Exercises 11:
Hybrid Parallel Programming
MPI+OpenMP

Today

- Review OpenMP and MPI programming
- Create a hybrid MPI+OpenMP program

- Setup:
  - Download ex11.tar from Moodle, scp it to the cluster, and unpack the tar file

Inner Product Program

- Look at serial.c
- This program computes the inner product of two vectors of all 1s
- Takes two command line inputs: len (vector length) and trials (number of trials to perform for timing average)
- Uses OpenMP just for timing functions

- Compile and run:

```
gcc -fopenmp -o serial serial.c
./serial 10000000 1000
```

Task 1: OpenMP parallelization

- Open openmp.c
- Add the appropriate OpenMP pragma to parallelize the inner product computation loop

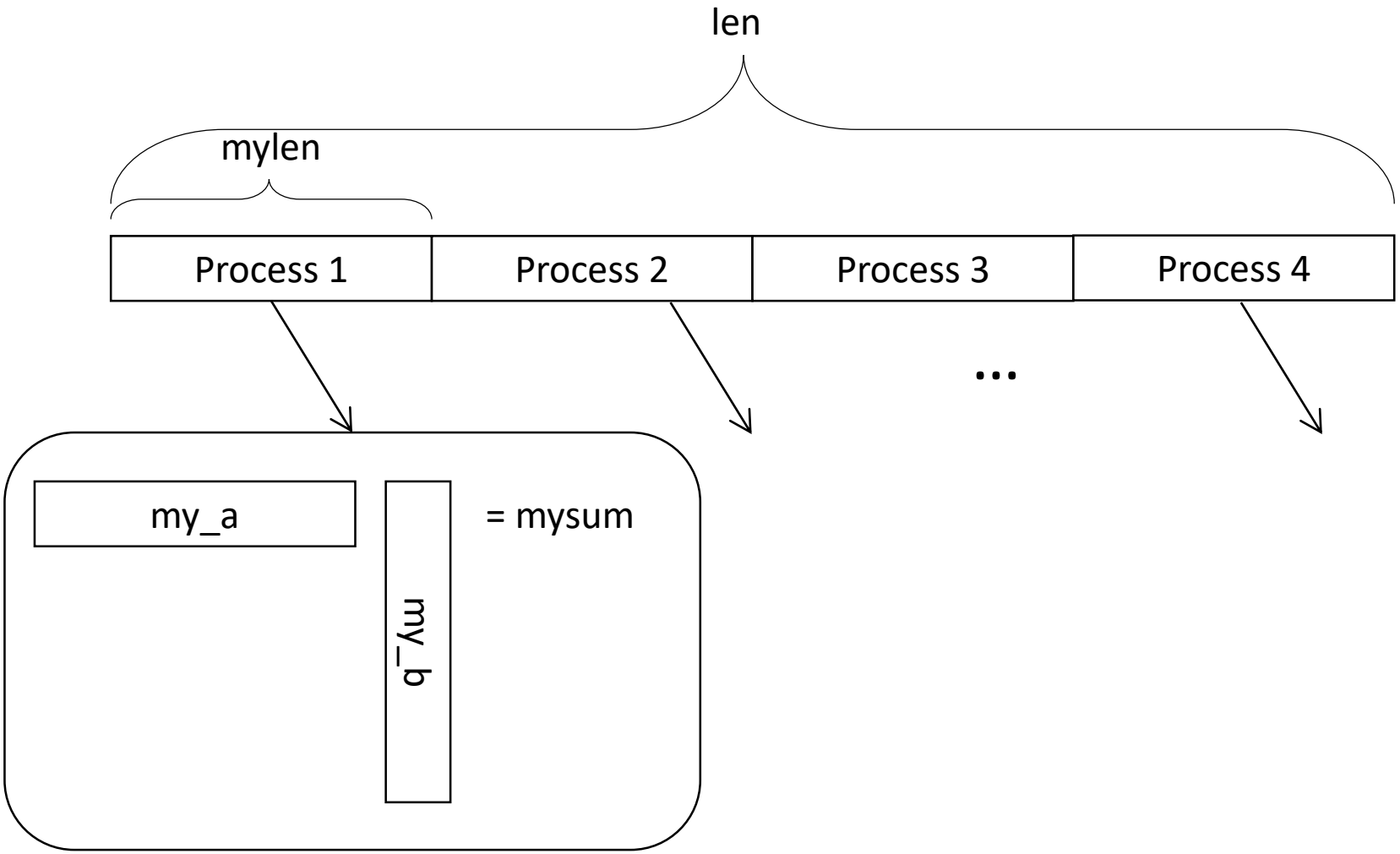
- Compile and run, e.g.,

```
gcc -fopenmp -o openmp openmp.c
export OMP_NUM_THREADS=4
./openmp 10000000 1000
```

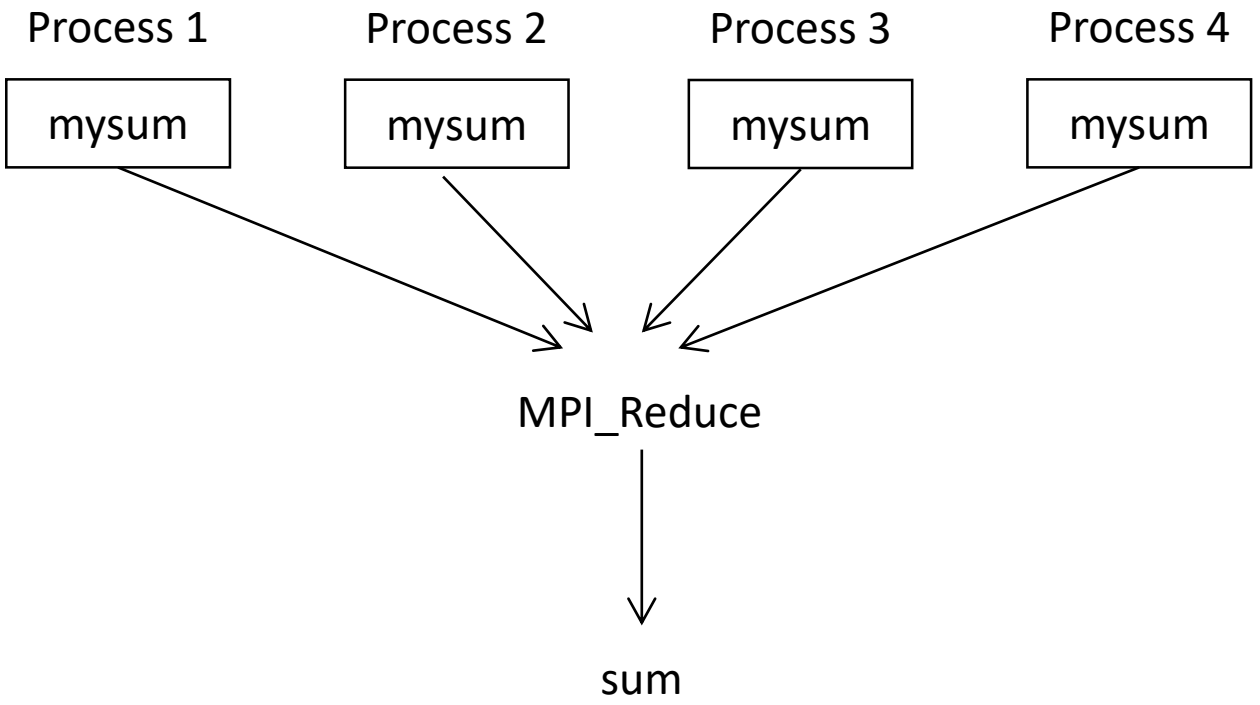
Task 2: MPI parallelization

- Open mpi.c

Task 2: MPI Parallelization



Task 2: MPI Parallelization



Task 2: MPI parallelization

- Open mpi.c

- Write the code to do the inner product computation
  - For-loop to do local computation, followed by collective to get the global sum

- Compile and run, e.g.,

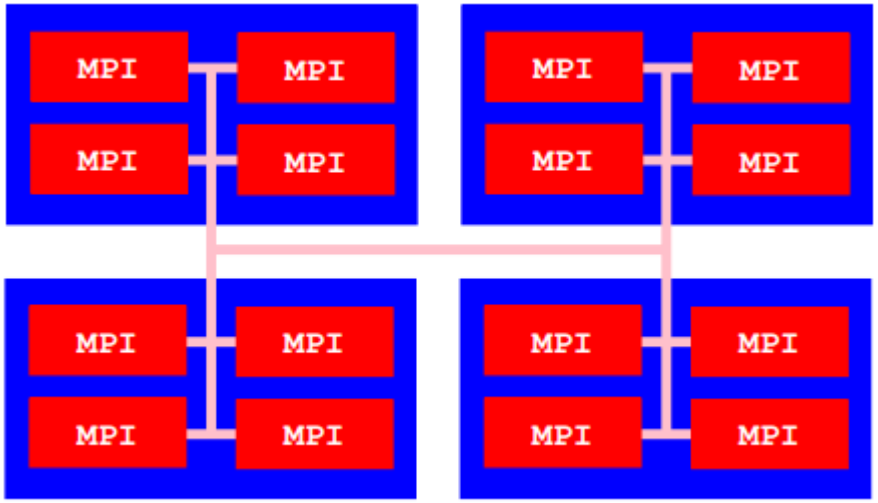
```
module load openmpi
mpicc -o mpi mpi.c
mpirun -n 4 ./mpi 10000000 1000
```

MPI_Reduce syntax

```
MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)
```

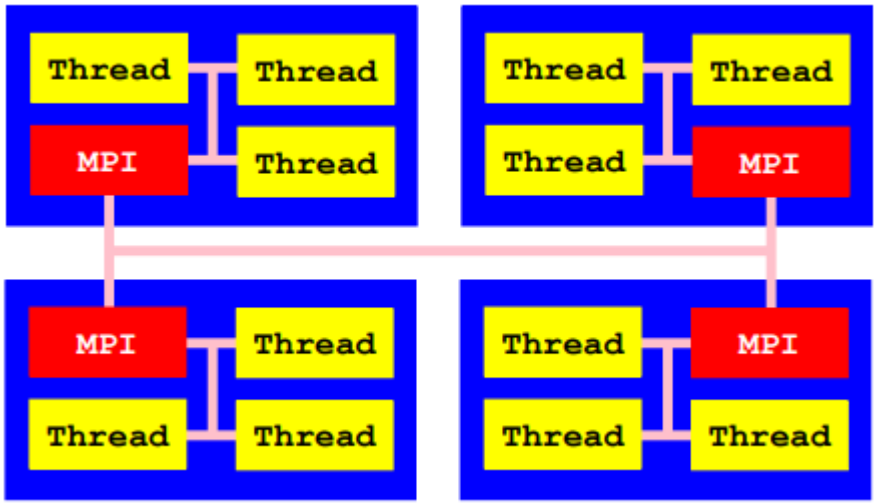
MPI on Clusters: "Flat MPI"

- On a cluster of multiprocessors (or multicore processors), we can execute a parallel program by having a MPI process executing in each processor (or core).
- It may be the case that multiple MPI processes execute in the same multiprocessor (or multicore processor), but still the interactions among those processes are based on message passing.



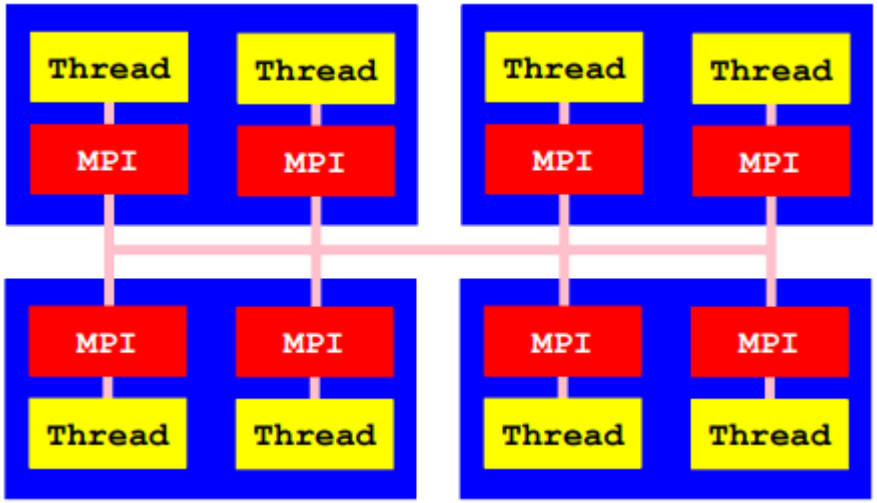
Hybrid Programming: MPI+OpenMP

- Different approach: hybrid program in which only one MPI process executes in each multiprocessor (or multicore processor)
- Launch a set of threads equal to the number of processors (or cores) in each machine to execute the parallel regions of the program.



Hybrid Programming: MPI+OpenMP

- Can combine both strategies and adapt the division between MPI processes and threads
  - Want to optimize the use of the available resources



Why Hybrid Programming?

- Architecture of parallel machines is overwhelmingly shared-memory multicore machines at the node level
  - Trends: more cores per CPU, less memory available per core

- Hybrid programming model is a natural fit

Why Hybrid Programming?

- Hybrid programming induces less communication among different nodes and increases performance of each node without having to increase memory requirements.
  - Using pure MPI can have large memory cost
    - Memory overhead from MPI itself
    - Pure MPI might require replicated data within a CPU/node
- Applications with two levels of parallelism may use MPI processes to exploit large grain parallelism, occasionally exchanging messages to synchronize information and/or share work, and use threads to exploit medium/small grain parallelism by resorting to a shared address space.
- Applications with constraints that may limit the number of MPI processes that can be used may take advantage of OpenMP to exploit the remaining computational resources.
- Applications for which load balancing is hard to achieve with only MPI processes may benefit from OpenMP to balance work, by assigning a different number of threads to each MPI process as a function of its load

Potential Advantages of Hybrid programming

- **Avoiding data replication**: Since threads can share data within a node, *if* any data needs to be replicated between processes, we can avoid this.
- **Light-weight** : Threads are lightweight and thus you reduce the meta-data associated with processes.
- **Reduction in number of messages** : A single process within a node can communicate with other processes, reducing number of messages between nodes (and thus reducing pressure on the Network Interface Card).
- **Faster communication** : Since threads communicate using shared memory, you can avoid using point-to-point MPI communication within a node.

- *Whether or not MPI+OpenMP will be faster (or use significantly less memory) than pure MPI depends heavily on the application.*

Hybrid Programming

- The simplest and safe way to combine MPI with OpenMP is to **never use the MPI calls inside the OpenMP parallel regions**. In this case, there is no problem with the MPI calls, given that only the master thread is active during all MPI communications

```
main(int argc, char **argv) {
  ...
  MPI_Init(&argc, &argv);
  ... // master thread only --> MPI calls here
  #pragma omp parallel
  {
    ... // team of threads --> no MPI calls here
  }
  ... // master thread only --> MPI calls here
  MPI_Finalize();
  ...
}
```

```
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
```

Task 3: Hybrid Parallelization

- We will just use the basic approach – no calls to MPI within OpenMP parallel regions

Task 3: Hybrid Parallelization

- We will just use the basic approach – no calls to MPI within OpenMP parallel regions

- Copy your completed `mpi.c` file to a new file called `hybrid.c`

```
cp mpi.c hybrid.c
```

- Open the file `hybrid.c`
  - Add OpenMP pragma to parallelize the local inner product computation
  - Modify the print statement at the end so that you print off the number of OpenMP threads in addition to the number of MPI tasks
    - Can use a "dummy" parallel region at the beginning to set this, as in `openmp.c`

Task 3: Hybrid Parallelization

- Compile and run, e.g.,

```
mpicc -fopenmp -o hybrid hybrid.c
export OMP_NUM_THREADS=4
mpirun -n 2 ./hybrid 10000000 1000
```

```
carson@r3d3:[~/HPC_W22/ex11/sols] mpirun -n 2 ./hybrid_sol 10000000 1000
Hybrid version: sum  =  10000000.000000, time=0.010988
MPI processes: 2, OpenMP threads: 4
```

Task 4: Try different setups

- Submit a job to the cluster using 2 nodes
  - Look at job.sh; feel free to modify
  - Compare different combinations of MPI processes and threads
    - E.g.,
      - 8 MPI processes, 1 thread per process
      - 4 MPI processes, 2 threads per process
      - 2 MPI processes, 4 threads per process
      - 1 MPI process, 8 threads per process

Results (On [r41, r42])

```
Serial version: sum  =  100000000.000000, time=0.265286

Hybrid version: sum  =  100000000.000000, time=0.063387
MPI processes: 8, OpenMP threads: 1

Hybrid version: sum  =  100000000.000000, time=0.060399
MPI processes: 4, OpenMP threads: 2

Hybrid version: sum  =  100000000.000000, time=0.060718
MPI processes: 2, OpenMP threads: 4

Hybrid version: sum  =  100000000.000000, time=0.062499
```

MPI Thread Safe

- If a program is parallelized so that it has MPI calls inside OpenMP parallel regions, then multiple threads can call the same MPI communications and at the same time. For this to be possible, it is **necessary that the MPI implementation be thread safe**.
  - Support for this since MPI-2

```
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  if (tid == id1)
    mpi_calls_f1(); // thread id1 makes MPI calls here
  else if (tid == id2)
    mpi_calls_f2(); // thread id2 makes MPI calls here
  else
    do_something();
}
```

MPI Support for Multithreading

```
int MPI_Init_thread(int *argc, char ***argv, int required,
  int *provided)
```

- `MPI_Init_thread()` initializes the MPI execution environment (similarly to `MPI_Init()`) and defines the support level for multithreading:
  - `required` is the aimed support level
  - `provided` is the support level provided by the MPI implementation
- The support level for multithreading can be:
  - `MPI_THREAD_SINGLE` – no multithreading
  - `MPI_THREAD_FUNNELED` – only the master thread can make MPI calls
  - `MPI_THREAD_SERIALIZED` – all threads can make MPI calls, but only one thread at a time can be in such state
  - `MPI_THREAD_MULTIPLE` – all threads can make simultaneous MPI calls without any constraints (MPI is thread safe)

MPI_THREAD_FUNNELED

- With support level MPI_THREAD_FUNNELED only the master thread can make MPI calls. One way to ensure this is to protect the MPI calls with the omp master directive.
- However, the omp master directive does not define any implicit synchronization barrier among all threads in the parallel region (at entrance or exit of the omp master directive) in order to protect the MPI call.

```
#pragma omp parallel
{
  ...
  #pragma omp barrier // explicit barrier at entrance
  #pragma omp master // only the master thread makes the MPI call
    mpi_call();
  #pragma omp barrier // explicit barrier at exit
  ...
}
```

MPI_THREAD_SERIALIZED

- With support level MPI_THREAD_SERIALIZED all threads can make MPI calls, but only one thread at a time can be in such state. One way to ensure this is to protect the MPI calls with the omp single directive that allows to define code blocks that should be executed only by one thread.
- However, the omp single directive does not define an implicit synchronization barrier at the entrance of the directive. In order to protect the MPI call, it is necessary to set an explicit omp barrier directive at entrance of the omp single directive.

```
#pragma omp parallel
{
  ...
  #pragma omp barrier // explicit barrier at entrance
  #pragma omp single // only one thread makes the MPI call
    mpi_call(); // explicit barrier at exit
  ...
}
```

MPI_THREAD_MULTIPLE

- With support level MPI_THREAD_MULTIPLE all threads can make simultaneous MPI calls without any constraints. Given that the implementation is thread safe, there is no need for any additional synchronization mechanism among the threads in the parallel region in order to protect the MPI calls.

```
#pragma omp parallel private(tid)
{
  tid = omp_get_thread_num();
  ...
  if (tid == id1)
    mpi_calls_f1(); // thread id1 makes MPI calls here
  else if (tid == id2)
    mpi_calls_f2(); // thread id2 makes MPI calls here
  else
    do_something();
  ...
}
```

MPI_THREAD_MULTIPLE

- The communication among threads of different MPI processes raises the problem of identifying the thread that is involved in the communication (as MPI communications only include arguments to identify the ranks of the MPI processes).
- A simple way to solve this problem is to use the tag argument to identify the thread involved in the communication.

```
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
...
#pragma omp parallel num_threads(NTHREADS) private(tid)
{
  tid = omp_get_thread_num();
  ...
  if (my_rank == 0) // MPI process 0 sends NTHREADS messages
    MPI_Send(a, 1, MPI_INT, 1, tid, MPI_COMM_WORLD);
  else if (my_rank == 1) // MPI process 1 receives NTHREADS messages
    MPI_Recv(b, 1, MPI_INT, 0, tid, MPI_COMM_WORLD, &status);
  ...
}
```

MPI Shared Memory

- Since MPI-3, MPI includes the "MPI shared memory model", which allows shared memory programming
- An alternative to MPI + OpenMP:
  - Distributed/shared memory programming using MPI + MPI

- See, e.g., https://software.intel.com/content/dam/develop/external/us/en/documents/an-introduction-to-mpi-3-597891.pdf