CME 213

Spring 2017

Eric Darve
MVAPICH library

- `mvapich2/2.3a-intel-16` on Certainty currently has a bug that prevents `mpirun` from working on more than 24 processes.
- Use `mvapich2/2.2b-intel-16` instead to run on more than 24 processes.
- However, only `mvapich2/2.3a` has CUDA support. `nvprof` needs CUDA support.
- You can also use `mpirun_rsh` with `mvapich2/2.3a`. 
Process mapping

- It is important to control the binding and mapping of the processes.
- How do you map processes to nodes?
- Binding: which hardware thread runs a process is controlled by the OS. Should a process be run by?
  - a specific hardware thread
  - any hardware thread on a core
  - any hardware thread on a socket
  - any hardware thread on a NUMA domain
- Mapping: a thread is assigned to which resource on a node, e.g., which socket?
Why mapping? Optimization!

- Mapping of processes is important.
- You may want to use many nodes to use more memory.
- You may want to have all threads on the same socket for faster shared memory access.
- You may want to have threads on different sockets to better utilize the memory buses.
- You may want to have as many processes per node as the number of GPU processors.
- You may want to use OpenMP with your MPI code.
Number of processes and nodes to use

`#PBS -l nodes=4:ppn=24`

Reserving 4 nodes, each with 24 hardware threads.

`mpiexec -np 96 -ppn 24 ./mpi_hello`

Create 96 processes, with 24 processes per node.

`mpiexec -np 16 -ppn 4 ./mpi_hello`

Create 16 processes, with 4 processes per node.
Process binding

- OS is responsible for assigning a hardware thread to each process.
- How do you control this mapping?

(bind-to object)

- Specify the size of the hardware element to restrict each process to.

- This determines how the OS can migrate a process. Does the process stay with the same hardware thread or is it allowed to migrate to another thread (say on the same socket)?
-bind-to

Options are [mpiexec bind-to --help]
numa bind to numa domain
socket bind to socket
core bind to core
hwthread bind to hardware thread
l1cache bind to process on L1 cache domain
l2cache bind to process on L2 cache domain
l3cache bind to process on L3 cache domain
map-by

map-by object
Skip over object between bindings.
Options are the same as bind-to.

Usage example:

mpiexec -bind-to hwthread -map-by socket -np 4 -ppn 4 ./a.out

Add

    HYDRA_TOPO_DEBUG=1 mpiexec ...

for DEBUG information.
Map by socket

mpiexec --bind-to hwthread --map-by socket

process 0 binding:  1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
process 1 binding:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
process 2 binding:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
process 3 binding:  0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Socket 0       Socket 1
Map by core

mpiexec -bind-to hwthread -map-by core
process 0 binding: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
process 1 binding: 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
process 2 binding: 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
process 3 binding: 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Core 0  Core 1
Map by hardware thread

mpiexec -bind-to hwthread -map-by hwthread
process 0 binding: 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
process 1 binding: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
process 2 binding: 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
process 3 binding: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Core 0 Thread 0

Core 0 thread 1
Bind to socket

`mpiexec -bind-to socket -map-by socket`

- process 0 binding: 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
- process 1 binding: 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1
- process 2 binding: 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
- process 3 binding: 0 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1

Spread processes across sockets

`mpiexec -bind-to socket -map-by core`

- process 0 binding: 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
- process 1 binding: 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
- process 2 binding: 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
- process 3 binding: 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0

Bunch processes inside each socket
If you have a memory intensive application that requires 4 processes, would you place the 4 processes on the same socket or on different sockets?
Start the presentation to activate live content.

Do you see this message in presentation mode, install the add-in or get help at PolEv.com/app.

If you have 4 processes that require inter-process communication, would you place them on the same socket or different sockets?
You are running on a node that is linked with other processes (OS, MPI, etc), is it better to bind a process to a hardware thread? Or bind a process to a socket? Or not specify any binding?

**Start the presentation to activate live content**

If you see this message in presentation mode, install the add-in or get help at PollEv.com/app
You want to run a program with 2 processes that requires 25 GB of memory per process, on computer nodes with 24 hardware threads and 64 GB of memory. How many processes would you assign per node?
MPI
Point-to-point communication
Point-to-point communication

• There are a few technical details to understand regarding communication.

• This is important to understand whether a deadlock may occur in your program or not.

• Two main concepts:
  • Blocking/non-blocking
  • Synchronous/asynchronous (not common in practice)
Two ways to communicate

1) using an MPI system buffer

To optimize the communication the MPI library uses two different strategies for communication: buffered and non-buffered.
2) Without a buffer

- Send()
- MPI: send data over network
- Main program thread continues

Main thread cannot make progress until the communication is completed

Possibly long wait time
What is the difference?

- Send and Recv are blocking operations:
  - The call does not return until the resources become available again
  - Send: data in buffer can be changed
  - Recv: data in buffer is available and can be used

- Send – If MPI uses a separate system buffer, the data in `smessage` (user buffer space) is copied (fast); then the main thread resumes.

- If MPI does not use a separate system buffer, the main thread must wait until the communication over the network is complete.

- Recv – If communication happens before the call, the data is stored in an MPI system buffer, and then simply copied into the user provided `rmessage` when `recv()` is called.

- The user cannot decide whether a buffer is used or not; the MPI library makes that decision based on the resources available and other factors.
With MPI buffer

With MPI buffer

Core 0

Core 1

Core 0

Core 1

No buffer is used; the thread needs to wait for the recv()

send()

recv()

Copy to buffer and resume

recv()
Deadlocks
## Deadlocks

- Because we use blocking routines, deadlocks can occur:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Deadlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recv()</td>
<td>Recv()</td>
<td>Always</td>
</tr>
<tr>
<td>Send()</td>
<td>Send()</td>
<td></td>
</tr>
<tr>
<td>Send()</td>
<td>Recv()</td>
<td>Depends on whether a buffer is used or not</td>
</tr>
<tr>
<td>Recv()</td>
<td>Recv()</td>
<td>Secure</td>
</tr>
<tr>
<td>Recv()</td>
<td>Send()</td>
<td></td>
</tr>
</tbody>
</table>

- See MPI codes in `mpi_deadlock/` and diagram on next slide.

- Secure implementation: code is guaranteed to never deadlock; this should be true independent of whether buffers are used or not.
Ring communication
Code with deadlock

```c
int rank_sender = rank == 0 ? world_size - 1 : rank - 1;
MPI_Recv(&number_recv, 1, MPI_INT, rank_sender,
          0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("Process %d received \t %2d from process %d\n", rank,
       number_recv, rank_sender);

int rank_receiver = rank == world_size - 1 ? 0 : rank + 1;
MPI_Send(&number_send, 1, MPI_INT, rank_receiver,
         0, MPI_COMM_WORLD);
printf("Process %d sent \t\t %2d to process %d\n", rank,
       number_send, rank_receiver);
```
“Non-secure” code; can potentially deadlock

```
// Receive from the lower process and send to the higher process.
int rank_receiver = rank == world_size - 1 ? 0 : rank + 1;
MPI_Send(&number_send, 1, MPI_INT, rank_receiver, 0, MPI_COMM_WORLD);
printf("Process %d sent \t\t %d to    process %d\n", rank,
       number_send, rank_receiver);

int rank_sender = rank == 0 ? world_size - 1 : rank - 1;
MPI_Recv(&number_recv, 1, MPI_INT, rank_sender, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
printf("Process %d received \t %d from process %d\n", rank,
       number_recv, rank_sender);
```
if(rank % 2 == 0) {
    int rank_receiver = rank==world_size-1 ? 0 : rank + 1;
    MPI_Send(&number_send, 1, MPI_INT, rank_receiver, 0, MPI_COMM_WORLD);
} else {
    int rank_sender = rank==0 ? world_size-1 : rank-1;
    MPI_Recv(&number_recv, 1, MPI_INT, rank_sender, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}

if(rank % 2 == 1) {
    int rank_receiver = rank==world_size-1 ? 0 : rank + 1;
    MPI_Send(&number_send, 1, MPI_INT, rank_receiver, 0, MPI_COMM_WORLD);
} else {
    int rank_sender = rank==0 ? world_size-1 : rank-1;
    MPI_Recv(&number_recv, 1, MPI_INT, rank_sender, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
Variant using `MPI_Sendrecv`

```c
int rank_receiver = rank == world_size - 1 ? 0 : rank + 1;
int rank_sender = rank == 0 ? world_size - 1 : rank - 1;
MPI_Sendrecv(&number_send, 1, MPI_INT, rank_receiver, 0,
                 &number_recv, 1, MPI_INT, rank_sender, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```
MPI_Sendrecv

MPI_Sendrecv(void *sendbuf, int sendcount,
    MPI_Datatype sendtype,
    int dest, int sendtag,
    void *recvbuf, int recvcount,
    MPI_Datatype recvtype,
    int source, int recvtag,
    MPI_Comm comm, MPI_Status *status)
A pair of blocking Send/Recv will synchronize the two processes.
where every process first sends receives will always deadlock
transfer never starts before MPI
NON-BLOCKING COMMUNICATIONS
Blocking vs non-blocking

Blocking = you process one communication/task at a time.

Non blocking: communicate while doing something else. Regularly check whether comm has completed.
Blocking vs non-blocking

- Blocking communications are convenient:
  - Simple to use.
  - Issue command; once code returns, you know that the task is done (at least the resource is usable).
  - Efficient.

- However, this is too restrictive.
  - When communications are happening, you probably want to do something else, such as do some useful computations or issue other communications. This is called overlapping communications and computations.
  - When you are waiting for several communications to complete, you may be able to do something as soon as one of them completes.

- Non-blocking communications can be used for that purpose:
  - Non-blocking routines return almost immediately.
  - Test routines are used to check the status of the communication.
Finite-element analysis

- **2D rectangular domain**
- **General domain**

- MPI communication using `Send` and `Recv`. If buffers are used the program will run correctly. Otherwise, deadlocks are possible.
- It’s easier to launch all the communications in a non-blocking manner, and then test to check whether they have completed or not.
Gather ring using non-blocking communications

See MPI code: `gather_ring.C`
// Vector to store the status of the non-blocking sends
vector<MPI_Request> send_req(nproc-1);
for(int i=0; i<nproc-1; ++i) {
    // Send to the right: Isend
    int* p_send = &numbers[(rank - i + nproc) % nproc];
    MPI_Isend(p_send, 1, MPI_INT, rank_receiver, 0, MPI_COMM_WORLD, &send_req[i]); // We can proceed; no need to wait now.
    // Receive from the left: Recv
    int* p_recv = &numbers[(rank - i - 1 + nproc) % nproc];
    MPI_Recv(p_recv, 1, MPI_INT, rank_sender, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    // We need to wait; we cannot move forward until we have that data.
}
Non-blocking versions of send and recv

Replace: MPI_send → MPI_Isend

```c
int MPI_Isend(void* buf, int count,
               MPI_Datatype datatype,
               int dest, int tag,
               MPI_Comm comm, MPI_Request *request)
```

`MPI_Request`* use to get information later on about the status of that operation.

What does I stand for?
Immediate
Testing and waiting

There is a similar non-blocking receive:

```c
int MPI_Irecv(void* buf, int count,
               MPI_Datatype datatype,
               int source, int tag,
               MPI_Comm comm, MPI_Request *request)
```

Test the status of the request using:

```c
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status)
```

Flag is 1 if request has been completed, 0 otherwise.

Wait until request completes:

```c
int MPI_Wait(MPI_Request *request, MPI_Status *status)
```
Is it possible to deadlock a code with blocking communications?
Application: bucket and sample sort

• Bucket sort is a simpler parallel algorithm.
• Assume we have a sequence of integers in the interval \([a,b]\].
• Split \([a,b]\) into \(p\) sub-intervals.
• Move each element to the appropriate bucket (prefix sum required again).
• Sort each bucket in parallel.
• Simple and efficient!
• A variant of this is the radix sort.
• Problem: how should we split the interval? This process may lead to intervals that are unevenly filled.
• **Improved version: sample (or splitter) sort.**
Videos of sorting algorithms

Radix sort
https://www.youtube.com/watch?v=LyRWppObda4

15 sorting algorithms
https://www.youtube.com/watch?v=kPRA0W1kECg
Shared memory: prefix sum required for the final bucket assignment.
Distributed memory

**MPI_Allgather()**

**MPI_Alltoall() and MPI_Alltoallv()**

Initial element distribution

Local sort & sample selection

Sample combining

Global splitter selection

Final element assignment
Main challenges

- Load-balancing: the sub-sequences are not always of equal sizes. Depending on the distribution of data, we can get load-imbalance of up to 2x.
- The **AllToAll** communication is a bit complicated because the number of data per process is not the same. Hence, we need **MPI_Alltoallv**.
- This requires an **MPI_Alltoall** to first gather the size of the data to be sent.
- Similarly, the test section at the end requires **MPI_Gatherv**.

\[
\text{Efficiency} = \frac{T_{seq}}{n \cdot T_{par}}
\]
Scalability

1 process
   Efficiency: 0.591695; runtime seq: 4.51704, par: 7.63407

2 processes
   Efficiency: 0.587918; runtime seq: 4.47443, par: 3.80532

6 processes
   Efficiency: 0.627524; runtime seq: 4.49705, par: 1.19439

12 processes
   Efficiency: 0.839512; runtime seq: 4.47041, par: 0.443751

24 processes
   Efficiency: 0.789635; runtime seq: 6.97855, par: 0.368237

48 processes
   Efficiency: 0.221131; runtime seq: 6.82009, par: 0.642538

The efficiency is excellent, but limited by the fact that we need to sort the vector essentially twice. This explains the slowdown with n=1. Notice that the efficiency increases after that. This is due to a more efficient use of the cache when using multiple cores.