Advanced Topics in Numerical Analysis: High Performance Computing

Distributed memory algorithms (MPI)

Georg Stadler, Dhairya Malhotra
Courant Institute, NYU

Spring 2019, Monday, 5:10–7:00PM, WWH #1302

April 22, 2019

Outline

Organization issues

Submitting jobs through a scheduler

Summary of previous class

MPI collectives

Organization

Scheduling:

- (Short) homework assignment #5 posted, due next week; you are asked to provide an update on your final project
- There will be one more (last) homework assignment

Topics today:

- Job schedulers: SLURM
- Collective communication in MPI and many examples

Outline

Organization issues

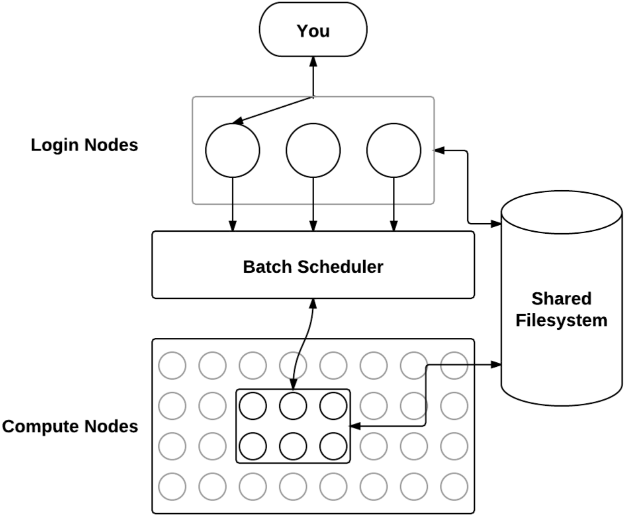
Submitting jobs through a scheduler

Summary of previous class

MPI collectives

Submitting jobs through a scheduler (e.g., on Prince)

Overview of HPC cluster



Submitting jobs on Prince

Prince user guide: https://wikis.nyu.edu/display/NYUHPC/Clusters+-+Prince

Batch facilities: SGE, LSF, SLURM. Prince uses SLURM, and these are some of the basic commands:

- submit/start a job: `sbatch jobscript`
- submit/start a job (interactive):
  `srun <options> --pty /bin/bash`
- see status of my job: `squeue -u USERNAME`
- cancel my job: `scancel JOBID`
- see all jobs on machine: `squeue | less`

Submitting jobs on Prince

Some basic rules:

- Don't run on the login node!
- Don't abuse the shared file system.

Submitting jobs on Prince

```
#!/bin/bash
#SBATCH --nodes=1               \# total number of mpi tasks
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --time=5:00:00
#SBATCH --mem=2GB
#SBATCH --job-name=myTest
#SBATCH --mail-type=END         \# email me when the job finishes
#SBATCH --mail-user=first.last@nyu.edu
#SBATCH --output=slurm_%j.out

module purge
module load ...
./myexecutable
```

Outline

More MPI

Last Class

- The distributed memory computing model
- Sources of parallelism
- Send and recv communication in MPI
- Communication costs (Postal model):
  - Latency + 1/bandwidth * message length

Parallelism and locality

- Moving data (through network or memory hierarchy) is slow
- Real world problems often have parallelism and locality, e.g.,
  - objects move independently from each other ("embarrassingly parallel")
  - objects mostly influence other objects nearby
  - dependence on distant objects can be simplified
  - Partial differential equations have locality properties
- Applications often exhibit parallelism at multiple levels

Parallelism and locality—examples

Examples from last class:

- Conway's game of life—parallelism through domain decomposition
- Particle systems (background forces, neighbor forces, far-field forces) — domain decomposition
- Sparse/dense matrix-vector multiplication–row-wise storage
- PDE solution (elliptic/hyperbolic/parabolic)

MPI Send/Recv Modes

MPI send modes:

- Standard Send (MPI_Send): return when the send array can be re-used.
- Buffered Send (MPI_Bsend): returns when message is copied to a secondary buffer (send array can be re-used). Buffering requires extra overhead and should be avoided.
- Synchronous Send (MPI_Ssend): returns when the message has been received by the receiving process (send array can be re-used).
- Non-blocking Send (MPI_Isend): returns immediately, the send array cannot be reused until MPI_Wait() returns.
- Other send modes

MPI recv modes:

- Standard Recv (MPI_Revc): return when the message has been received
- Standard Non-blocking Recv (MPI_Irevc): return immediately, the recv array cannot be used until MPI_Wait returns.

Deadlock-free send/recv patterns

Blocking send and recv

Process-0	**Process-1**
MPI_Sendrecv(...)	MPI_Sendrecv(...)
// $\cdots$ can use recv array now	// $\cdots$ can use recv array now

Blocking send, non-blocking recv

Process-0	**Process-1**
MPI_Irecv(... , recv_request)	MPI_Irecv(... , recv_request)
MPI_Send(...)	MPI_Send(...)
// $\cdots$ other code	// $\cdots$ other code
MPI_Wait(recv_request, ...)	MPI_Wait(recv_request, ...)
// $\cdots$ can use recv array now	// $\cdots$ can use recv array now

Deadlock-free send/recv patterns

Non-blocking send and recv

Process-0	**Process-1**
MPI_Irecv(... , recv_request)	MPI_Irecv(... , recv_request)
MPI_Isend(... , send_request)	MPI_Isend(... , send_request)
// ⋯ other code	// ⋯ other code
MPI_Wait(recv_request, ...)	MPI_Wait(recv_request, ...)
MPI_Wait(send_request, ...)	MPI_Wait(send_request, ...)
// ⋯ can use send/recv arrays	// ⋯ can use send/recv arrays

Outline

Organization issues

Submitting jobs through a scheduler

Summary of previous class

MPI collectives

MPI Collectives

- Calls that involve more than 2 processes (also called point-to-point)
- Could also be done with Sends and Recvs, but more efficient and concenient
- Can be one-to-all or all-to-all
- Every process needs to see the collective call to avoid hangs!
- Actual implementation depends on MPI library (and possibly on the network type)

Code examples: `https://github.com/NYU-HPC19/lecture11`

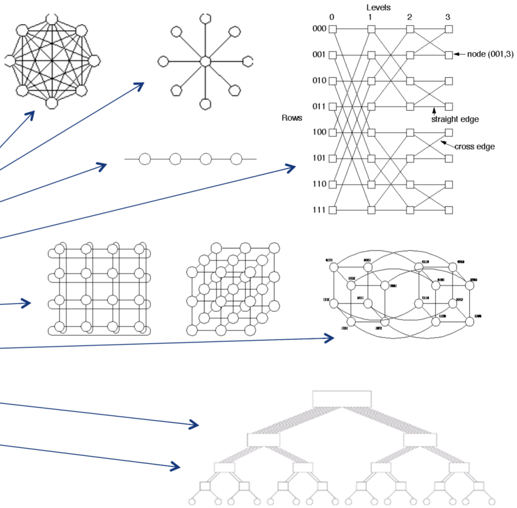
Tutorial: `http://mpitutorial.com/tutorials/`

Network types: topologies

Networks

Networks

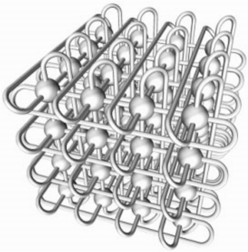
- Complete and star
- Arrays/rings
- Butterfly
- Mesh
- Hybercubes
- Trees
- Switches



Network types: metrics

- Diameter: maximum distance between any two nodes
- Connectivity: number of links needed to remove to isolate a node
- Bisection width: number of links to be removed to break network into equal parts
- Cost: Total number of links

Examples

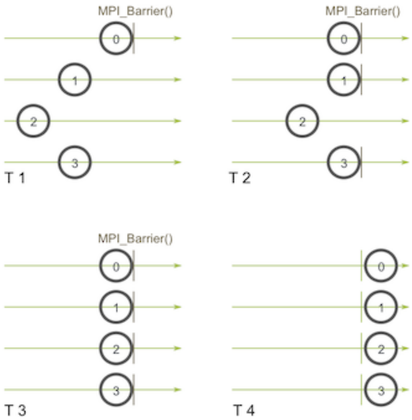


Network	Diameter	Bisection Width	Connectivity	Cost (No. of links)
Completely-connected	1	$p^2/4$	$p-1$	$p(p-1)/2$
Complete binary tree	$2\log((p+1)/2)$	1	1	$p-1$
Linear array	$p-1$	1	1	$p-1$
2-D mesh, no wraparound	$2(\sqrt{p}-1)$	$\sqrt{p}$	2	$2(p-\sqrt{p})$
2-D wraparound mesh	$2\lfloor\sqrt{p}/2\rfloor$	$2\sqrt{p}$	4	$2p$
Hypercube	$\log p$	$p/2$	$\log p$	$(p\log p)/2$
Wraparound k-ary d-cube	$d\lfloor k/2\rfloor$	$2k^{d-1}$	$2d$	dp

MPI Barrier

Synchronizes all processes. Other collective functions implicitly act as a synchronization. Used for instance for timing.

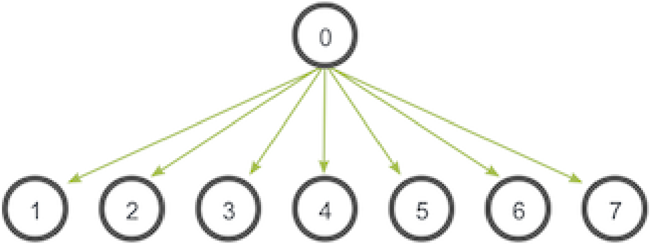
```
MPI_Barrier(MPI_Comm communicator)
```



MPI Broadcast

Broadcasts data from one to all processors. Every processor calls same function (although its effect is different).

`MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator)`

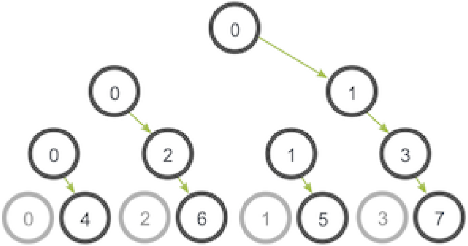


Actual implementation depends on MPI library.

MPI Broadcast

Broadcasts data from one to all processors. Every processor calls same function (although its effect is different).

```
MPI_Bcast(void* data, int count, MPI_Datatype
datatype, int root, MPI_Comm communicator)
```



Actual implementation depends on MPI library.

MPI Reduce

Reduces data from all to one processors. Every processor calls same function.

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

Possible Reduce operators:

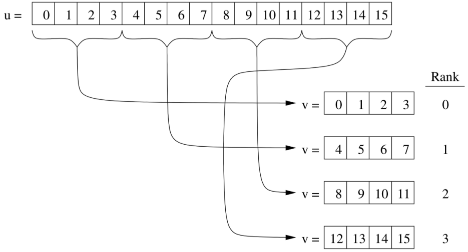
- MPI_MAX: Returns the maximum element.
- MPI_MIN: Returns the minimum element.
- MPI_SUM: Sums the elements.
- MPI_PROD: Multiplies all elements.
- MPI_LAND: Performs a logical and across the elements.
- MPI_LOR: Performs a logical or across the elements.
- MPI_BAND: Performs a bitwise and across the bits of the elements.
- MPI_BOR: Performs a bitwise or across the bits of the elements.
- MPI_MAXLOC: Returns the maximum value and the rank of the process that owns it.
- MPI_MINLOC: Returns the minimum value and the rank of the process that owns it.

`MPI_Allreduce()`: Provides result of reduction too all processors.

MPI Scatter

Broadcasts different data from one to all processors. Every processor calls same function.

```
MPI_Scatter(void* sendbuff, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm
communicator)
```



Send arguments must be provided on all processors, but sendbuf can be NULL. Send/recv count are per processor. Variable-sized variant is `MPI_Scatterv`.

MPI Gather

Gathers different data from all to one processors. Every processor calls same function. Gather is (more or less) the opposite of scatter.

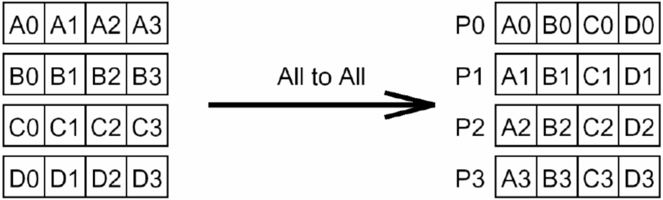
`MPI_Gather(void* sendbuff, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm communicator)`

`MPI_Allgather()` gathers from all processors to all processors. Variable-sized variant is `MPI_Gatherv`.

MPI All-to-all

Shares data from each to each processor.

```
MPI_Alltoall(void *sendbuf, int count, MPI_Datatype
sendtype, void *recvbuf, int recvcount, MPI_Datatype
recvtype, MPI_Comm comm)
```



Example: matrix-transpose or sorting. Variable-sized variant is called `MPI_Alltoallv`.