Advanced Topics in Numerical Analysis: High Performance Computing

MATH-GA 2012.001 & CSCI-GA 2945.001

Georg Stadler
Courant Institute, NYU
stadler@cims.nyu.edu

Spring 2017, Thursday, 5:10–7:00PM, WWH #512

March 30, 2017

Outline

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)

What should (not) be added to a repository?

Git tracks diff-files to keep its memory requirements small. Main rule: mostly add *source files that compile*.

- .c, .cpp, .f files YES!
- .tex files YES!
- .aux, .out, .dvi. . . files NO!
- compiled files, object files NO! (large, no diffs possible, conflicts)
- .pdf files YES/NO!
- large data files NO. . . sometimes maybe
- photos, movies etc. NO! (unless unavoidable)

My rule of thumb: Files in the repository are permanent, only the best should make it in there (it's not your trash can!) They should compile (code/Latex), be (more or less) cleaned up, unless it's avoidable only source/text files.

Some of my git wisedom

Should I have a few large repositories or many small ones?

- I recommend many small ones (like I use for this class).
- Easier to manage, commit messages easier to monitor.
- Small memory footprint and faster!
- It's easy to link two repositories (e.g., code libraries) using git submodules (look it up)!

Some of my git wisedom

Should I have a few large repositories or many small ones?

- I recommend many small ones (like I use for this class).
- Easier to manage, commit messages easier to monitor.
- Small memory footprint and faster!
- It's easy to link two repositories (e.g., code libraries) using git submodules (look it up)!

How often should you commit?

- As often as you like (in case of doubt, more often)
- Makes it easier to monitor changes, track down bugs
- If you collaborate, better to avoid conflicts
- For me: feels like a (small) achievement, supports clean/systematic working style (always look at diff before committing)

Graphical interface to git

Provided by bitbucket/github/gitlab. Locally, I use

```
$ gitk (--all)
```

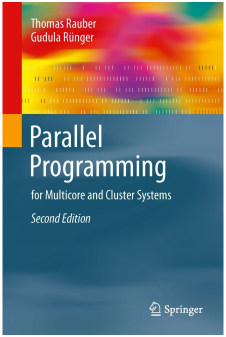
Outline

MPI Collectives

Recommended online resource:

http://mpitutorial.com/

Recommended reading: Chapter 5 in



Non-blocking MPI Send/Recv

- Non-blocking communication allows interlacing communication and computation.

  `MPI_ISend(..., MPI_Request *request)`

  `MPI_IRecv(..., MPI_Request *request))`

- Must check status to ensure that communication has finished.

  `MPI_Wait(MPI_Request *request, MPI_Status *status)`

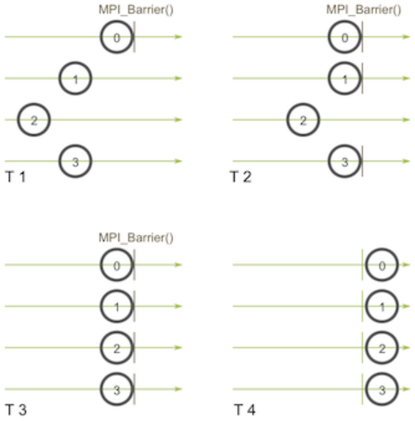
Comparison with mailing a letter:

- Blocking Send: drop off letter at the mail box (copied to MPI buffer)
- Nonblocking Send: letter on kitchen table is ready to be taken to the mail box (MPI starts taking care of message)
- Blocking Recv: Letter has arrived (it's in the desired memory location)
- Nonblocking Recv: I'm expecting a letter (keep checking till it arrives using `MPI_Wait()`)

MPI Barrier

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

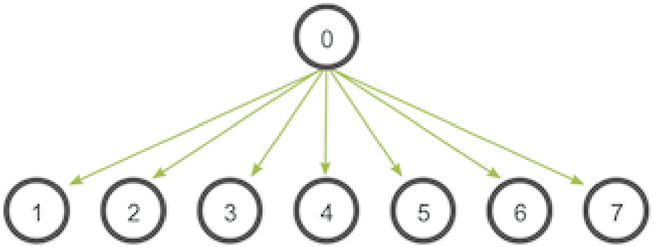
```
MPI_Barrier(MPI_Comm communicator)
```



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)
```

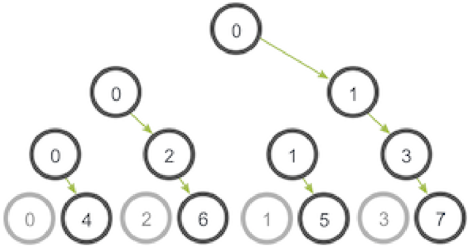


Actual implementation depends on MPI library.

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:

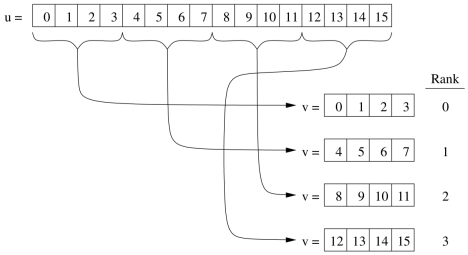
- MPI_MAX: Returns the maximum element.
- MPI_MIN: Returns the minimum element.
- MPI_SUM: Sums the elements.
- MPI_PROD: Multiplies all elements.
- MPI_LAND: Performs a logical and across the elements.
- MPI_LOR: Performs a logical or across the elements.
- MPI_BAND: Performs a bitwise and across the 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.

MPI Gather

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

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

Variant:

`MPI_Allgather()` gathers from all processors to all processors.

MPI_Bcast comparison

Let's compare a naive implementation of `MPI_Bcast` with the system implementation:

https://github.com/NYU-HPC17/lecture8

MPI_Bcast comparison

Let's compare a naive implementation of `MPI_Bcast` with the system implementation:

https://github.com/NYU-HPC17/lecture8

. . . and let's do it on Stampede!

Outline

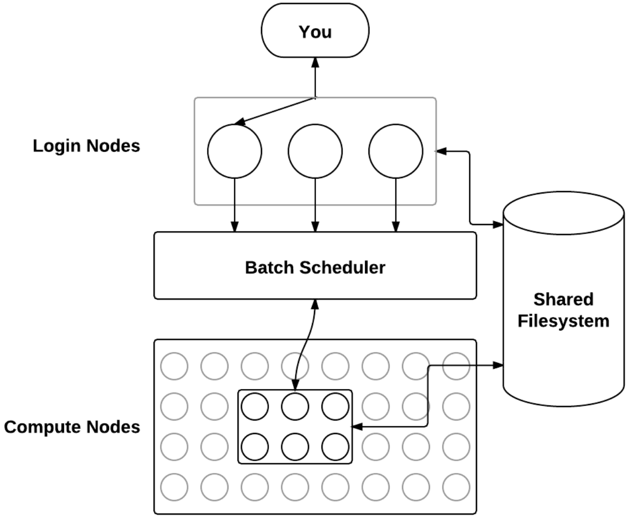
Summary of previous class

MPI Collectives

Submitting jobs through a scheduler

Submitting jobs on Stampede

Overview of HPC cluster



Submitting jobs on Stampede

Stampede user guide:
https://portal.tacc.utexas.edu/user-guides/stampede

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

- submit/start a job: `sbatch jobscript`
- see status of my job: `squeue -u USERNAME`
- cancel my job: `scancel JOBID`
- see all jobs on machine: `showq | less`

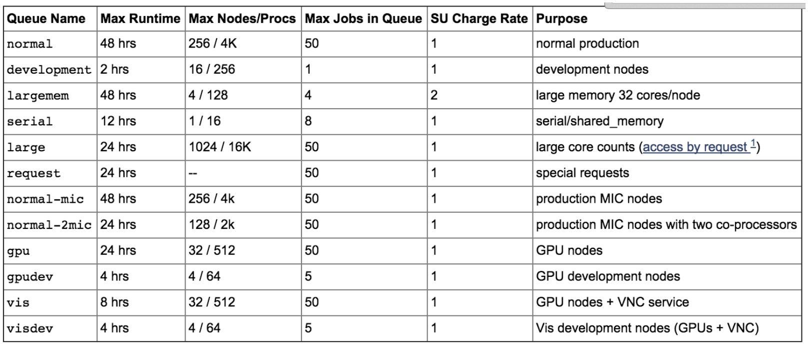
Submitting jobs on Stampede

Some basic rules:

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

Submitting jobs on Stampede

Available queues on Stampede



Queue Name	Max Runtime	Max Nodes/Procs	Max Jobs in Queue	SU Charge Rate	Purpose
`normal`	48 hrs	256 / 4K	50	1	normal production
`development`	2 hrs	16 / 256	1	1	development nodes
`largemem`	48 hrs	4 / 128	4	2	large memory 32 cores/node
`serial`	12 hrs	1 / 16	8	1	serial/shared_memory
`large`	24 hrs	1024 / 16K	50	1	large core counts (access by request [1])
`request`	24 hrs	--	50	1	special requests
`normal-mic`	48 hrs	256 / 4k	50	1	production MIC nodes
`normal-2mic`	24 hrs	128 / 2k	50	1	production MIC nodes with two co-processors
`gpu`	24 hrs	32 / 512	50	1	GPU nodes
`gpudev`	4 hrs	4 / 64	5	1	GPU development nodes
`vis`	8 hrs	32 / 512	50	1	GPU nodes + VNC service
`visdev`	4 hrs	4 / 64	5	1	Vis development nodes (GPUs + VNC)

Submitting jobs on Stampede

Example job script (in git repo for lecture5)

```
#!/bin/bash
#SBATCH -J myMPI              \# job name
#SBATCH -o myMPI.o            \# output and error file name
#SBATCH -n 32                 \# total number of mpi tasks
#SBATCH -p development        \# queue -- normal, development, etc.
#SBATCH -t 01:30:00           \# run time (hh:mm:ss) - 1.5 hours
#SBATCH --mail-user=username@tacc.utexas.edu
#SBATCH --mail-type=begin     \# email me when the job starts
#SBATCH --mail-type=end       \# email me when the job finishes
ibrun ./a.out                 \# run the MPI executable
```