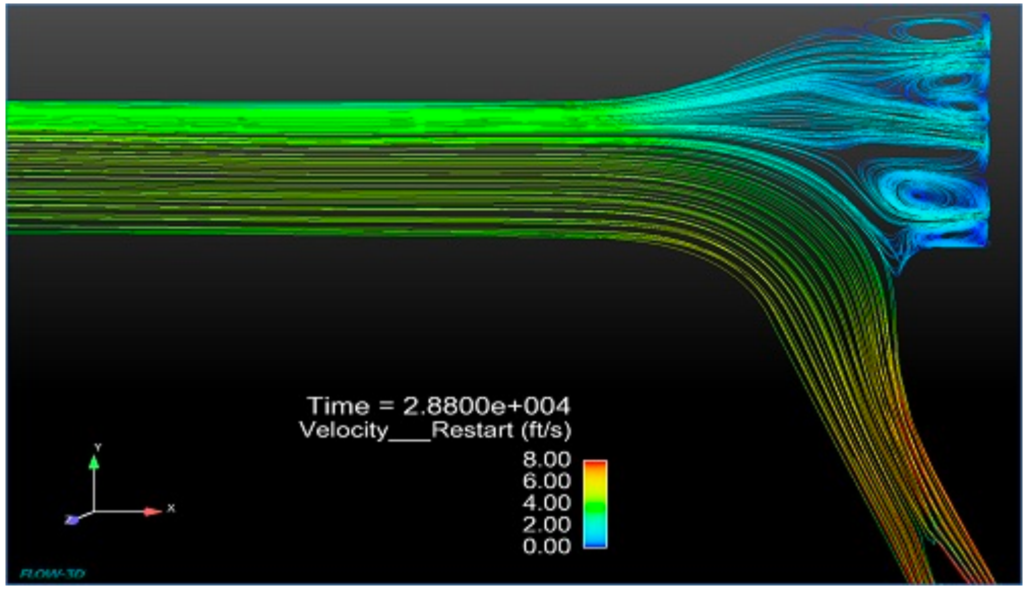
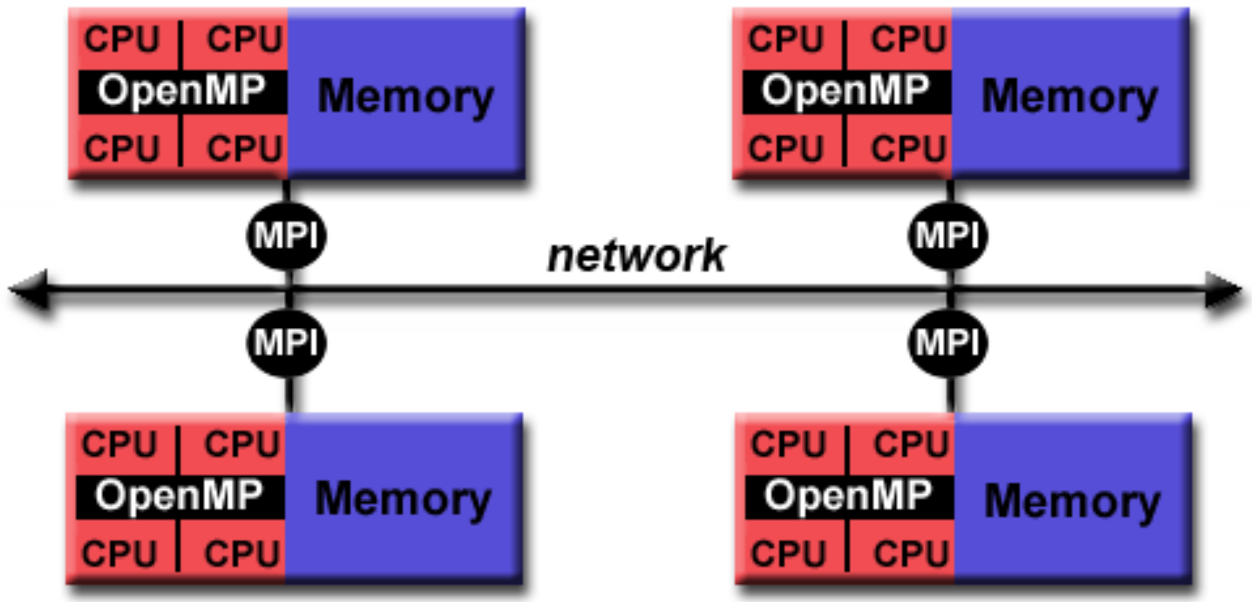
Hybrid MPI/OpenMP Programming, Compiling and Execution – An Introduction

Carlos J. BARRIOS HERNANDEZ, PhD.
@carlosjaimebh

Big Problems : Smart Solutions





Computational Dynamic Fluids Problems

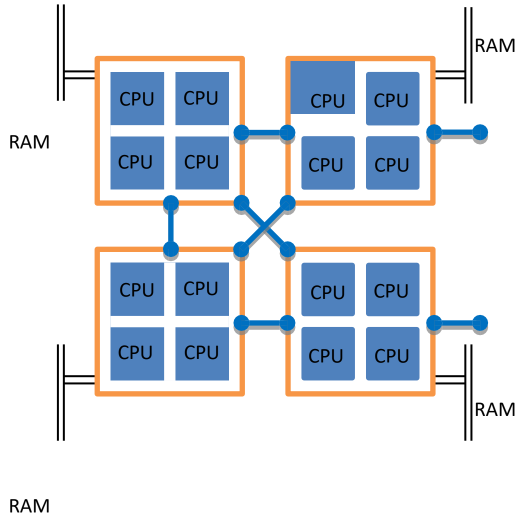
Overview

- Architectural Considerations
- Single and multilevel parallelism.
- Example of MPI-OpenMP buildup.
- Compilation and running.
- Performance suggestions.
- Code examples.

Architectural Considerations

RAM Arrangement on GUANE-1
(and the must part of the clusters)

- *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 4 sockets (chips)
- *Multiple cores per socket*
  - socket (chip) has 4/6 or 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*
  - 4 cores sharing the socket have fastest access to attached memory



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?

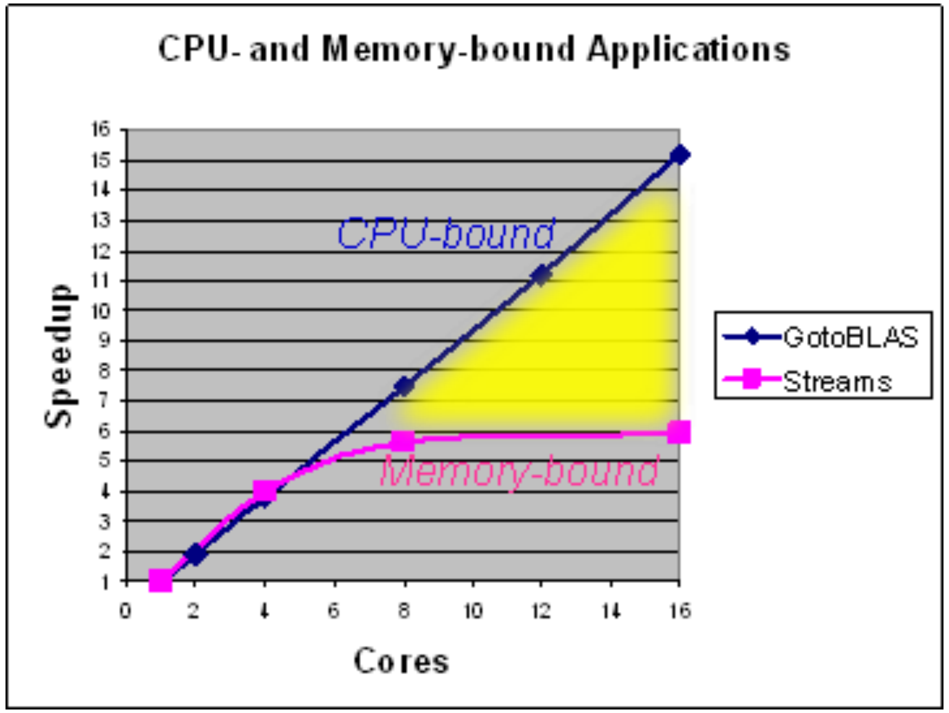
- Eliminates domain decomposition at node level
- Automatic memory coherency at node level
- Lower (memory) latency and data movement within node
- Can synchronize on memory instead of barrier
- Efficient Energy Consumption

Why not hybrid?

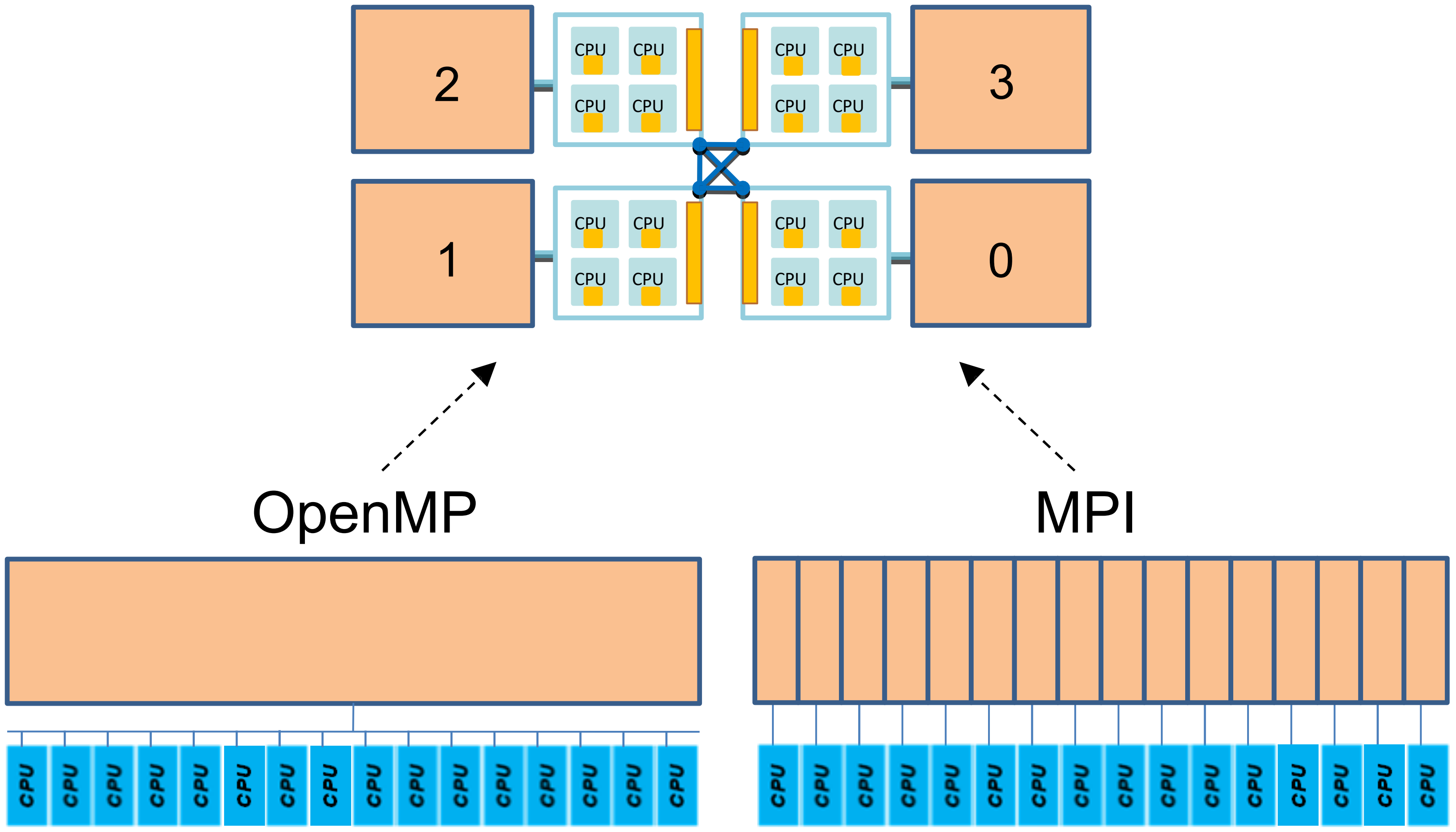
- An SMP algorithm created by aggregating MPI parallel components on a node (or on a socket) may actually run slower
- Possible waste of effort

Motivation for Hybrid

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



Two Views of a Node



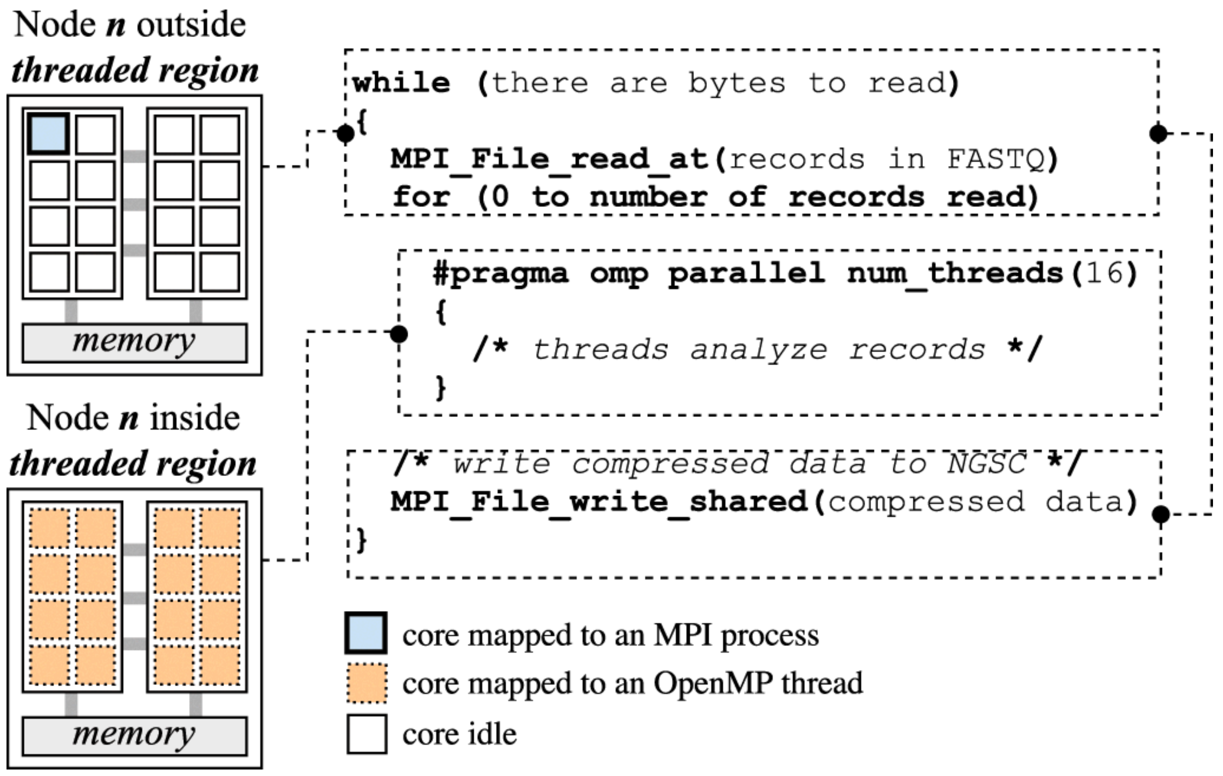
Two Views = Two Ways to Write Parallel Programs

- 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***

What is ***Hybridization***?

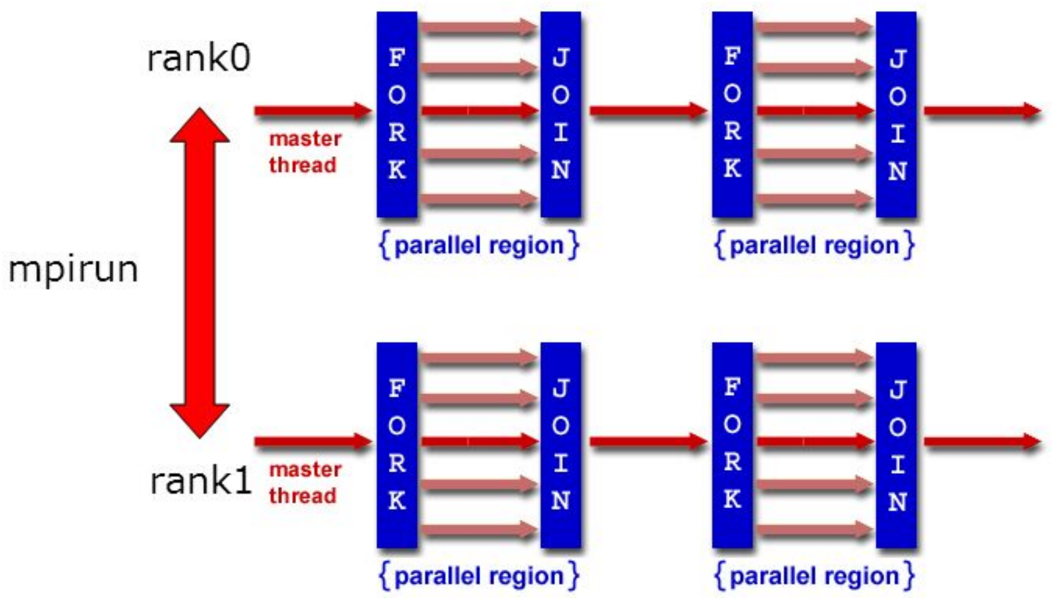
- the use of inherently different models of programming in a complimentary manner, in order to achieve some benefit not possible otherwise;
- a way to use different models of parallelization in a way that takes advantage of the good points of each;

B. Estrade <estrabd@lsu.edu>, HPC @ LSU – High Performance Computing Workshop

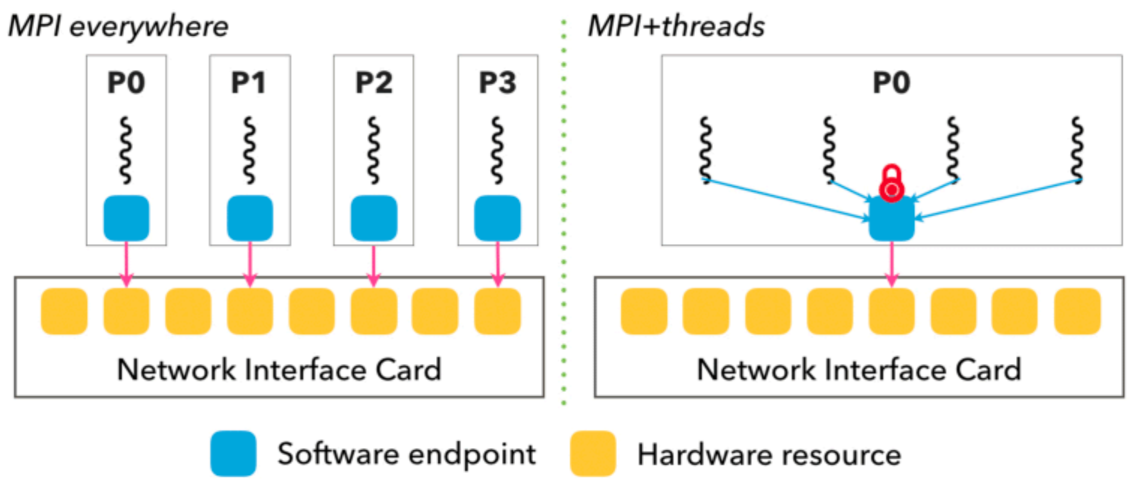


```
while (there are bytes to read)
{
  MPI_File_read_at(records in FASTQ)
  for (0 to number of records read)
    #pragma omp parallel num_threads(16)
    {
      /* threads analyze records */
    }
  /* write compressed data to NGSC */
  MPI_File_write_shared(compressed data)
}
```

◻ Each MPI process spawns multiple OpenMP threads

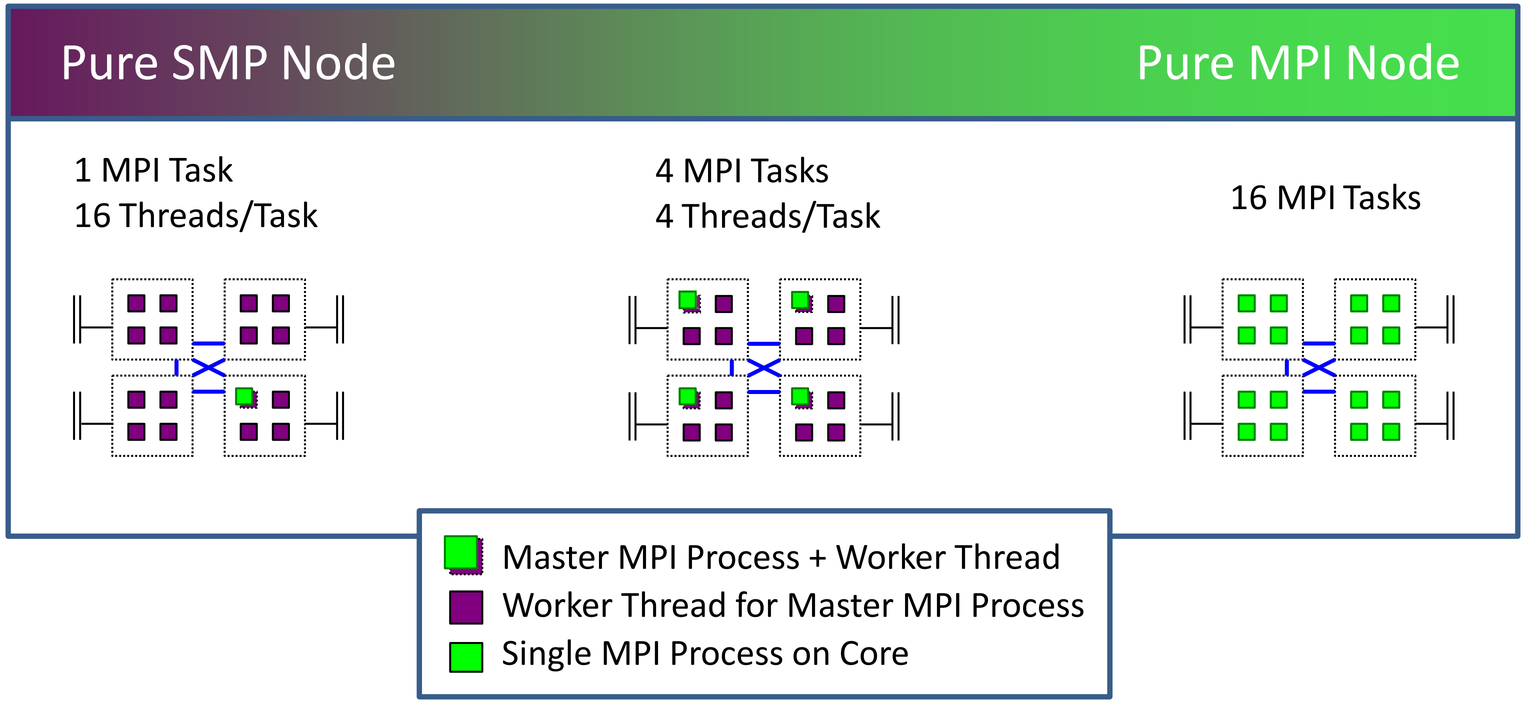


Some Possible MPI + Thread Configurations



- 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 Ranger, e.g.
- Treat each *socket* as an SMP
  - launch one MPI process on each socket
  - create parallel threads sharing same-socket memory
  - typically want 4 threads/socket on GUANE-1, 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

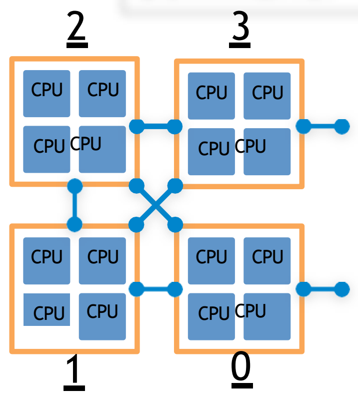
NUMA control is managed by the kernel.

NUMA control can be changed with numactl.

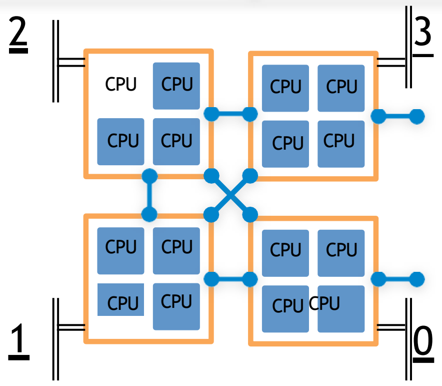
NUMA Operations

- Process Affinity and Memory Policy can be controlled at socket and core level with numactl.

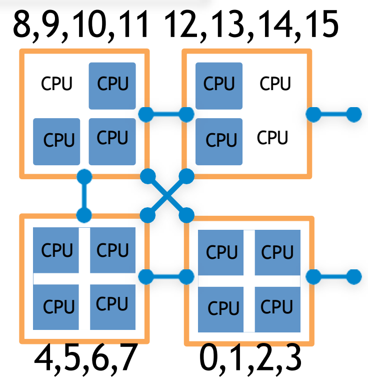
```
Command:      numactl   < options socket/  ore >   ./a.out
```



Process: Socket References
process assignment
-N



Memory: Socket References
memory allocation
-l -i --preferred -m
(local, interleaved, pref., mandatory)



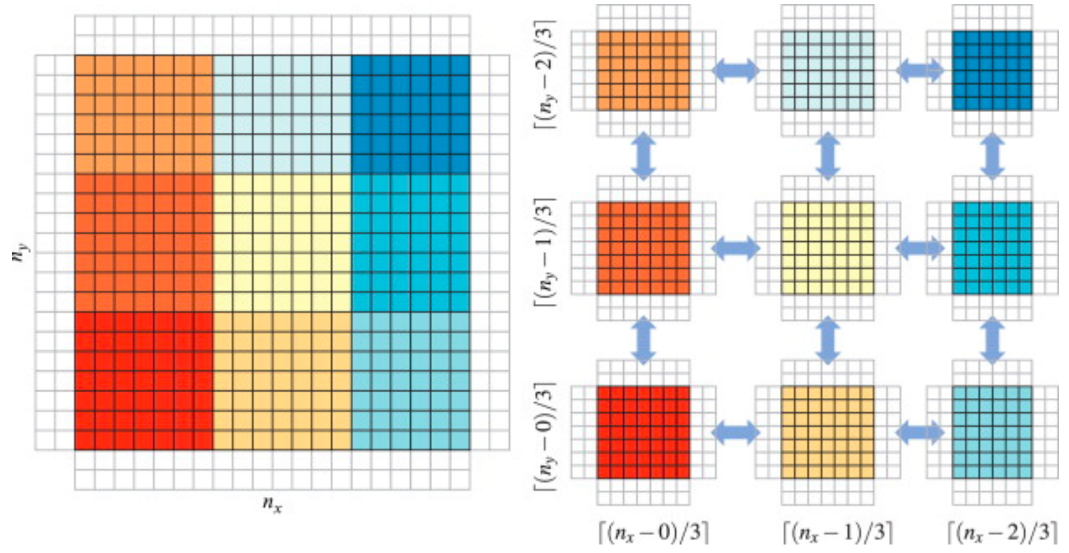
Process: Core References
core assignment
-C

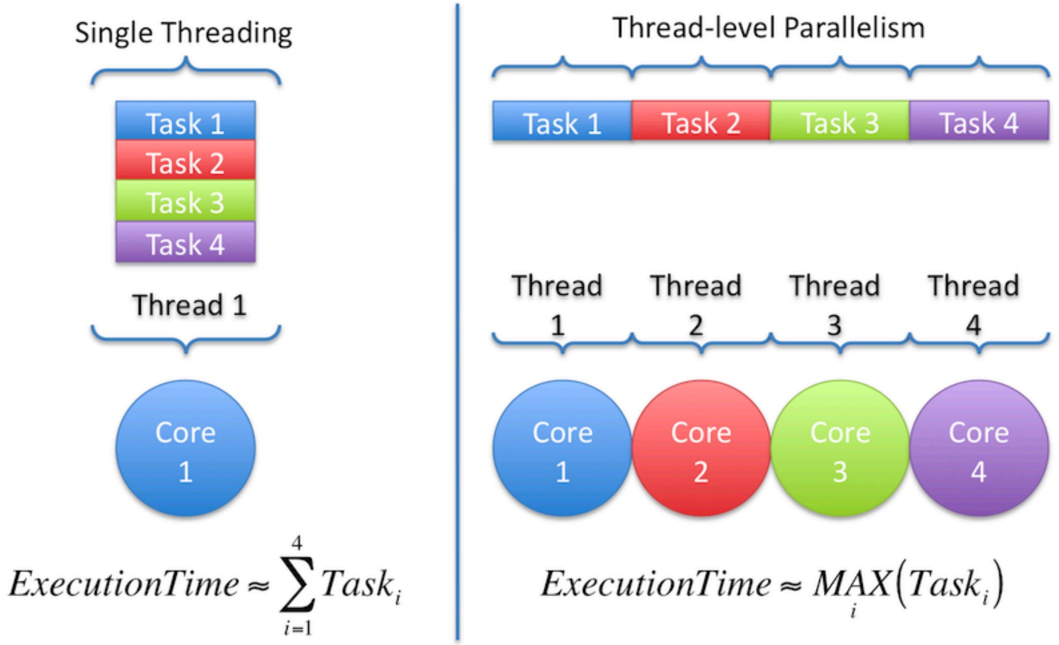
Process Affinity and Memory Policy

- 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

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

Single level parallelism


n_y
n_x
$\lceil (n_y-2)/3 \rceil$
$\lceil (n_y-1)/3 \rceil$
$\lceil (n_y-0)/3 \rceil$
$\lceil (n_x-0)/3 \rceil$
$\lceil (n_x-1)/3 \rceil$
$\lceil (n_x-2)/3 \rceil$

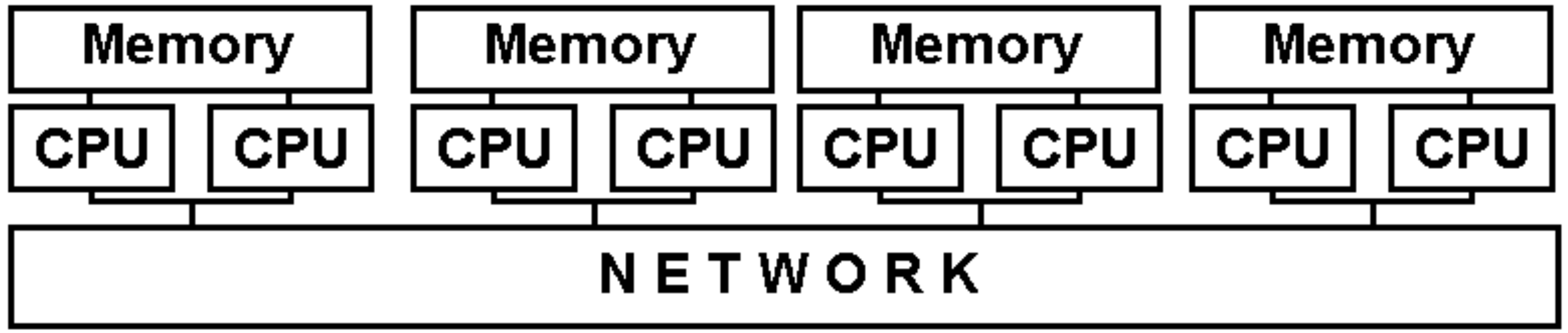


$$ExecutionTime \approx \sum_{i=1}^{4} Task_i$$

$$ExecutionTime \approx \underset{i}{MAX}(Task_i)$$

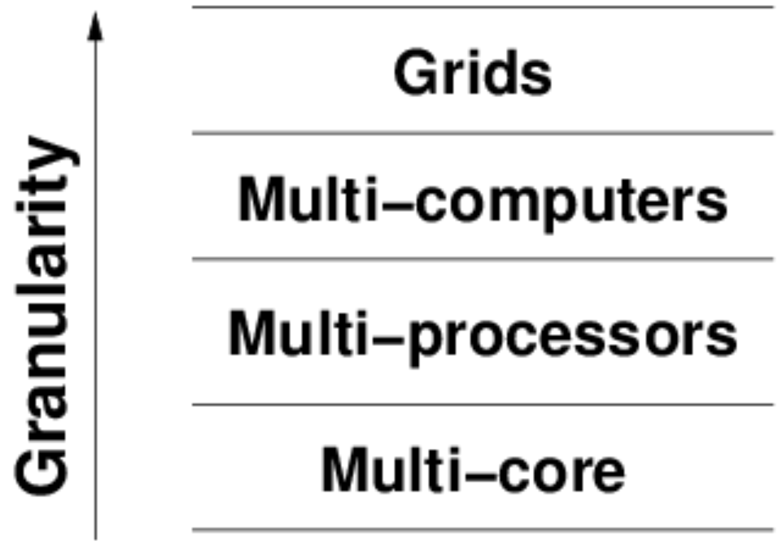
- Shared memory computers
  - N processors, single system image
  - thread-based parallelism - OpenMP, shmem
  - message-based parallelism - MPI
- Distributed memory computers
  - nodes with local memory, coupled via network
  - message-based parallelism – MPI
  - partitioned global space – UPC, Coarray Fortran

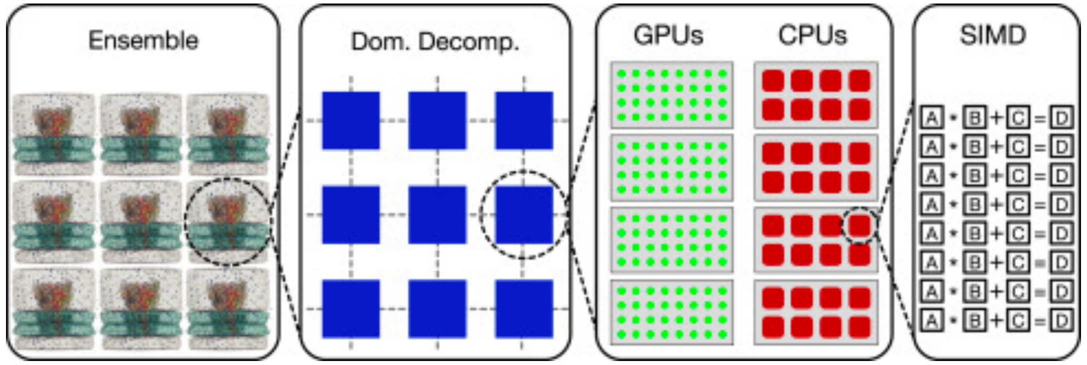
Remember: Shared-Distributed memory



- Each node has N processors that share memory
- Nodes loosely connected (network)
- CHPC:
  - 8, 12, 16, 20, 24 core cluster nodes

Multilevel parallelism





Example: GROMACS

- Coarse and fine grain level
  - coarse – nodes, processors, fine – CPU cores
  - MPI - nodes, CPU sockets
    OpenMP, pthreads, shmem – CPU cores
  - OpenMP works best with processing intensive loops
- Multilevel advantages
  - memory limitations – extra memory for each copy of executable on the node
  - process vs. thread overhead
  - message overhead
  - portability, ease to maintain (can disable OpenMP)

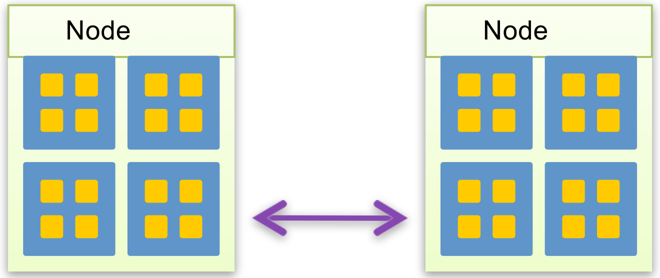
Remember MPI and OpenMP

- MPI (Message Passing Interface)
  - standardized library (not a language)
  - collection of processes communicating via messages
  - available for most architectures
  - http://www.mpi-forum.org/

- OpenMP
  - API for shared memory programming
  - available on most architectures as a compiler extension (C/C++, Fortran)
  - includes compiler directives, library routines and environment variables
  - www.openmp.org

MPI with OpenMP -- Messaging

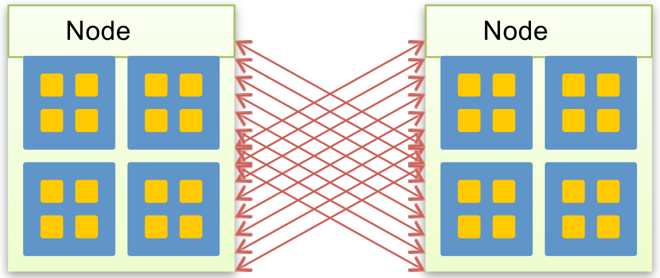
Single-threaded messaging



rank to rank

MPI from serial region or a single thread within parallel region

Multi-threaded messaging



rank-thread ID to any rank-thread ID MPI from multiple threads within parallel region Requires thread-safe implementation

Processes vs. threads

- Process
  - have own address space
  - can have multiple threads

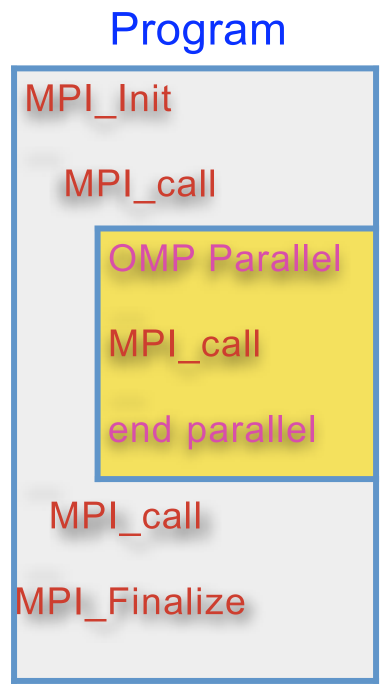
- MPI
  - many processes
  - shared-nothing architecture
  - explicit messaging
  - implicit synchronization
  - all or nothing parallelization

- Thread
  - execute within process
  - same address space
  - share process's stack
  - thread specific data

- OpenMP
  - 1 process, many threads
  - shared-everything architecture
  - implicit messaging
  - explicit synchronization
  - incremental parallelism

Hybrid – Program Model

- Start with MPI initialization
- Create OMP parallel regions within MPI task (process).
  - Serial regions are the master thread or MPI task.
  - MPI rank is known to all threads
- Call MPI library in serial and parallel regions.
- Finalize MPI



Hello World Example

```c
#include <stdio.h>
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int iam = 0, np = 1;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  //omp_set_num_threads(4);

#pragma omp parallel default(shared) private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d from process %d out of %d on %s\n",
           iam, np, rank, numprocs, processor_name);
  }

  MPI_Finalize();
}
```

Compilation, Execution and output

Compilation

```
mpicc -fopenmp hello.c -o hello then ran using
export OMP_NUM_THREADS=4
```

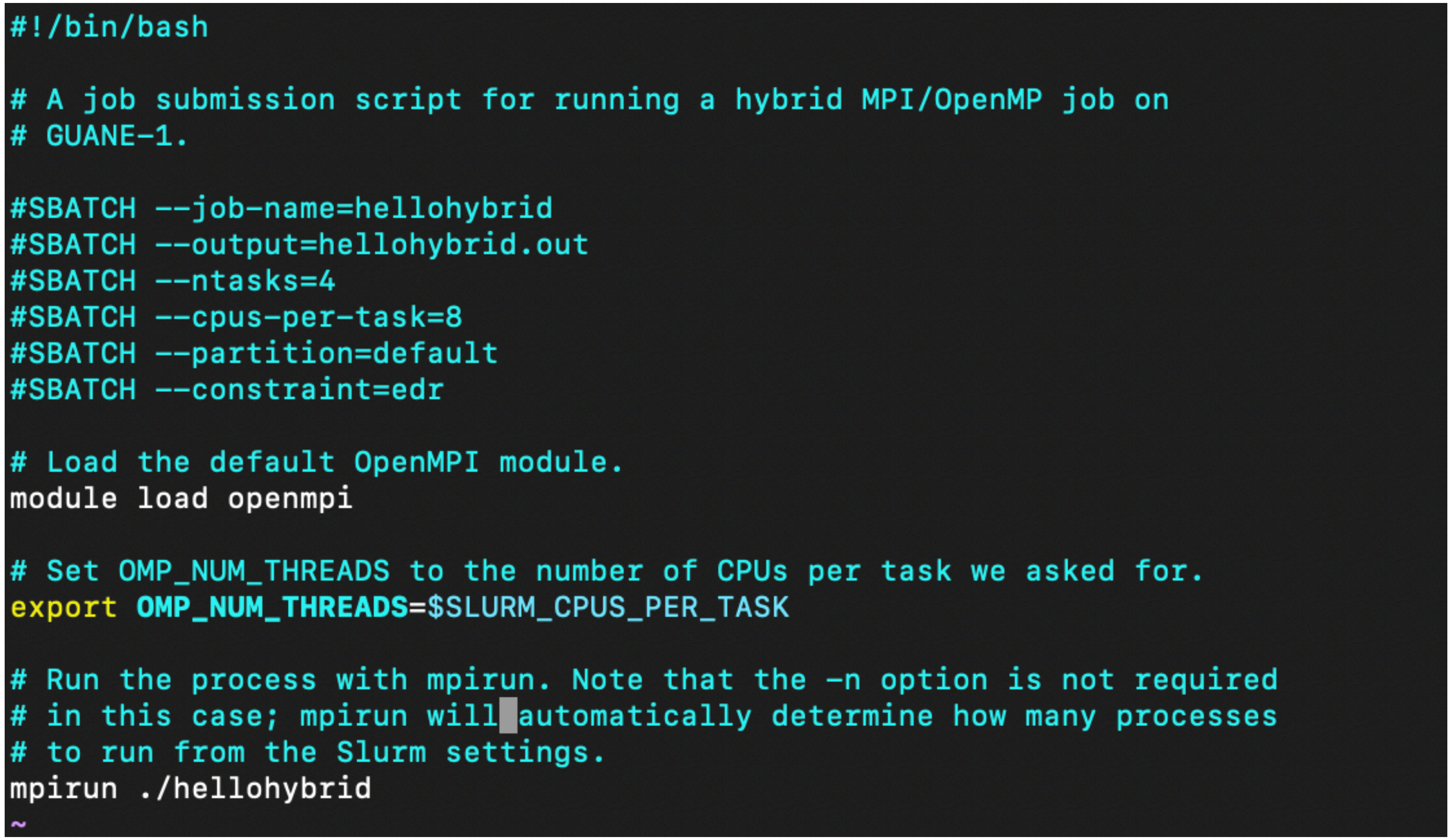
Execution

```
mpirun ./hello -np 2 -x OMP_NUM_THREADS
```

- Here is the output I am getting:

```
Hello from thread 0 out of 4 from process 0 out of 1 on GUANE-09
Hello from thread 2 out of 4 from process 0 out of 1 on GUANE-09
Hello from thread 1 out of 4 from process 0 out of 1 on GUANE-09
Hello from thread 3 out of 4 from process 0 out of 1 on GUANE-09
```

However, the sbatch file…



```
#!/bin/bash

# A job submission script for running a hybrid MPI/OpenMP job on
# GUANE-1.

#SBATCH --job-name=hellohybrid
#SBATCH --output=hellohybrid.out
#SBATCH --ntasks=4
#SBATCH --cpus-per-task=8
#SBATCH --partition=default
#SBATCH --constraint=edr

# Load the default OpenMPI module.
module load openmpi

# Set OMP_NUM_THREADS to the number of CPUs per task we asked for.
export OMP_NUM_THREADS=$SLURM_CPUS_PER_TASK

# Run the process with mpirun. Note that the -n option is not required
# in this case; mpirun will automatically determine how many processes
# to run from the Slurm settings.
mpirun ./hellohybrid
~
```

Another Example: Pi

- Calculation of value of π using integral:

$$\int_0^1 \frac{dx}{x^2+1} = \frac{\pi}{4}$$

- trapezoidal rule
- simple loop easy to parallelize both with MPI and OpenMP

Serial code

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
const int N = 10000000000;
const double h = 1.0/N;
const double PI = 3.141592653589793238462643;
double x,sum,pi,error,time; int i;

time = ctimer();
sum = 0.0;
for (i=0;i<=N;i++){
  x = h * (double)i;
  sum += 4.0/(1.0+x*x);}

pi = h*sum;
time += ctimer();

error = pi - PI;
error = error<0 ? -error:error;
printf("pi = %18.16f +/- %18.16f\n",pi,error);
printf("time = %18.16f sec\n",time);
return 0;}
```

- **User-defined timer**
- **Calculation loop**
- **Print out result**

OpenMP code

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
const int N = 10000000000;
const double h = 1.0/N;
const double PI = 3.141592653589793238462643;
double x,sum,pi,error,time; int i;

time = -ctimer();
sum = 0.0;
#pragma omp parallel for shared(N,h),private(i,x),reduction(+:sum)
for (i=0;i<=N;i++){
  x = h * (double)i;
  sum += 4.0/(1.0+x*x);}

pi = h*sum;
time += ctimer();

.......

return 0;}
```

- **OpenMP directive**

MPI code

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
const int N = 10000000000;
const double h = 1.0/N;
const double PI = 3.141592653589793238462643;
double x,sum,pi,error,time,mypi; int i;
int myrank,nproc;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);

time = -ctimer();
sum = 0.0;
for (i=myrank;i<=N;i=i+nproc){
  x = h * (double)i;
  sum += 4.0/(1.0+x*x);}
mypi = h*sum;
MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
time += ctimer();
......
return 0;}
```

- **MPI initialization**
- **Distributed loop**
- **Global reduction**

MPI-OpenMP code

```
#include <stdio.h>
#include <math.h>
#include "timer.h"
int main(int argc, char *argv[]){
const int N = 10000000000;
const double h = 1.0/N;
const double PI = 3.141592653589793238462643;
double x,sum,pi,error,time,mypi; int i;
int myrank,nproc;

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
MPI_Comm_size(MPI_COMM_WORLD,&nproc);

time = -ctimer();
sum = 0.0;

#pragma omp parallel for shared(N,h,myrank,nproc),private(i,x),reduction(+:sum)
for (i=myrank;i<=N;i=i+nproc){
  x = h * (double)i;
  sum += 4.0/(1.0+x*x);}
mypi = h*sum;
MPI_Reduce(&mypi,&pi,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);
time += ctimer();
......
return 0;}
```

- **OpenMP directive to parallelize local loop using threads**

- **Sum local values of π**

Compilation

- GNU, PGI, Intel compilers, OpenMP with –fopenmp, -mp, -openmp switch

- MPICH2, MVAPICH2, OpenMPI or Intel MPI

  `module load mpich2` **MPICH2**
  `module load mvapich2` **MVAPICH2**
  `module load openmpi` **OpenMPI**
  `module load impi` **Intel MPI**

```
mpicc -mp=numa source.c -o program.exe (PGI)
mpif90 -openmp source.f -o program.exe (Intel)
mpif90 -fopenmp source.f -o program.exe (GNU)
```

Third party libraries

- BLASes and FFTW are threaded
- Intel compilers:

```
-I…./pkg/fftw/std_intel/include
-lfftw3 -lfftw3_omp
-L.../sys/pkg/fftw/std_intel/lib
-Wl,-rpath=/…/sys/installdir/intel/mkl/lib/intel64
-L/…/sys/installdir/intel/mkl/lib/intel64
-lmkl_intel_lp64 -lmkl_intel_thread -lmkl_core -liomp5  -lpthread
```

- PGI compilers:

```
-I/…/sys/pkg/fftw/std_pgi/include
-lfftw3 -lfftw3_omp
-L/…/sys/pkg/fftw/std_pgi/lib -lacml_mp
```

- MKL ScaLAPACK w/ Intel

```
-Wl,-rpath=/…/sys/installdir/intel/mkl/lib/intel64
-L/uufs…/sys/installdir/intel/mkl/lib/intel64
-lmkl_scalapack_ilp64 -lmkl_intel_ilp64 -lmkl_core
-lmkl_intel_thread -lmkl_blacs_intelmpi_ilp64 -liomp5 -lpthread -lm
```

https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor

Running

- Ask for #MPI processes
- Use SLURM environment variables to get OpenMP thread count
- Interactive batch (asking for 2 nodes, 2 tasks/node)

```
srun -n 4 -N 2 -t 1:00:00 -p kingspeak -A chpc -pty
/bin/tcsh -l
… wait for prompt …

set TPN=`echo $SLURM_TASKS_PER_NODE | cut -f 1 -d \(`
set PPN=`echo $SLURM_JOB_CPUS_PER_NODE | cut -f 1 -d \(`
@ THREADS = ( $PPN / $TPN )
mpirun -genv OMP_NUM_THREADS=$THREADS -np $SLURM_NTASKS
./program.exe
```

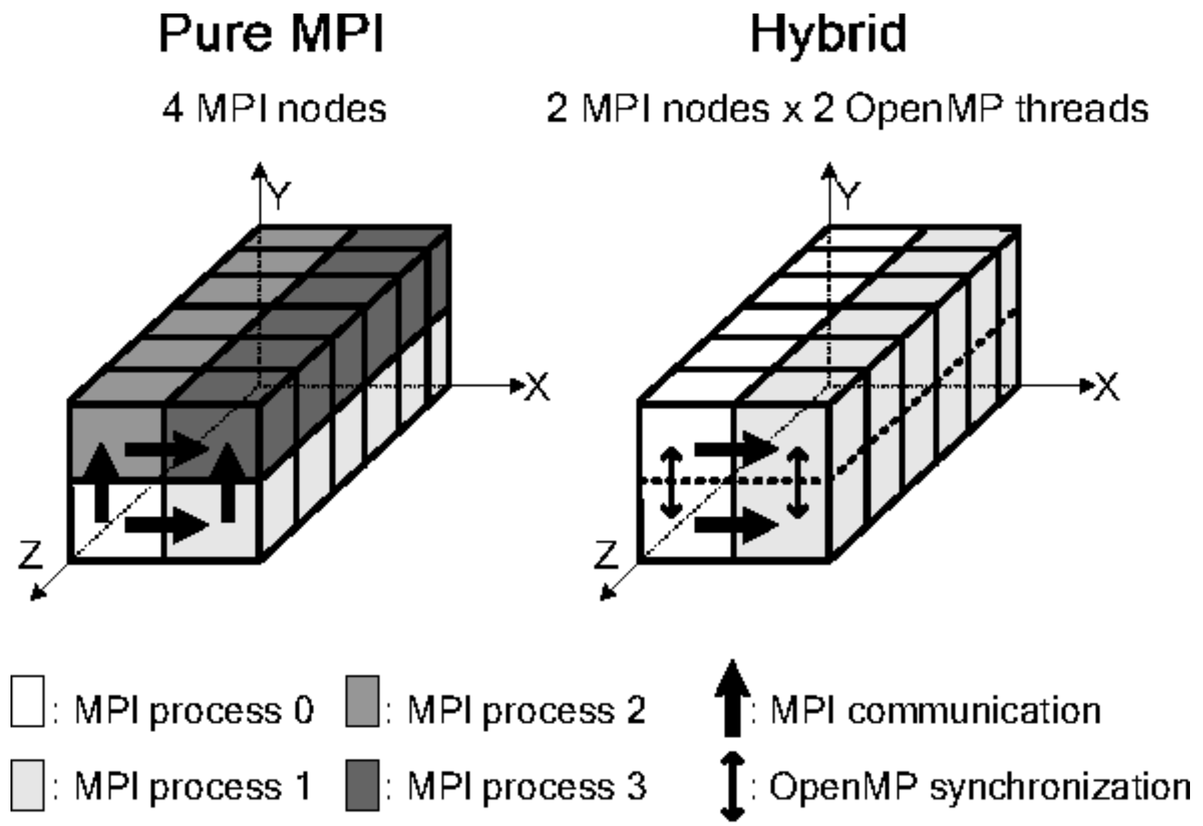
- Non-interactive batch
  - same thing, except in a Slurm script

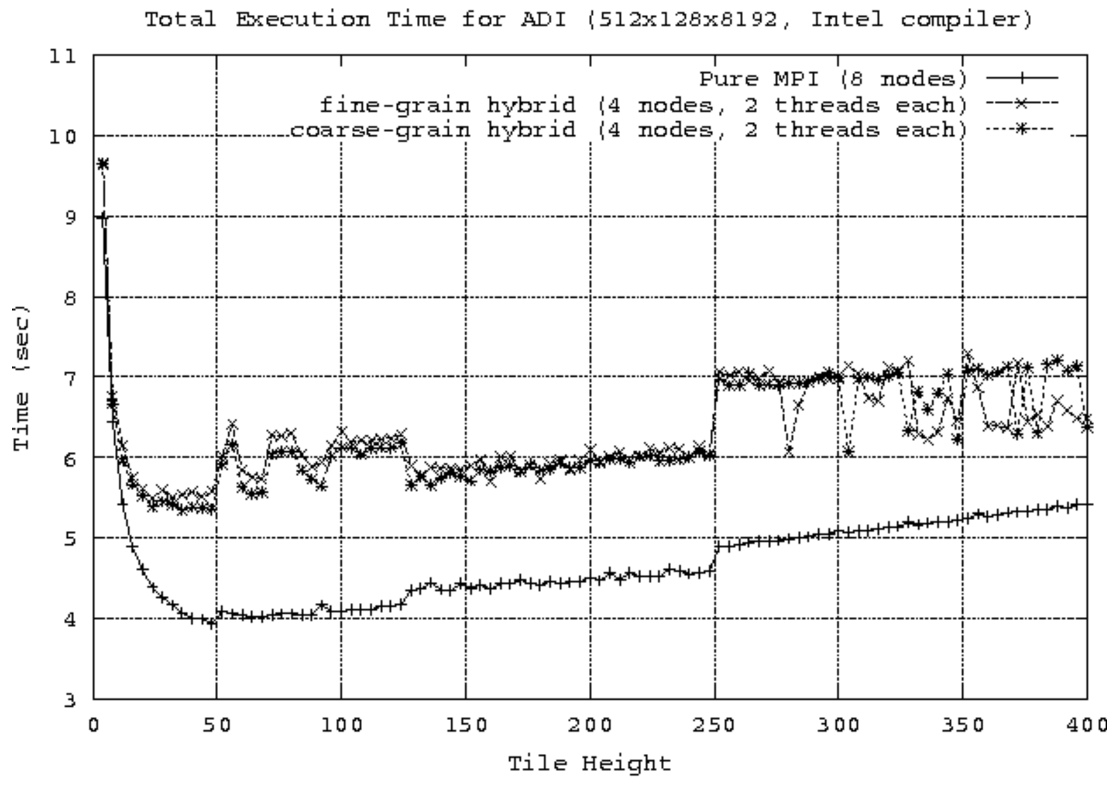
Running – process pinning

- Current NUMA architectures penalize memory access on neighboring CPU sockets
- Distribute and bind processes to CPU sockets

- Intel compilers can also pin threads to cores
  - `module load intel mvapich2`
  - `mpirun -genv KMP_AFFINITY granularity=fine,compact,1,0 -genv MV2_BINDING_POLICY scatter -genv MV2_BINDING_LEVEL socket`
    - `-genv OMP_NUM_THREADS 8 -np 4`
- Intel MPI binds processes to sockets by default
  - `Module load intel impi`
  - `mpirun -x KMP_AFFINITY granularity=fine,compact,1,0`
    - `-genv OMP_NUM_THREADS 8 -np 4`
  - or use `I_MPI_PIN_DOMAIN=socket`

Performance Comparison





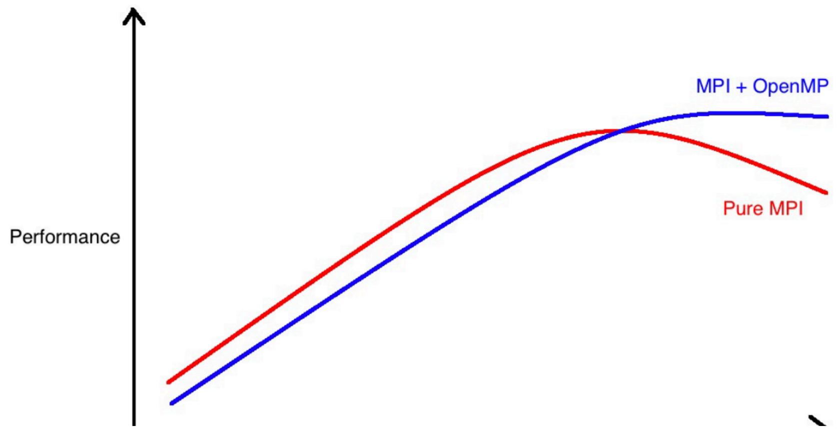
DOI:[10.1109/IPDPS.2004.1302919](), Corpus ID: 5129233

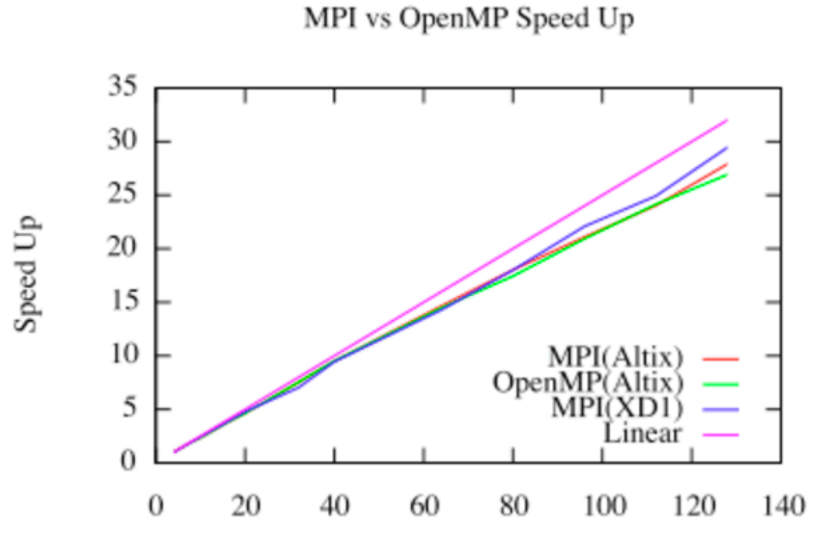
Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters, [Nikolaos Drosinos](), [Nectarios Koziris]()

General multilevel approach

- Parallelize main problem using MPI
- task decomposition
  - frequencies in wave solvers
- domain decomposition
  - distribute atoms in molecular dynamics
  - distribute mesh in ODE/PDE solvers
- Exploit internal parallelism with OpenMP
- use profiler to find most computationally intense areas
  - internal frequency loop in wave solvers
  - local force loop in MD
  - local element update loop in ODE/PDE solvers
- measure the efficiency to determine optimal number of threads to use
- Intel AdvisorXE can be helpful (`module load advisorxe`)

Attention!





Modeling pulse propagation and scattering in a dispersive medium: Performance of MPI/OpenMP hybrid code

•DOI: 10.1145/1188455.1188555

- Not every MPI program will benefit from adding threads
  - Not worth with loosely parallel codes (too little communication)
  - Overhead with thread creation about 10^4 flops
  - Time with different node/thread count to get the best performing combination
- MPI communication within OpenMP
  - Can be tricky if each thread communicates
  - Some MPI implementations still have trouble with MPI_THREAD_MULTIPLE

Four MPI threading models

- MPI_THREAD_SINGLE
  - only non-threaded section communicates
- MPI_THREAD_FUNNELLED
  - process may be multithreaded but only master thread communicates
- MPI_THREAD_SERIALIZED
  - multiple threads may communicate but only one at time
- MPI_THREAD_MULTIPLE
  - all threads communicate

Example of single thread communication.

- Complex norm routine

```
int main(int argc, char **argv){
.......
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
.......

double _Complex stabWmnorm(double *Wm, double _Complex *stab, int
size)
{
  double _Complex norm, vec, norml;
  int i;
  norml = 0 + I*0;
  #pragma omp parallel for private(i,vec) reduction(+:norml)
  for (i=0;i<size;i++)
  {
     vec = stab[i]*Wm[i];
     norml = norml + vec*conj(vec);
  }
  MPI_Allreduce(&norml,&norm,1,MPI_DOUBLE_COMPLEX,MPI_SUM,MPI_COMM_WORLD);

  return sqrt(norm);
}

MPI_Finalize();
```

Parallel OpenMP for loop

MPI communication outside OpenMP

Multiple threads comm.
- initialization

- Special MPI_Init
  - Returns variable thread_status which indicates what level of threading is supported

```
int thread_status;

MPI_Init_thread(&argc, &argv,MPI_THREAD_MULTIPLE,&thread_status);
if (thread_status!=MPI_THREAD_MULTIPLE)
{
 printf("Failed to initialize MPI_THREAD_MULTIPLE\n");
 exit(-1);
}

...

MPI_Finalize();
```

Multiple threads point-to- point communication

```
#pragma omp parallel private(iis,niip,iip,iisf)
{
 double _Complex *ne, *nh; int comlab, mythread, nthreads;
 MPI_Status statx[fwdd->Nz];
 MPI_Request reqx[fwdd->Nz];

#ifdef _OPENMP
 mythread = omp_get_thread_num(); nthreads = omp_get_num_threads();
#endif

 ne = (double _Complex *)malloc(sizeof(double _Complex)*3*Nxy);

 comlab=mythread*10000; // different tag for each proc/thread

 for (iis=mythread; iis < Ncp[0]; iis+=nthreads)
 {
   if (cpuinfo[0] == iip)
   {
    MPI_Isend( &ne[0], Nxy, MPI_DOUBLE_COMPLEX, Dp[0], comlab, MPI_COMM_WORLD, reqx[Nreqi[0]]);
    Nreqi[0]++;
   }
   else if (cpuinfo[0] == Dp[0])
   {
    MPI_Irecv(&Ebb[ie[0]*Nxy], Nxy, MPI_DOUBLE_COMPLEX, iip, comlab, MPI_COMM_WORLD, reqx[Nreqi[0]]);
    Nreqi[0]++;
   }
   MPI_Waitall(Nreqi[0], &reqx[0], &statx[0]);
 }

 free(ne);

}
```

Start parallel OpenMP section

Data structures for non-blocking communication

Find thread # and # of threads

Allocate local thread arrays

Each thread does different iteration of this loop

Each communication pair has unique tag

Finalize non-blocking communication

Free local thread arrays

End OpenMP parallel section

-> use message tag to differentiate between threads

Multiple threads collective communication

```
MPI_Comm comm_thread[NOMPCPUS];

#pragma omp parallel private(iis,niip,iip,iisf)
{
 double _Complex *ne; int mythread, nthreads

#ifdef _OPENMP
 mythread = omp_get_thread_num(); nthreads = omp_get_num_threads();
#endif

 ne = (double _Complex *)malloc(sizeof(double _Complex)*3*Nxy);

 for(ithr=0;ithr<nthreads;ithr++)
 {
   #pragma omp barrier // synchronize so that each process gets the right thread
   if (ithr==mythread) MPI_Comm_dup(comm_domain,&comm_thread[mythread]);
 }
 for (iis=mythread; iis < Ncp[0]; iis+=nthreads)
 {
    … calculate ne …
    MPI_Gatherv( &ne[indgbp[iic]],Nxy_loc,MPI_DOUBLE_COMPLEX, &Gb[ie[ic]*Nxy2 + iit2], Nxy_rec,
    Nxy_disp, MPI_DOUBLE_COMPLEX, Dp[ic],comm_thread[mythread]);
 }

 for(ithr=0;ithr<nthreads;ithr++)
 {
   if (ithr==mythread) MPI_Comm_free(&comm_thread[mythread]);
 }

 free(ne);
}
```

Start parallel OpenMP section

Local thread variables

Find thread # and # of threads

Allocate local thread arrays

Per thread communicator

Each thread does different iteration of this loop

Thread communicator

Free thread communicators

Free local thread arrays

End OpenMP parallel section

-> use communicators to differentiate between threads

Further Use

- Mixed MPI-OpenMP has become commonplace
- reduces memory footprint per core
- better locality of memory access per core
- faster inter-node communication – larger messages, smaller overhead
- More Complex Codes Needs More Hybrid Solutions (Smart Solutions)
- Also we can mix CUDA+OpenMP+MPI
- … or use OpenACC, OMPSs…

Another MPI-OpenMP example

- Master-worker code
  - good for parallelization of problems of varying run time
  - master feeds workers with work until all is done
- Disadvantage – master does not do any work
- Run two OpenMP threads on the master
  - distribute work
  - do work
- Critical section at the work selection
- Can run also on single processor nodes

Master-worker MPI- OpenMP implementation

```
int main(int argc, char **argv){
.......
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
.......
master = numprocs - 1;
.......
if (myid == master) {
.......
omp_set_num_threads(2);
#pragma omp parallel sections private(request) {
#pragma omp section {
.......
#pragma omp critical (gen_work) {
    work = generate_work(&work_data, num_tasks, work_array, job_flag);
}
.......
}
#pragma omp section{
.......
#pragma omp critical (gen_work){
    work = generate_work(&work_sl_data, num_tasks, work_array, job_flag);
}
.......
}
#pragma omp barrier
.......
}
else {
.......
}
.......
MPI_Barrier(world); MPI_Finalize();}
```

Master section

Master thread master processor

Critical section – work generation

Worker thread of the master processor

Critical section – work generation

End OpenMP sections

Workers - send work requests and receive work

Conclusions

- You need to know your platform (architecture features)
- It is possible to achieve single and multilevel parallelism
- Compilation, running is easy (however it is possible to be differences between platforms)
- Scalability Guaranteed
- However, be careful

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: Ranger User Guide, see numa section. www.tacc.utexas.edu/services/userguides/ranger
- 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
- http://www.chpc.utah.edu/short_courses/mpi_omp