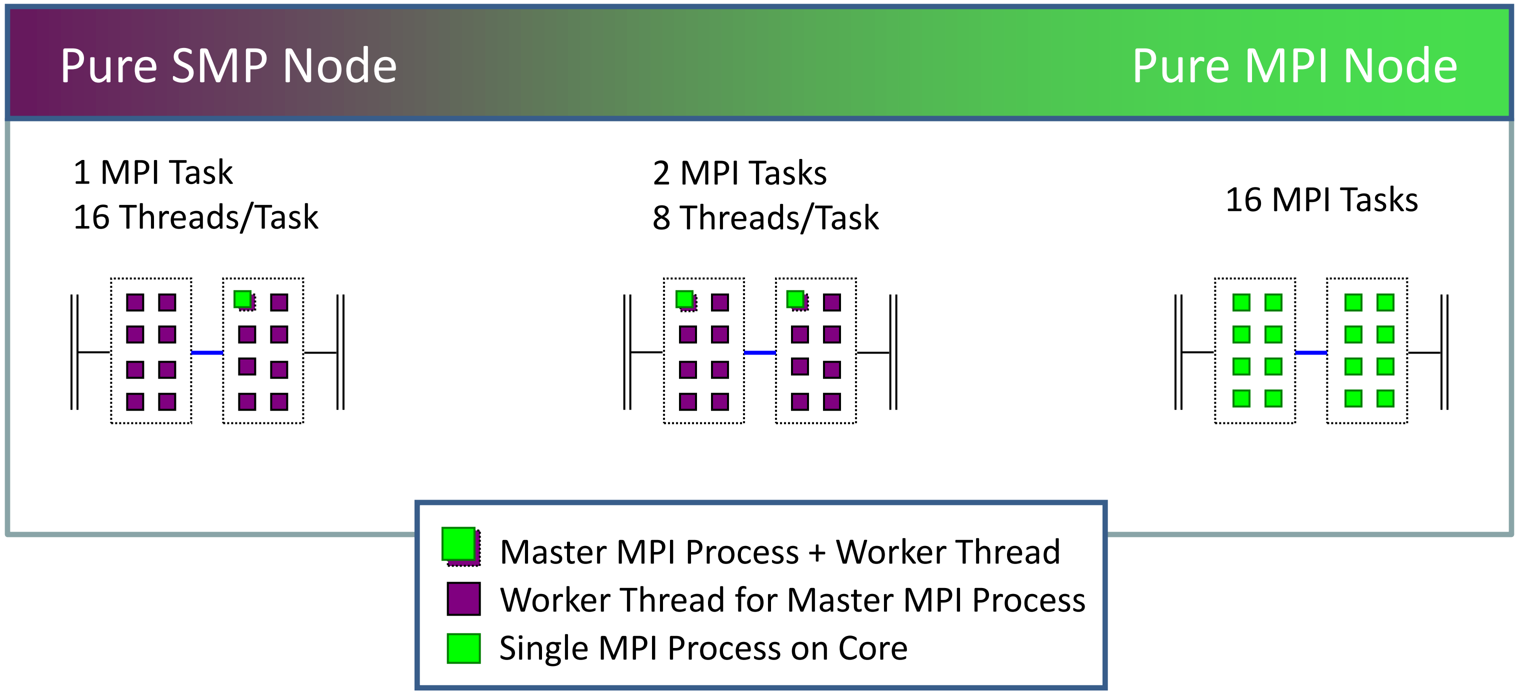
Two Views = Two Ways to Write Parallel Programs

- 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



🟩 Master MPI Process + Worker Thread
🟪 Worker Thread for Master MPI Process
🟩 Single MPI Process on Core

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.

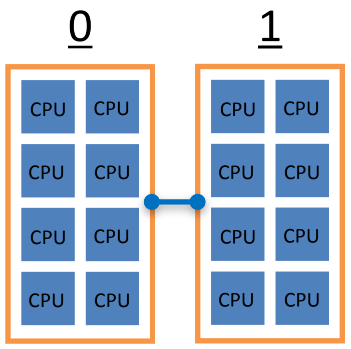
NUMA control can be changed with numactl.

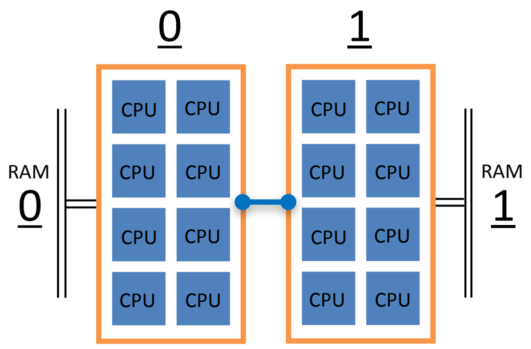
Process Affinity and Memory Policy

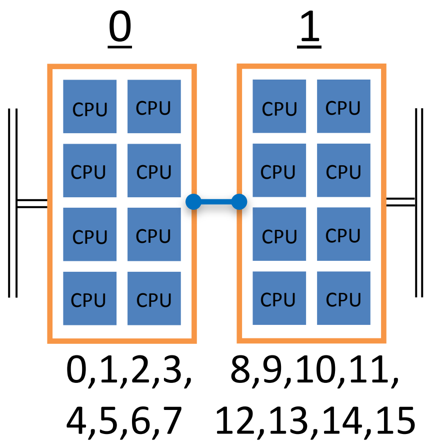
- One would like to set the *affinity* of a process for a certain socket or core, and the *allocation* of data in memory relative to a socket or core
- Individual users can alter kernel policies
  (setting Process Affinity and Memory Policy == PAMPer)
  - users can PAMPer their own processes
  - root can PAMPer any process
  - careful, libraries may PAMPer, 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
```







For a Process: Socket Control	For a Process's Memory: Socket Control	For a Process: Core Control
socket assignment -N	memory allocation -l -i --preferred -m (local, interleaved, preferred, mandatory)	core assignment -C

Quick Guide to numactl

Socket Affinity	-N	{0,1}	Execute process on cores of this (these) socket(s) only.
Memory Policy	-l	no argument	Allocate on current socket; fallback to any other if full.
Memory Policy	-i	{0,1}	Allocate round robin (interleave) on these sockets. No fallback.
Memory Policy	--preferred=	{0,1} select one	Allocate on this socket; fallback to any other if full.
Memory Policy	-m	{0,1}	Allocate only on this (these) socket(s). No fallback.
Core Affinity	-C	{0,1,2,3,4,5,6,7, 8,9,10,11,12,13, 14,15}	Execute process on this (these) core(s) only.

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**
- PAMPering at **job level**
  - controls behavior (e.g., process-core affinity) for ALL processes
  - no simple/standard way to control *thread*-core affinity with numactl

job script (Bourne shell)	job script (C shell)
`#SBATCH -n 2 -N 2`	`#SBATCH -n 2 -N 2`
`...`	`...`
`export OMP_NUM_THREADS=16`	`setenv OMP_NUM_THREADS 16`
`...`	`...`
`ibrun numactl -i all ./a.out`	`ibrun numactl -i all ./a.out`

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**
- PAMPering 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

job script (Bourne shell)	job script (C shell)
`#SBATCH -n 4 -N 2`	`#SBATCH -n 4 -N 2`
`...`	`...`
`export OMP_NUM_THREADS=8`	`setenv OMP_NUM_THREADS 8`
`...`	`...`
`ibrun tacc_affinity ./a.out`	`ibrun tacc_affinity ./a.out`

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 known to all threads

Call MPI in single-threaded regions

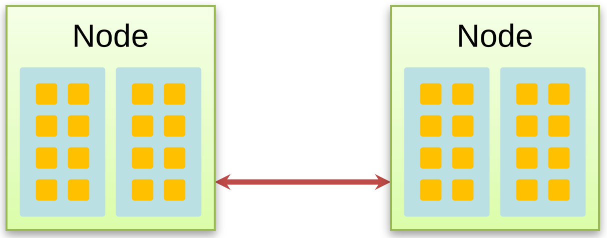
Finalize MPI

```
MPI_Init
...
MPI_Call
...
    OMP parallel
    ...
    MPI_Call
    ...
    end parallel
...
MPI_Call
...
MPI_Finalize
```

Types of MPI Calls Among Threads

Single-threaded messaging

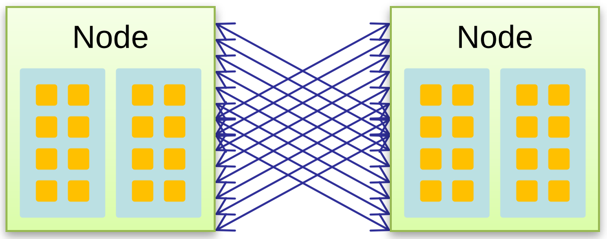
- Call MPI from a serial region
- Call MPI from a single thread within a parallel region



Rank to rank

Multi-threaded messaging

- Call MPI from multiple threads within a parallel region
- Requires an implementation of MPI that is thread-safe



rank-thread ID to rank-thread ID

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

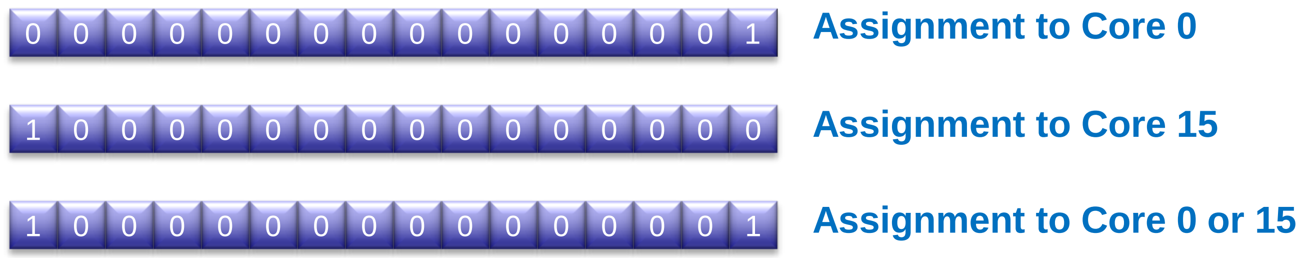
```
call mpi_init_thread( MPI_THREAD_MULTIPLE, iprovided, ierr)
call mpi_comm_rank(MPI_COMM_WORLD, irank, ierr)
call mpi_comm_size( MPI_COMM_WORLD, nranks, ierr)
if (iprovided >= MPI_THREAD_MULTIPLE) then    ! All threads can call MPI
!$OMP parallel private(j, ithread, nthreads)
   nthreads=OMP_GET_NUM_THREADS()
   ithread   =OMP_GET_THREAD_NUM()
   call pwork(ithread, irank, nthreads, nranks…)
   if(irank == 0) then
      call mpi_send(ithread,1,MPI_INTEGER, 1,  ithread, MPI_COMM_WORLD, ierr)
   else
      call mpi_recv(          j,1,MPI_INTEGER, 0,  ithread, MPI_COMM_WORLD, istat, ierr)
      print*, "Yep, this is ",irank," thread ", ithread," I received from ", j
   endif
!$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
- This is the *only* way to set affinities and policies that differ per *thread*
- To make scheduling assignments, set bits in a mask:



Code Example for Scheduling Affinity

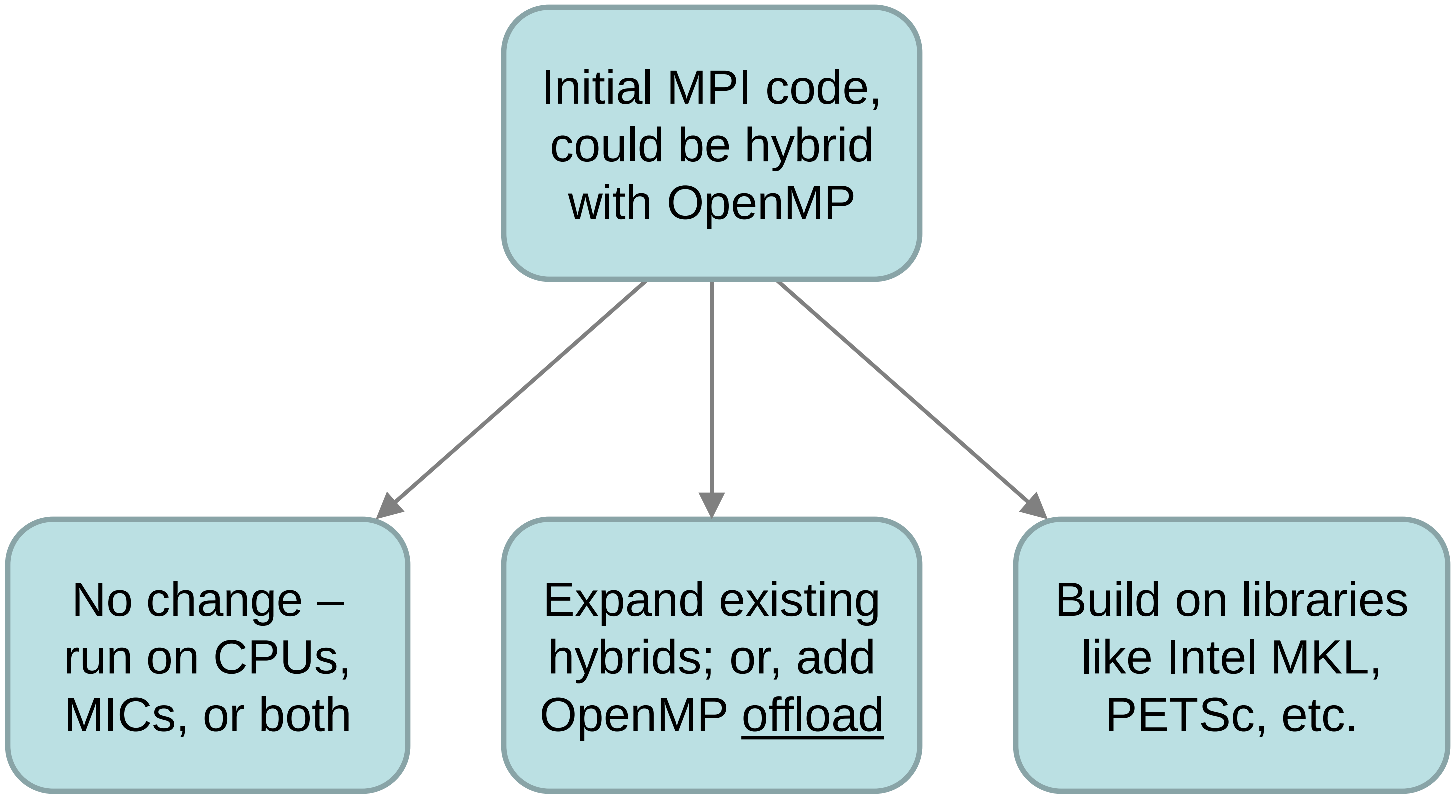
```
...
#include <spawn.h>                //C API parameters and prototypes
...
int icore=3;                      //Set core number
cpu_set_t cpu_mask;               //Allocate mask
...
CPU_ZERO(     &cpu_mask);         //Set mask to zero
CPU_SET(icore,&cpu_mask);         //Set mask with core #

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

```
#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

```
#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]; } }
```

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

Pros and Cons of MIC Programming Models

- Offload engine: <u>MIC serves as coprocessor for the host</u>
  - *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; difficult to retain data on card
- "Symmetric" #1: <u>Everything is just an MPI core</u>
  - *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: <u>MIC and host are just different SMPs</u>
  - *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; not much experience with OpenMP-based hybrids scaling to 60+ cores

Quick Guide to KMP_AFFINITY

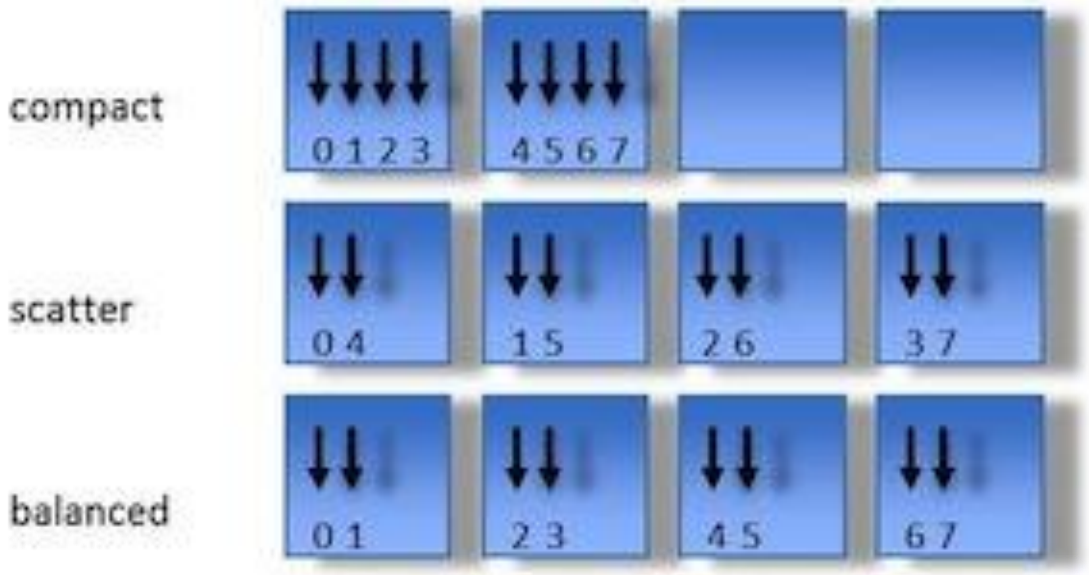
- Set this environment variable to influence thread affinity generally
- Useful for CPU and/or MIC models based on OpenMP (SMP, offload)

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

Type	Effect
compact	Pack threads close to each other.
explicit	Use the proclist modifier to pin threads.
none	Does not pin threads.
scatter	Round-robin threads to cores.
balanced (Phi only)	Use scatter, but keep OMP thread ids consecutive.

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 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.
- 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).

  http://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/node162.htm
- Intel Corp.: Thread Affinity Interface (Linux and Windows), from the Intel Fortran Compiler User and Reference Guides.

  http://www.intel.com/software/products/compilers/docs/fmac/doc_files/source/extfile/optaps_for/common/optaps_openmp_thread_affinity.htm

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_SERIALIZED < MPI_THREAD_MULTIPLE
```

MPI-2’s MPI_Init_thread

Syntax:

```
call MPI_Init_thread(                                   irqd, ipvd, ierr)
int MPI_Init_thread (int *argc, char ***argv, int rqd, int *pvd)
int MPI::Init_thread(int& argc, char**& argv, int rqd)
```

- Input: **rqd**, or “required” (integer)
  - Indicates the desired level of thread support
- Output: **pvd**, or “provided” (integer)
  - Indicates the available level of thread support
- If thread level **rqd** is supported, the call returns **pvd = rqd**
- Otherwise, **pvd** returns the highest provided level of support

MPI-2 Thread Support Levels

Support Levels	Description
MPI_THREAD_SINGLE	Only one thread will execute.
MPI_THREAD_FUNNELED	Process may be multi-threaded, but only the main thread will make MPI calls (calls are “funneled” to main thread). *Default*
MPI_THREAD_SERIALIZE	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”).
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, with no restriction.

Example: Single-Threaded MPI Calls

Fortran

```fortran
include 'mpif.h'
program hybsimp


call MPI_Init(ie)
call MPI_Comm_rank(...irk,ie)
call MPI_Comm_size(...isz,ie)
!Setup shared mem, comp/comm

!$OMP parallel do
  do i=1,n
    <work>
  enddo

!Compute & communicate
call MPI_Finalize(ierr)
end
```

C

```c
#include <mpi.h>
int main(int argc,
  char **argv) {
int rank, size, ie, i;
ie= MPI_Init(&argc,&argv[]);
ie= MPI_Comm_rank(...&rank);
ie= MPI_Comm_size(...&size);
//Setup shared mem, comp/comm

#pragma omp parallel for
  for(i=0; i<n; i++){
    <work>
  }

// compute & communicate
ie= MPI_Finalize();
}
```

Funneled MPI Calls via Master

- Must have support for **MPI_THREAD_FUNNELED 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

Fortran	C
`include 'mpif.h'`	`#include <mpi.h>`
`program hybmas`	`int main(int argc,`
	`  char **argv) {`
	`int rank, size, ie, i;`
`!$OMP parallel`	`#pragma omp parallel`
	`{`
`  !$OMP barrier`	`  #pragma omp barrier`
`  !$OMP master`	`  #pragma omp master`
	`  {`
`  call MPI_<Whatever>(...,ie)`	`    ie= MPI_<Whatever>(...);`
`  !$OMP end master`	`  }`
`  !$OMP barrier`	`  #pragma omp barrier`
`!$OMP end parallel`	`}`
`end`	`}`

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

Example: Serialized MPI Calls and OpenMP

Fortran	C
`include 'mpif.h'`	`#include <mpi.h>`
`program hybsing`	`int main(int argc,`
	`  char **argv) {`
	`int rank, size, ie, i;`
`call MPI_Init_thread( &`	`ie= MPI_Init_thread(`
`MPI_THREAD_SERIALIZED,ipvd,ie)`	`MPI_THREAD_SERIALIZED,ipvd);`
`!$OMP parallel`	`#pragma omp parallel`
	`{`
`  !$OMP barrier`	`  #pragma omp barrier`
`  !$OMP single`	`  #pragma omp single`
	`  {`
`  call MPI_<Whatever>(...,ie)`	`    ie= MPI_<Whatever>(...);`
`  !$OMP end single`	`  }`
`  !Don't need OMP barrier`	`  //Don't need omp barrier`
`!$OMP end parallel`	`}`
`end`	`}`

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

Fortran

```fortran
include 'mpif.h'
program hybsing


!$OMP parallel

   if (ithread .eq. 0) then
   call MPI_<Whatever>(...,ie)
   else
     <work>
   endif

!$OMP end parallel
end
```

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>
   }
}
}
```