Message Passing Interface (MPI) Programming

Aiichiro Nakano

Collaboratory for Advanced Computing & Simulations
Department of Computer Science
Department of Physics & Astronomy
Department of Quantitative & Computational Biology
University of Southern California

Email: anakano@usc.edu

MPI: Standard parallel programming language

Preparation

Minimal knowledge required for the hands-on projects in this course:

- **Able to log in & use the Discovery computing cluster at USC Center for Advanced Research Computing (CARC) at the level of its "getting started" tutorial—logging in; transferring files; installing software; running jobs,…:**
  https://www.carc.usc.edu/user-guides/hpc-systems/discovery/getting-started-discovery
- **Use shell commands to interact with the operating system at the level of "Chapter 1—Introduction to the Command Line" of *Effective Computation in Physics* by Scopatz and Huff; USC students have free access to the book through Safari Online:** https://libraries.usc.edu/databases/safari-books

Chapter 1. Introduction to the Command Line

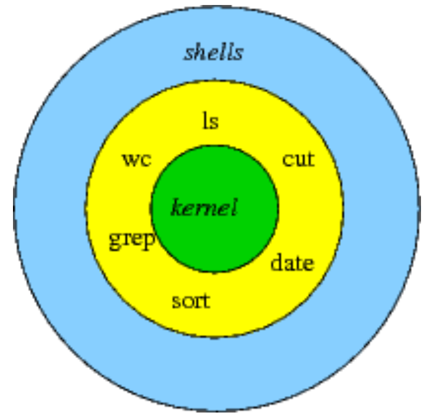
The command line, or *shell*, provides a powerful, transparent interface between the user and the internals of a computer. At least on a Linux or Unix computer, the command line provides total access to the files and processes defining the state of the computer—including the files and processes of the operating system.

- **Need to log in from USC secure network or USC VPN if off campus**
  https://itservices.usc.edu/vpn

How to Use USC CARC Cluster

System: Intel/AMD-based computing cluster

https://carc.usc.edu



Shell is a language you speak with the operating system

Log in

```
> ssh anakano@discovery.usc.edu
```

Alternatively, you can use `discovery2.usc.edu`

To use MPI library:

Use text editor like vim, nano, emacs

If using Bash shell, add these in .bashrc

```
module purge
module load usc
```

To set up standard software environment

Type `echo $0` to find which shell you are using

Compile an MPI program

```
> mpicc -o mpi_simple mpi_simple.c
```

Execute an MPI program

```
> mpirun -n 2 mpi_simple
```

To find absolute path to mpicc command

```
[anakano@discovery ~]$ which mpicc
/spack/2206/apps/linux-centos7-x86_64_v3/gcc-11.3.0/openmpi-4.1.4-4w23jca/bin/mpicc
[anakano@discovery ~]$ more /proc/cpuinfo
```

To find processor information

Email carc-support@usc.edu for assistance

VPN Issue

- **It is now required to use VPN (virtual private network) to access Discovery from off-campus:**
  https://itservices.usc.edu/vpn
- **Cisco AnyConnect software for VPN on Mac may have a DNS (domain name system) problem, which could be bypassed using IP addresses instead of login server names (note discovery.usc.edu is a generic name for the two login servers, discovery1 and discovery2)**

  discovery1.usc.edu: 10.72.0.13

  discovery2.usc.edu: 10.72.0.14

Submit a Slurm Batch Job

Slurm (Simple Linux Utility for Resource Management): Open-source job scheduler that allocates compute resources on clusters for queued jobs

Prepare a script file, mpi_simple.sl

```
#!/bin/bash
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2
#SBATCH --time=00:00:10
#SBATCH --output=mpi_simple.out
#SBATCH -A anakano_429
mpirun -n $SLURM_NTASKS ./mpi_simple
```

Class project account; type `myaccount` to check all accounts

Total number of processes = ntasks-per-node × nodes

Submit a Slurm job

```
discovery: sbatch mpi_simple.sl
Submitted batch job 63695
```

Check the status of a Slurm job

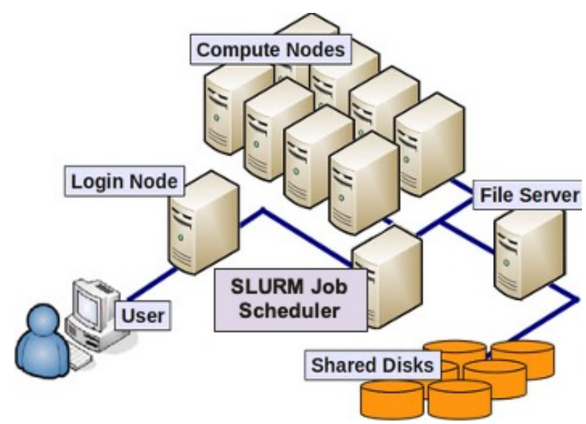
```
discovery: squeue -u anakano
JOBID PARTITION       NAME     USER ST       TIME  NODES  NODELIST(REASON)
63695      main  mpi_simple  anakano PD       0:00      1  (Resources)
```

Cancel a Slurm job

```
discovery: scancel 63695
```

Check the output

```
discovery: more mpi_simple.out
n = 777
```



For detailed explanation, see the lecture note

https://aiichironakano.github.io/cs596/02MPI.pdf

Interactive Job at CARC

When debugging your MPI program, you may want to access computing nodes interactively, so that you can edit, compile & run MPI program in real time unlike the batch job

Reserve 2 processors for 20 minutes

```
[anakano@discovery cs596]$ salloc -n 2 -t 20
salloc: Granted job allocation 63754
salloc: Waiting for resource configuration
salloc: Nodes d05-05 are ready for job
[anakano@d05-05 cs596]$ mpirun -n 2 ./mpi_simple
n = 777
[anakano@d05-05 cs596]$ exit
exit
salloc: Relinquishing job allocation 63754
[anakano@discovery cs596]$
```

Note you are now using a computing node named d05-05

Back to the login node

Type `less /proc/cpuinfo` **to find what kind of node you got**

Symbolic Link to Work Directory

- **Your home directory has small (but enough for assignments) quota (type `myquota` to confirm), so use the scratch file system (`/scratch1/anakano` for user `anakano`) if needed**
- **It is convenient to make a symbolic link to a directory you use often, rather than typing its long absolute path every time**

```
[anakano@discovery ~]$ ln -s /scratch1/anakano/cs596 cs596
[anakano@discovery ~]$ ls -lt
total 81985
lrwxrwx--- 1 anakano anakano 22 Aug 23 12:14 cs596 -> /scratch1/anakano/cs596
drwxrwx--- 3 anakano anakano  1 Aug 20 10:07 FFTW
lrwxrwx--- 1 anakano anakano 16 Aug 14 15:48 scr -> /scratch1/anakano
...
[anakano@discovery ~]$ cd cs596
[anakano@discovery cs596]$ pwd -P
/scratch1/anakano/cs596
```

symbolic link (`ln -s`) — source (`/scratch1/anakano/cs596`) — alias (`cs596`)

This directory has been created as
mkdir /scratch1/anakano/cs596

Instead of typing `cd /scratch1/anakano/cs596`

Print physical working directory (`pwd -P`)

File Transfer

- **Use secure file transfer protocol to transfer files between your laptop and Discovery**

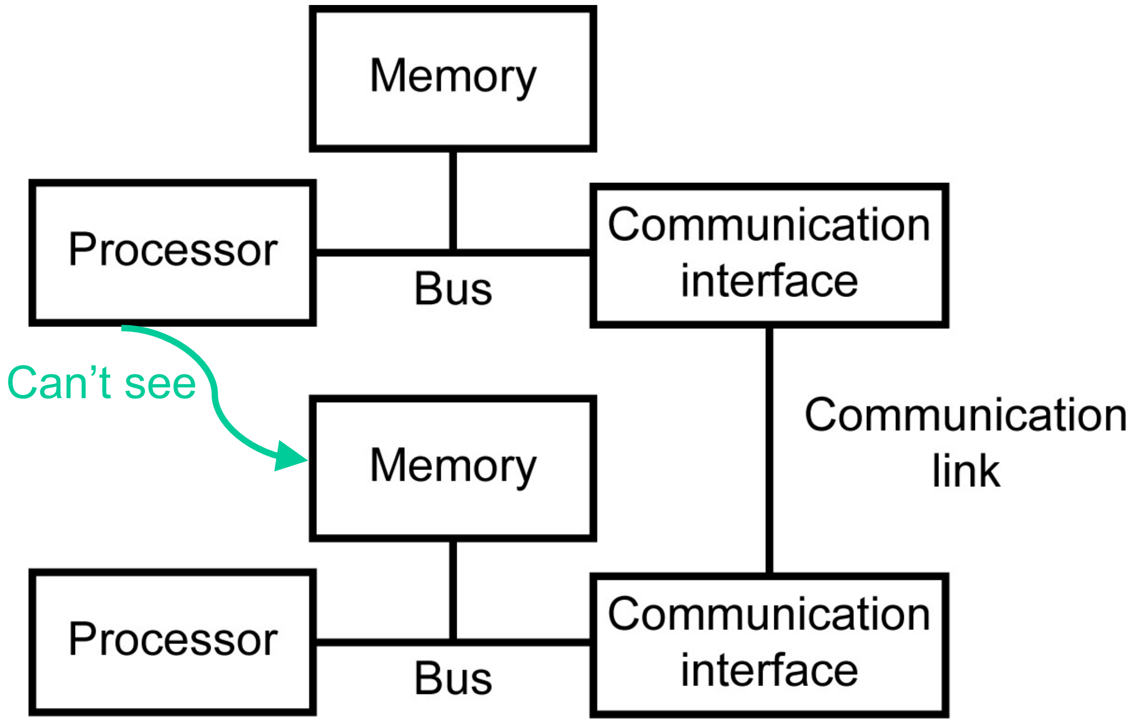
```
macbook-pro $ sftp anakano@discovery.usc.edu
Connected to discovery.usc.edu.
sftp> cd cs596
sftp> put md.*
sftp> ls
md.c    md.h    md.in
sftp> exit
macbook-pro $
```

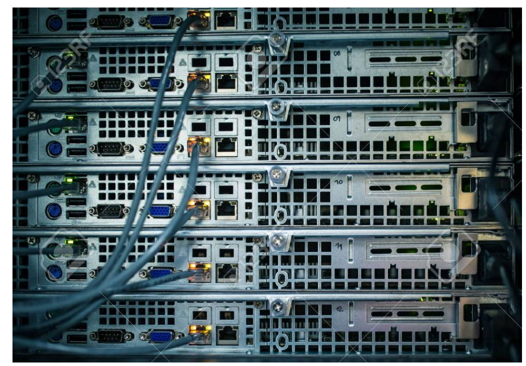
Transfer files from local computer (your laptop) to remote computer (Discovery) — annotation for `put md.*`

Check whether the files have been transferred — annotation for `ls`

- **To transfer files from remote computer to local computer, use `get` instead**

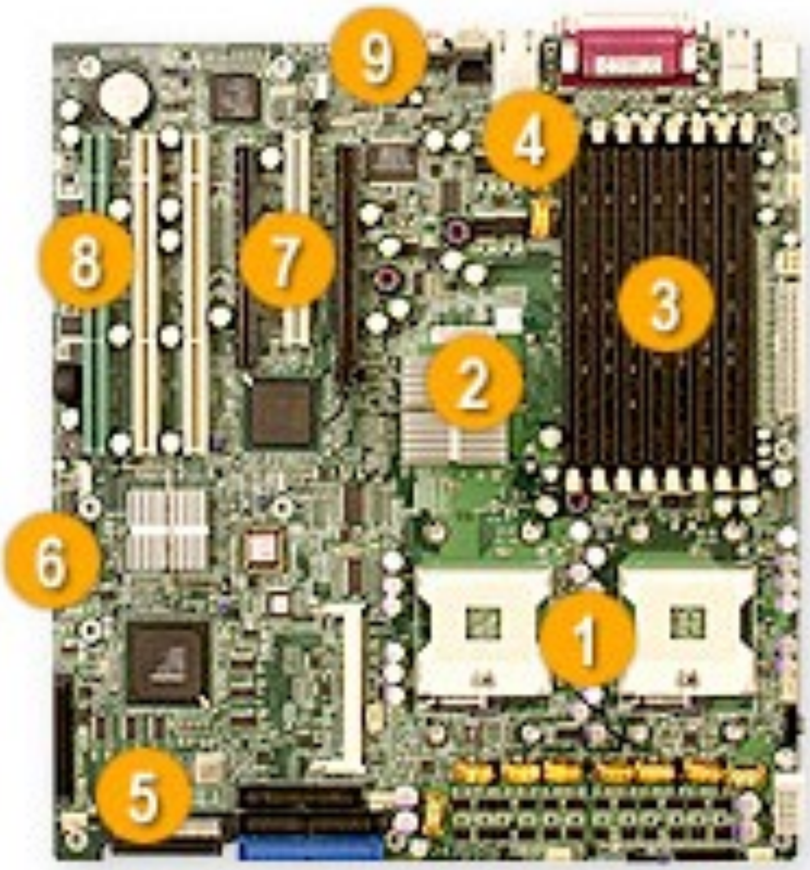
Parallel Computing Hardware





- **Processor**: Executes arithmetic & logic operations.
- **Memory**: Stores program & data.
- **Communication interface**: Performs signal conversion & synchronization between communication link and a computer.
- **Communication link**: A wire capable of carrying a sequence of bits as electrical (or optical) signals.

Motherboard



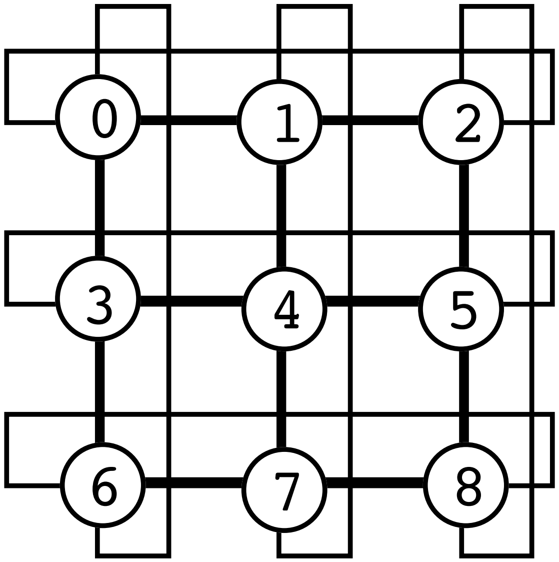
Key Features

1. Dual Intel® Xeon™ EM64T Support up to 3.60 GHz
2. Intel® E7525 (Tumwater) Chipset
3. Up to 16GB DDRII-400 SDRAM
4. Intel® 82546GB Dual-port Gigabit Ethernet Controller
5. Adaptec AIC-7902 Dual Channel Ultra320 SCSI
6. 2x SATA Ports via ICH5R SATA Controller
7. 1 (x16) & 1 (x4) PCI-Express, 1 x 64-bit 133MHz PCI-X, 2 x 64-bit 100MHz PCI-X, 1 x 32-bit 33MHz PCI Slots
8. Zero Channel RAID Support
9. AC'97 Audio, 6-Channel Sound

Supermicro X6DA8-G2

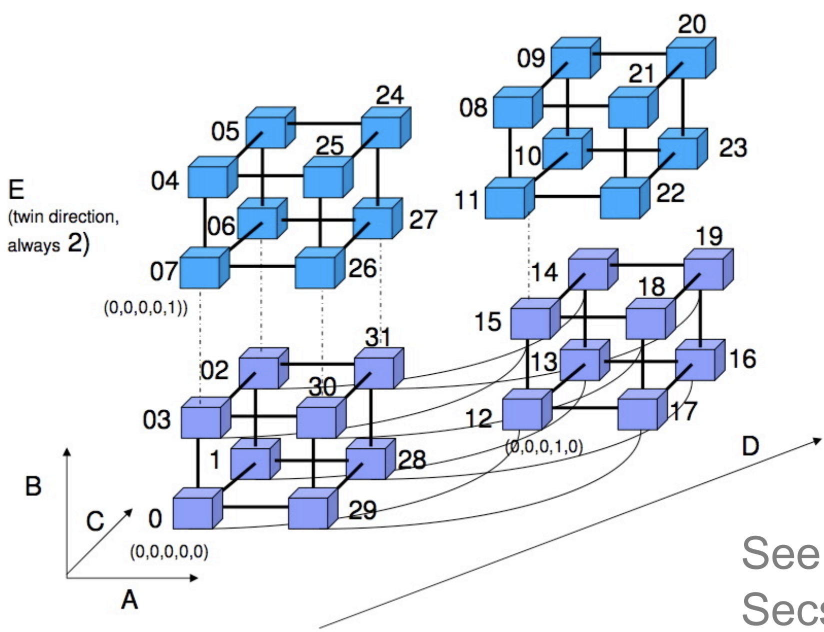
Communication Network

Mesh (torus)

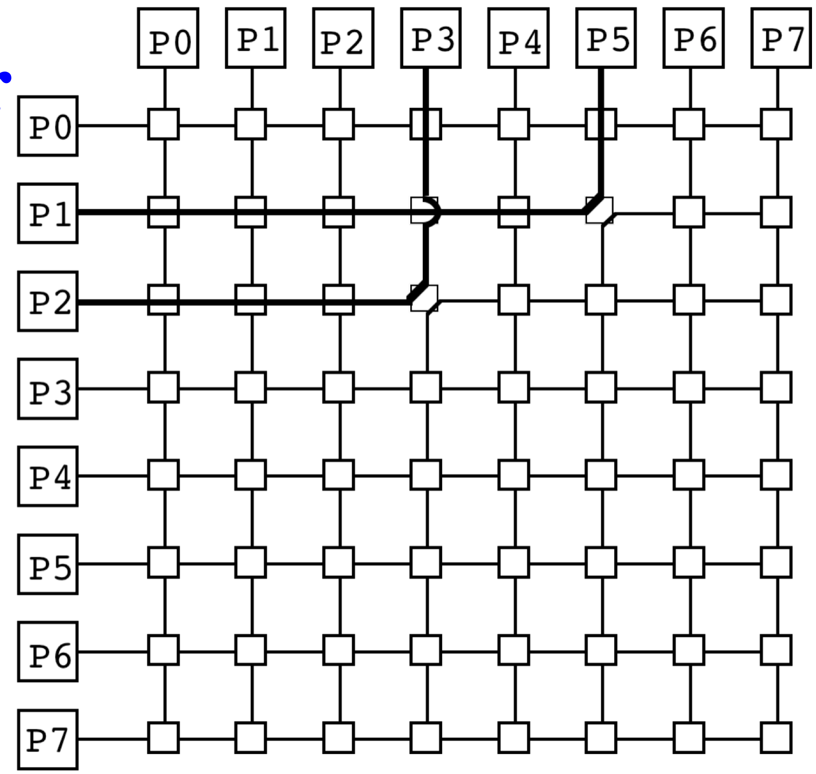


NEC Earth Simulator (640x640 crossbar)

IBM Blue Gene/Q (5D torus)



Crossbar switch



See Grama,
Secs. 2.4.2-2.4.4

Message Passing Interface

MPI (Message Passing Interface)

A standard message passing system that enables us to write & run applications on parallel computers

Download for Unix & Windows:

`http://www.mcs.anl.gov/mpi/mpich`

Compile

```
> mpicc -o mpi_simple mpi_simple.c
```

Run

```
> mpirun -np 2 mpi_simple
```

MPI Programming

mpi_simple.c: Point-to-point message send & receive

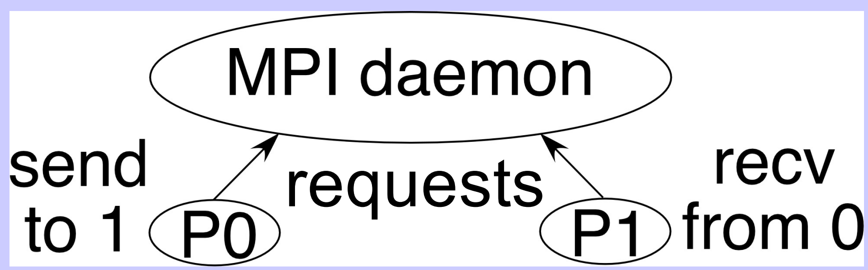
```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
  MPI_Status status;
  int myid;
  int n;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid == 0) {
    n = 777;
    MPI_Send(&n, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
  }
  else {
    MPI_Recv(&n, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
    printf("n = %d\n", n);
  }
  MPI_Finalize();
 return 0;
}
```

MPI rank

Matching message tags

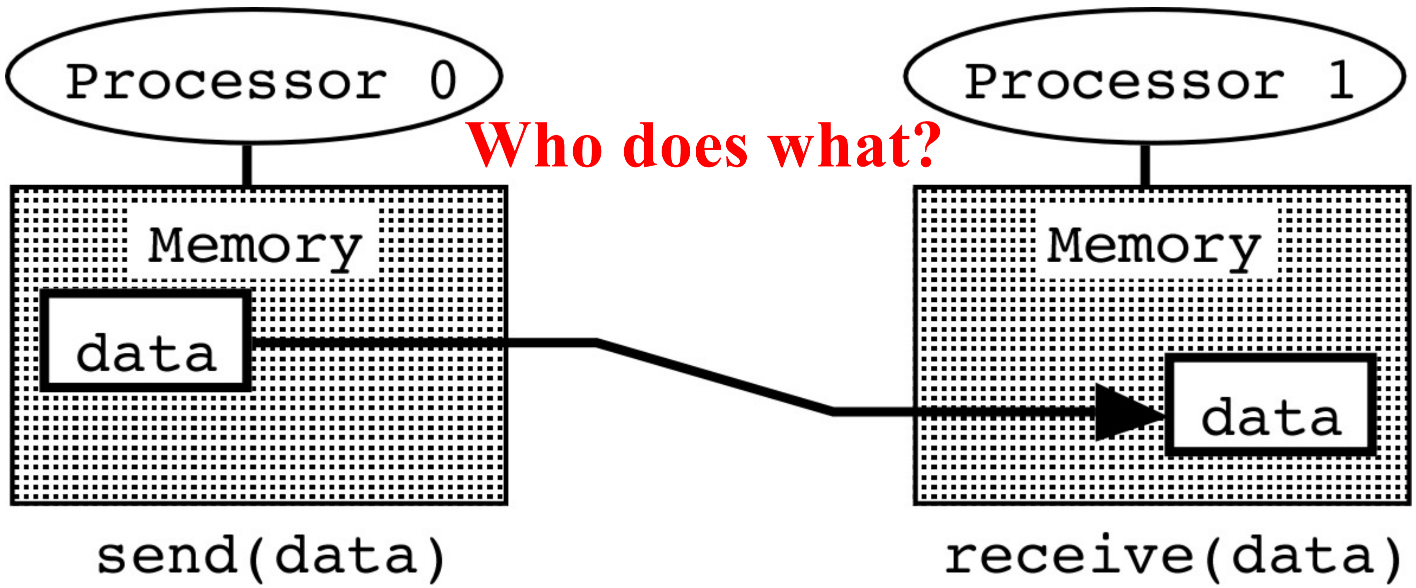
Data triplet

To/from whom



cf. p. 3 of https://aiichironakano.github.io/cs596/02MPI.pdf

Single Program Multiple Data (SPMD)



Process 0

```
if (myid == 0) {
  n = 777;
  MPI_Send(&n,...);
}
else {
  MPI_Recv(&n,...);
  printf(...);
}
```

Process 1

```
if (myid == 0) {
  n = 777;
  MPI_Send(&n,...);
}
else {
  MPI_Recv(&n,...);
  printf(...);
}
```

Single Program Multiple Data (SPMD)

What really happens?

node 1 node 2 … node 3 node 4 …

mpi_simple mpi_simple

rank0 rank1

start executing line-by-line independently

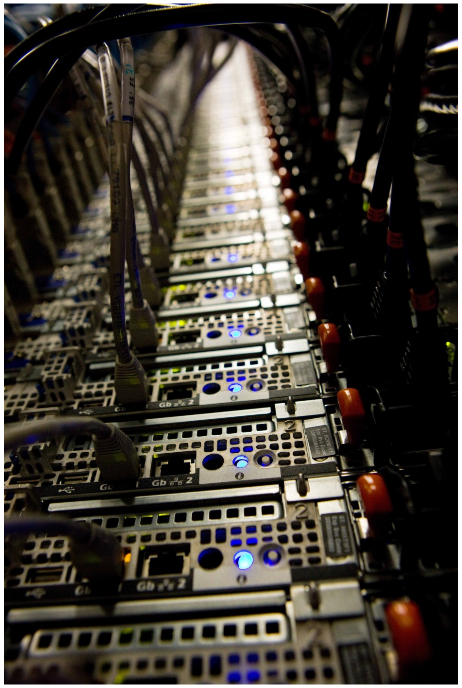
sbatch allocates backend nodes

discovery

%mpirun -n 2 mpi_simple

%ssh discovery.usc.edu

My laptop



MPI Minimal Essentials

We only need `MPI_Send()` & `MPI_Recv()` within `MPI_COMM_WORLD`

```
MPI_Send(&n, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
MPI_Recv(&n, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
```

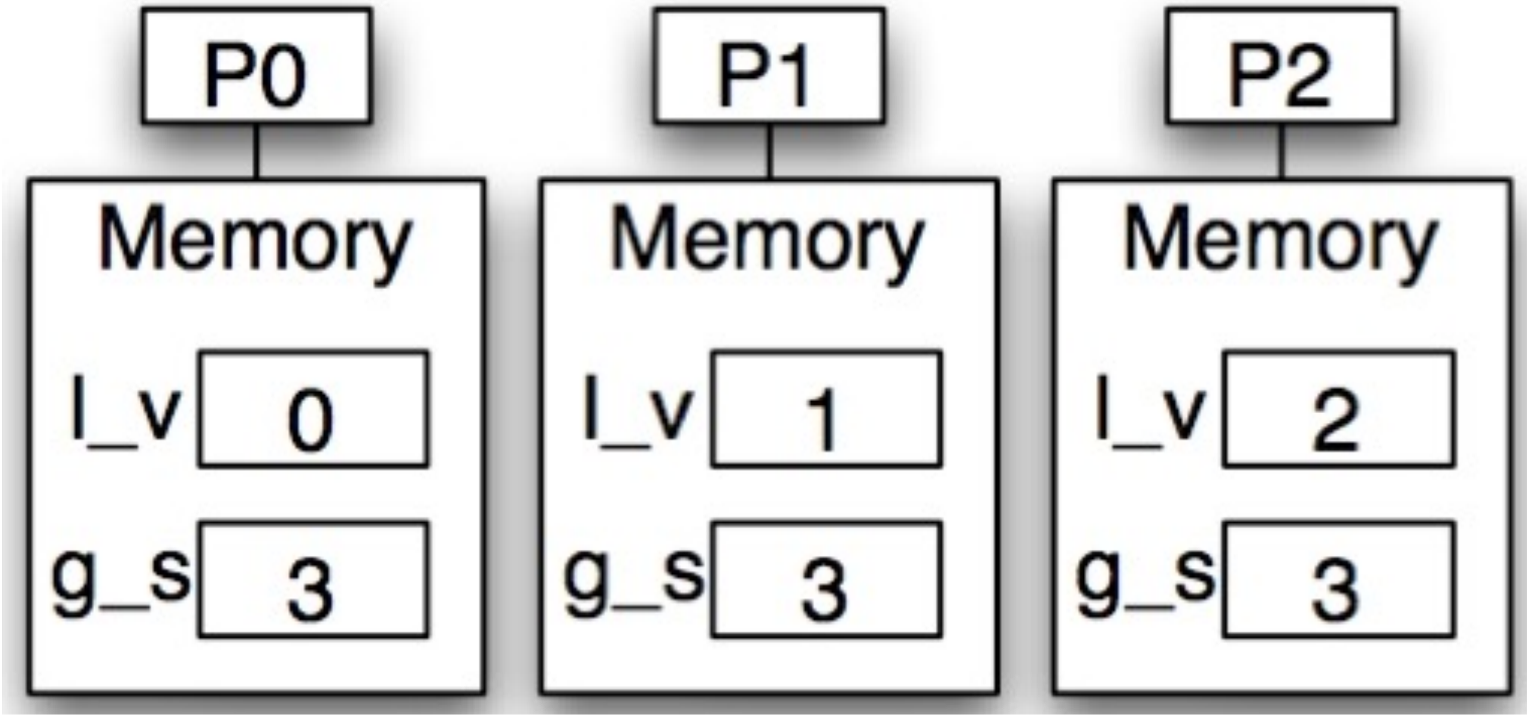
Data triplet (&n, 1, MPI_INT) — **To/from whom** (1 or 0, 10, MPI_COMM_WORLD) — **Information** (&status)

Global Operation

All-to-all reduction: Each process contributes a partial value to obtain the global summation. In the end, all the processes will receive the calculated global sum.

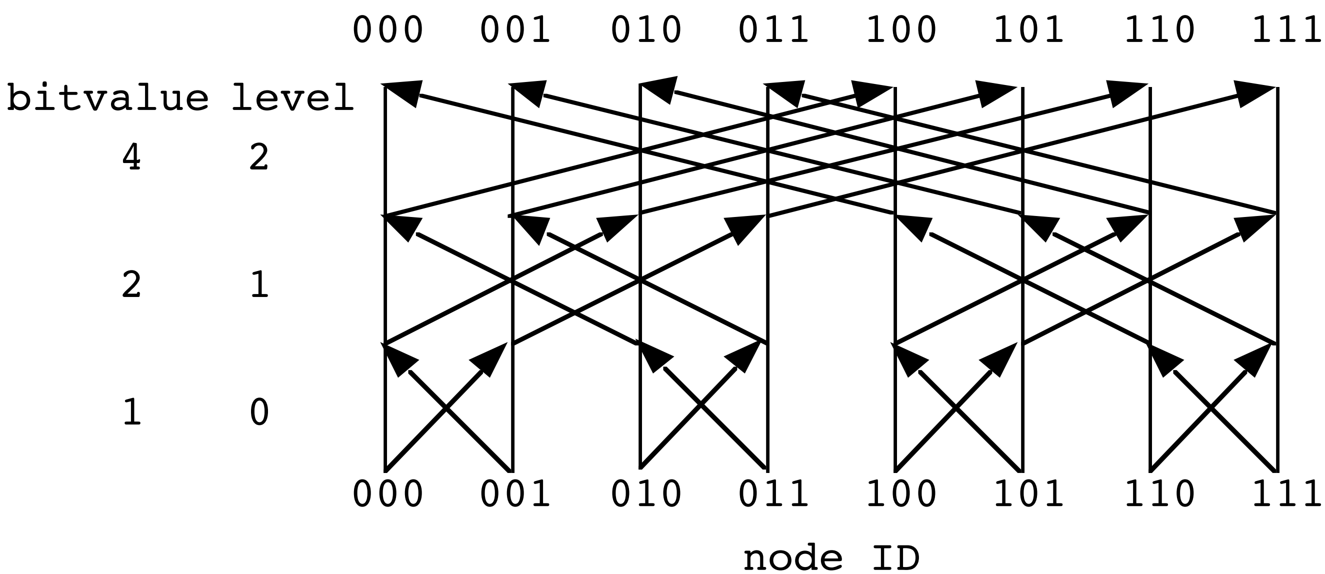
```
MPI_Allreduce(&local_value, &global_sum, 1, MPI_INT, MPI_SUM,
MPI_COMM_WORLD)
```

```
int l_v, g_s;  // local variable & global sum
l_v = myid;    // myid is my MPI rank
MPI_Allreduce(&l_v, &g_s, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
```



Hypercube Algorithm

Hypercube algorithm: Communication of a reduction operation is structured as a series of pairwise exchanges, one with each neighbor in a hypercube (**butterfly**) structure. Allows a computation requiring all-to-all communication among p processes to be performed in $\log_2 p$ steps.

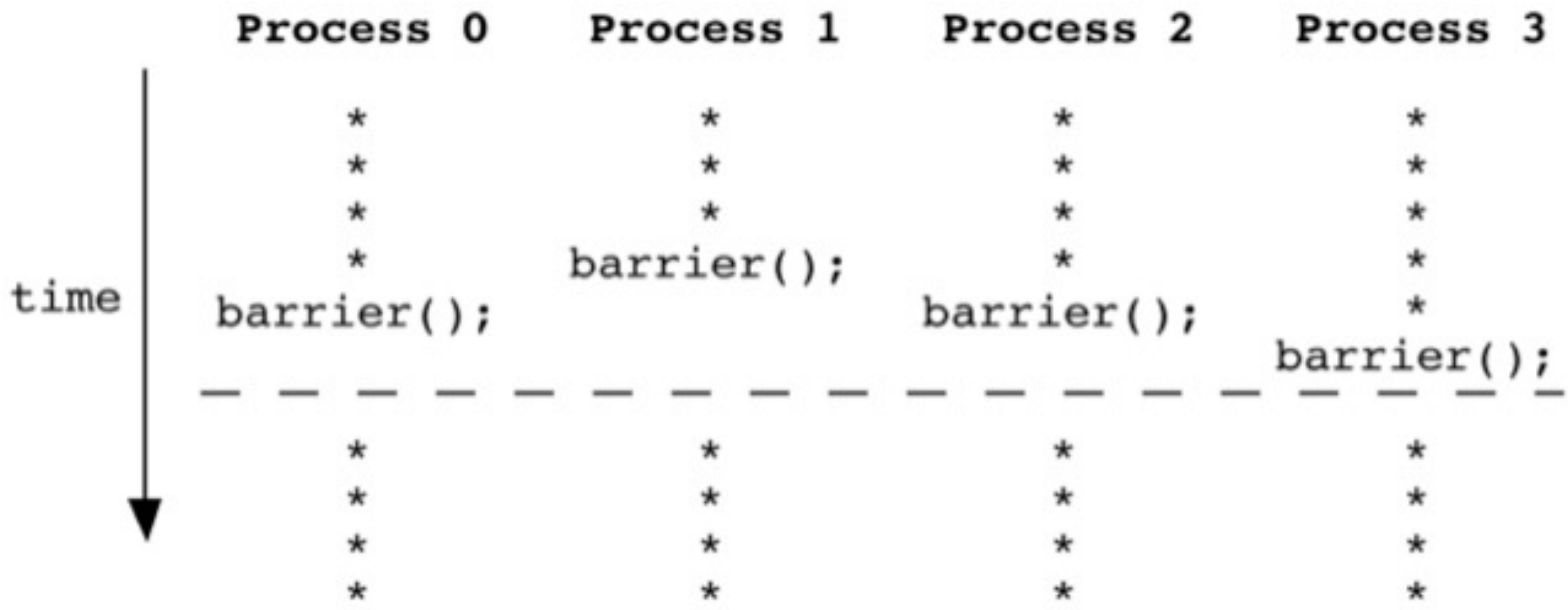


Butterfly network

```
a000 + a001 + a010 + a011 + a100 + a101 + a110 + a111
=  ((a000 + a001) + (a010 + a011))
+  ((a100 + a101) + (a110 + a111))
 ②                 ①       ⓪
```

Barrier

```
<A>;
barrier();
<B>;
```



MPI_Barrier(MPI_Comm communicator)

Useful for debugging (but would slow down the program)

MPI Communication

MPI communication functions:

1. **Point-to-point**

   `MPI_Send()`

   `MPI_Recv()`

2. **Global**

   `MPI_Allreduce()`

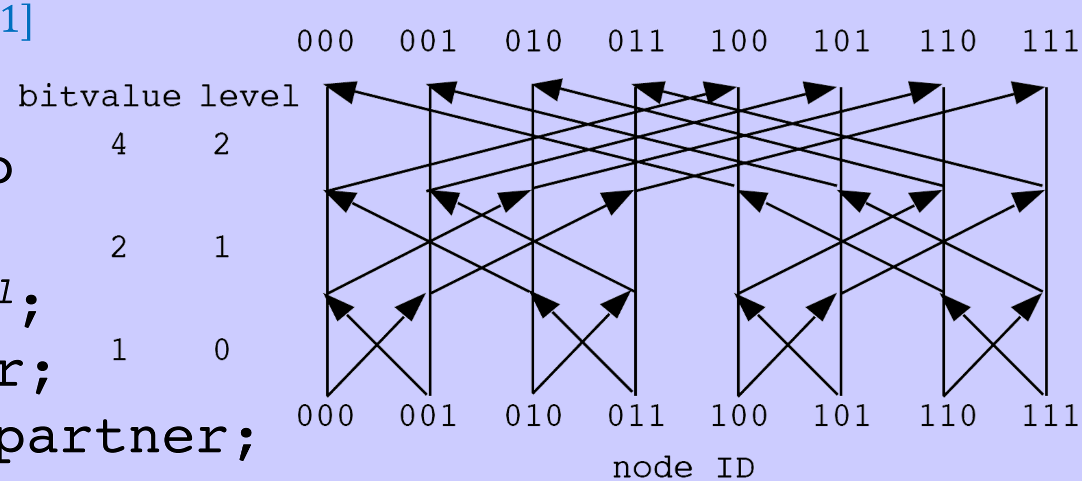
   `MPI_Barrier()`

   `MPI_Bcast()`

Hypercube Template

```
procedure hypercube(myid, input, log2P, output)
begin
  mydone := input;
  for l := 0 to log2P-1 do
  begin
    partner := myid XOR 2^l;
    send mydone to partner;
    receive hisdone from partner;
    mydone = mydone OP hisdone
  end
  output := mydone
end
```

$myid \in [0, P-1]$



level l	bitvalue 2^l
0	001
1	010
2	100

Associative operator (*e.g.*, sum, max)

(*a* OP *b*) OP *c* = *a* OP (*b* OP *c*)

Exclusive OR

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

(a = 0: a XOR b = b; a = 1: a XOR b = $\bar{b}$)

abcdefg XOR 0000100 = abcd$\bar{e}$fg

In C, ^ (caret operator) is bitwise XOR applied to int

Driver for Hypercube Test

```
#include "mpi.h"
#include <stdio.h>
int nprocs;  /* Number of processes */
int myid;    /* My rank */

double global_sum(double partial) {
  /* Implement your own global summation here */
}

int main(int argc, char *argv[]) {
  double partial, sum, avg;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  partial = (double) myid;
  printf(”Rank %d has %le\n", myid, partial);
  sum = global_sum(partial);
  if (myid == 0) {
    avg = sum/nprocs;
    printf("Global average = %le\n", avg);
  }
  MPI_Finalize();
  return 0;
}
```

Global variables are visible in both *global_sum*() & *main*()

Who am I?

How big is the world? (see p. 5 in MPI lecture note)

C Implementation of global_sum( )

```
double mydone = partial;
for (int bitvalue=1; bitvalue<nprocs; bitvalue*=2)
{
   int partner = myid ^ bitvalue;
   send mydone to partner;
   receive hisdone from partner;
   mydone = mydone + hisdone;
}
return mydone;
```

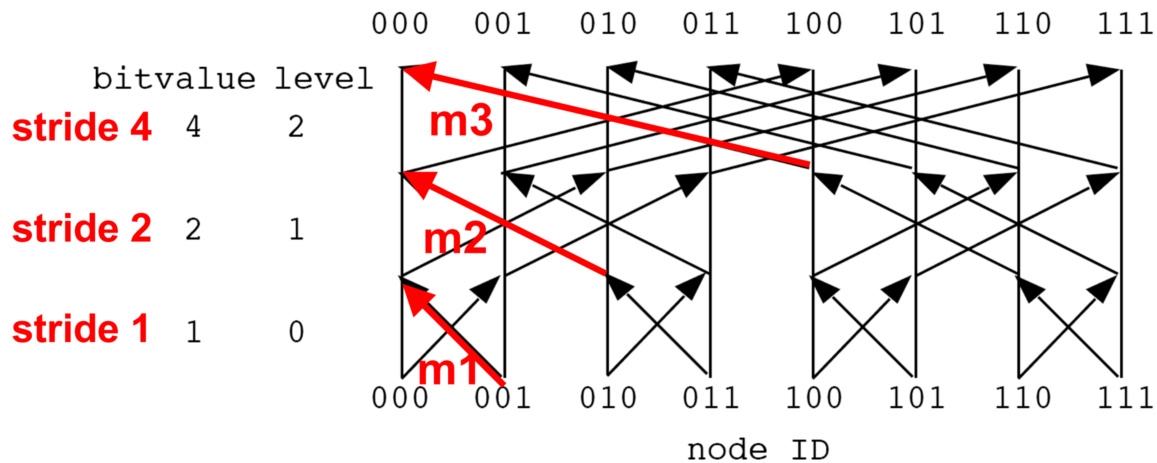
Multiplied by 2

Implement with MPI_Send() & MPI_Recv()

level	2^{l}	bitvalue
0	001	1
1	010	2
2	100	4

Use *bitvalue* as counter & bitmask

Recursive stride doubling



It is recommended to use distinct labels (tags) for different messages, *e.g.*, bitmask (= stride) as a tag

MPI Send & Receive Revisited

Starting address of data

# of data items

Data type

Source/destination rank

Tag (label) *cf.* email subject

Communicator (to be discussed)

```
MPI_Send(&n, 1, MPI_INT, 1, 10, MPI_COMM_WORLD);
MPI_Recv(&n, 1, MPI_INT, 0, 10, MPI_COMM_WORLD, &status);
```

Data triplet (`&n, 1, MPI_INT`) — **To/from whom** (`0, 10, MPI_COMM_WORLD`) — **Information** (`&status`)

MPI_Datatype	C data type
MPI_CHAR	char
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
...	...

MPI_Status status;
Filled with information about the received message

status.MPI_SOURCE	Source process rank
status.MPI_TAG	Tag of the received message
...	...

- **Only tag-matching message passing between matching source/destination pair of ranks take place**
- **It is recommended to use distinct tags for different messages to avoid accidental receipt of unintended messages**

Sample Slurm Script

Run two MPI runs in a single Slurm job

```
#!/bin/bash
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --time=00:00:59
#SBATCH --output=global.out
#SBATCH -A anakano_429

mpirun -n $SLURM_NTASKS ./global_avg
mpirun -n 4 ./global_avg
```

`mpicc -o global_avg global_avg.c`

Total number of processors
= ntasks-per-node (4) × nodes (2) = 8

- **Type** `sbatch global_avg.sl` **in the directory where the executable** `global_avg` **resides, or** `cd` **(change directory) to where it is**

Output of `global.c`

- **4-processor job**

```
Rank 0 has 0.000000e+00
Rank 1 has 1.000000e+00
Rank 2 has 2.000000e+00
Rank 3 has 3.000000e+00
Global average = 1.500000e+00
```

- **8-processor job**

```
Rank 0 has 0.000000e+00
Rank 1 has 1.000000e+00
Rank 2 has 2.000000e+00
Rank 3 has 3.000000e+00
Rank 5 has 5.000000e+00
Rank 6 has 6.000000e+00
Rank 4 has 4.000000e+00
Rank 7 has 7.000000e+00
Global average = 3.500000e+00
```

Actual output is random order in ranks — Why?

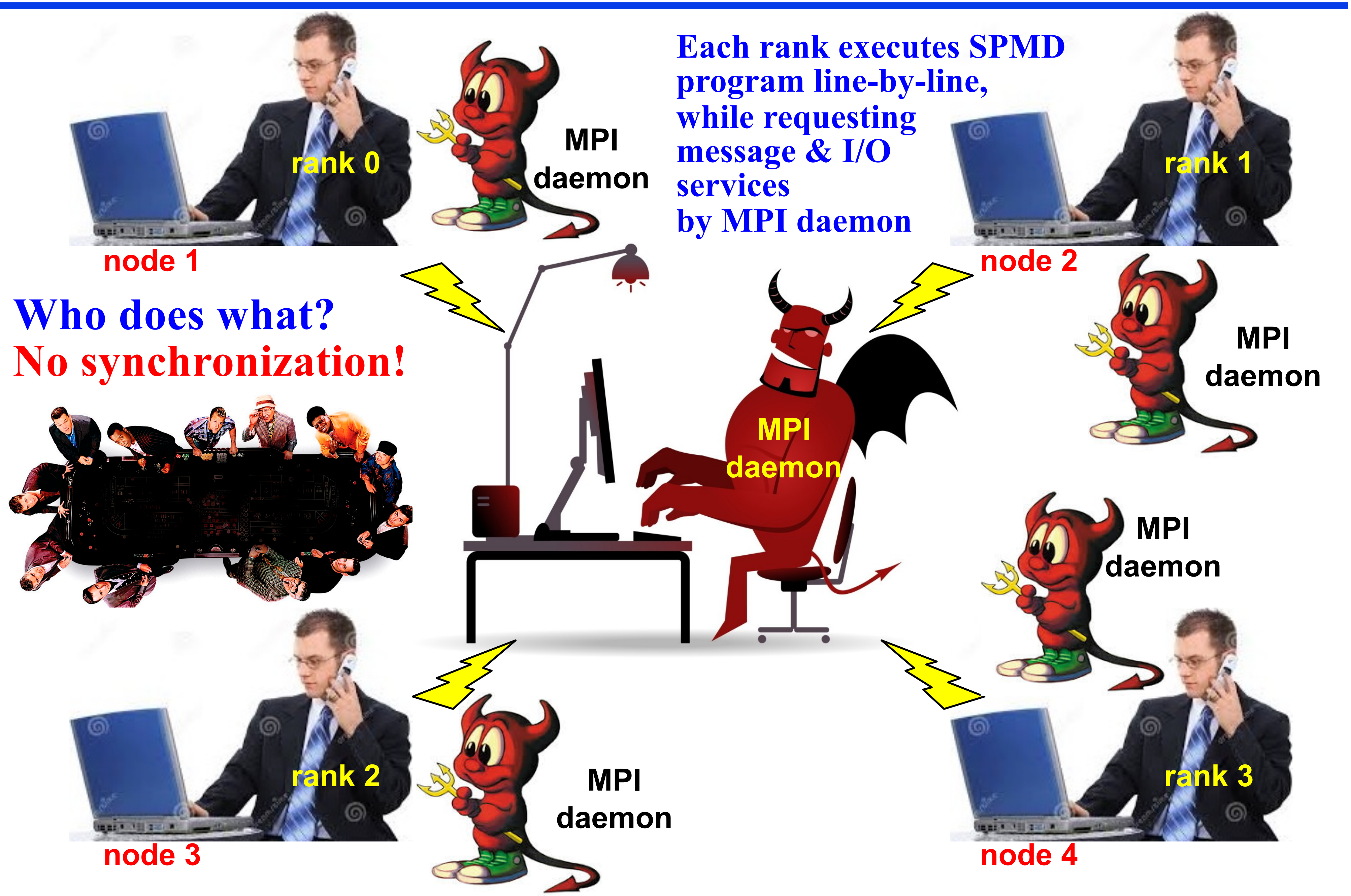
References on Hypercube Algorithms

1. https://en.wikipedia.org/wiki/Hypercube_(communication_pattern)
2. I. Foster, *Designing and Building Parallel Programs* (Addison-Wesley, 1995) Chap. 11 — Hypercube algorithms: https://www.mcs.anl.gov/~itf/dbpp/text/node123.html

Distributed-Memory Parallel Computing

Each rank executes SPMD program line-by-line, while requesting message & I/O services by MPI daemon

Who does what?
No synchronization!



Communicator

mpi_comm.c: Communicator = process group + context

```
#include "mpi.h"
#include <stdio.h>
#define N 64
int main(int argc, char *argv[]) {
  MPI_Comm world, workers;
  MPI_Group world_group, worker_group;
  int myid, nprocs;
  int server, n = -1, ranks[1];
  MPI_Init(&argc, &argv);
  world = MPI_COMM_WORLD;
  MPI_Comm_rank(world, &myid);     // My rank in the world (see next slide)
  MPI_Comm_size(world, &nprocs);   // How big is the world?
  server = nprocs-1;               // The last guy becomes the server
  MPI_Comm_group(world, &world_group);
  ranks[0] = server;               // Note the rest will become workers
  MPI_Group_excl(world_group, 1, ranks, &worker_group);
  MPI_Comm_create(world, worker_group, &workers);
  MPI_Group_free(&worker_group);   // Release resources no longer needed
  if (myid != server).             // All, except for the server, are workers
    MPI_Allreduce(&myid, &n, 1, MPI_INT, MPI_SUM, workers);
  printf("process %2d: n = %6d\n", myid, n);
  MPI_Comm_free(&workers);
  MPI_Finalize();
  return 0;
}
```

Usage

- **Avoid accidental match of unintended Send-Receive pairs**
- **Global operations in a subgroup of processes**

Code at https://aiichironakano.github.io/cs596/src/mpi/
For detail, see p. 4 in https://aiichironakano.github.io/cs596/02MPI.pdf

Example: Ranks in Different Groups

World Rank	Institution*	Country /Region	National Rank	Total Score	Score on Alumni
1	Harvard University		1	100	100
2	Stanford University		2	72.1	41.8
3	Massachusetts Institute of Technology (MIT)		3	70.5	68.4
4	University of California-Berkeley		4	70.1	66.8
5	University of Cambridge		1	69.2	79.1
51	University of Southern California		33	31	31.7

```
MPI_Comm_rank(world, &usc_world);
MPI_Comm_rank(us, &usc_national);
```

Rank is relative in each communicator!

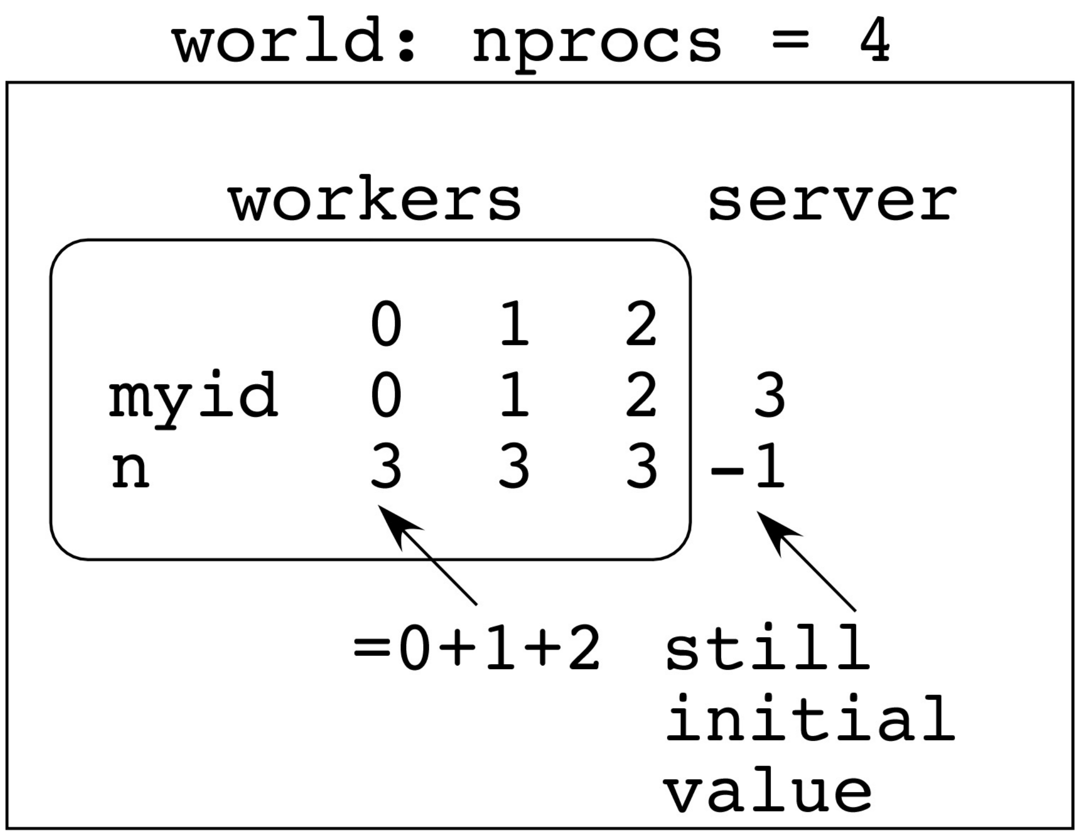
Output from mpi_comm.c

Slurm script

```
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=2
...
mpirun -n $SLURM_NTASKS./mpi_comm
```

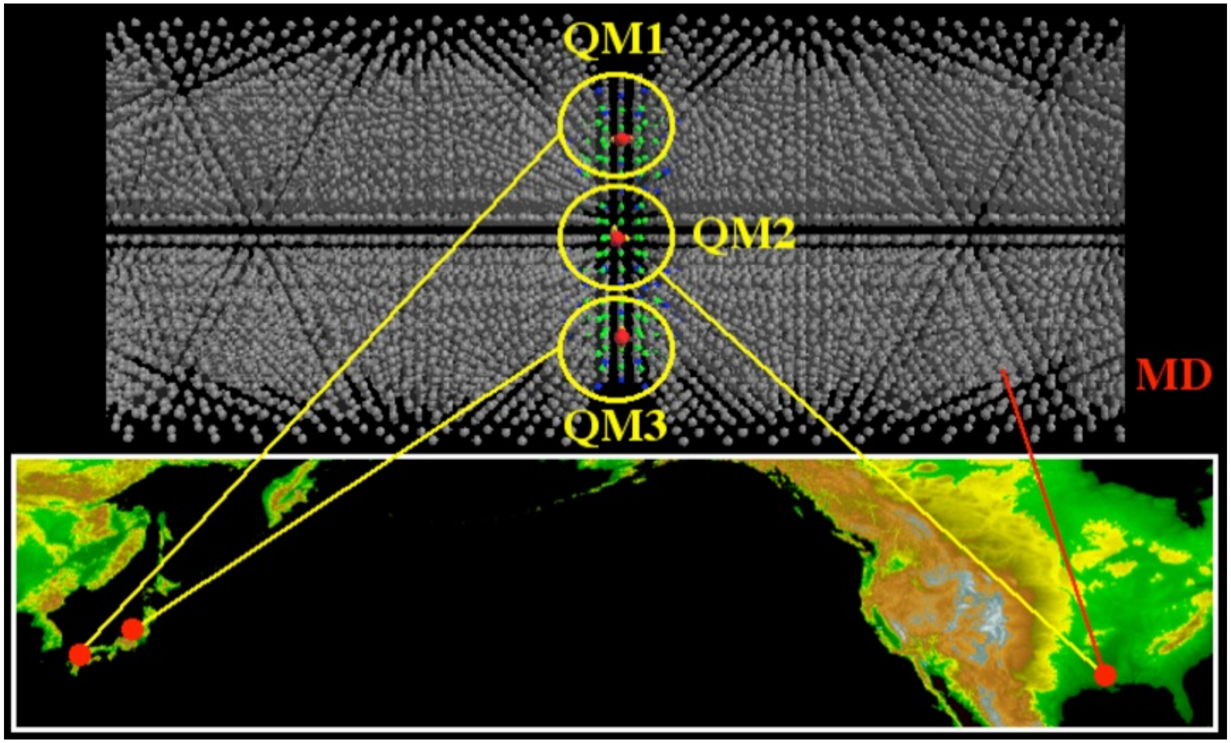
```
process  3: n =     -1
process  0: n =      3
process  1: n =      3
process  2: n =      3
```

What Has Happened?



Grid Computing & Communicators

H. Kikuchi *et al.*, "Collaborative simulation Grid: multiscale quantum-mechanical/classical atomistic simulations on distributed PC clusters in the US & Japan, *IEEE/ACM SC02*



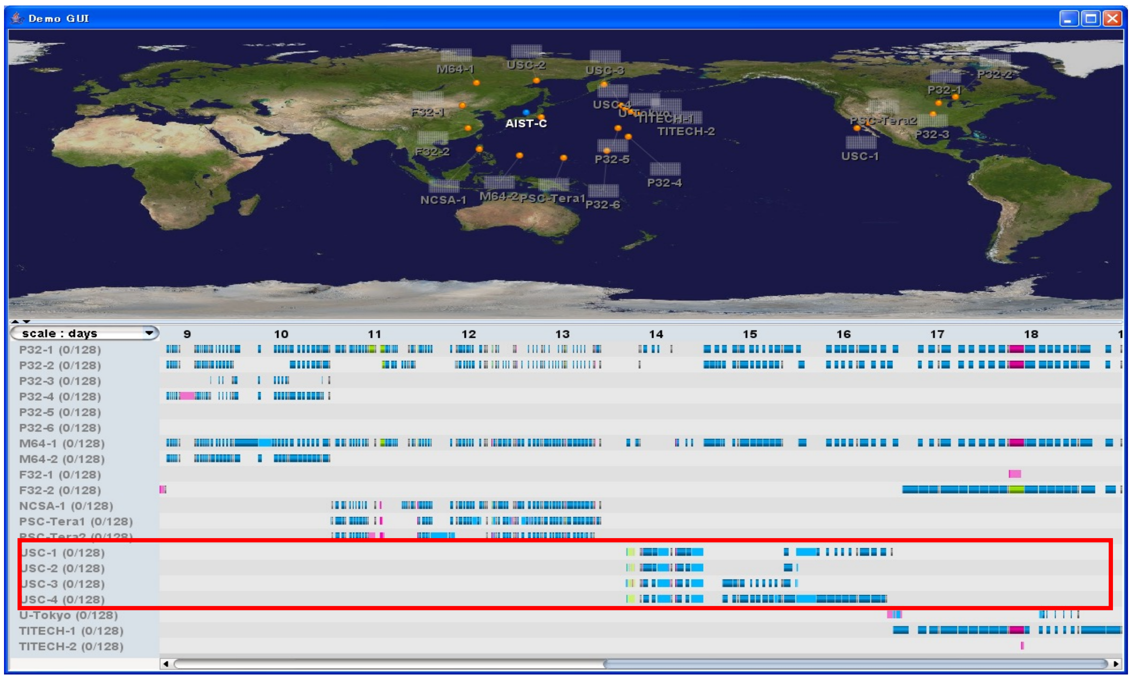
Communicator = a nice migration path to distributed computing

- **Single MPI program run with the Grid-enabled MPI implementation, MPICH-G2**
- **Processes are grouped into MD & QM groups by defining multiple MPI communicators as subsets of MPI_COMM_WORLD; a machine file assigns globally distributed processors to the MPI processes**

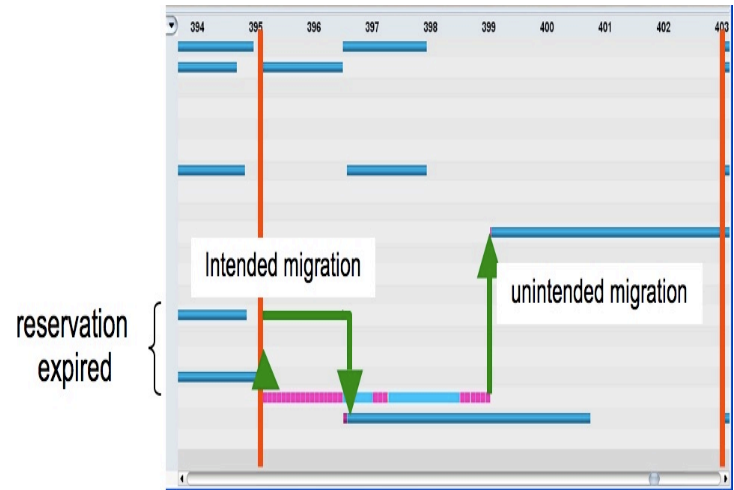
Global Grid QM/MD

- *One of the largest (153,600 cpu-hrs) sustained Grid supercomputing* **at 6 sites in the US (USC, Pittsburgh, Illinois) & Japan (AIST, U Tokyo, Tokyo IT)**

USC



Automated resource migration & fault recovery

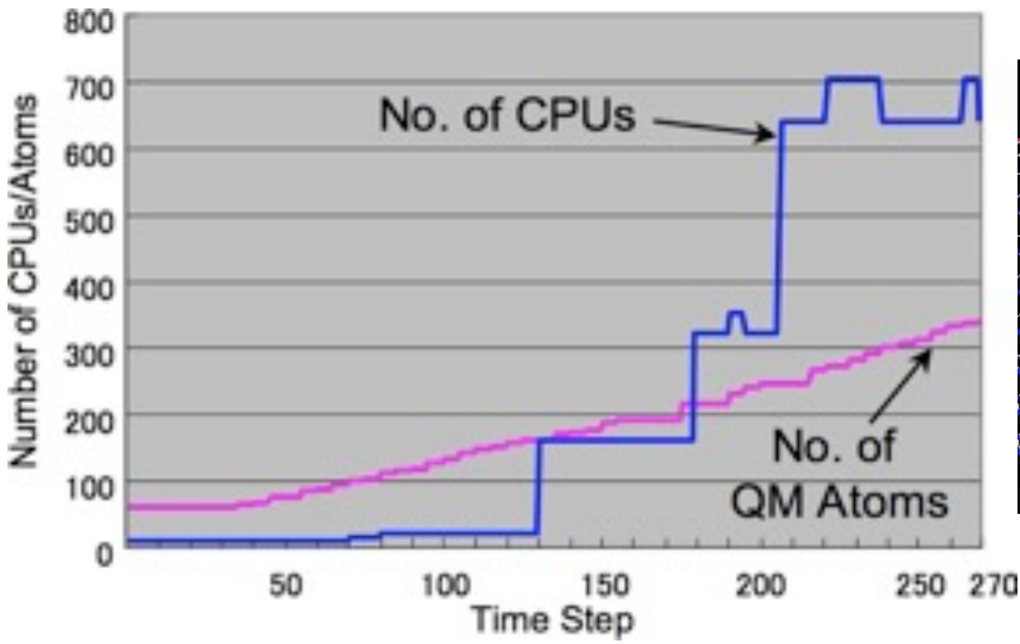


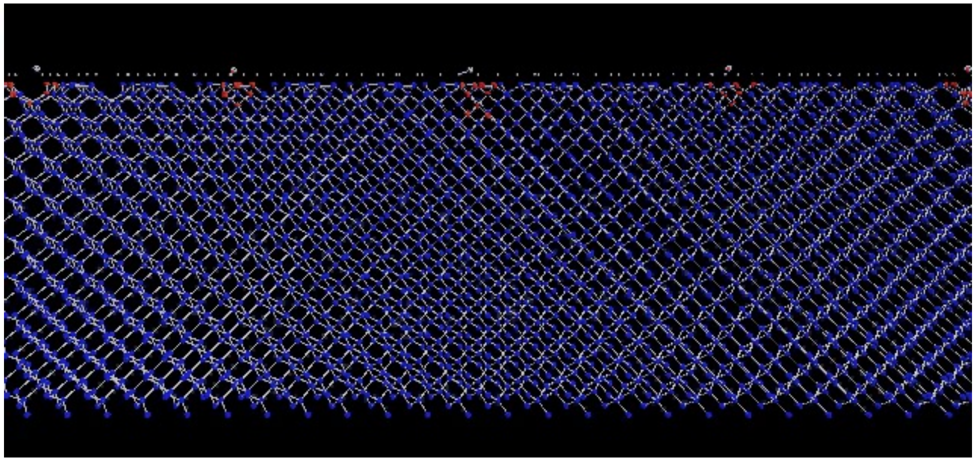
Takemiya *et al*., "Sustainable adaptive Grid supercomputing: multiscale simulation of semiconductor processing across the Pacific," *IEEE/ACM SC06*

Sustainable Grid Supercomputing

- Sustained (> months) supercomputing (> 10^3 CPUs) on a Grid of geographically distributed supercomputers
- Hybrid Grid remote procedure call (GridRPC) + message passing (MPI) programming
- Dynamic allocation of computing resources on demand & automated migration due to reservation schedule & faults

Ninf-G GridRPC: ninf.apgrid.org; MPICH: www.mcs.anl.gov/mpi

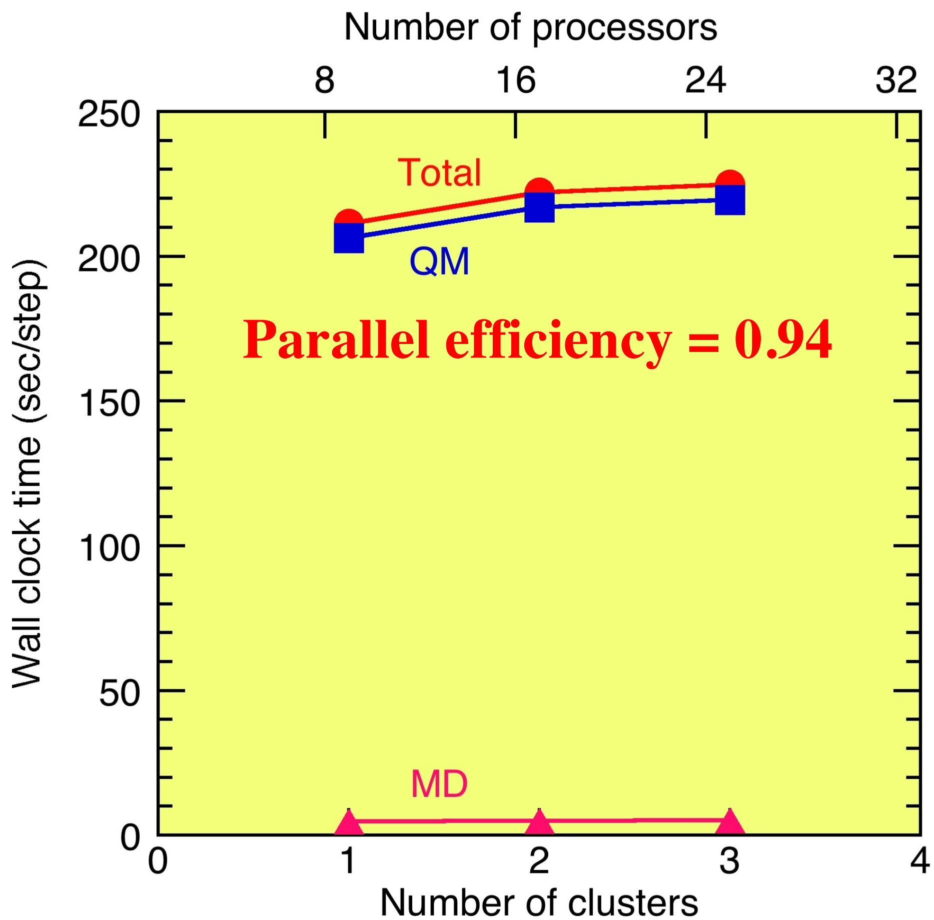




Multiscale QM/MD simulation of high-energy beam oxidation of Si

Computation-Communication Overlap

H. Kikuchi *et al.*, "Collaborative simulation Grid: multiscale quantum-mechanical/classical atomistic simulations on distributed PC clusters in the US & Japan, *IEEE/ACM SC02*



$$\frac{\text{Earth's circumference}}{\text{Light speed}} = \frac{40{,}000\ [\text{km}] = 4\times10^{7}[\text{m}]}{3\times10^{8}[\text{m/s}]} = 0.1\ \text{s} = 100\ \text{ms}$$

Try on Discovery:

`traceroute www.u-tokyo.ac.jp`

vs. `ping hpc-transfer.usc.edu`

- **How to overcome 200 ms latency & 1 Mbps bandwidth?**
- **Computation-communication overlap: To hide the latency, the communications between the MD & QM processors have been overlapped with the computations using asynchronous messages**

Synchronous Message Passing

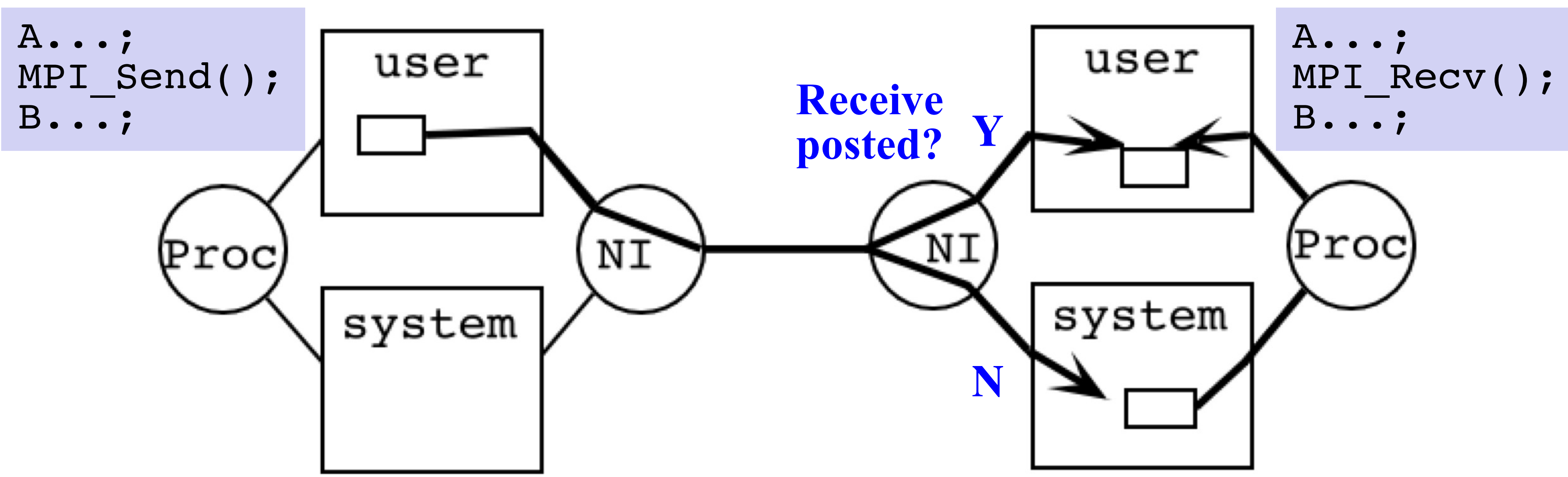
`MPI_Send()`: (blocking), synchronous

- Safe to modify original data immediately on return
- Depending on implementation, it may return whether or not a matching receive has been posted, or it may block (especially if no buffer space available)

`MPI_Recv()`: blocking, synchronous

- Blocks for message to arrive
- Safe to use data on return

Experienced a lot of blocking on iPSC/860 with 12 MB user & 4 MB system memory per node



Asynchronous Message Passing

Allows computation-communication overlap

`MPI_Isend()`: non-blocking, asynchronous

- Returns immediately whether or not a matching receive has been posted
- Not safe to modify original data immediately (use `MPI_Wait()` system call)

`MPI_Irecv()`: non-blocking, asynchronous

- Does not block for message to arrive
- Cannot use data before checking for completion with `MPI_Wait()`

`MPI_Irecv()` is just a "request" for data delivery, when a matching message arrives

```
A...;
MPI_Isend();
B...;
MPI_Wait();
C...;  // Reuse the send buffer
```

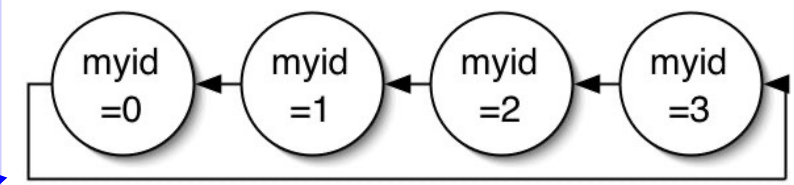
```
A...;
MPI_Irecv();
B...;
MPI_Wait();
C...;  // Use the received message
```

See p. 6 in https://aiichironakano.github.io/cs596/02MPI.pdf

Program irecv_mpi.c

```
#include "mpi.h"
#include <stdio.h>
#define N 1000
int main(int argc, char *argv[]) {
  MPI_Status status;
  MPI_Request request;
  int send_buf[N], recv_buf[N];
  int send_sum = 0, recv_sum = 0;
  long myid, left, Nnode, msg_id, i;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &Nnode);
  left  = (myid + Nnode - 1) % Nnode;
  for (i=0; i<N; i++) send_buf[i] = myid*N + i; // Compose a big message
  MPI_Irecv(recv_buf, N, MPI_INT, MPI_ANY_SOURCE, 777, MPI_COMM_WORLD,
            &request); // Post a receive
  /* Perform tasks that don't use recv_buf */
  MPI_Send(send_buf, N, MPI_INT, left, 777, MPI_COMM_WORLD);
  for (i=0; i<N; i++) send_sum += send_buf[i];
  MPI_Wait(&request, &status); // Complete the receive
  /* Now it's safe to use recv_buf */
  for (i=0; i<N; i++) recv_sum += recv_buf[i];
  printf("Node %d: Send %d Recv %d\n", myid, send_sum, recv_sum);
  MPI_Finalize();
  return 0;
}
```

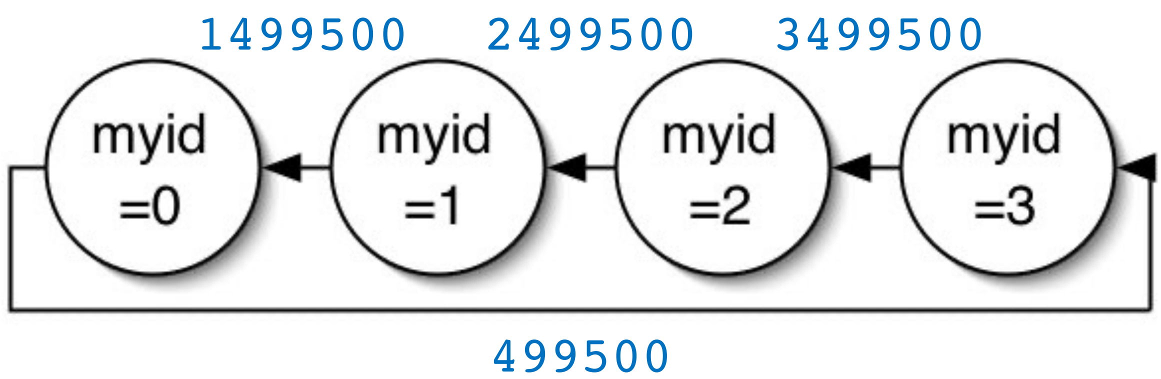
Wrap-around/torus
***via* modulo (%) operator**
(*cf.* periodic boundary condition)



Code at https://aiichironakano.github.io/cs596/src/mpi/

Output from `irecv_mpi.c`

```
Node 1: Send 1499500 Recv 2499500
Node 3: Send 3499500 Recv 499500
Node 0: Send 499500 Recv 1499500
Node 2: Send 2499500 Recv 3499500
```



Multiple Asynchronous Messages

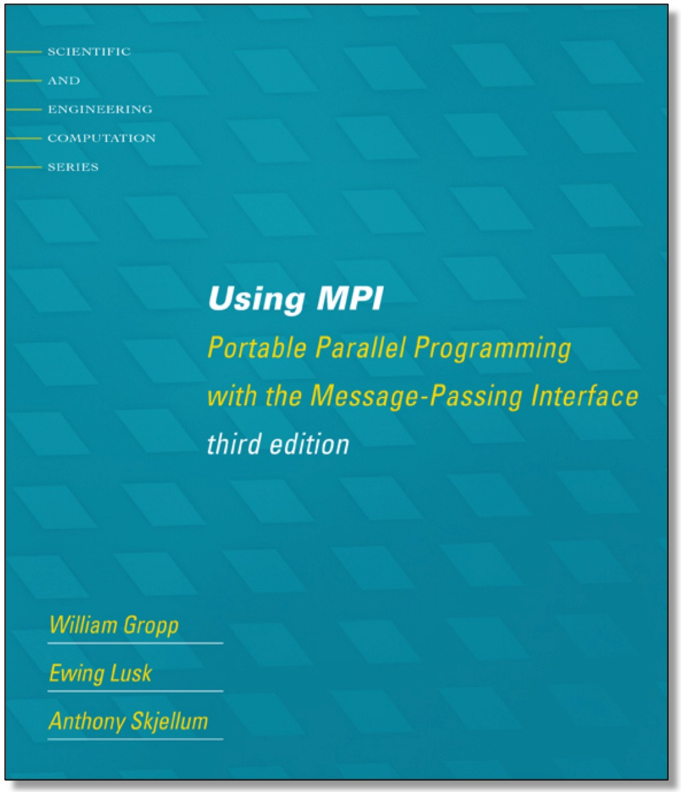
```
MPI_Request requests[N_message];
MPI_Status statuses[N_message];
MPI_Status status;
int index;

/* Wait for all messages to complete */
MPI_Waitall(N_message, requests, statuses);

/* Wait for any specified messages to complete */
MPI_Waitany(N_message, requests, &index, &status);
```

returns the index (∈ [0,N_message-1]) of the message that completed

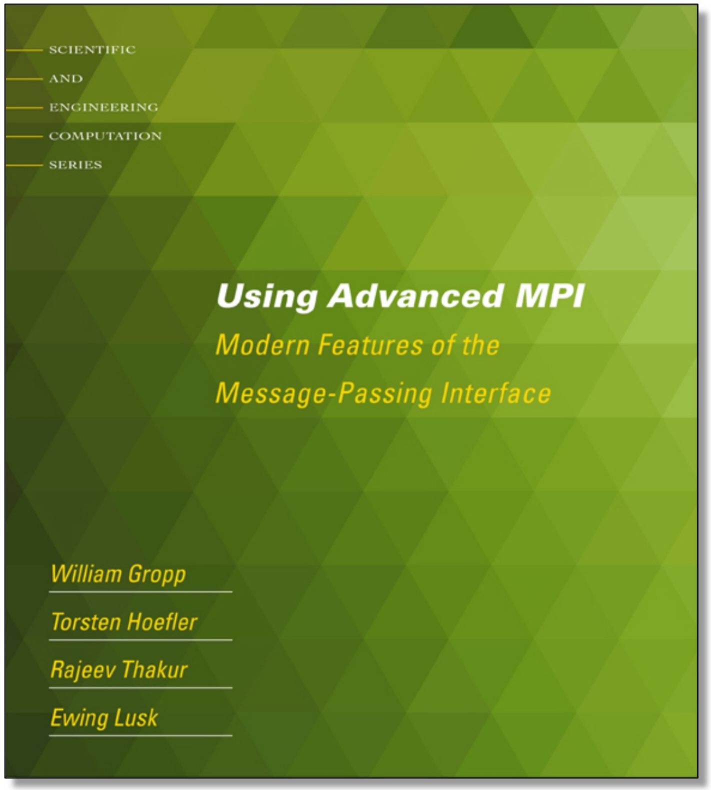
Polling `MPI_Irecv`

```
int flag;

/* Post an asynchronous receive */
MPI_Irecv(recv_buf, N, MPI_INT, MPI_ANY_SOURCE, 777,
          MPI_COMM_WORLD, &request);

/* Perform tasks that don't use recv_buf */
...

/* Polling */
MPI_Test(&request, &flag, &status); // Check completion
if (flag) { // True if message received
  /* Now it's safe to use recv_buf */
  ...
}
```

Where to Go from Here



Basic MPI



Advanced MPI, including MPI-3

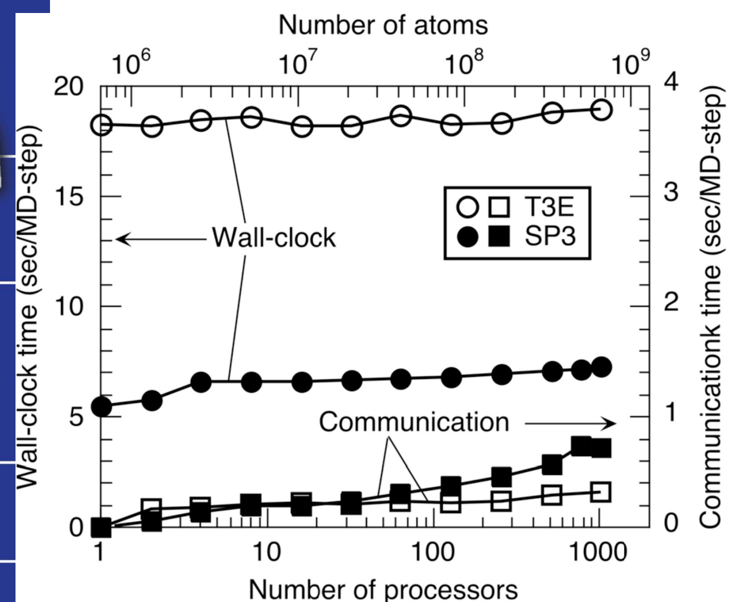
- **Complete MPI reference at** http://www.netlib.org/utk/papers/mpi-book/mpi-book.html
- **MPI is evolving (MPI-2 to MPI-3) to include advanced features like remote memory access (`MPI_Put()` & `MPI_Get()`; *cf.* `sftp`), parallel I/O and dynamic process management**
- **Various versions of MPI standard are specified at** https://www.mpi-forum.org/docs/

MPI Basics: Recap

- **Parallel computing = Who does what**
- **Single program multiple data (SPMD) programming: Do it with MPI rank (who am I) & selection constructs (`if`, *etc*.)**
- **Only need `MPI_Send()` & `MPI_Recv()` within communicators to implement any distributed-memory parallel computing**
- **Asynchronous message passing (`MPI_Isend()` & `MPI_Irecv()`) to overlap computation & communication**
- **You can survive professionally only with a few global communication functions, *e.g.*, `MPI_Allreduce()`, `MPI_Barrier()` & `MPI_Bcast()`**

Start using MPI for your research & projects!

It's not how fancy programming constructs you use, but a compelling application instead.



SC2001 SC01	Best Paper	Aiichiro Nakano, Rajiv K. Kalia, Priya Vashishta, Timothy J. Campbell, Shuji Ogata, Fuyuki Shimojo, and Subhash Saini Scalable atomistic simulation algorithms for materials research
	Best Student Paper	Shava Smallen, Henri Cazsanova and Francine Berman Applying Scheduling and Tuning to On-line Parallel Tomography
	ACM Gordon Bell Prize	See list of ACM Gordon Bell Prize winners
	Best Research Poster	Sumir Chandra, Johan Steensland, and Manish Parashar ??? If you know, please contact chair@SIGHPC.org