

# Message Passing Interface (MPI) Programming

---

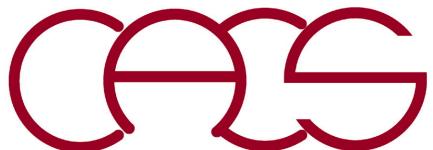
---

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](mailto: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>

**Log in**

> ssh anakano@discovery.usc.edu

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

**To use MPI library:**

**If using Bash shell, add these in .bashrc**

module purge

To set up standard  
software environment

module load usc

**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

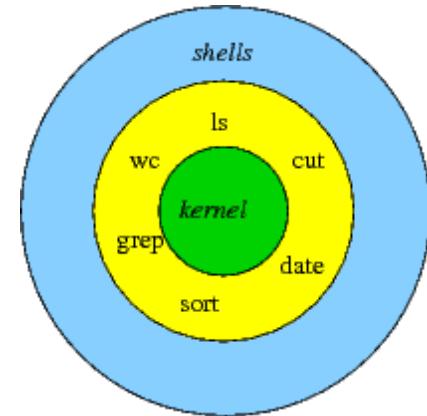
[anakano@discovery ~]\$ which mpicc

/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](mailto:carc-support@usc.edu) for assistance



Shell is a language you speak with  
the operating system

Type **echo \$0** to find  
which shell you are using

# 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

## 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
```

## 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
63695	main	mpi_simple	anakano	PD

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

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

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

## Cancel a Slurm job

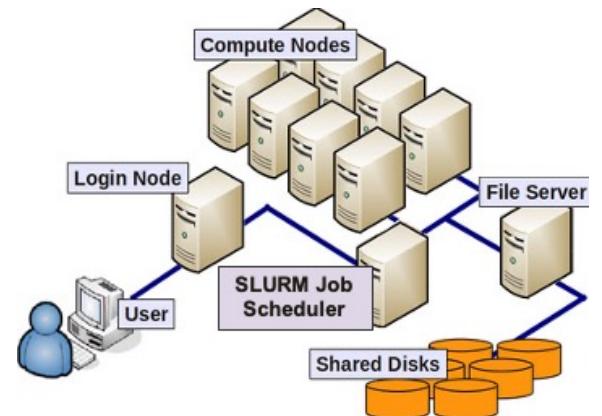
discovery: **scancel 63695**

## 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

symbolic link	source	alias
[anakano@discovery ~]\$ <b>ln -s /scratch1/anakano/cs596 cs596</b>		
[anakano@discovery ~]\$ <b>ls -lt</b>		This directory has been created as mkdir /scratch1/anakano/cs596
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 ~]\$ <b>cd cs596</b>		
[anakano@discovery cs596]\$ <b>pwd -P</b>		Instead of typing cd /scratch1/anakano/cs596
/scratch1/anakano/cs596		Print physical working directory

# 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.*      Transfer files from local computer (your laptop)
          to remote computer (Discovery)

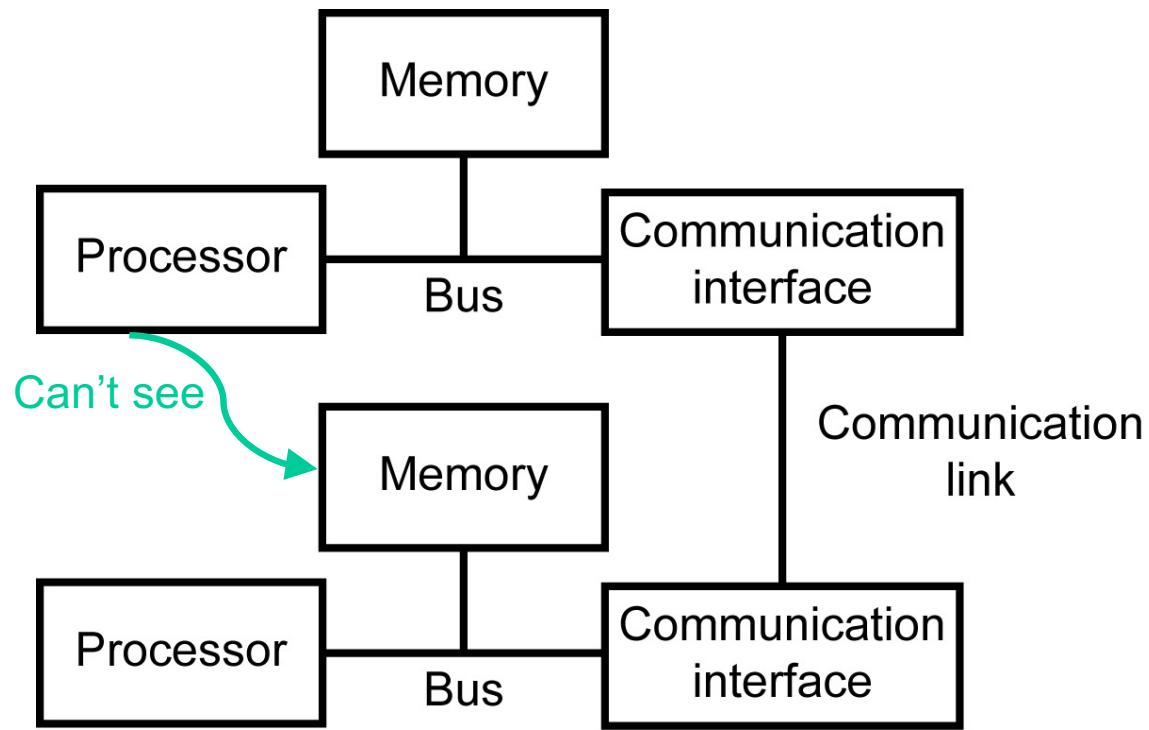
sftp> ls   — Check whether the files have been transferred
md.c    md.h    md.in

sftp> exit

macbook-pro $
```

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

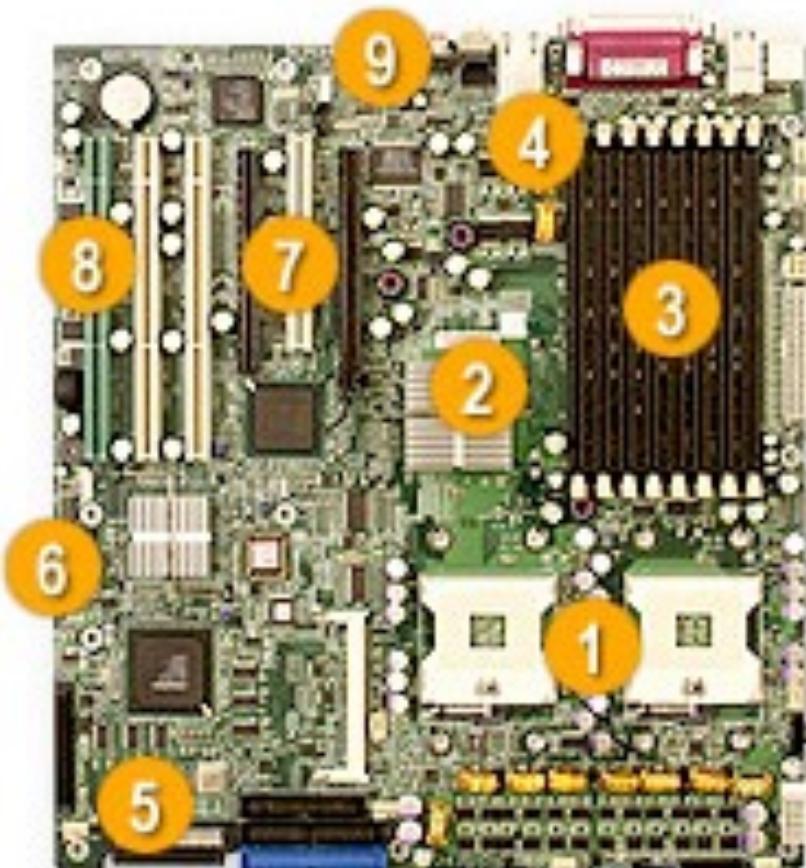
# Parallel Computing Hardware



- **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

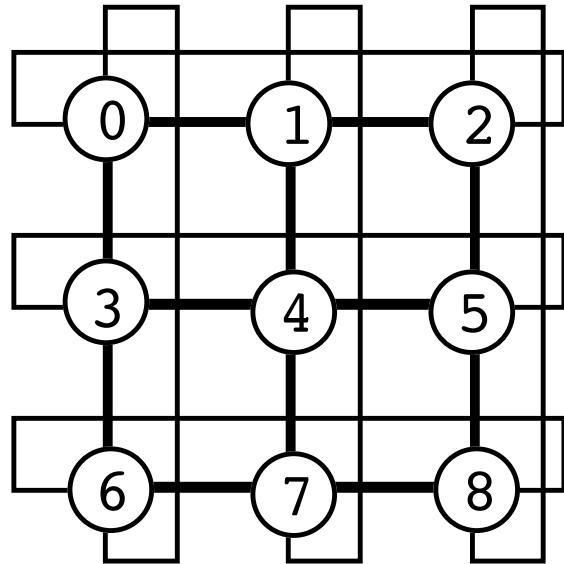


1. Dual Intel® Xeon™ EM64T Support up to 3.60 GHz
2. Intel® E7525 (Tumwater) Chipset
3. Up to 16GB DDRII-400 SDRAM
4. Intel® 82546GB Dual-port Gigabit Ethernet Controller
5. Adaptec AIC-7902 Dual Channel Ultra320 SCSI
6. 2x SATA Ports via ICH5R SATA Controller
7. 1 (x16) & 1 (x4) PCI-Express,  
1 x 64-bit 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

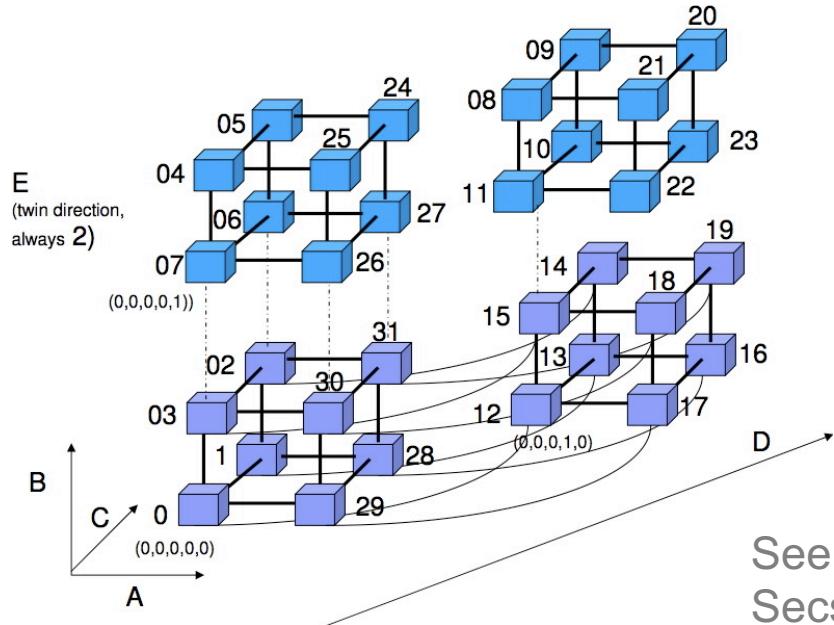
**Supermicro X6DA8-G2**

# Communication Network

Mesh  
(torus)



IBM Blue Gene/Q (5D torus)

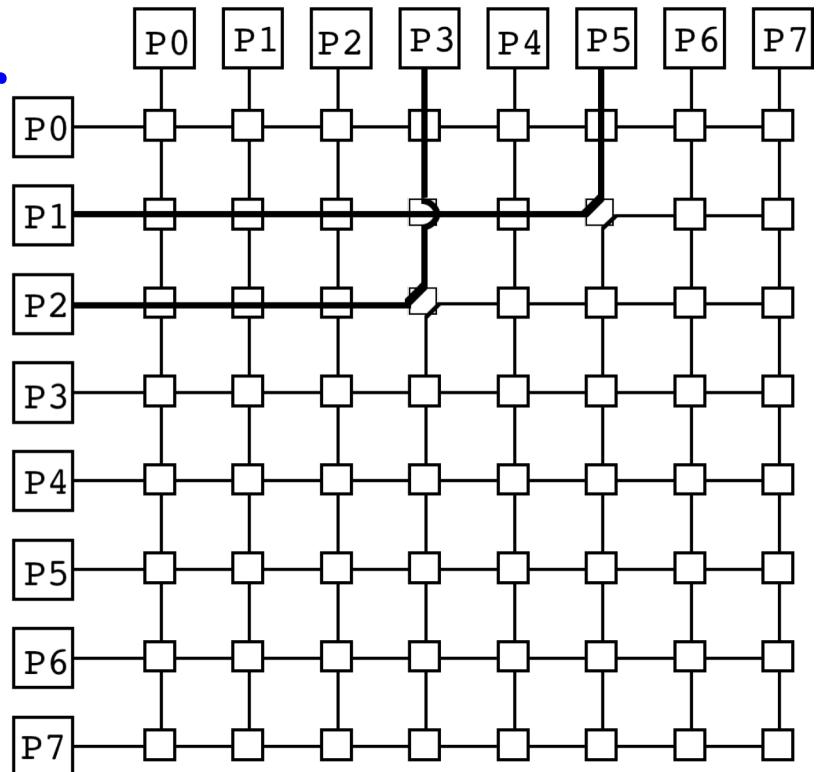


See [Gramma](#),  
Secs. 2.4.2-2.4.4

Crossbar  
switch



NEC Earth Simulator (640x640 crossbar)



# 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

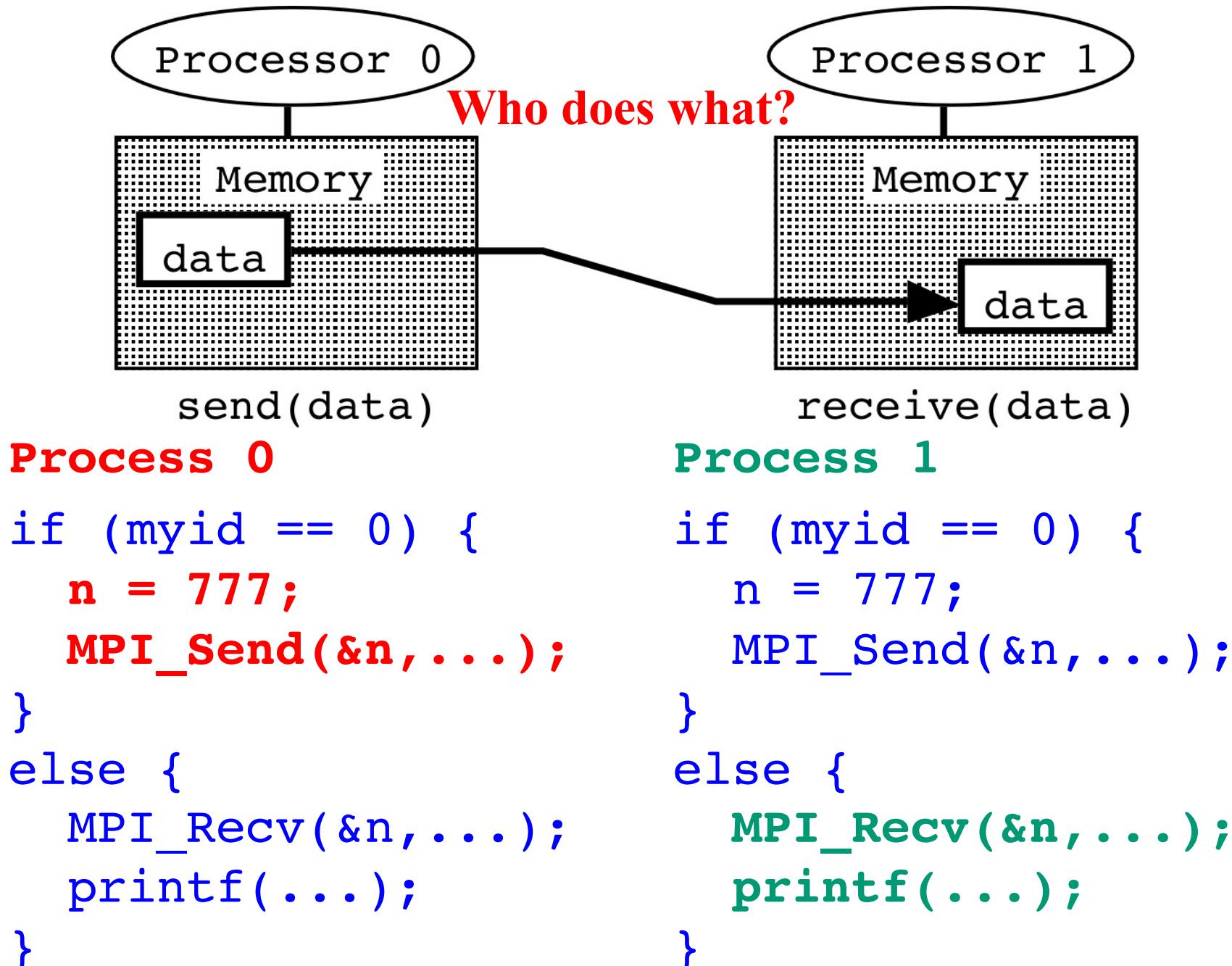
## mpi\_simple.c: Point-to-point message send & receive

```
#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;
}
```

The diagram illustrates the MPI point-to-point message exchange between MPI ranks P0 and P1. It shows two MPI daemon components, one for each rank. Arrows indicate the flow of requests from the MPI code to the daemons and the responses from the daemons back to the MPI code.

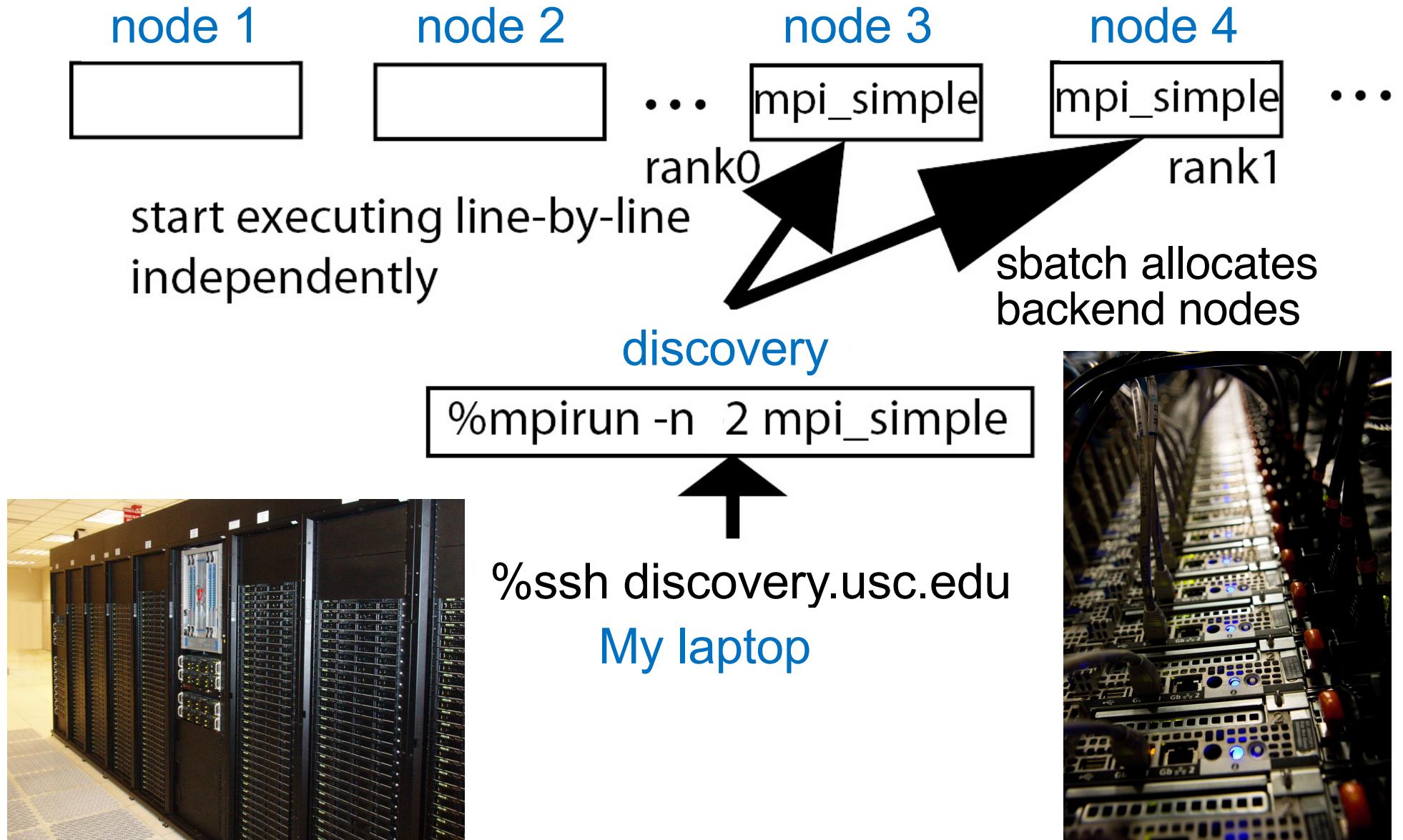
- MPI rank:** Labels the MPI ranks P0 and P1.
- Matching message tags:** Labels the tag values 10 and 10 used in the MPI\_Send and MPI\_Recv calls.
- Data triplet:** Labels the data being sent/received, which is the variable `n`.
- To/from whom:** Labels the destination (rank 1 for P0, rank 0 for P1) and source (rank 0 for P0, rank 1 for P1) of the message.
- MPI daemon:** Labels the two MPI daemon components, one for P0 and one for P1.
- send to 1** and **recv from 0**: Arrows indicating the direction of the message exchange between the MPI code and the MPI daemon.

# Single Program Multiple Data (SPMD)



# Single Program Multiple Data (SPMD)

What really happens?



# 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);
```

The diagram illustrates the breakdown of MPI communication parameters into three categories: Data triplet, To/from whom, and Information. Brackets under the code group the parameters into these categories. The first two parameters of each call (the data type and its count) are grouped under 'Data triplet'. The next two parameters (the destination rank and source rank) are grouped under 'To/from whom'. The final parameter of each call (the status variable) is grouped under 'Information'.

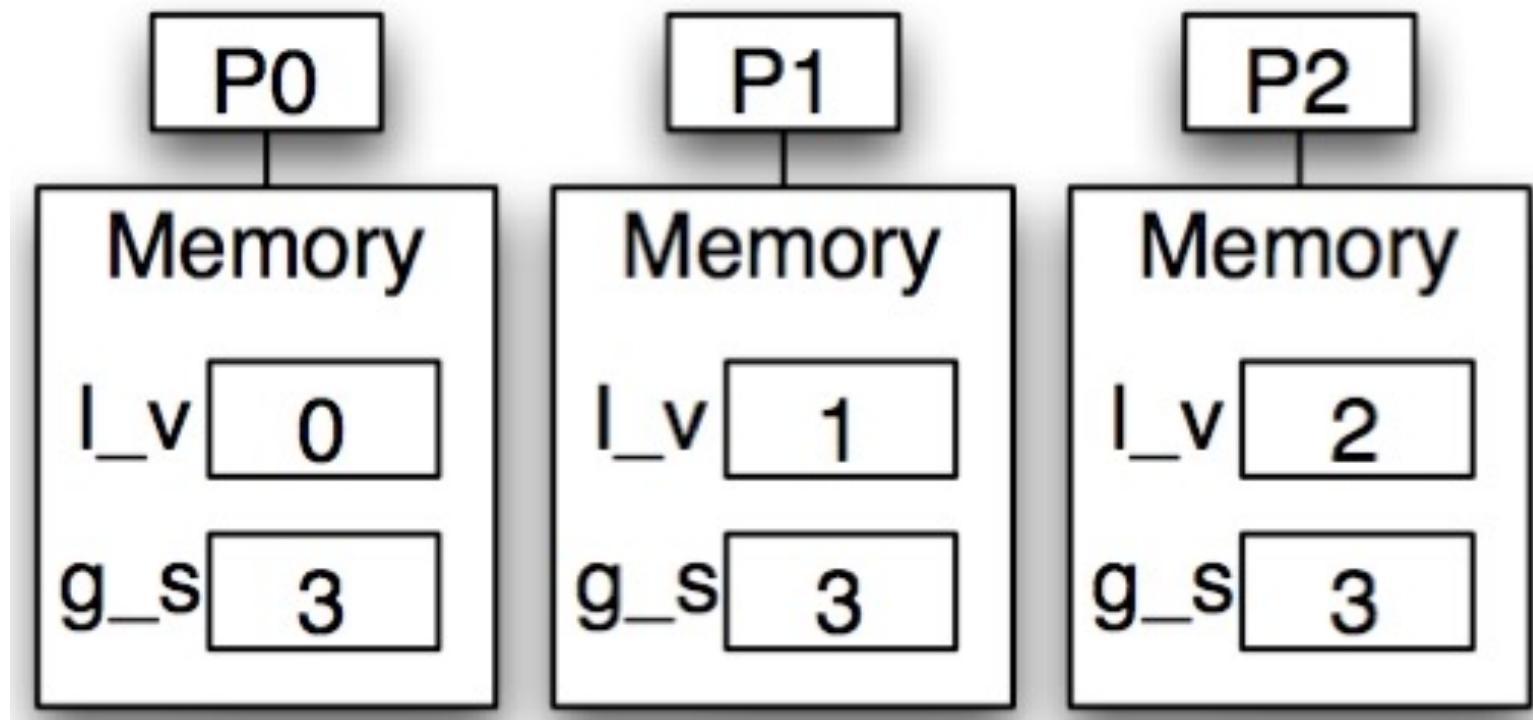
Data triplet                  To/from whom                  Information

# 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.

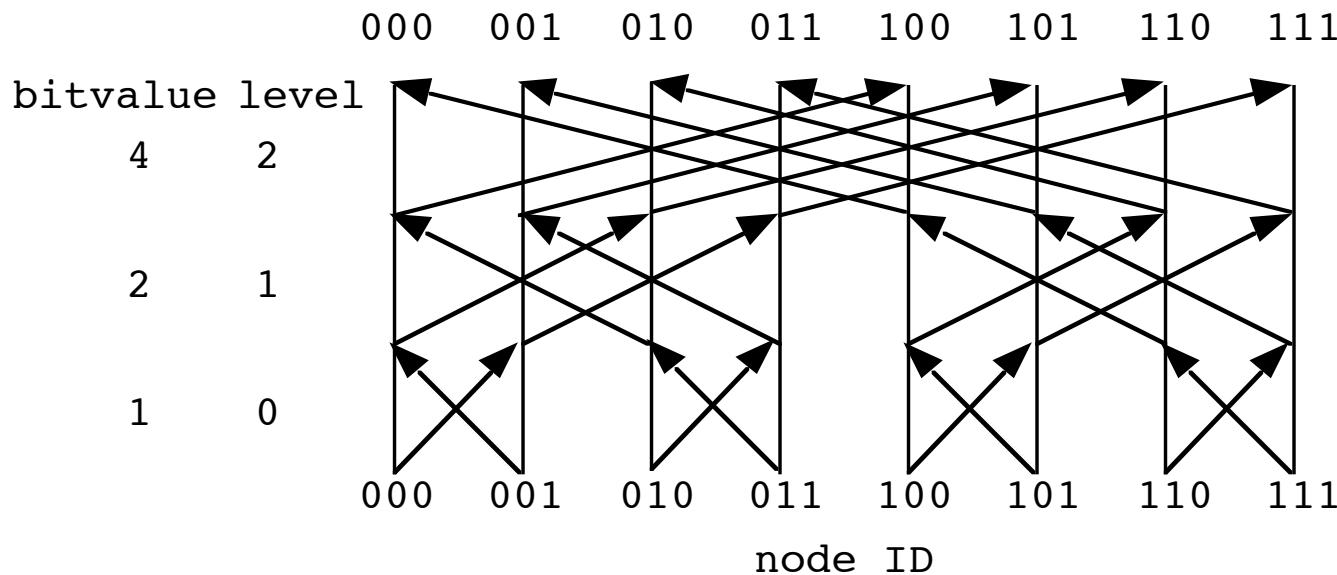
```
MPI_Allreduce(&local_value, &global_sum, 1, MPI_INT, MPI_SUM,  
MPI_COMM_WORLD)
```

```
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.



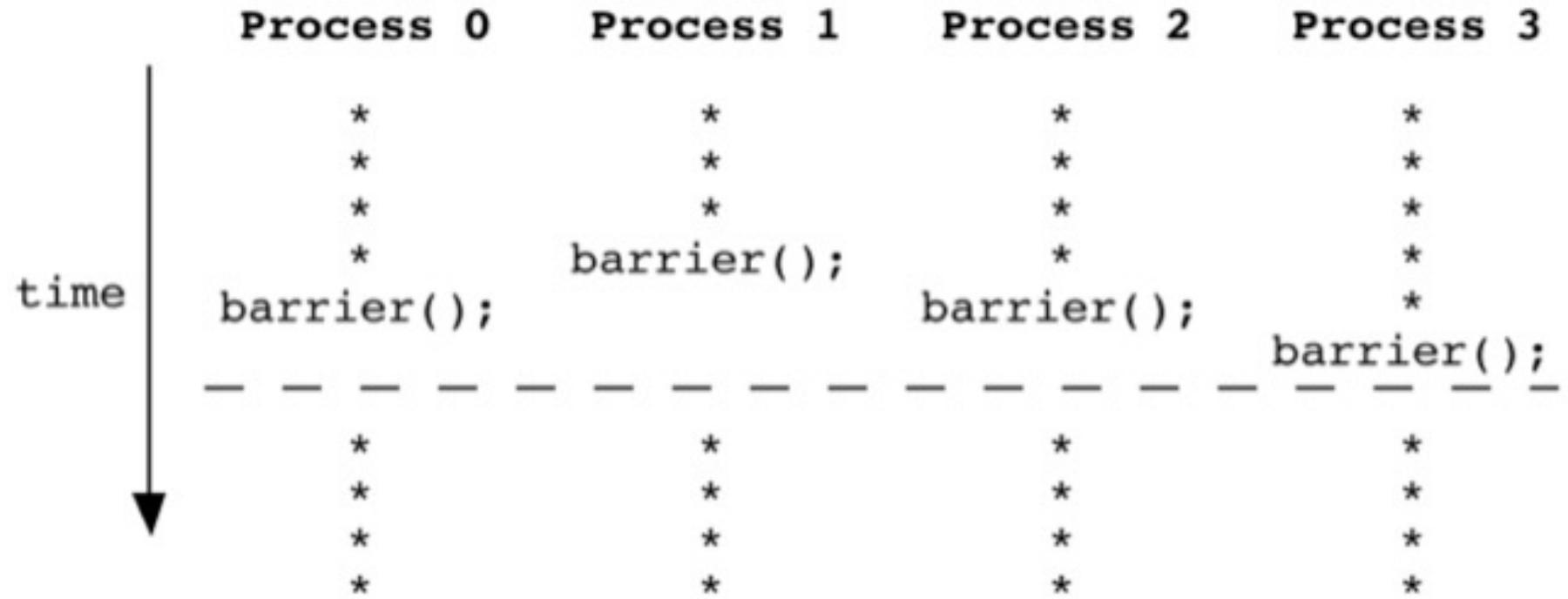
Butterfly network

$$\begin{aligned} & a_{000} + a_{001} + a_{010} + a_{011} + a_{100} + a_{101} + a_{110} + a_{111} \\ = & ((a_{000} + a_{001}) + (a_{010} + a_{011})) \\ + & ((a_{100} + a_{101}) + (a_{110} + a_{111})) \end{aligned}$$

(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()`

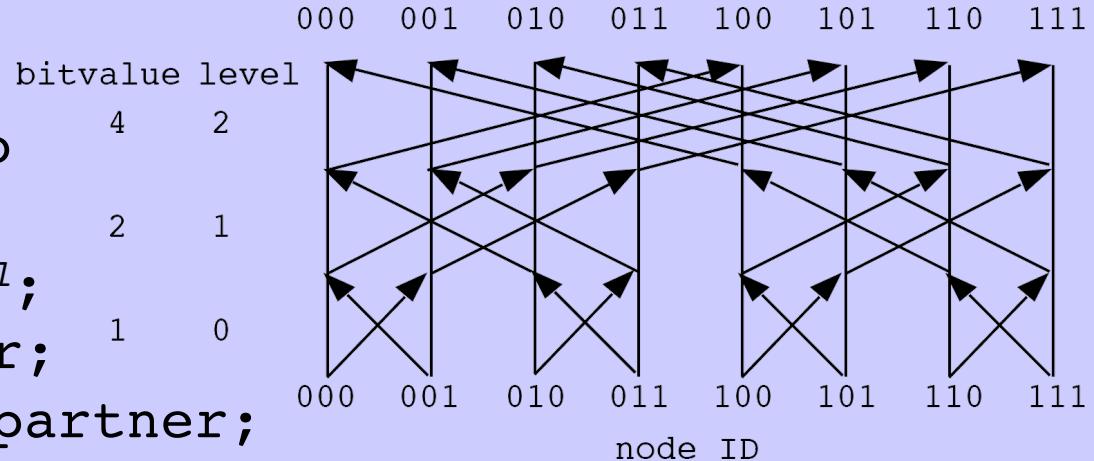
`MPI_Bcast()`

# Hypercube Template

```

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

```



level	$\frac{1}{2^1}$	bitvalue
0	0	001
1	1	010
2		100

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

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

$$(a \text{ OP } b) \text{ OP } c = a \text{ OP } (b \text{ OP } c)$$

$$abcdefg \text{ XOR } 0000100 = abcde\bar{f}g$$

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); Who am I?
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs); How big is the world? (see
    partial = (double) myid; p. 5 in MPI lecture note)
    printf("Rank %d has %le\n", myid, partial);
    sum = global_sum(partial);
    if (myid == 0) {
        avg = sum/nprocs;
        printf("Global average = %d\n", avg);
    }
    MPI_Finalize();
    return 0;
}
```

Global variables are visible in both *global\_sum()* & *main()*

# C Implementation of global\_sum()

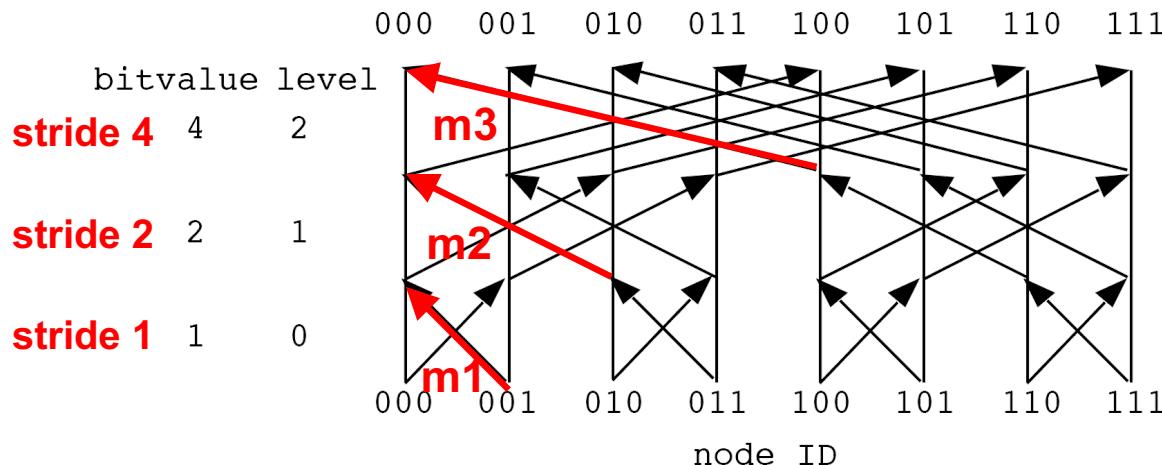
```
double mydone = partial;
for (int bitvalue=1; bitvalue<nprocs; bitvalue*=2)
{
    int partner = myid ^ bitvalue;
    send mydone to partner;
    receive hisdone from partner;
    mydone = mydone + hisdone;
}
return mydone;
```

Multiplied by 2

level	$\frac{1}{2^1}$	bitvalue
0	001	1
1	010	2
2	100	4

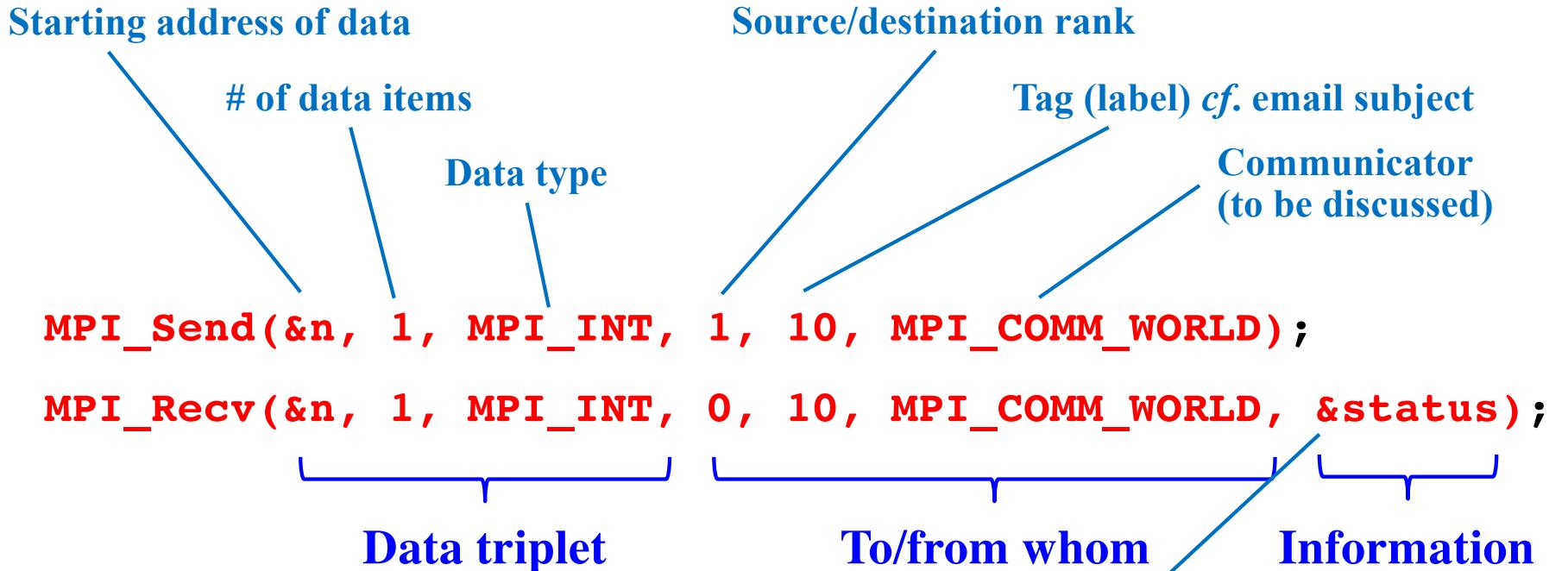
Implement with MPI\_Send() & MPI\_Recv()

Use *bitvalue* as counter & bitmask



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

# MPI Send & Receive Revisited



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      mpicc -o global_avg global_avg.c

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

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\\_\(communication pattern\)](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



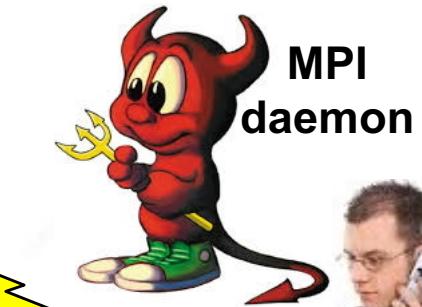
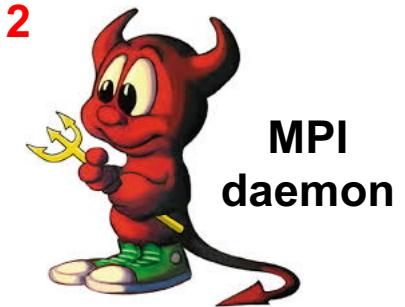
node 1



Who does what?  
No synchronization!



node 2



node 4

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

# 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	USA	1	100	100
2	Stanford University	USA	2	72.1	41.8
3	Massachusetts Institute of Technology (MIT)	USA	3	70.5	68.4
4	University of California-Berkeley	USA	4	70.1	66.8
5	University of Cambridge	UK	1	69.2	79.1

51	University of Southern California	USA	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

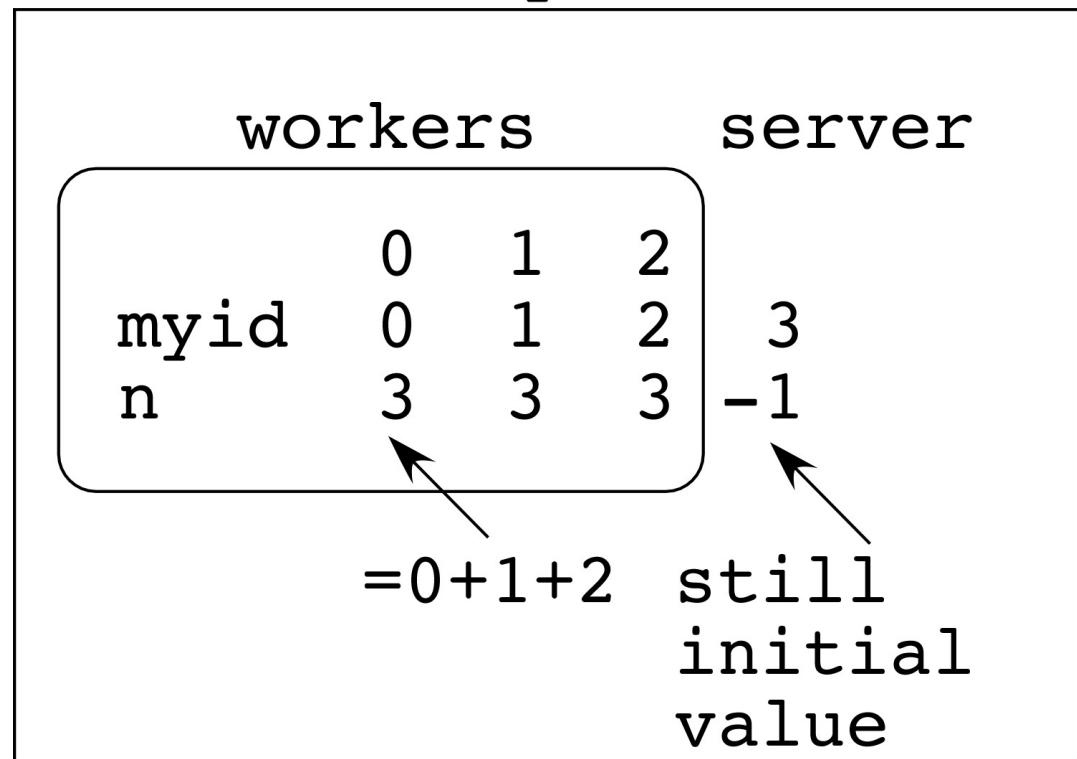
**Slurm script**

```
#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
```

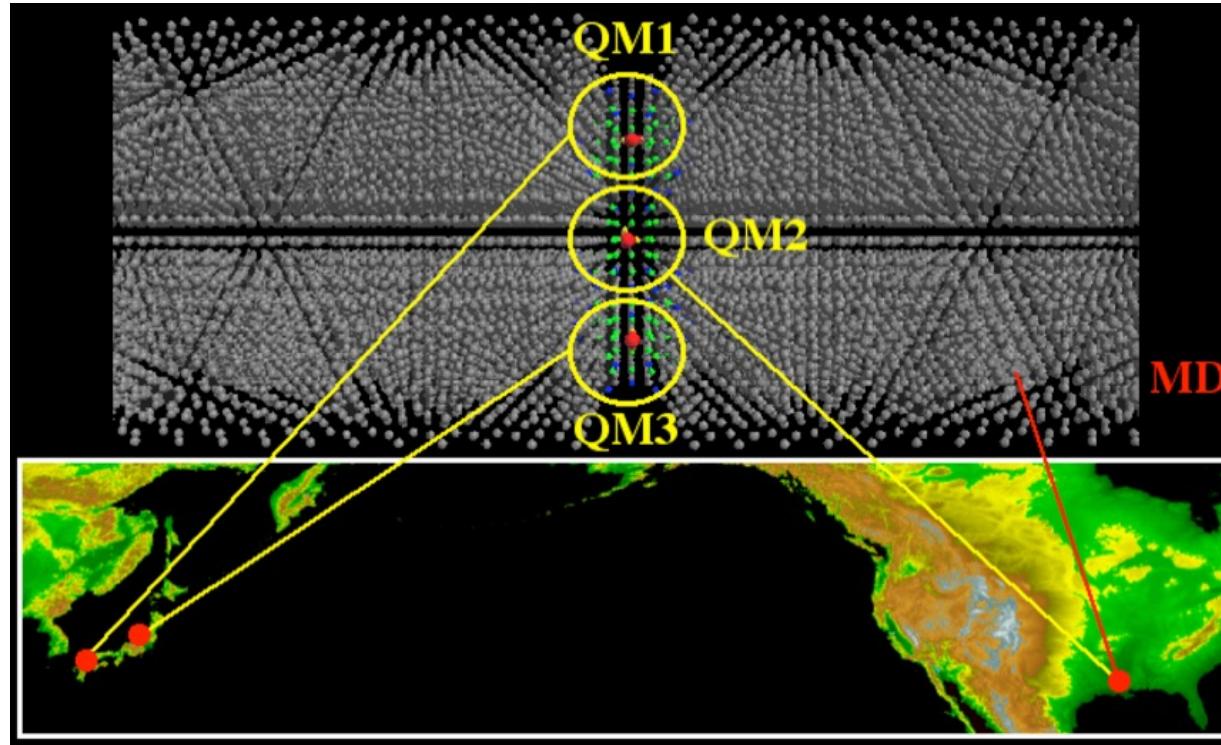
world: nprocs = 4

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

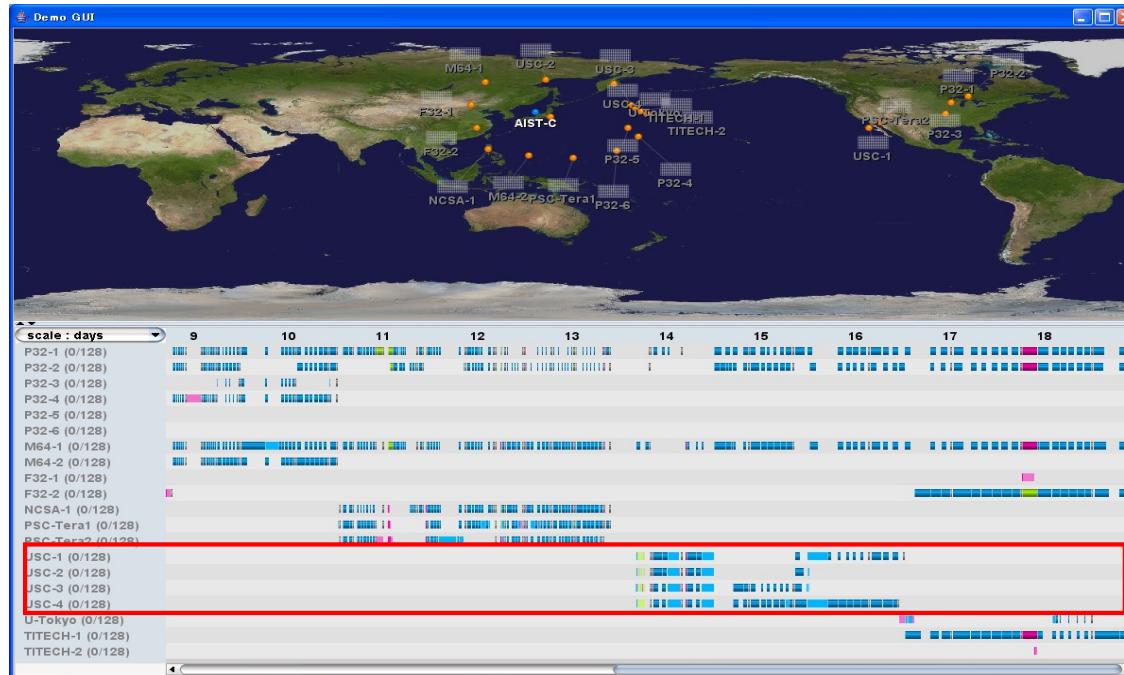


**Communicator = a nice migration path to distributed computing**

- 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

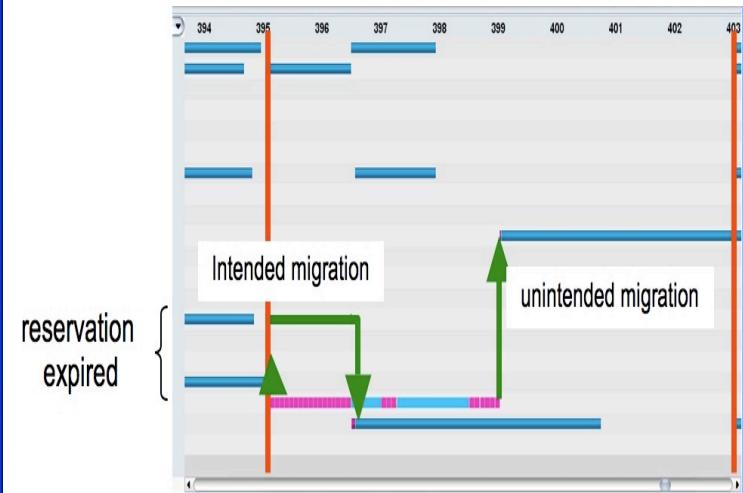
# Global Grid QM/MD

- *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)*



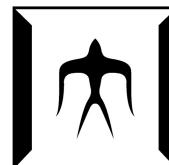
USC

Automated  
resource migration  
& fault recovery



東京大学  
THE UNIVERSITY OF TOKYO

**AIST**



**SC06**  
POWERFUL BEYOND IMAGINATION

**USC**

**NCSA**

PITTSBURGH  
SUPERCOMPUTING  
CENTER

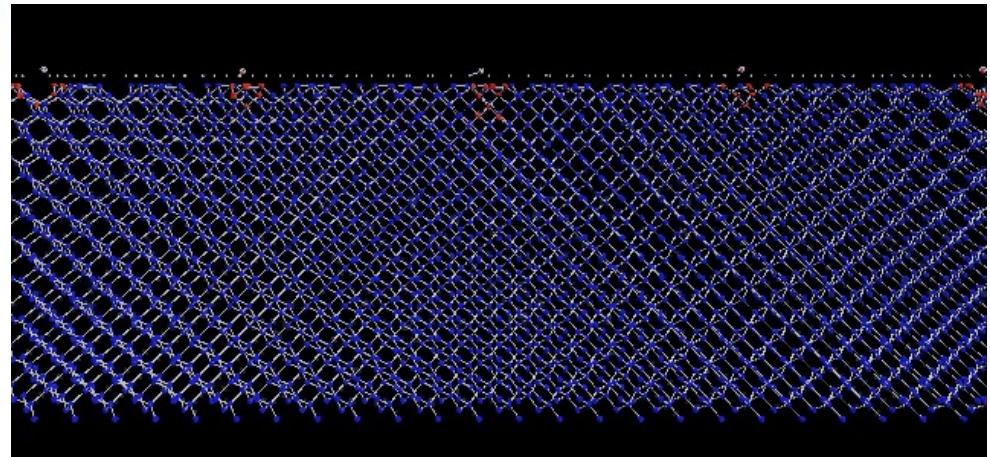
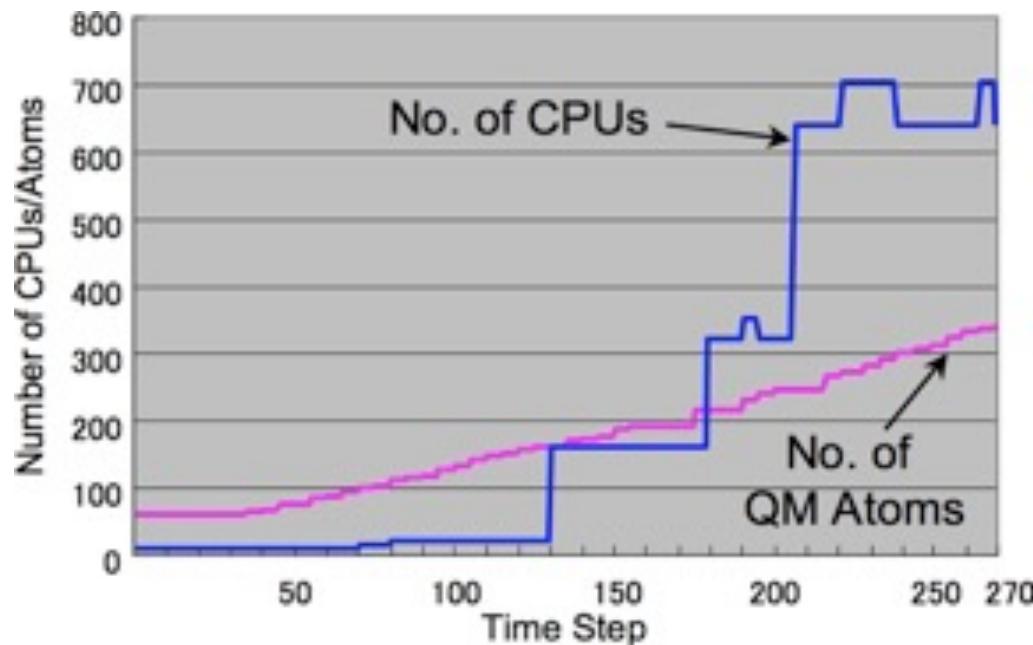
Takemiya et al., "Sustainable adaptive Grid supercomputing: multiscale simulation of semiconductor processing across the Pacific," IEEE/ACM SC06

# 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



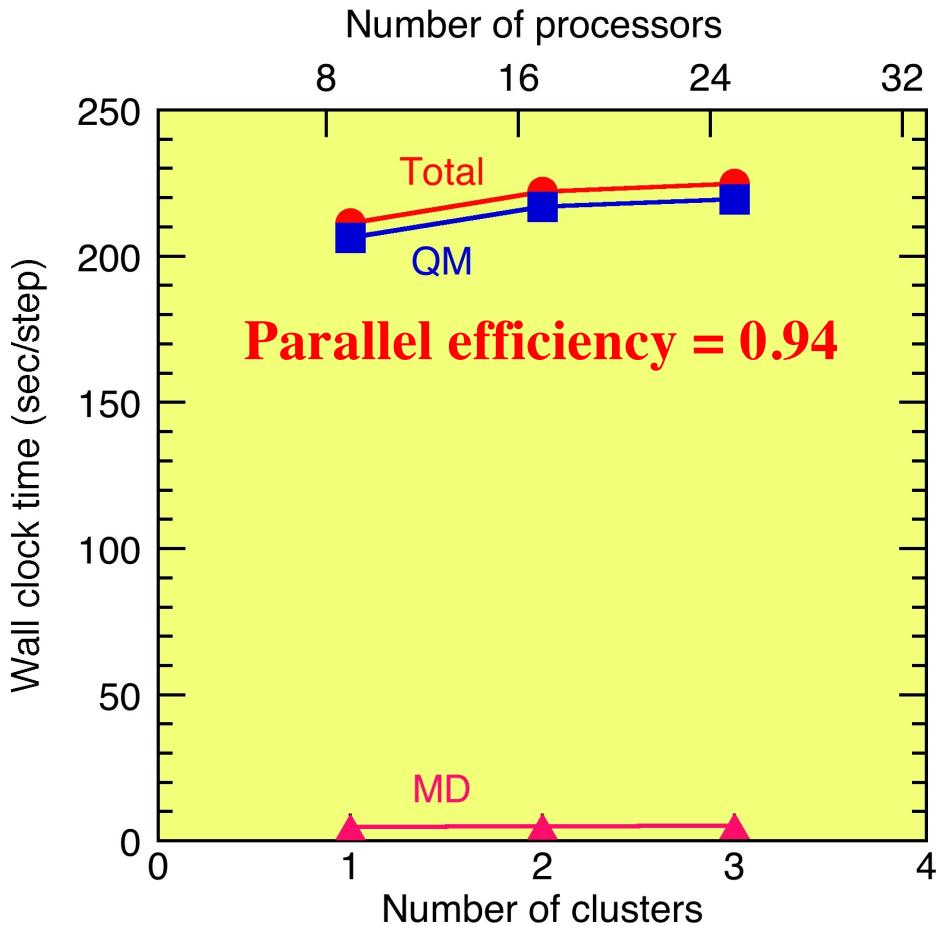
Ninf-G GridRPC: [ninf.apgrid.org](http://ninf.apgrid.org); MPICH: [www.mcs.anl.gov/mpi](http://www.mcs.anl.gov/mpi)



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"



$$\frac{\text{Earth's circumference}}{\text{Light speed}} = \frac{40,000 \text{ [km]}}{3 \times 10^8 \text{ [m/s]}} = 4 \times 10^7 \text{ [m]} = 0.1 \text{ s} = 100 \text{ ms}$$

**Try on Discovery:**  
traceroute [www.u-tokyo.ac.jp](http://www.u-tokyo.ac.jp)  
**vs.** ping [hpc-transfer.usc.edu](http://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

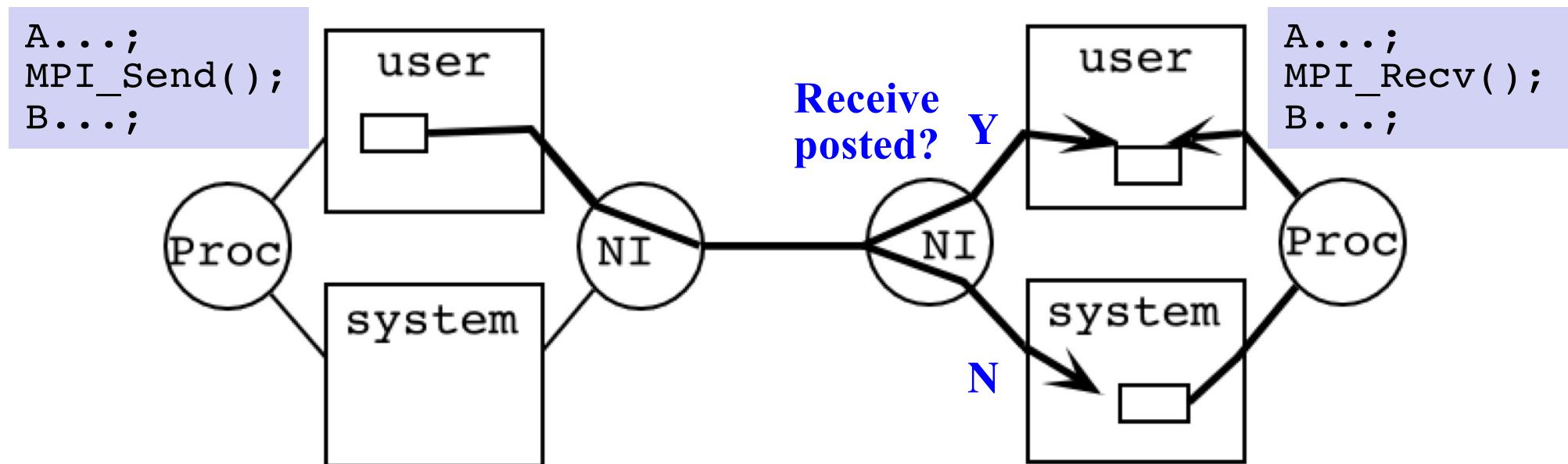
- Safe to modify original data immediately on return
- Depending on implementation, it may return whether or not a matching receive has been posted, or it may block (especially if no buffer space available)

**`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

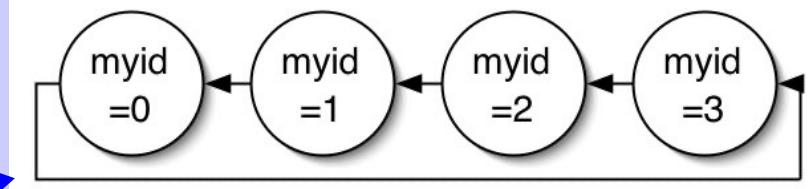
```
A...;  
MPI_Isend();  
B...;  
MPI_Wait();  
C...; // Reuse the send buffer
```

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

# 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;
}
```

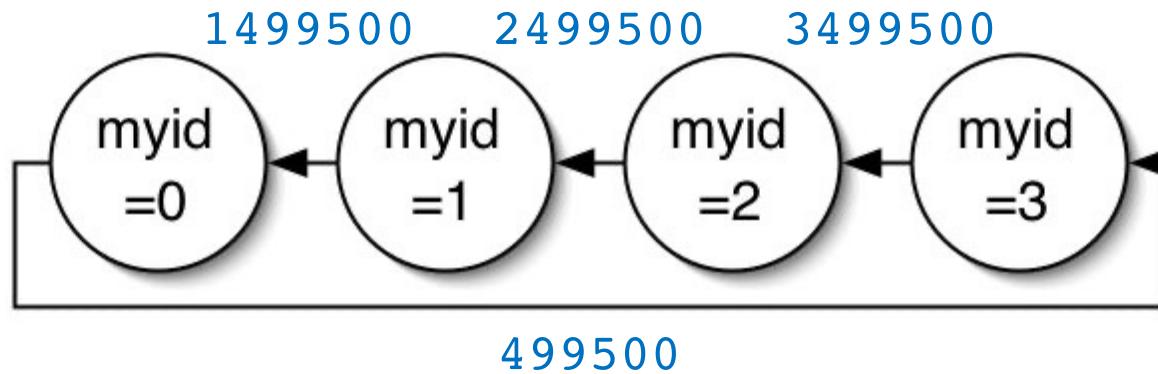
Wrap-around/torus  
via modulo (%) operator  
(cf. periodic boundary condition)



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
```

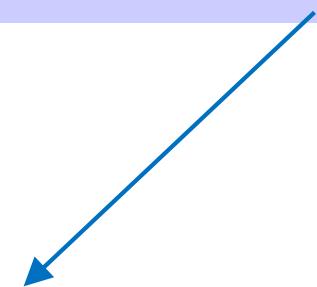


# Multiple Asynchronous Messages

```
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 ( $\in [0, N\_message-1]$ ) of the message that completed

# Polling MPI\_Irecv

---

```
int flag;

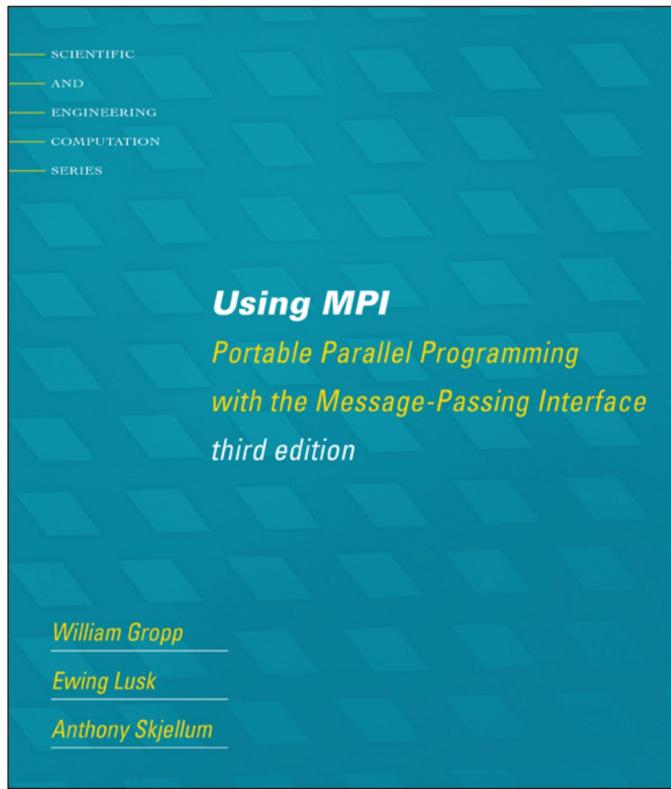
/* Post an asynchronous receive */
MPI_Irecv(recv_buf, N, MPI_INT, MPI_ANY_SOURCE, 777,
          MPI_COMM_WORLD, &request);

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

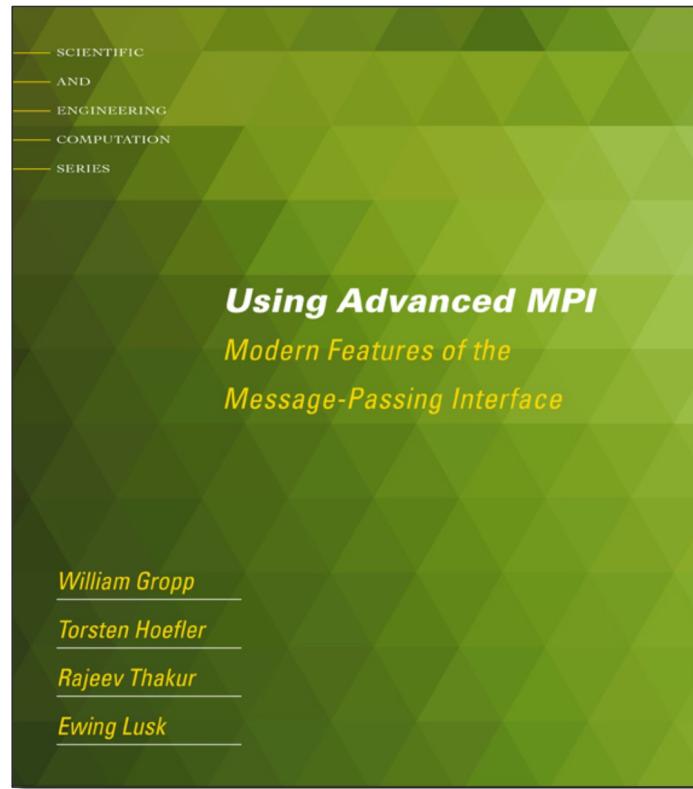
/* Polling */
MPI_Test(&request, &flag, &status); // Check completion
if (flag) { // True if message received
    /* Now it's safe to use recv_buf */
    ...
}
```

# 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

---

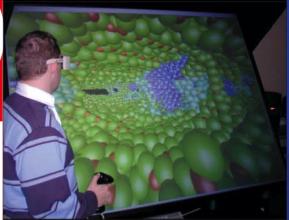
- Parallel computing = Who does what
- Single program multiple data (SPMD) programming: Do it with MPI rank (who am I) & selection constructs (*if, etc.*)
- Only need `MPI_Send()` & `MPI_Recv()` within communicators to implement any distributed-memory parallel computing
- Asynchronous message passing (`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!

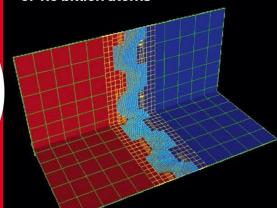
# 20 Years-Unleashing the Power of HPC

# SC2001

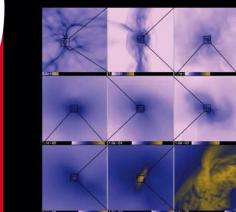
2001 Chair  
Charles Slocomb  
Denver, CO



A discrete particle simulation of 1.5 billion atoms



Adaptive mesh simulation of advecting sinusoidal density contours



Adaptive mesh simulation of star formation



The MDM system

Adaptive mesh simulation of a spherical shock

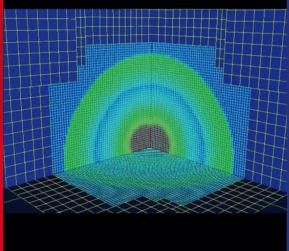


Figure 6: Real part of an example radial function. In this example  $\lambda = 1$ , so the radial function is symmetric about  $r_1 = r_2$ .

Solution of a three body quantum mechanics problem

2001 Chair  
Charles Slocomb  
Denver, CO



## 2001

**Notable Systems first mentioned this year in the proceedings:**

- SGI Origin 3000
- Sun Fire 6000
- ASCI White
- Blue Horizon
- ASCI Blue Mountain

**Notable Processors:**

- MIPS R 12000
- Intel Pentium 4
- Intel Itanium



A WINE-2 system board

**Noteworthy Architecture Topics:**

- Cache coherence through snooping
- Application speedups through custom on-the-fly FPGA function units
- Interactive program steering
- Grid-enabled parallel computing

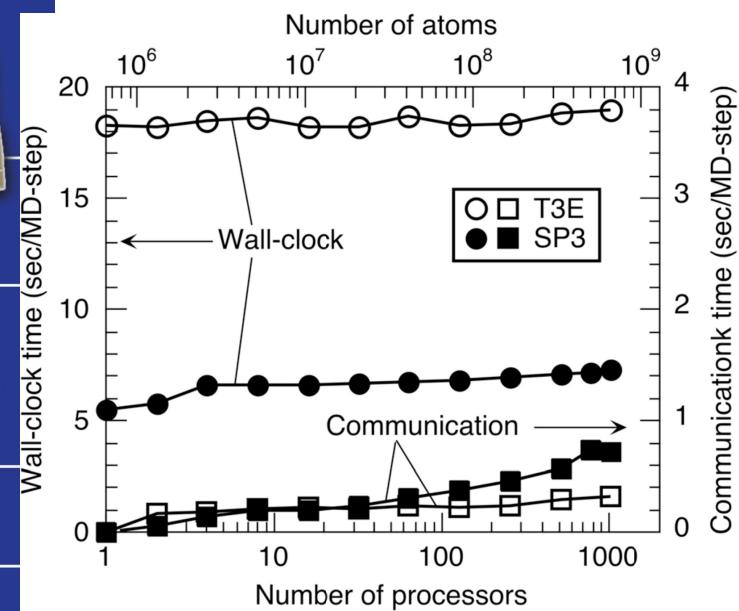
**Notable Programming Languages:**

- HDL
- PThreads

**Research Machines:**

- CPlant

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



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