Hybrid Heterogeneous Computing

Victor Eijkhout

PCSE
2016

TACC
THE UNIVERSITY OF TEXAS AT AUSTIN
Texas Advanced Computing Center
THE UNIVERSITY OF TEXAS AT AUSTIN

Memory Hierarchy

- Distributed and Shared Memory Parallel Paradigms in HPC
  - MPI: addresses data movement in distributed memory (between processes-- executables)
  - OpenMP: addresses data access in shared memory (among threads in an executable)

Memory Hierarchy

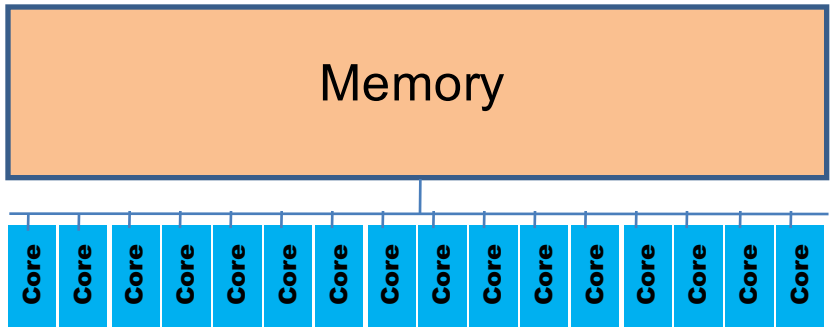
Let's think about using MPI and MPI together.

- Processes/Threads and Data on Nodes
  - Hardware: On a node, both use the same hardware.
  - Control: It's about location, location, location.
  - Speed: Location and Organization
  - Simultaneous Access:
    MPI makes copies
    OpenMP uses locks

Node Views

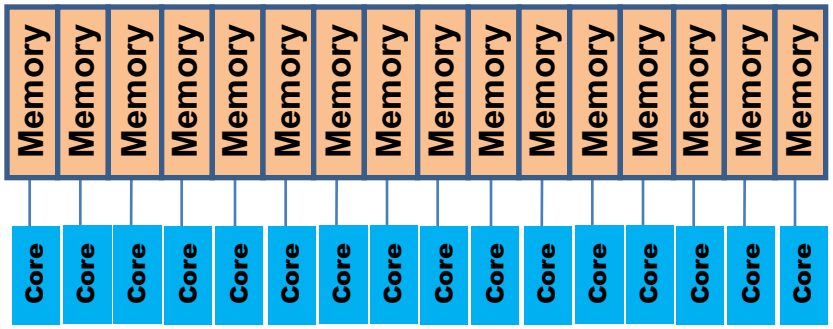
Paradigms

OpenMP



Run a bunch of threads in shared memory(spawned by a single a.out).

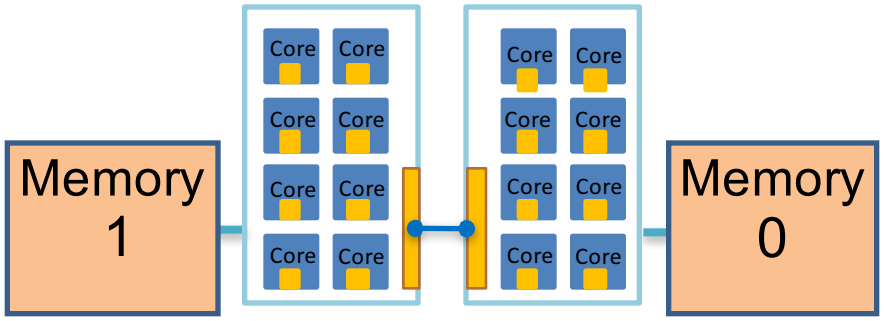
MPI



Run a bunch of a.out's as distributed memory paradigm

Hardware

Stampede, host Node: **N**on-**U**niform **M**emory **A**ccess (NUMA)



Reality of Process Location & Data Storage

2 sockets each with 8 cores

TACC

THE UNIVERSITY OF TEXAS AT AUSTIN

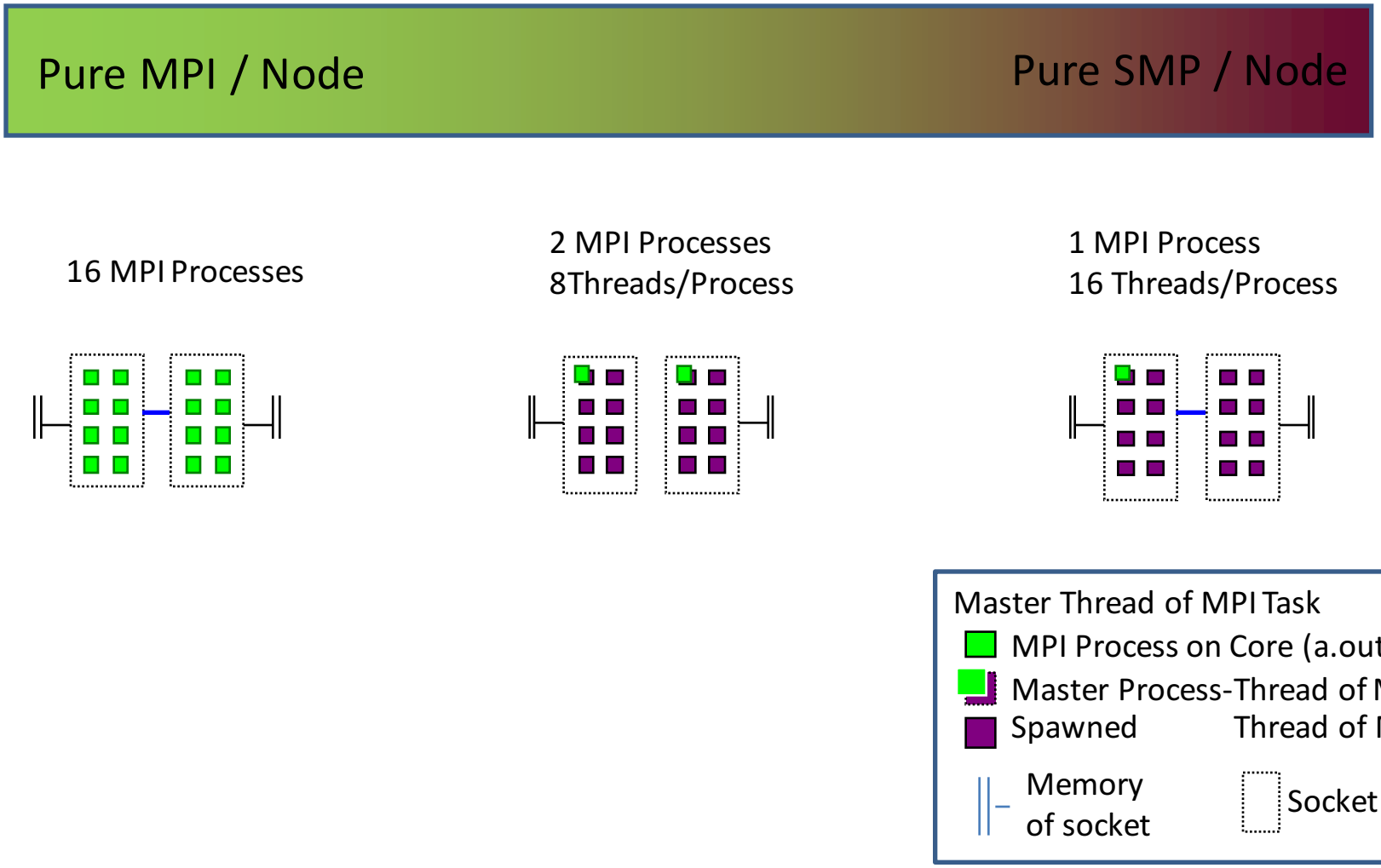
MPI & OpenMP == Hybrid

- Developed as SPMD (put it all in a single code)
- Use MPI task as a container for OpenMP threads*.
  - Makes sense since MPI tasks don't share memory.
  - OpenMP threads can access all of memory within the task
  - Implies the obvious—Use MPI parallelism across Nodes and OpenMP within a Nodes.
  - But, you may see multiple MPI tasks on a node!

*Actually you can initiate MPI within a parallel region–
but there are restrictions and there is no reasonable use case.

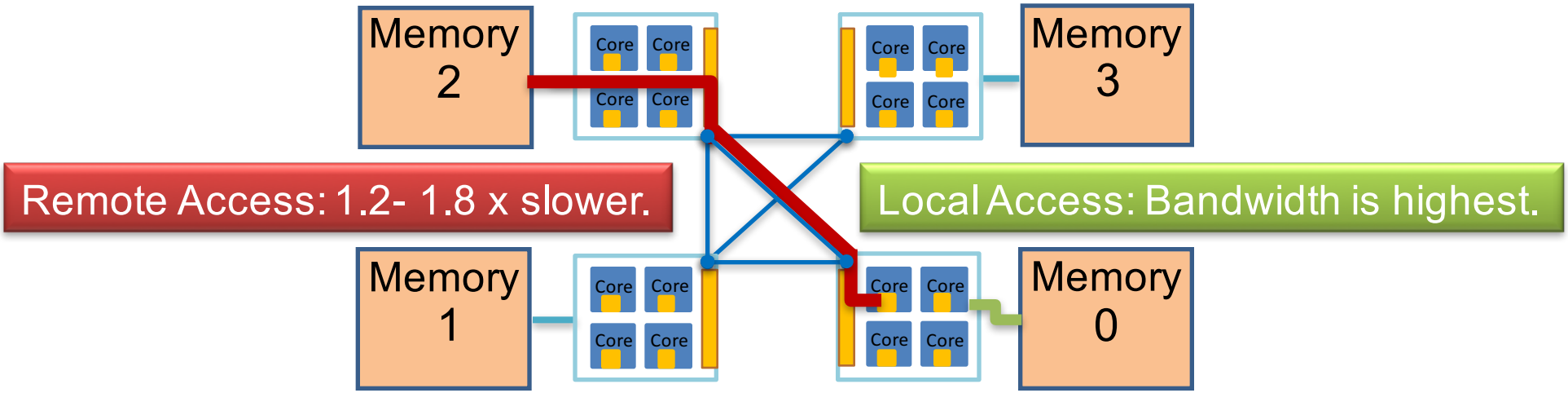
Modes of MPI/OpenMP Operation

Hardware View



Motivation for Memory Locality

TACC's Ranger system has 4 sockets per node with an asymmetric interconnect. Shared Memory Access can vary significantly in this NUMA system.



- MPI tasks on each (of 4) sockets guarantee no remote access and each MPI-task's OpenMP threading is always local.

Pure OpenMP codes or hybrid codes with a single MPI-task per node:

- If thread access is mainly local, each thread should initialize its portion of data. Do NOT initialize in serial section of code.
- If each thread accesses all of memory, try interleave memory policy.

Hybrid – Program Model

- Start with special 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 or parallel regions.
- Finalize MPI

Program

```
MPI_init_thread
...
  MPI_call
      OMP Parallel
      ...
      MPI_call
      ...
      end Parallel
...
  MPI_call
...
MPI_finalize
```

From 2.1 MPI Standard

- MPI calls can be thread-safe:
- Concurrent executing threads can call MPI library routines with the outcome as if the calls "executed in some order, even if their execution is interleaved".
- Blocking MPI calls will block for the calling thread only, allowing another thread to execute, if available. (Only the encountering thread waits on MPI block call.)
- The calling thread will be blocked until the event on which it is waiting occurs. (Thread is unrunnable and MPI triggers the release.)
- Once the blocked communication is enabled and can proceed, then the call will complete and the thread will be marked runnable, within a finite time. (Thread release is guaranteed, making it runnable within the OpenMP realm.)
- A blocked thread will not prevent progress of other runnable threads on the same process, and will not prevent them from executing MPI calls.

Thread safety not guaranteed

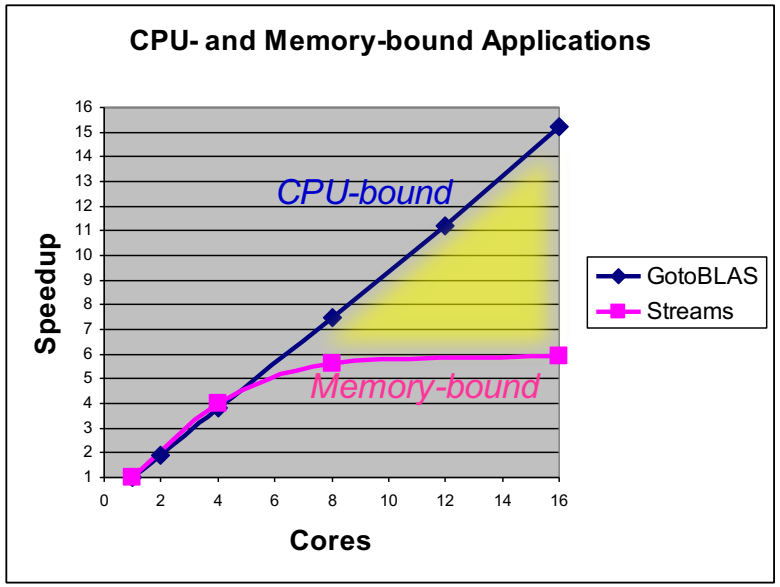
- MPI and Threads Section 12.4
- Implementation not obliged to fulfill all "requirements" of section 12.4

- Thread-safe MPI implementation guaranteed not to interfere with the progress of another thread– there may be interleaving of execution.

Hybrid - Motivation

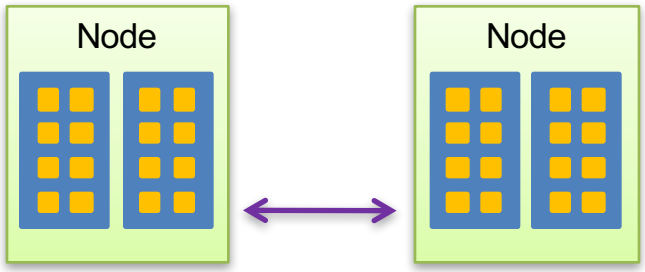
- Optimize Node Memory Traffic
  - Combine MPI Send/Recs
- Load Balancing
  - OpenMP has runtime scheduling
  - Aggregated work can be balanced
- Reduce number of MPI tasks
  - Aggregated Sends are better*
  - Fewer tasks for collectives
  - Less buffered space and less data copies

Mileage will vary with type of App.



MPI with OpenMP -- Messaging

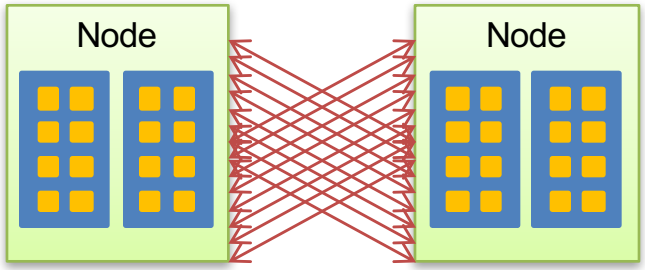
Multi-threaded:
MPI through Master
(may not be in parallel region)



rank to rank

MPI from serial region or a single thread within parallel region

Multi-threaded:
MPI with all/any thread



rank-thread ID to any rank-thread ID

MPI from multiple threads within parallel region
<u>Requires thread-safe implementation</u>

Hybrid Coding – MPI with Master

Fortran

```
include 'mpif.h'
use omp_lib
program single_thread

call MPI_Init(ierr)
call MPI_Comm_rank (...,irank,ierr)
call MPI_Comm_size (...,isize,ierr)

! MPI with Master thread (here)

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

! MPI with Master thread (or here)

call MPI_Finalize(ierr)
end
```

C

```
#include <mpi.h>
#include <omp.h>
int main(int argc, char **argv){

ierr= MPI_Init(&argc,&argv[]);
ierr= MPI_Comm_rank (...,&rank);
ierr= MPI_Comm_size (...,&size);

//MPI with Master (here)

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

// MPI with Master (or here)

ierr= MPI_Finalize();
}
```

Hybrid Coding – MPI with threads

Fortran

```
include 'mpif.h'
use omp_lib
program multi_thread

call MPI_Init_thread(MPI_THREAD_MULTIPLE,
                                iprovided,ierr)

!  MPI  with Master thread

!$OMP parallel

  !$OMP barrier      !may be necessary

    call MPI_<Whatever>(...,ierr)   {any/all threads}

!$OMP end parallel

!  MPI  with Master thread

end
```

C

```
#include <mpi.h>
#include <omp.h>
int main(int argc, char **argv){

MPI_Init_thread(...,MPI_THREAD_MULTIPLE,
                                iprovided)

//MPI with Master

  #pragma omp parallel
  {
   #pragma omp barrier   //maybe

    ierr=MPI_<Whatever>(...)   {any/all threads}

  }

//MPI with Master

}
```

MPI2 MPI_Init_thread

Syntax:

```
call MPI_Init_thread(required, provided, ierr)
int  MPI_Init_thread(int  *argc, char  ***argv, int required, int *provided)
int  MPI::Init_thread(int& argc, char**& argv, int required)
```

Support Levels	Description
MPI_THREAD_SINGLE	Only one thread will execute.
MPI_THREAD_FUNNELED	Process may be multi-threaded, but only main thread will make MPI calls (calls are "funneled" to main thread).
MPI_THREAD_SERIALIZE	Process may be multi-threaded, any thread can make MPI calls, but threads cannot execute MPI calls concurrently (all MPI calls must be ”serialized").
MPI_THREAD_MULTIPLE	Multiple threads may call MPI, no restrictions.

If possible, the call will return provided = required. Otherwise, the highest level of support will be provided.

MPI Calls Funneled through Main Thread

- MPI_THREAD_FUNNELED
- MPI Main Thread is the thread that called MPI_INIT -- Usually Master *
- Probably requires OMP_BARRIER <u>before</u> and <u>after</u> MPI call. Remember, there is no implicit barrier in a master construct (OMP_MASTER).
- With barriers, other threads will be sleeping.

*Technically, MPI_Init_thread can be called in a parallel region by only one thread and that thread becomes the MPI main thread– this is complicated and nobody does this.

Funneling through Master

Fortran

```
include 'mpif.h'
program hybmas


!$OMP parallel

  !$OMP barrier
  !$OMP master

  call MPI_<whatever>(…,ierr)
  !$OMP end master

  !$OMP barrier

!$OMP end parallel
end
```

C

```
#include <mpi.h>
int main(int argc, char **argv){
 int rank, size, ierr, i;

#pragma omp parallel
{
  #pragma omp barrier
  #pragma omp master
  {
   ierr=MPI_<Whatever>(…)
  }

  #pragma omp barrier

}
}
```

MPI Call within Single

- MPI_THREAD_SERIALIZED
- Probably requires OMP_BARRIER at beginning, since there is an implicit barrier in SINGLE workshare construct (OMP_SINGLE).
- All other threads will be sleeping.

- (The simplest case is for any thread to execute a single mpi call, e.g. with the "single" omp construct. See next slide.)

Serialize through Single

Fortran

```
include 'mpif.h'
program hybsing
call mpi_init_thread(MPI_THREAD_THREADED,
                     iprovided,ierr)
!$OMP parallel

  !$OMP barrier
  !$OMP single

   call MPI_<whatever>(...,ierr)
  !$OMP end single

  !!OMP barrier  not required

!$OMP end parallel
end
```

C

```
#include <mpi.h>
int main(int argc, char **argv){
int rank, size, ierr, i;
mpi_init_thread(MPI_THREAD_THREADED,
iprovided)
#pragma omp parallel
{
  #pragma omp barrier
  #pragma omp single
  {
   ierr=MPI_<Whatever>(...)
  }

  //pragma omp barrier not required

}
}
```

Overlapping Communication and Work

- One core might saturate the PCI-e ←→network bus. Why use all to communicate?
- Communicate with one or several threads.
- Work with others during communication.
- Need at least MPI_THREAD_FUNNELED support.
- Can be difficult to manage and load balance!

Overlapping Communication and Work

Fortran

```
include 'mpi.h'
program hybover


!$OMP parallel

  if (ithread == 0) then
    call MPI_<whatever>(…,ierr)
  else
    <work>
  endif

!$OMP end parallel
end
```

C

```
#include <mpi.h>
int main(int argc, char **argv){
 int rank, size, ierr, i;

#pragma omp parallel
{
  if (thread == 0){
    ierr=MPI_<Whatever>(…)
  }
  if(thread != 0){
    work
  }

}
```

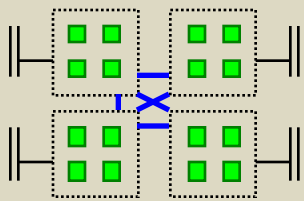
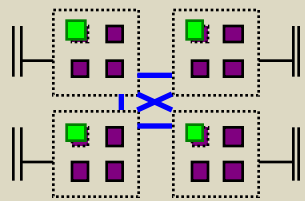
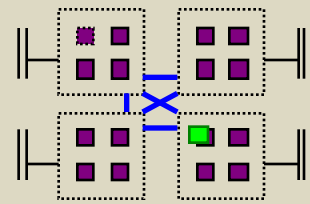
Thread-rank Communication

```
use omp_lib                                                    !  Uses just 2 ranks
include "mpif.h"
...
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)
...
!$OMP parallel private(j, ithread, nthreads) num_threads(2)
...
   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,  istatus,ierr)

      write(*, '("Recd rank:thread",2i4,"  from thread:",i2)' ) irank,ithread,j
   endif

!$OMP END PARALLEL
end
                                  ! compile:  mpixx –openmp –O2 –xhost prog.xx
```

Communicate between ranks.

Threads use tags to differentiate.

Modes of MPI/OpenMP Operation

	 16 MPI Processes	 4 MPI Processes 4Threads/Process	 1 MPI Process 16 Threads/Process
Process (a.out) Location	Use defaults (no concern)	Force 4 a.out's across sockets	Force thread-level affinity
Storage Access	FORCE* local	FORCE* local	Application dependent

Use numactl to position a.out's and API within code for threads (if necessary).

Affinity

- With two MPI tasks per node & 8 threads per task, you want one MPI task per socket
- Use “ibrun tacc_affinity yourprogram” (wrapper around ”numactl”)

“First touch”

- When you allocate memory, it is not actually allocated.
- It is allocated the first time you touch it (read or write)
- Wrong way:
  - Initialize array from one thread
  - Process it from all threads
  - Array may be allocated on one socket, accessed from the other: slow(ish)

numactl Quick Guide

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