西北工業大學

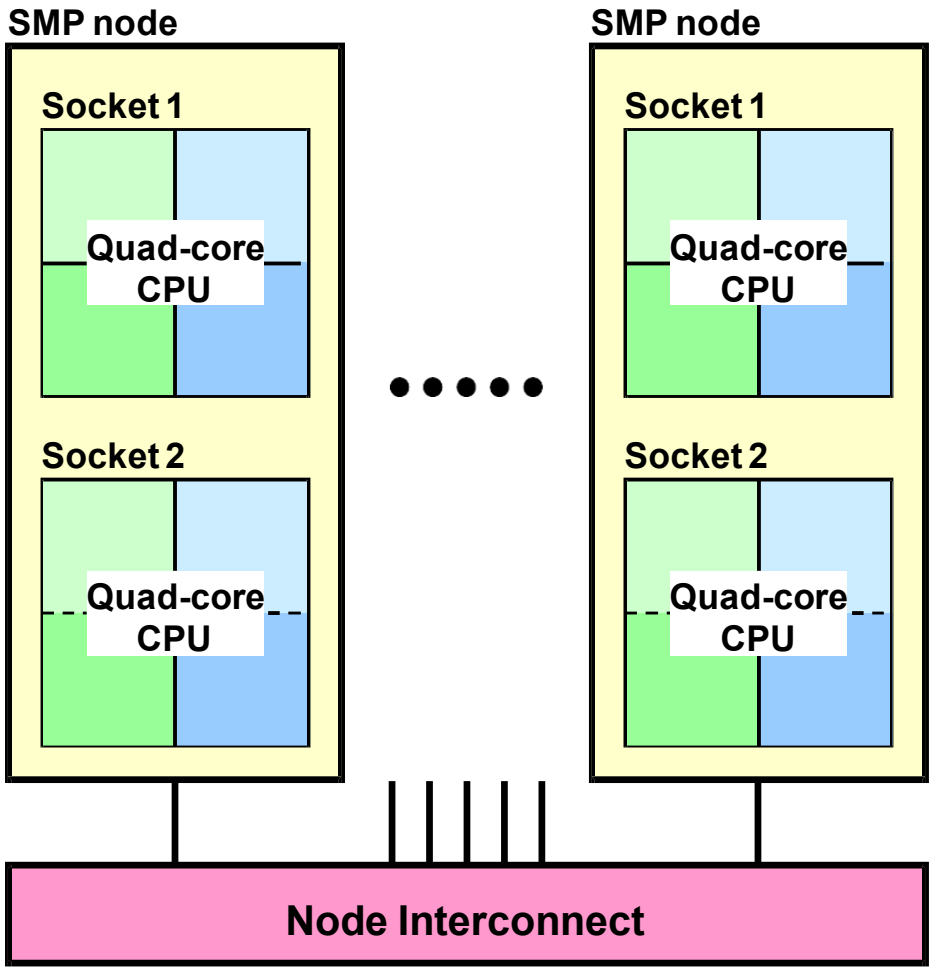
Parallel Computing

Hybrid MPI and OpenMP Parallel Programming

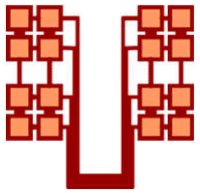
Objectives

- difference between message passing (MPI) and shared memory (OpenMP) approaches
- why hybrid?
- a straightforward approach to combine both MPI and OpenMP in parallel programming
- example hybrid code, compile and execute hybrid code on clusters

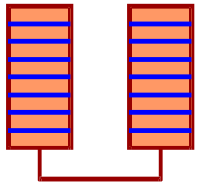
Motivation



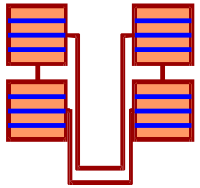
- Which programming model is fastest?
- MPI everywhere?



- Fully hybrid MPI & OpenMP?



- Something between? (Mixed model)

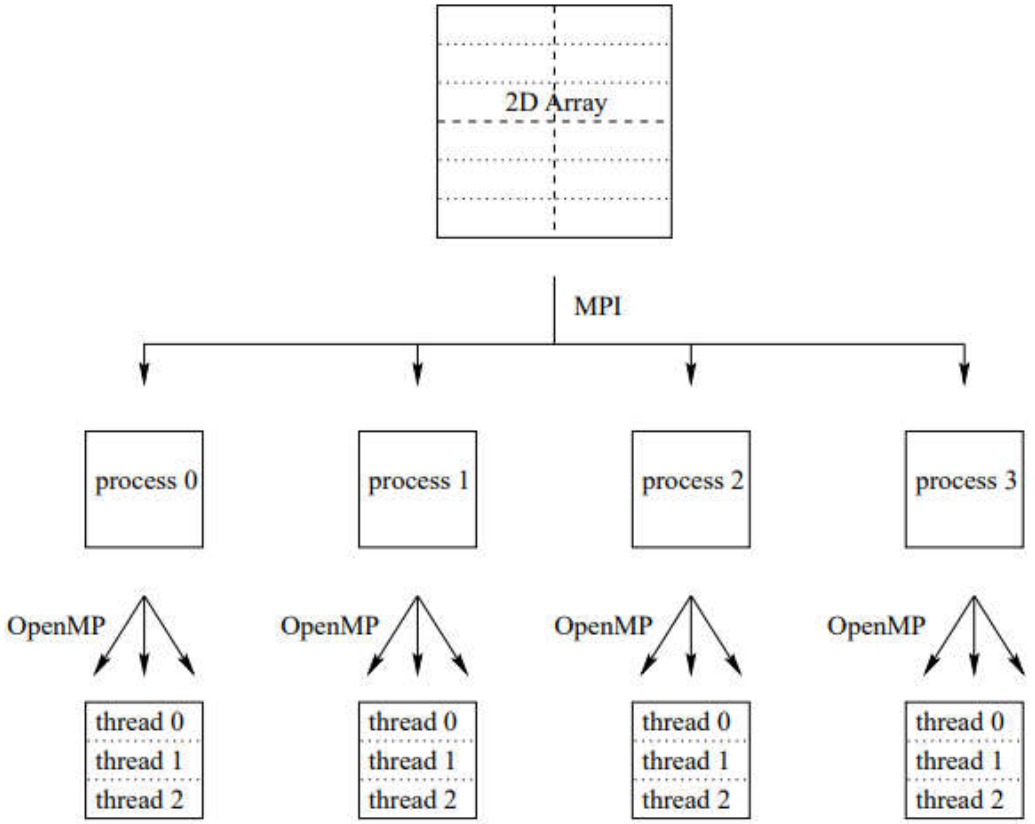


- Often hybrid programming **slower** than pure MPI
  - Examples, Reasons, …

?

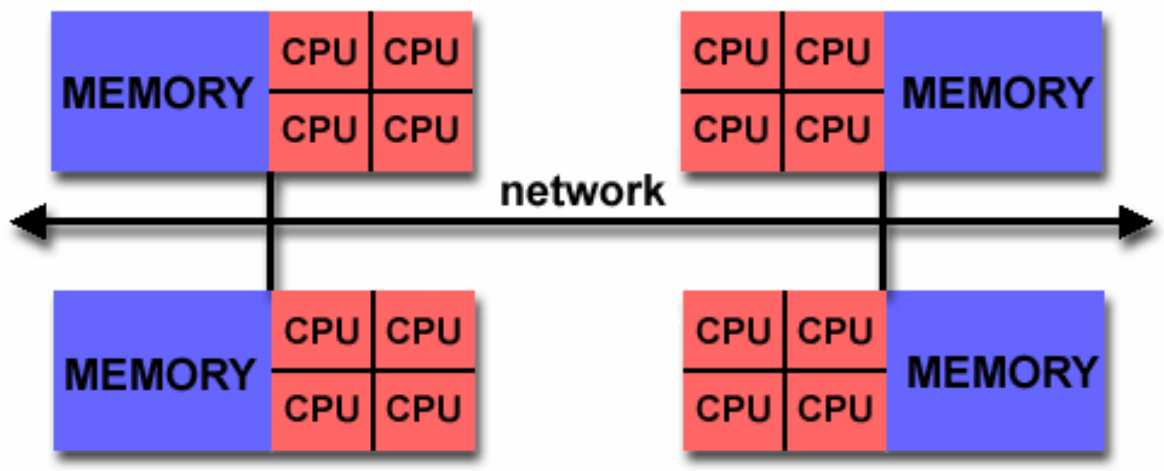
Motivation

- For example, a mixed mode program may allow the data placement policies of MPI to be utilized with the finer grain parallelism of OpenMP.
- The majority of mixed mode applications involve a hierarchical model; MPI parallelization occurring at the top level, and OpenMP parallelization occurring below



a 2D grid which has been divided geometrically between four MPI processes. These subarrays have then been further divided between three OpenMP threads. This model closely maps to the architecture of an SMP cluster, the MPI parallelization occurring between the SMP nodes and the OpenMP parallelization within the nodes.

Hybrid Distributed-Shared Memory Architecture



- Computer cluster basics, Employ both shared and distributed memory architectures

- The shared memory component is usually a cache coherent SMP node. Processors on a given SMP node can address that node's memory as global.

- The distributed memory component is the networking of multiple SMP nodes. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.

MPI

- standard for distributed memory communications
- provides an explicit means to use message passing on distributed memory clusters
- specializes in packing and sending complex data structures over the network
- data goes to the process
- synchronization must be handled explicitly due to the nature of distributed memory

OpenMP

- a shared memory paradigm, implicit intra-node communication
- efficient utilization of shared memory SMP systems
- easy threaded programming, supported by most major compilers
- the process goes to the data, communication among threads is implicit

MPI *vs.* OpenMP

- **Pure MPI Pros:**
  - Portable to distributed and shared memory machines.
  - Scales beyond one node
  - No data placement problem
- **Pure MPI Cons:**
  - Explicit communication
  - High latency, low bandwidth
  - Difficult load balancing
- **Pure OpenMP Pros:**
  - Easy to implement parallelism
  - Implicit Communication
  - Low latency, high bandwidth
  - Dynamic load balancing
- **Pure OpenMP Cons:**
  - Only on shared memory node or machine
  - Scale within one node
  - data placement problem

Why Hybrid: employ the best from both approaches

- MPI makes inter-node communication relatively easy
- MPI facilitates efficient inter-node scatters, reductions, and sending of complex data structures
- Since program state synchronization is done explicitly with messages, correctness issues are relatively easy to avoid

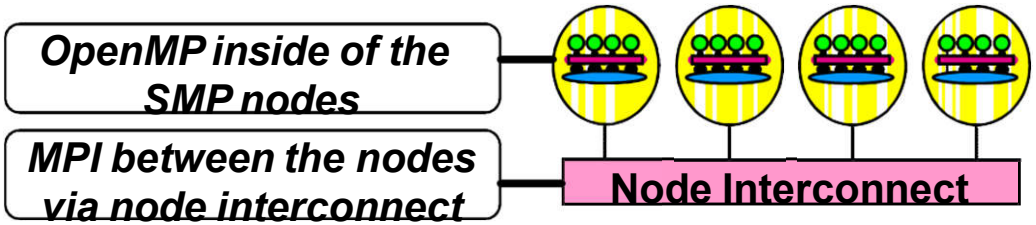
- OpenMP allows for high performance, and relatively straightforward, intra-node threading
- OpenMP provides an interface for the concurrent utilization of each SMP's shared memory, which is much more efficient that using message passing
- Program state synchronization is implicit on each SMP node, which eliminates much of the overhead associated with message passing

Overall Goal:

to reduce communication needs and memory consumption, or improve load balance

Major Programming models on hybrid systems

- Pure MPI (one MPI process on each core)
- Hybrid: **MPI + OpenMP**
  - shared memory OpenMP
  - distributed memory MPI

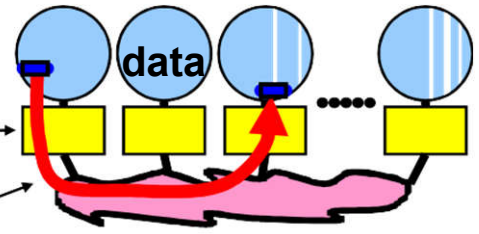


Hybrid: MPI message passing + **MPI-3.0 shared memory programming**

- Other: PGAS (UPC, Coarray Fortran, ….) / together with MPI

MPI

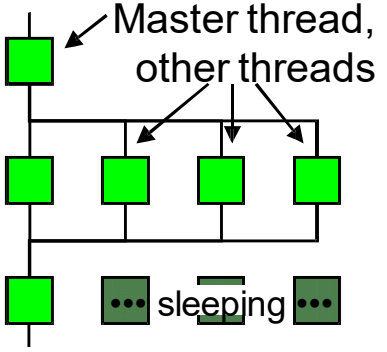
Sequential program on each core



Explicit **M**essage **P**assing by calling MPI_Send & MPI_Recv

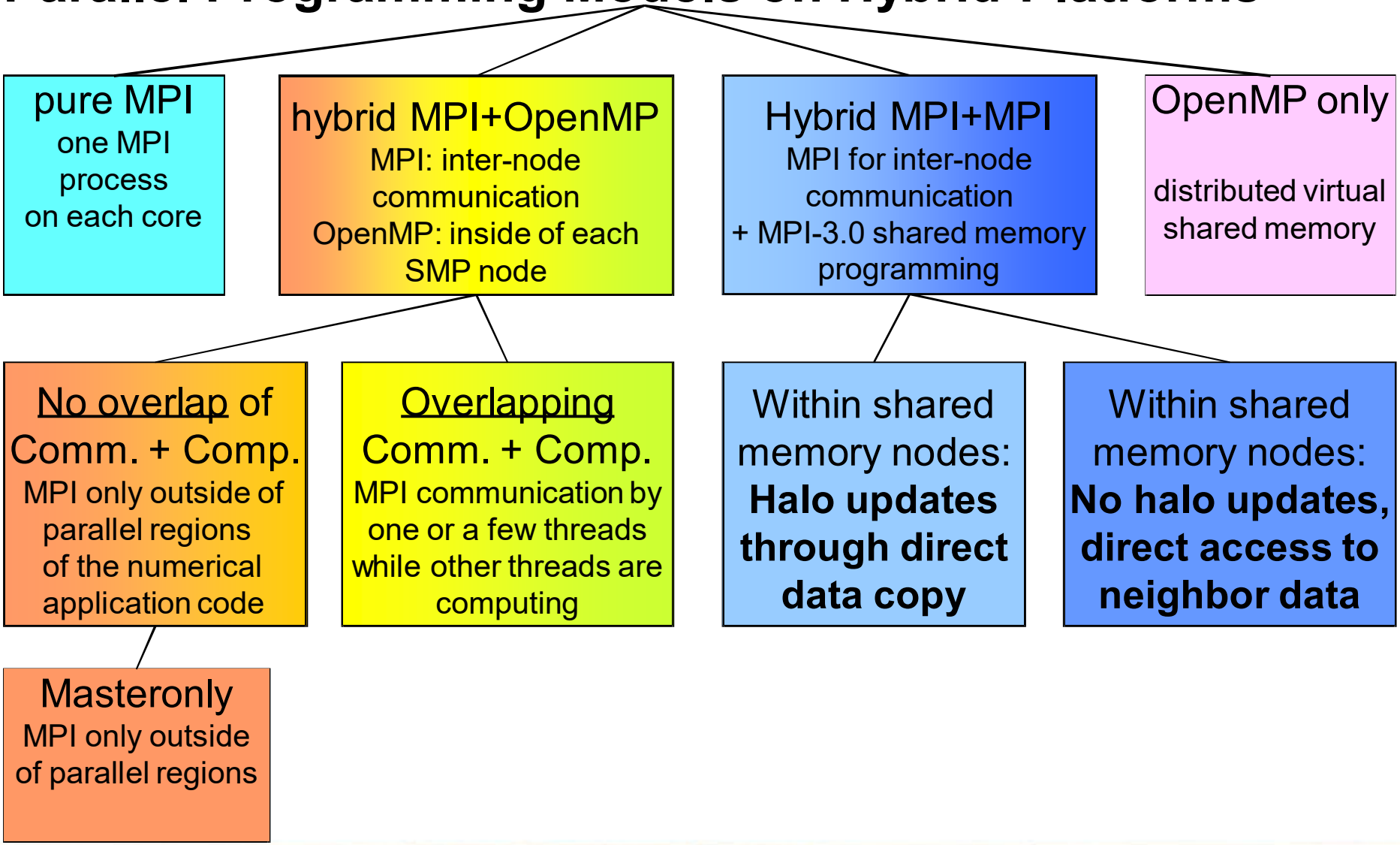
OpenMP (shared data)

```
some_serial_code
#pragma omp parallel for
for (j=…;…; j++)
        block_to_be_parallelized
again_some_serial_code
```



Parallel Programming Models

Parallel Programming Models on Hybrid Platforms



Pure MPI

pure MPI
one MPI process on each core

Advantages

- No modifications on existing MPI codes
- MPI library need not to support multiple threads

Major problems

- Does MPI library uses internally different protocols?
  - **Shared memory inside of the SMP nodes**
  - **Network communication between the nodes**
- Does application topology fit on hardware topology?
- Unnecessary MPI-communication inside of SMP nodes!

Discussed in detail later on in the section **Mismatch Problems**

MPI-only: example

There are no threads in the system

- E.g., there are no OpenMP parallel regions

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```

mpirun -N 4 ./mpi_only

Hybrid MPI+OpenMP Master only Style

Masteronly
MPI only outside of parallel regions

Advantages

- No message passing inside of the SMP nodes
- No topology problem

```
for (iteration ….)
{
  #pragma omp parallel
     numerical code
  /*end omp parallel */

  /* on master thread only */
     MPI_Send (original data
     to halo areas
       in other SMP nodes)
     MPI_Recv (halo data
     from the neighbors)
} /*end for loop
```

Major Problems

- All other threads are sleeping while master thread communicates!
- inter-node bandwidth?
- MPI-lib must support at least MPI_THREAD_FUNNELED

Thread-safety quality of MPI libraries

MPI rules with OpenMP/ Automatic SMP-parallelization

- Special MPI-2 Init for multi-threaded MPI processes:

```
int MPI_Init_thread(   int * argc, char ** argv[],
                       int thread_level_required,
                       int * thead_level_provided);
int MPI_Query_thread(  int * thread_level_provided);
int MPI_Is_main_thread(int * flag);
```

- REQUIRED values (increasing order):
  - **MPI_THREAD_SINGLE:** **Only one thread will execute**
  - **THREAD_MASTERONLY:** **MPI processes may be multi-threaded,**
    (virtual value, **but only master thread will make MPI-calls**
    not part of the standard) **AND only while other threads are sleeping**
  - **MPI_THREAD_FUNNELED:** **Only master thread will make MPI-calls**
  - **MPI_THREAD_SERIALIZED:** **Multiple threads may make MPI-calls,**
    but only one at a time
  - **MPI_THREAD_MULTIPLE:** **Multiple threads may call MPI,**

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

hybrid-hello

```
#include <stdio.h>
#include <mpi.h>
#include <omp.h>
int main(int argc, char *argv[])
{
   int my_id, omp_rank;
   int provided, required=MPI_THREAD_FUNNELED;
   MPI_Init_thread(&argc, &argv, required, &provided);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);

 #pragma omp parallel private(omp_rank)
   {
      omp_rank = omp_get_thread_num();
      printf("I'm thread %d in process %d\n", omp_rank, my_id);
   }
   if (my_id == 0) {
      printf("\nProvided thread support level: %d\n", provided);
      printf("  %d - MPI_THREAD_SINGLE\n", MPI_THREAD_SINGLE);
      printf("  %d - MPI_THREAD_FUNNELED\n", MPI_THREAD_FUNNELED);
      printf("  %d - MPI_THREAD_SERIALIZED\n", MPI_THREAD_SERIALIZED);
      printf("  %d - MPI_THREAD_MULTIPLE\n", MPI_THREAD_MULTIPLE);
   }
   MPI_Finalize();
   return 0;
}
```

multi-threaded processes

Create threads with openmp

```
I'm thread 0 in process 0

Provided thread support level: 1
  0 - MPI_THREAD_SINGLE
  1 - MPI_THREAD_FUNNELED
  2 - MPI_THREAD_SERIALIZED
  3 - MPI_THREAD_MULTIPLE
I'm thread 0 in process 1
```

MPI_THREAD_FUNNELED

- All MPI calls are made by the master thread
  - Outside the OpenMP parallel regions
  - In OpenMP master regions

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_FUNNELED, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
    for (i = 0; i < 100; i++)
        compute(buf[i]);

    /* Do MPI stuff */

    MPI_Finalize();

    return 0;
}
```

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

MPI_THREAD_SERIALIZED

- Only one thread can make MPI calls at a time
  - ♦ Protected by OpenMP critical regions

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_SERIALIZED, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
#pragma omp critical
        /* Do MPI stuff */
    }

    MPI_Finalize();

    return 0;
}
```

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

MPI_THREAD_MULTIPLE

- Any thread can make MPI calls any time (restrictions apply)

```
int main(int argc, char ** argv)
{
    int buf[100], provided;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

#pragma omp parallel for
    for (i = 0; i < 100; i++) {
        compute(buf[i]);
        /* Do MPI stuff */
    }

    MPI_Finalize();

    return 0;
}
```

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.

Calling MPI inside of OMP MASTER

- Inside of a parallel region, with "**OMP MASTER**"
- Requires MPI_THREAD_FUNNELED,
  i.e., only master thread will make MPI-calls
- **Caution:** There isn't any synchronization with "OMP MASTER"!
  Therefore, "**OMP BARRIER**" normally necessary to guarantee, that data or buffer space from/for other threads is available before/after the MPI call!

```
#pragma omp barrier
#pragma omp master
        MPI_Xxx(...);

#pragma omp barrier
```

- But this implies that all other threads are sleeping!
- The additional barrier implies also the necessary cache flush!

… the barrier is necessary–example with MPI_Recv

```
#pragma omp parallel
{
      #pragma omp for nowait
      for (i=0; i<1000; i++)
                  a[i] = buf[i];

     #pragma omp barrier
     #pragma omp master
                MPI_Recv(buf,...);
     #pragma omp barrier

     #pragma omp for nowait
           for (i=0; i<1000; i++)
                c[i] = buf[i];

}
```

No barrier inside

Barriers needed to prevent data races

Example: Thread support within Open MPI

- In order to enable thread support in Open MPI, configure with:

```
configure --enable-mpi-threads
```

- This turns on:
  - Support for full `MPI_THREAD_MULTIPLE`
  - internal checks when run with threads (`--enable-debug`)

```
configure --enable-mpi-threads --enable-progress-threads
```

- This (additionally) turns on:
  - Progress threads to asynchronously transfer/receive data per network BTL.
- Additional Feature:
  - Compiling **with** debugging support, but **without** threads will check for recursive locking

Overlapping Communication and Computation

MPI communication by one or a few threads while other threads are computing

```
if (my_thread_rank < …) {
   MPI_Send/Recv….
     i.e., communicate all halo data
} else {
   Execute those parts of the application
     that do not need halo data
     (on non-communicating threads)
}

Execute those parts of the application
  that  need halo data
  (on all threads)
```

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

Hybrid MPI + MPI-3 shared memory

Hybrid MPI+MPI
MPI for inter-node communication
+ MPI-3.0 shared memory programming

Advantages

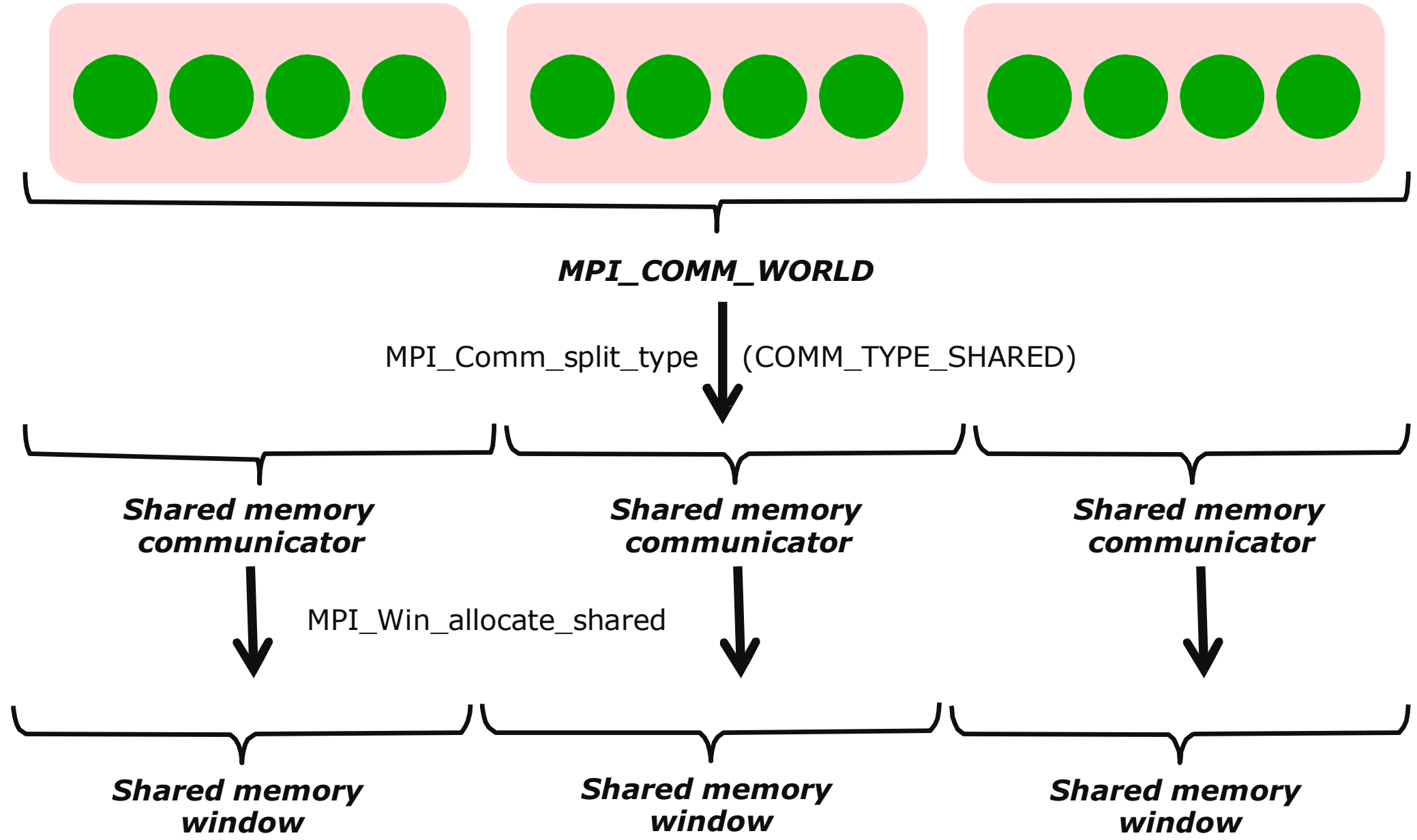
- No message passing inside of the SMP nodes
- Using only one parallel programming standard
- No OpenMP problems (e.g., thread-safety isn't an issue)

Major Problems

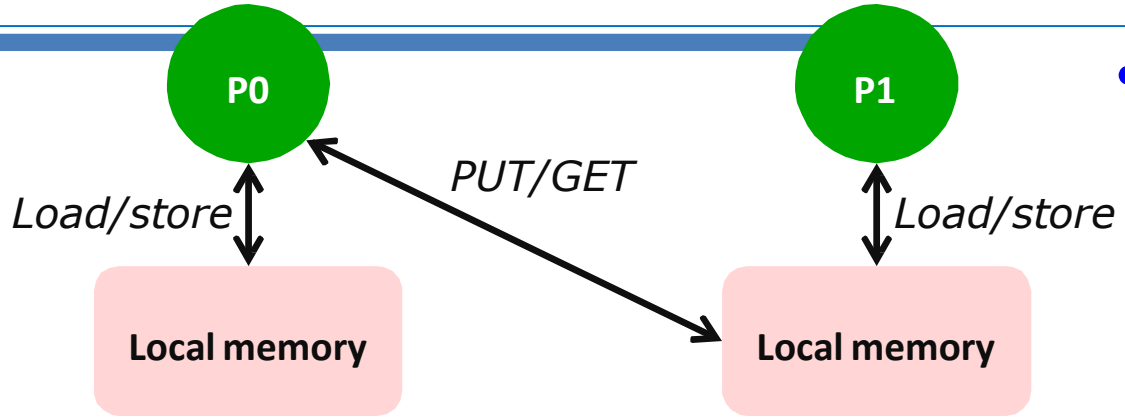
- Communicator must be split into shared memory islands
- To minimize shared memory communication overhead:
  Halos (or the data accessed by the neighbors) must be stored in
  MPI shared memory windows
- Same work-sharing as with pure MPI

- MPI-3 allows different processes to allocate shared memory through MPI
  - ♦ MPI_Win_allocate_shared
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads

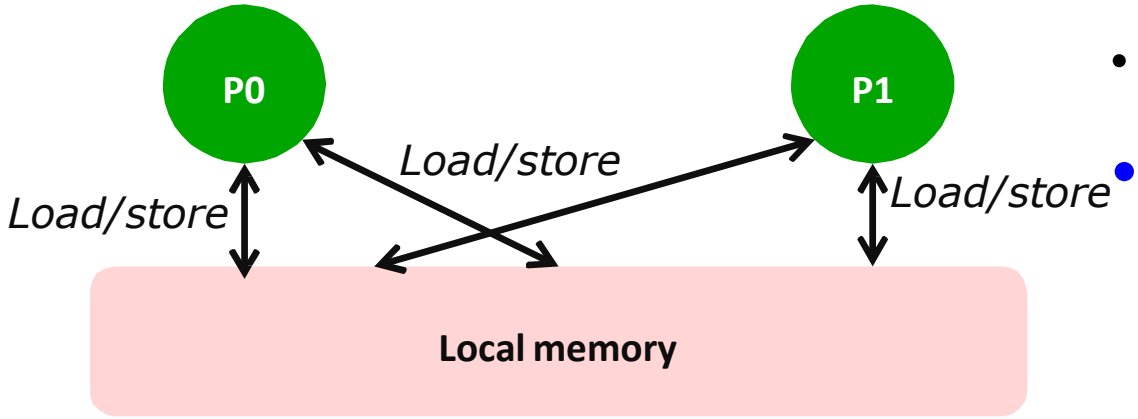
Creating Shared Memory Regions in MPI



Regular RMA windows vs. Shared memory windows



Traditional RMA windows



Shared memory windows

- Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
  - ♦ E.g., x[100] = 10
- All of the existing RMA functions can also be used on such memory for more advanced semantics such as atomic operations
- Can be very useful when processes want to use threads only to get access to all of the memory on the node

♦ You can create a shared memory window and put your shared data

Shared Arrays With Shared Memory Windows

```
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, .., &comm);
    MPI_Win_allocate_shared(comm, ..., &win);

    MPI_Win_lockall(win);

    /* copy data to local part of shared memory */
    MPI_Win_sync(win);

    /* use shared memory */

    MPI_Win_unlock_all(win);

    MPI_Win_free(&win);
    MPI_Finalize();
    return 0;
}
```

MPI_Get(…)
MPI_Put(…)

Pure OpenMP (on the cluster)

OpenMP only
distributed virtual shared memory

- Distributed shared virtual memory system needed
- Must support clusters of SMP nodes, e.g.,
  - Shared memory parallel inside of SMP nodes
  - Communication of modified parts of pages at OpenMP flush (part of each OpenMP barrier)

by rule of thumb:

Communication may be 10 times slower than with MPI

Why not Hybrid?

- OpenMP code performs worse than pure MPI code within node
  - all threads are idle except one while MPI communication
  - data placement, cache coherence
  - critical section for shared variables
- Possible waste of effort?

A Common Hybrid Approach

- From sequential code, parallel with MPI first, then try to add OpenMP.
- From MPI code, add OpenMP
- From OpenMP code, treat as serial code.
- Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks.
- Could use MPI inside parallel region with thread-safe MPI.

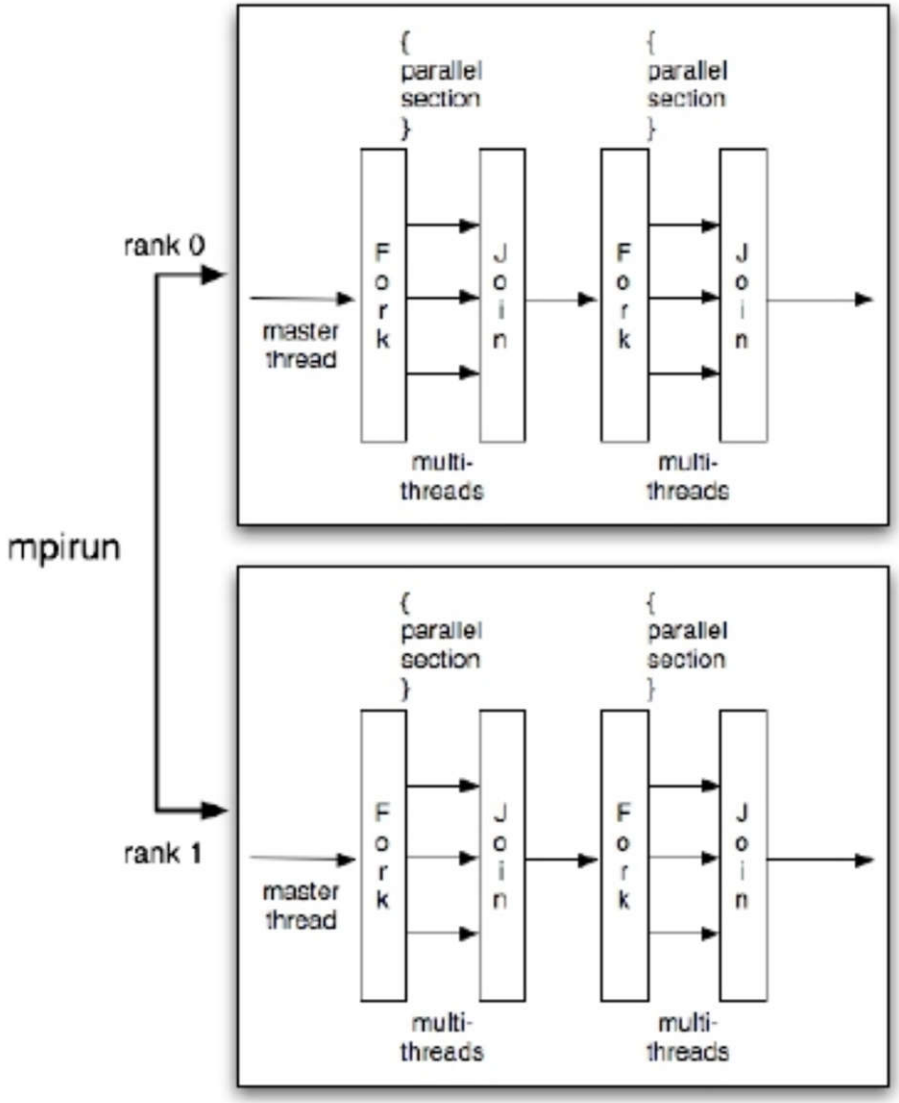
Hybrid – Program Model

- 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

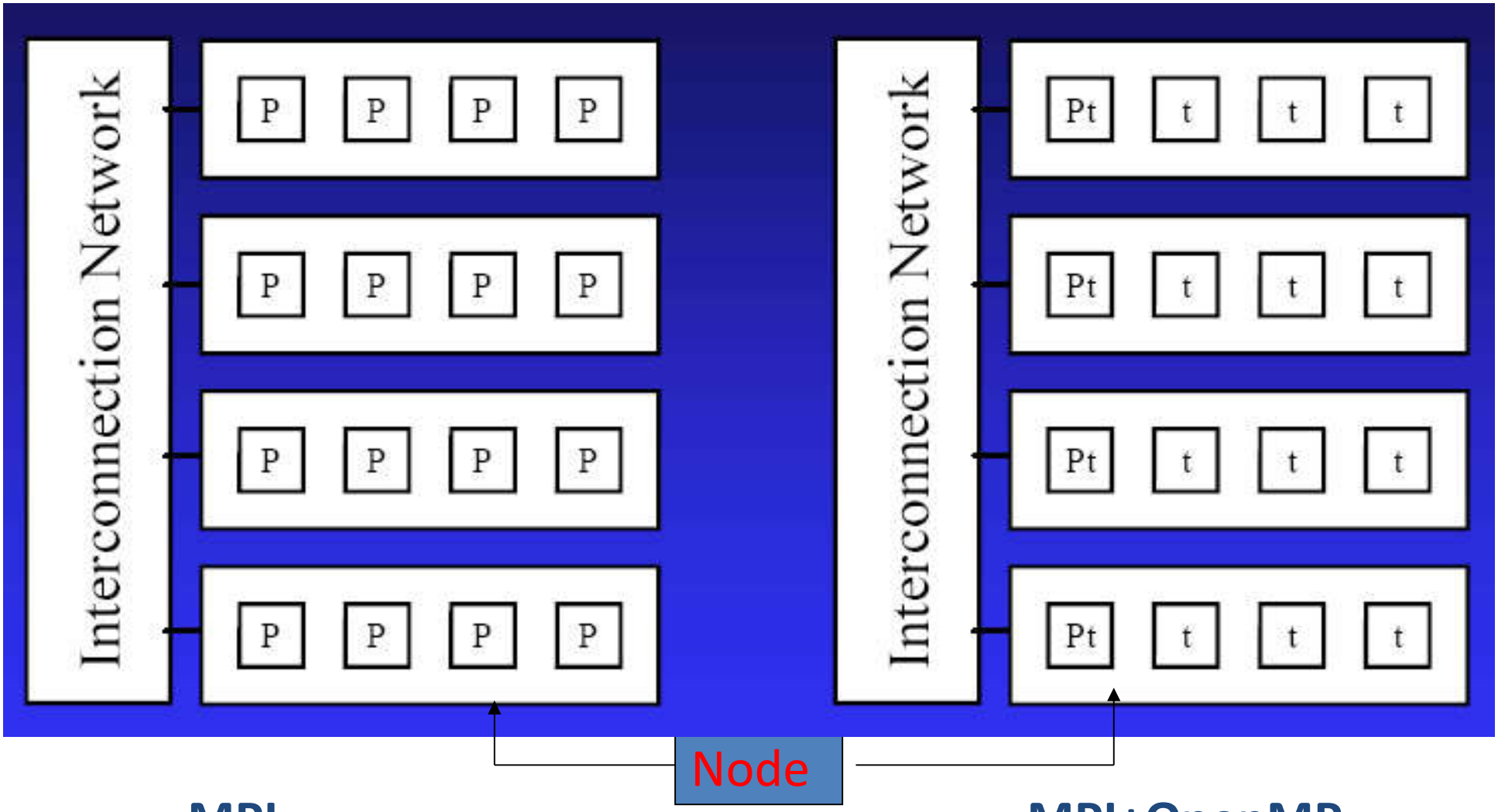
```
#include <stdio.h>
#include <mpi.h>
#include <omp.h>

int main(int argc, char *argv[])
{
   …
    /* Initialize MPI with thread support. */
     MPI_Init_thread(&argc, &argv,
            MPI_THREAD_MULTIPLE, &provided);
   /* some computation and MPI communication
*/
          // start OpenMP within node
       for i=1,n
         … computation
       endfor
    /* some computation and MPI communication
*/
     do communication
    MPI_Finalize();
    return 0;
}
```

Hybrid MPI+OpenMP Programming

Each MPI process spawns multiple OpenMP threads



MPI vs. MPI+OpenMP



16 cpus across 4 nodes
4 MPI processes per node

16 cpus across 4 nodes
1 MPI process and 4 threads per node

How to compile, link and run

- Use appropriate OpenMP compiler switch (-openmp, -fopenmp, -mp, -qsmp=openmp, …) and MPI compiler script (if available)
- Link with MPI library
  - Usually wrapped in MPI compiler script
  - If required, specify to link against thread-safe MPI library
    - **Often automatic when OpenMP or auto-parallelization is switched on**
- Running the code
  - Highly non-portable! Consult system docs! (if available…)
  - If you are on your own, consider the following points
  - Make sure OMP_NUM_THREADS etc. is available on all MPI processes
    - **Start "env VAR=VALUE … <YOUR BINARY>" instead of your binary alone**
  - Figure out how to start fewer MPI processes than cores on your nodes

Running code

Running the code

More examples (with mpiexec)

- Example for using mpiexec on a dual-socket quad-core cluster:

```
$ export OMP_NUM_THREADS=8
$ mpirun -n 4 ./a.out
```

- Same but 2 MPI processes per node:

```
$ export OMP_NUM_THREADS=4
$  mpirun -n 4./a.out
```

Where do the threads run?

- Pure MPI:

```
$ export OMP_NUM_THREADS=1 # or nothing if serial code
$  mpirun -n 1 ./a.out
```

Example:Mixing MPI and OpenMP

```
#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);

  #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();
}
```

Compiling and Linking Mixed MPI and OpenMP Programs

mpicc -fopenmp hello.c -o hello

Running Mixed Programs

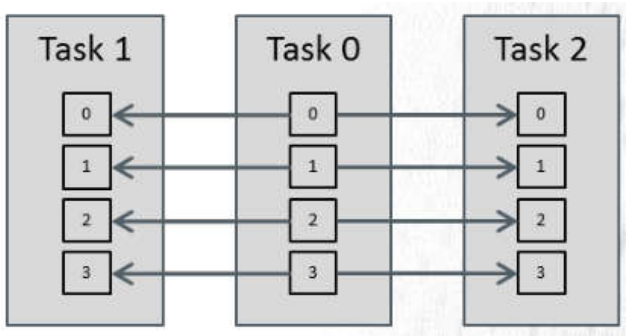
export OMP_NUM_THREADS=4

mpirun -np 2 ./hello

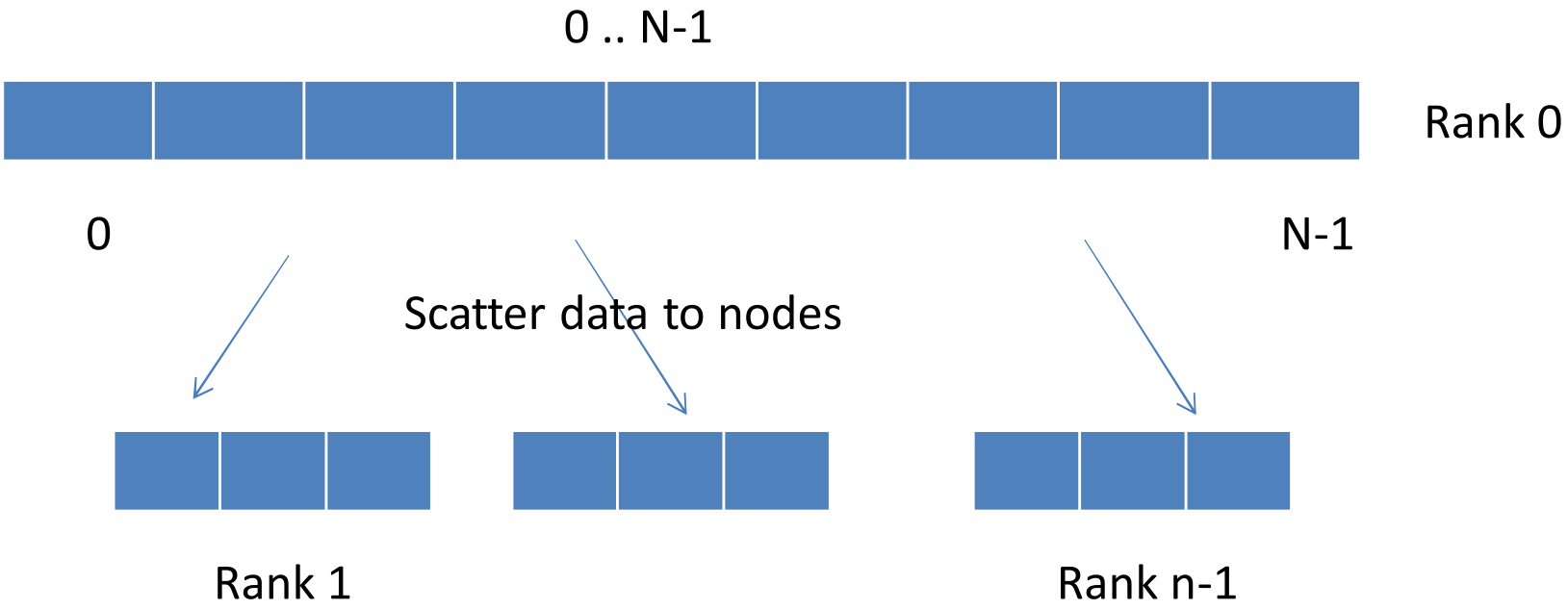
Multiple thread communication

```
…
int main(int argc, char *argv[])
{
    int provided, rank, ntasks;
    int tid, nthreads, msg, i;

    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

   #pragma omp parallel private(msg, tid, nthreads, i)
   {
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();

    if (rank == 0) {
       #pragma omp single
        {
            printf("%i threads in master rank\n", nthreads);
        }
       for (i = 1; i < ntasks; i++)
            MPI_Send(&tid, 1, MPI_INTEGER, i, tid, MPI_COMM_WORLD);
    } else {
        MPI_Recv(&msg, 1, MPI_INTEGER, 0, tid, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        printf("Rank %i thread %i received %i\n", rank, tid, msg);
    }
}
    MPI_Finalize();
    return 0;
}
```

- hybrid program where each OpenMP thread communicates using MPI.
- the threads of task 0 send their thread ID to the corresponding threads in other tasks (see the picture).



Example: sum numbers




```
sendcounts = new int[numprocs];
displs = new int[numprocs];
```

RANK =0

```
if (0 == myid)          {
            data = new int[n];
            int sum = 0;
            for (int i = 0; i < n; ++i)                     {
                        data[i] = rand() % 100;
                        sum += data[i];
            }
            sendcounts = new int[numprocs];
            displs = new int[numprocs];
            count = n / numprocs;
            remainder = n - count * numprocs;
            int prefixSum = 0;
            for (int i = 0; i < numprocs; ++i)                          {
                        sendcounts[i] = (i < remainder) ? count + 1 : count;
                        displs[i] = prefixSum;
                        prefixSum += sendcounts[i];
            }
}
```

Scatter the data to process

```
MPI_Scatterv(data, sendcounts, displs, MPI_INT, res, myBlockSize, MPI_INT, 0, MPI_COMM_WORLD);
```

For every process

```
int total = 0;
#pragma omp parallel for reduction(+:total)
        for (int i = 0; i < myBlockSize; ++i)
                total += res[i];
```

Reduce by root process

```
MPI_Reduce(&total, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
```

Example: sum numbers

```cpp
#include <mpi.h>
#include <iostream>

using namespace std;

int main(int argc, char **argv)
{
    int myid, numprocs, n, count, remainder, myBlockSize, result;
    int* data = NULL;
    int* sendcounts = NULL;
    int* displs = NULL;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    if (0 == myid)
    {
        cin >> n;
        data = new int[n];

        int sum = 0;
        for (int i = 0; i < n; ++i)
        {
            data[i] = rand() % 100;
            cout << data[i] << ' ';
            sum += data[i];
        }
        cout << endl;
        cout << "Exact sum: " << sum << endl;

        sendcounts = new int[numprocs];
        displs = new int[numprocs];

        count = n / numprocs;
        remainder = n - count * numprocs;
        int prefixSum = 0;
        for (int i = 0; i < numprocs; ++i)
        {
            sendcounts[i] = (i < remainder) ? count + 1 : count;
            displs[i] = prefixSum;
            prefixSum += sendcounts[i];
        }
    }
```

```
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    if (0 != myid)
    {
        count = n / numprocs;
        remainder = n - count * numprocs;
    }
    myBlockSize = myid < remainder ? count + 1 : count;

    int* res = new int[myBlockSize];

    MPI_Scatterv(data, sendcounts, displs, MPI_INT, res, myBlockSize, MPI_INT, 0, MPI_COMM_WORLD);

    if (0 == myid)
    {
        delete[] sendcounts;
        delete[] displs;
        delete[] data;
    }

    int total = 0;
#pragma omp parallel for reduction(+:total)
    for (int i = 0; i < myBlockSize; ++i)
        total += res[i];
    delete[] res;

    MPI_Reduce(&total, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Finalize();

    if (myid == 0)
    {
        cout << "Computed sum: " << result << endl;
#if _DEBUG
        system("pause");
#endif
    }
    return EXIT_SUCCESS;
}
```

Example: Calculating π

- **Numerical integration**

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

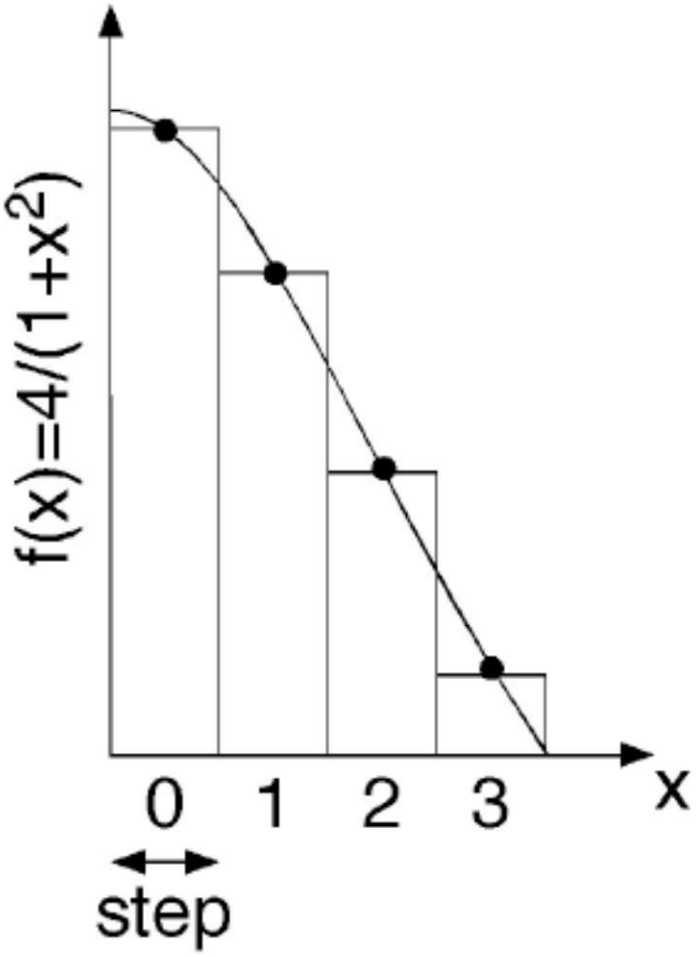
- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`

$x_i = (i+0.5)\Delta$ $(i = 0,\ldots,N\text{-}1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\Delta \cong \pi$$

```
#include <stdio.h>
#define NBIN 100000
void main() {
  int i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);}
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```



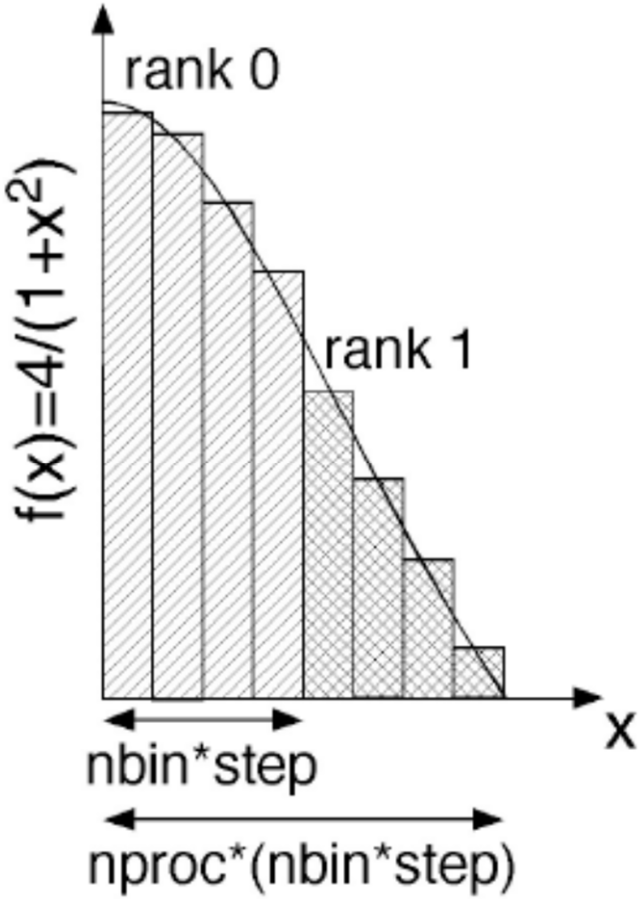
pi – MPI version

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>            /* MPI header file */
#define NUM_STEPS 100000000

int main(int argc, char *argv[]) {
   int nprocs;
   int myid;
   double start_time, end_time;
   int i;
   double x, pi;
   double sum = 0.0;
   double step = 1.0/(double) NUM_STEPS;

   /* initialize for MPI */
  MPI_Init(&argc, &argv);      /* starts MPI */
   /* get number of processes */
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
   /* get this process's number (ranges from 0 to nprocs - 1) */
   MPI_Comm_rank(MPI_COMM_WORLD, &myid);
```

```
   /* do computation */
   for (i=myid; i < NUM_STEPS; i += nprocs) {     /* changed */
      x = (i+0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
   }
   sum = step * sum;                          /* changed */
   MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);/* added */

/* print results */
   if (myid == 0) {
      printf("parallel program results with %d processes:\n", nprocs);
      printf("pi = %g  (%17.15f)\n",pi, pi);
   }

   /* clean up for MPI */
   MPI_Finalize();

   return 0;
}
```

MPI+OpenMP Calculation of π

- Each MPI process integrates over a range of width `1/nproc`, as a discrete sum of `nbin` bins each of width `step`
- Within each MPI process, `nthreads` OpenMP threads perform part of the sum as in `omp_pi.c`



MPI_OpenMP version

```
#define N 1000000000

int main( int argc, char* argv[] ){
   int rank, nproc;
   int i,low,up;
   double local = 0.0, pi, w, x;
   MPI_Status  status;
   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &nproc );
   MPI_Comm_rank( MPI_COMM_WORLD, &rank );
   w = 1.0/N;   //step
  low = rank*(N / nproc);
  up = low + N/nproc - 1;

   #pragma omp parallel for reduction(+:local) private(temp,i)
   for (i=low;i<up; i++)
  {
       x= (i+0.5)*w;
       local = local + 4.0/(1.0+x*x);
   }
   MPI_Reduce(&local, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,MPI_COMM_WORLD);
   if(rank==0)
     printf("pi = %.20f\n",pi*w);

   MPI_Finalize();
}
```

Compile and Run

- Compile (default gcc compilers on Compus Cluster)

```
mpicc -o pi-mpi pi-mpi.c
gcc -fopenmp -o pi-omp pi-omp.c
mpicc -fopenmp -o pi-hybrid pi-hybrid.c
```

- Run (qsub)

```
qsub -q mpi -n 8 --ppn=4 -r 10m -o pi-mpi.log -x OMP_NUM_THREADS=8
./pi-mpi
qsub -q threaded -n 8 -r 10m -o pi-omp.log ./pi-omp
qsub -q mpi -n 8 --ppn=1 --tpp=4 -r 10m -o pi-hybrid.log ./pi-hybrid
```

Results

- MPI

  MPI uses 8 processes:
  pi = 3.14159 (3.141592653589828)

- OpenMP

  OpenMP uses 8 threads:
  pi = 3.14159 (3.141592653589882)

- Hybrid

  mpi process 0 uses 4 threads
  mpi process 1 uses 4 threads
  mpi process 1 sum is 1.287 (1.287002217586605)
  mpi process 0 sum is 1.85459 (1.854590436003132)
  Total MPI processes are 2
  pi = 3.14159 (3.141592653589738)

Summary

- Computer systems in High-performance computing (HPC) feature a hierarchical hardware design (multi-core nodes connected via a network)
- OpenMP can take advantage of shared memory to reduce communication overhead
- Pure OpenMP performs better than pure MPI within node is a necessity to have hybrid code better than pure MPI across node.
- Whether the hybrid code performs better than MPI code depends on whether the communication advantage outcomes the thread overhead, etc. or not.
- There are more positive experiences of developing hybrid MPI/OpenMP parallel paradigms now. It's encouraging to adopt hybrid paradigm in your own application.

References

- http://openmp.org/sc13/HybridPP_Slides.pdf
- https://www.cct.lsu.edu/~estrabd/intro-hybrid-mpi-openmp.pdf
- http://www.cac.cornell.edu/education/Training/parallelMay2011/Hybrid_Talk-110524.pdf
- L. Smith and M. Bull, Development of mixed mode MPI / OpenMP applications, Scientific Programming, vol. 9, no. 2-3, pp. 83-98, 2001.