

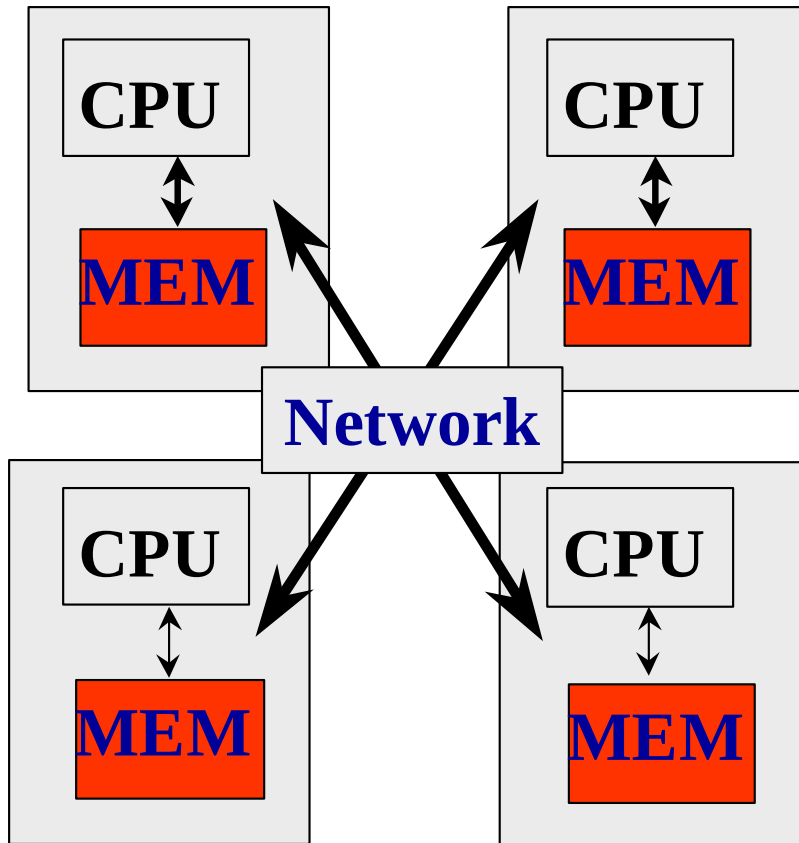
Programming for High Performance Computing

Programming Environment

Dec 22, 2011

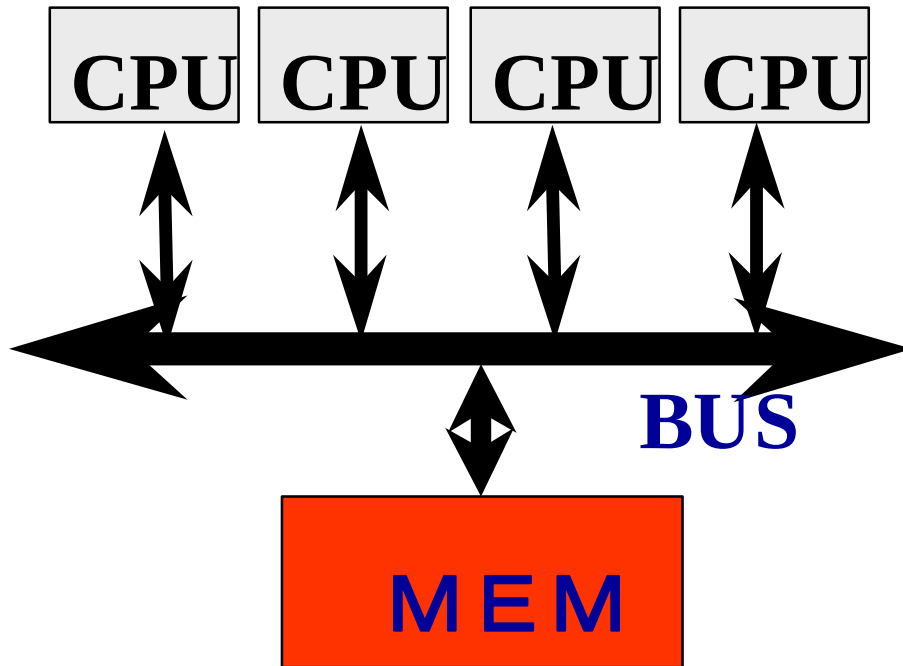
Osamu Tatebe

Distributed Memory Machine



- ◆ Parallel machine that consists computers (CPU and memory) connected by a network
- ◆ Parallel program is executed on each machine, communicating data (message) by the network
- ◆ MPP (Massively Parallel Processor)
 - ◆ Using proprietary technology
- ◆ PC Cluster
 - ◆ Using commodity technology

Shared Memory Machine



◆ Parallel machine such that CPUs access the same memory

◆ Parallel program is executed by accessing the data on the shared memory to communicate each other

◆ Large scale server, multicore CPU

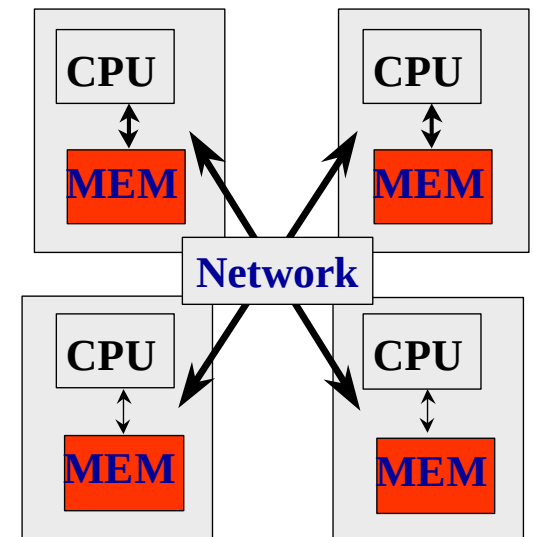
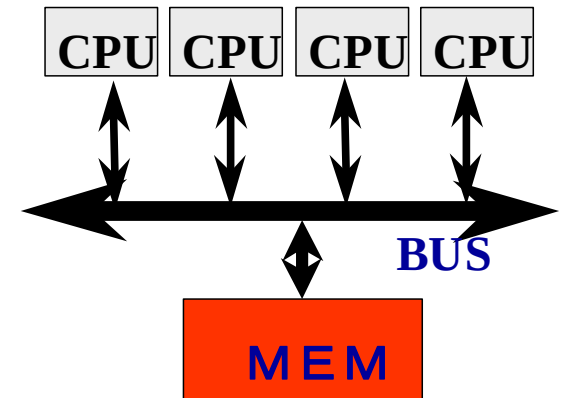
◆ UMA (uniform memory access) and NUMA (non-uniform memory access)

Benefit of parallel processing

- **Increase computational power**
 - More than one CPU
- **Increase memory BW**
 - Each CPU can access each memory
- **Increase I/O BW**
 - Each CPU can access each storage in parallel
- **Increase cache memory size**
 - Even though the data size does not fit in the cache size in a single processor, the cache memory can be efficiently used by dividing the data in the problem
- **Low cost**
 - When using microprocessors



PC Cluster Technology



Parallel Programming (1)

- Message passing programming
 - For Distributed memory machines
 - Can be used for shared memory machines
 - Complicated and rather difficult
 - Data transfer should be programmed
 - Scalable in terms of # processors
- Shared memory programming
 - For shared memory machines
 - Can be used for distributed shared memory (DSM) system on distributed memory machines
 - Easy to program (incremental from serial program)
 - Data transfer is taken care by the system
 - Often not scalable in terms of # processors

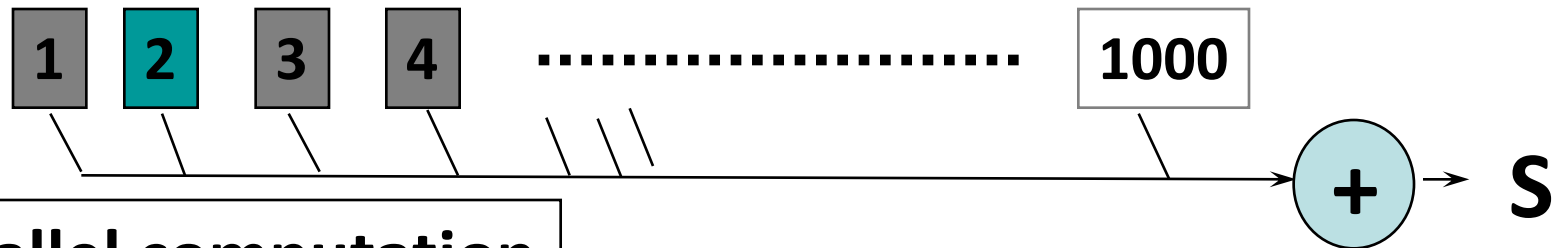
Parallel Programming (2)

- Message passing programming
 - MPI, PVM
- Shared memory programming
 - Multithread programming
 - Pthread, Solaris thread, NT thread
 - OpenMP
 - Annotation for parallel execution
 - Thread control, ... for shared memory machines
 - HPF
 - Annotation for parallel execution
 - Parallel construct
 - Distribution of array, ... for distributed memory machines
- Automatic parallelization
 - Parallelize serial programs by compiler
 - Parallelization analysis by compiler is limited
 - Sometimes hint is required for parallelization by annotation
- Fancy parallel programming languages

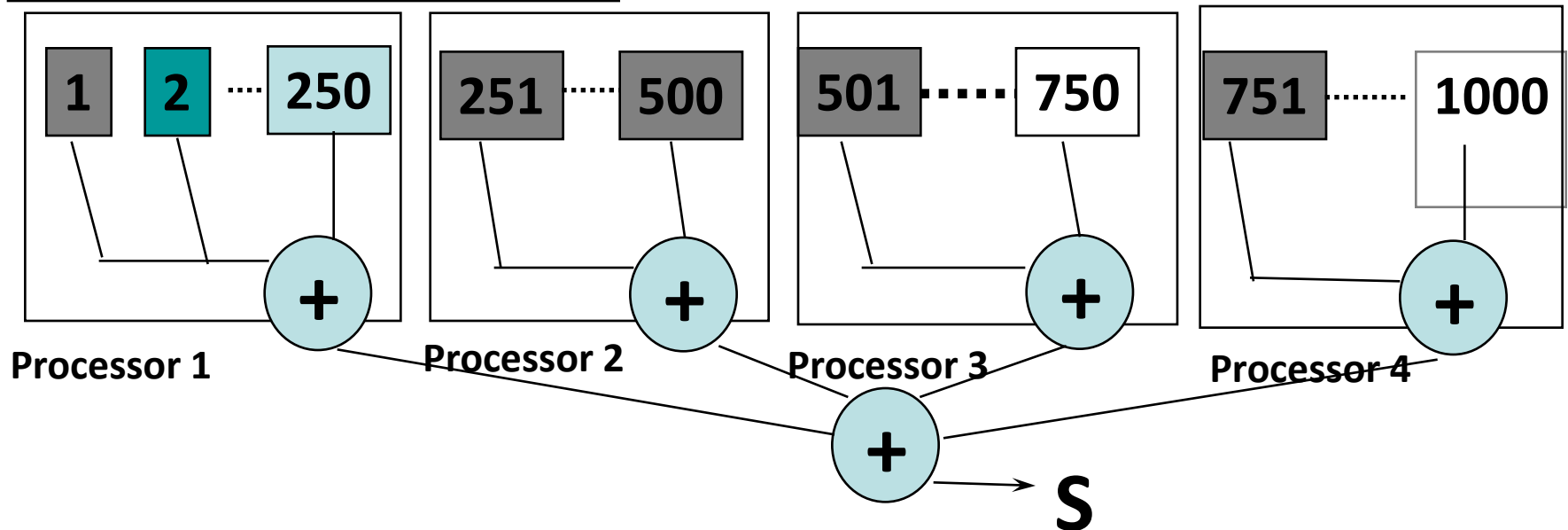
Example of parallel computing

for (i = 0; i < 1000; i++)
S += A[i]

Serial computation



Parallel computation



Multithread Programming

- Thread creation

Pthread, Solaris thread

```
for (t = 1; t < n_thd; t++){  
    r = pthread_create(&thd_main, t)  
}  
thd_main(0);  
for (t = 1; t < n_thd; t++){  
    pthread_join();  
}
```

- Divide the summation loop
- Atomic addition

```
double s; /* global */  
int n_thd; /* number of threads */  
int thd_main(int id)  
{ int c, b, e, i; double ss;  
  c = 1000 / n_thd;  
  b = c * id;  
  e = b + c;  
  ss = 0.0;  
  for (i = b; i < e; i++) ss += a[i];  
  pthread_lock();  
  s += ss;  
  pthread_unlock();  
  return (0);  
}
```


Programming in OpenMP

Just it is!

```
#pragma omp parallel for reduction(+:s)  
for (i = 0; i < 1000; i++) s += a[i];
```

About OpenMP

- Parallel programming model for shared memory machine
 - Extend the base languages (Fortran/C/C++) by using directive
- Specification discussed internationally (mainly US compiler vendors)
 - Oct. 1997 Fortran ver.1.0 API
 - Oct. 1998 C/C++ ver.1.0 API
 - July 2011 OpenMP 3.1
- URL
 - <http://www.openmp.org/>

Programming in MPI

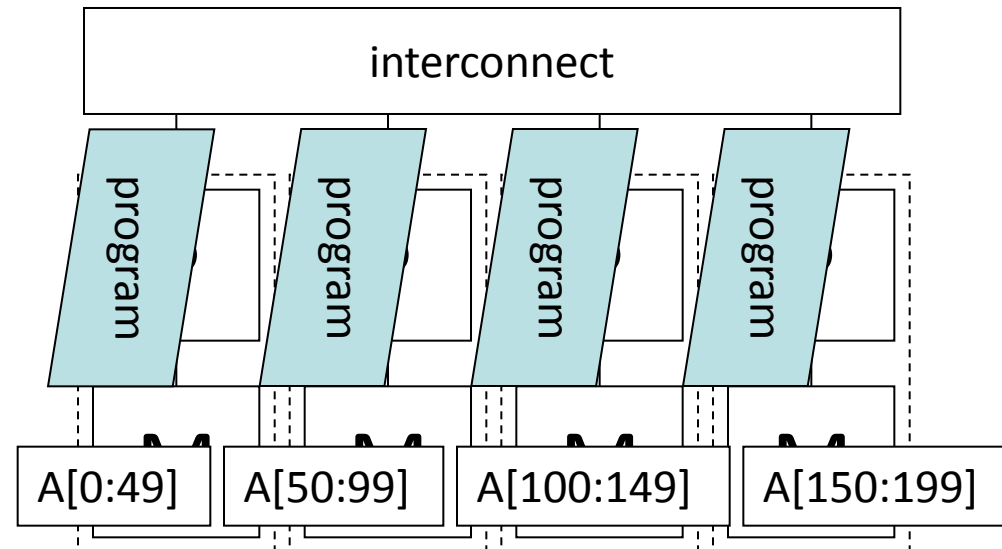
- MPI (Message Passing Interface)
- Standard programming model for distributed memory machines
 - Required for more than 100 nodes
 - Complicated but required for performance
 - Like assembly language
- Message passing (point-to-point communication)
 - Send/Receive
- Collective communication
 - Summation, ...

MPI – The Message Passing Interface

- Standard of message passing interface
- MPI-1.0 released in 1994
 - Portable parallel library, application
 - 8 communication modes, collective communication, communication domain, process topology
 - Defined more than 100 interfaces
 - C, C++, Fortran
 - Specification <http://www.mpi-forum.org/>
 - MPI-2.2 released in September, 2009
 - MPI-3 discussed
 - Japanese translation <http://phase.hpcc.jp/phase/mpi-j/ml/>

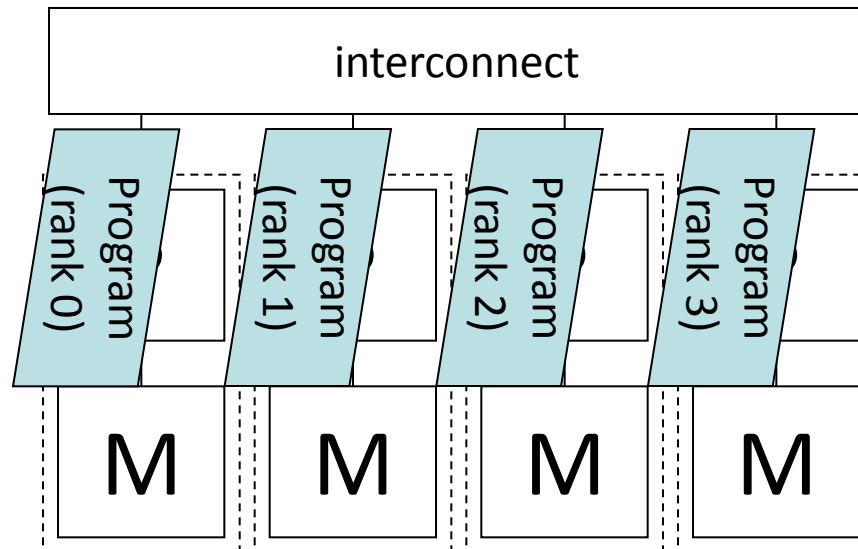
SPMD – Single Program, Multiple Data

- Parallel execution of the same single program independently (cf. SIMD)
- The same program but processes different data
- Parallel program is interacted with each other by message exchange



MPI execution model

- Execute the same program on each processor
 - Execution is not synchronous (if no communication happens)
- Each process has its own process rank
- Each process is communicated in MPI



Communicator (1)

- Communication domain
 - Set of processes
 - # processes, process rank
 - Process topology
 - 1D ring, 2D mesh, torus, graph
- `MPI_COMM_WORLD`
 - Initial communicator including all processes

Communicator (2)

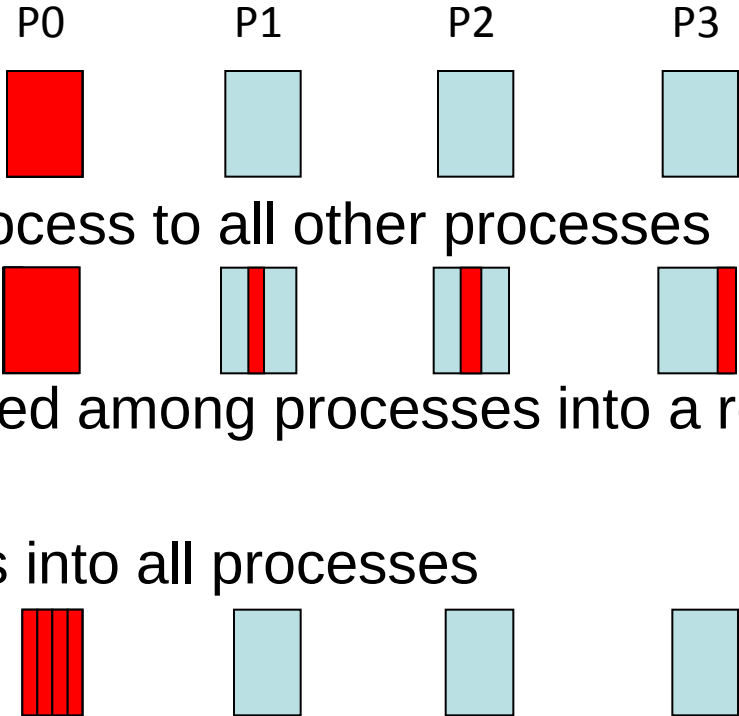
- “Scope” of collective communication (communication domain)
- Can divide set of processes
 - Two thirds of processes compute weather forecast, the rest one third compute the initial condition of the next iteration
- Intra-communicator and inter-communicator

Collective communication

- Message exchange among **all processes** specified by a communicator
- Barrier synchronization (no data transfer)
- Global communication
 - Broadcast, gather, scatter, gather to all, all-to-all scatter/gather
- Global reduction
 - Reduction (sum, maximum, logical and, ...), scan (prefix computation)

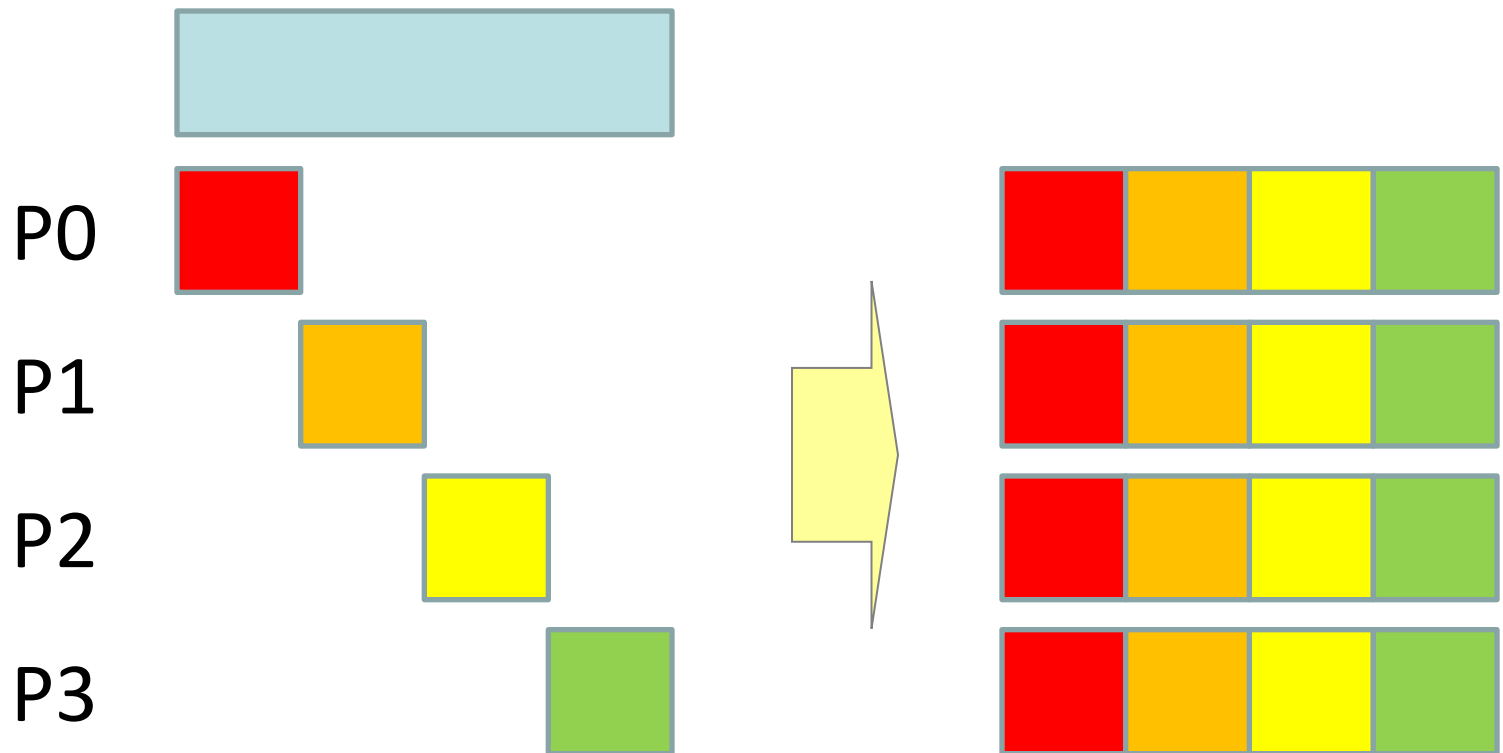
Global communication

- broadcast
 - Transfer $A[*]$ of the root process to all other processes
- gather
 - Gather sub arrays distributed among processes into a root process
 - Allgather gather sub arrays into all processes
- scatter
 - Scatter $A[*]$ of the root process to all processes
- Alltoall
 - Scatter/gather data from all processes to all processes
 - Distributed matrix transpose $A[:,*] \rightarrow A^T[:,*]$ (: means this dimension is distributed)



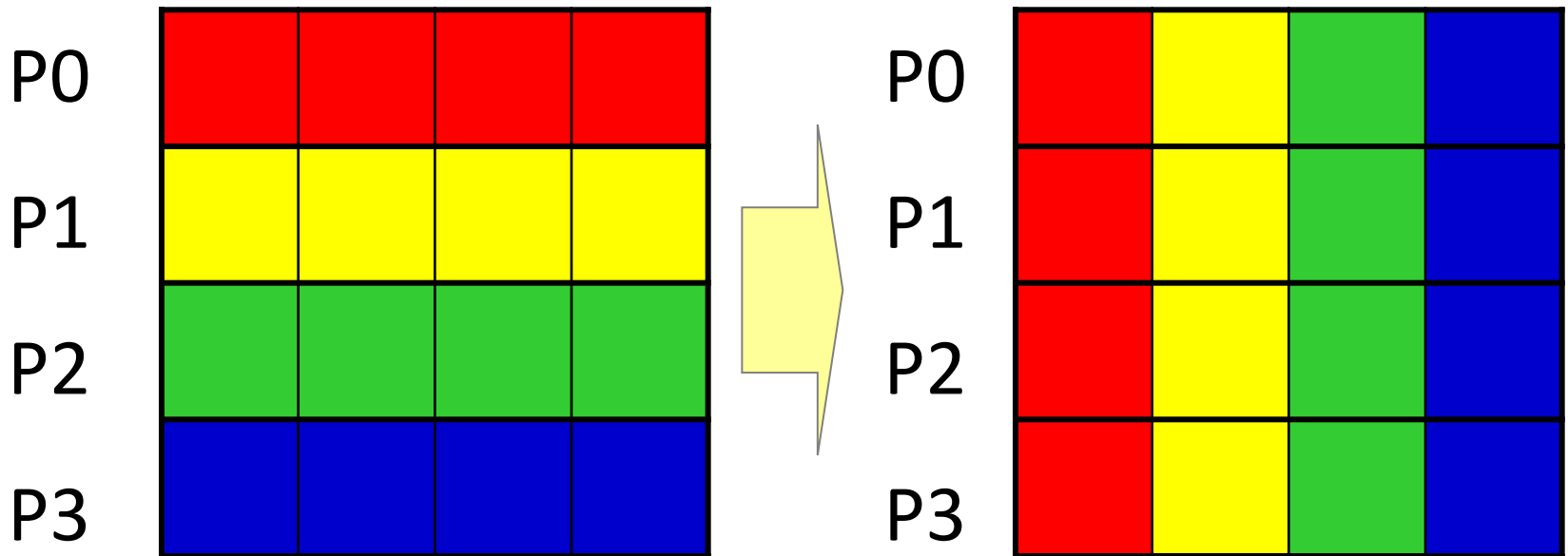
allgather

- Gather sub array of each process, and broadcast the whole array to all processes



alltoall

- Matrix transformation of (row-wise) distributed 2D array



Point-to-point communication (1)

- Data transfer among two process pair
 - Process A sends a data to process B (send)
 - Process B receives the data (from the process A) (recv)
- Data is typed
 - Basic data type, array, structure, vector, user-defined data type
- Send and the corresponding receive are specified by Communicator, message tag, process rank of source and destination

Point-to-point communication (2)

- Semantics of blocking communication
 - Send call returns when the send buffer can be reused
 - Receive call returns when the receive buffer is available
- When `MPI_Send(A, . . .)` returns, A can be safely modified
 - It may be that A is just copied into the communication buffer of the sender
 - It does not mean message transfer completion

Non-blocking point-to-point communication

- Nonblocking communication
 - post-send, complete-send
 - post-receive, complete-receive
- Post-{send,recv} initiates the send/receive operations
- Complete-{send,recv} waits for the completion
- It enables the overwrap of computation and communication to improve performance
 - Multithread programming also enables the overwrapping, but nonblocking communication often more efficient

Sample program (1): hostname

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char *argv[])
{
    int rank, len;
    char name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(name, &len);
    printf("%03d %s\n", rank, name);
    MPI_Finalize();
    return (0);
}
```


Explanation

- Include **mpi.h** to use MPI
- Each process executes main function
- SPMD (single program, multiple data)
 - A single program is executed on each node
 - Each program accesses different data (ie. data in their own running process)
- Initialize the MPI process
 - **MPI_Init**

Explanation (continued)

- Obtain the process rank
 - **MPI_Comm_rank**(MPI_COMM_WORLD, &rank);
 - Obtain the self rank in the communicator MPI_COMM_WORLD
 - Communicator is an opaque object. The information can be access by API
- Obtain the node name
 - **MPI_Get_processor_name**(name, &len);
- All process should finalize the MPI process
MPI_Finalize();

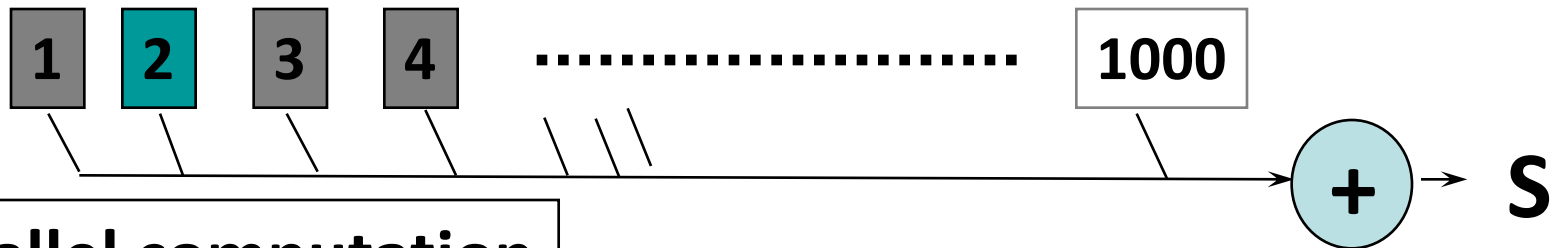
Operation for communicator

- int **MPI_Comm_size**(MPI_Comm *comm*, int **size*);
 - Returns the total number of processes *size* in the communicator *comm*
- int **MPI_Comm_rank**(MPI_Comm *comm*, int **rank*);
 - Returns the process rank *rank* in the communicator *comm*

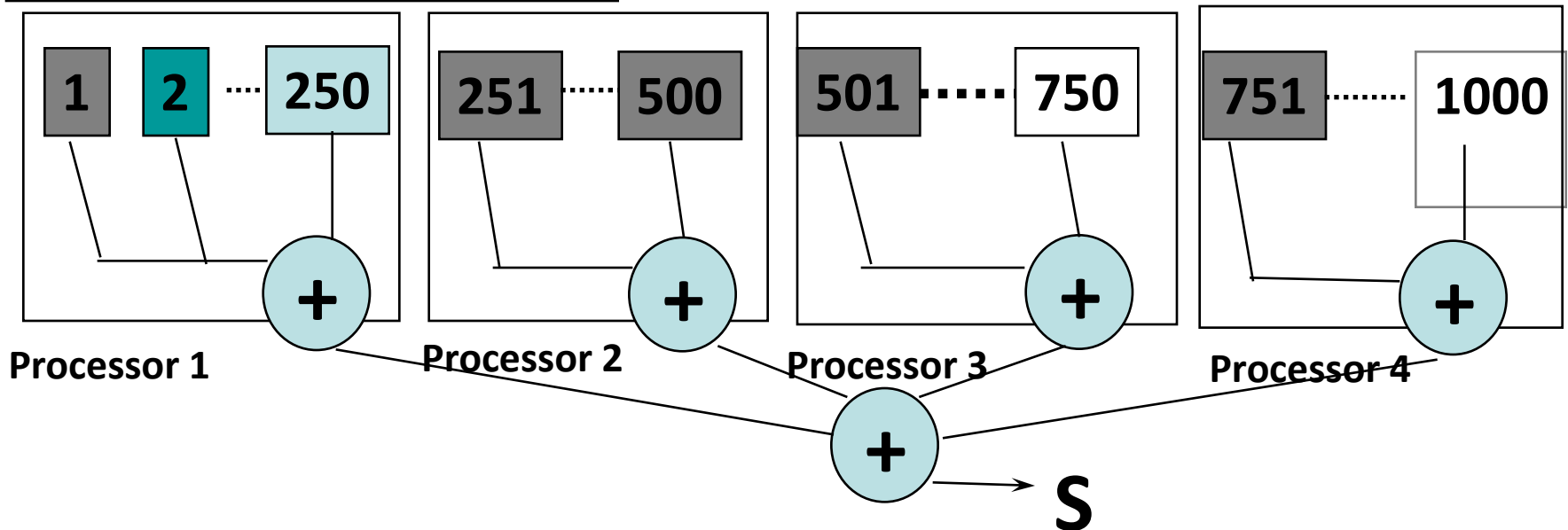
Sample program (2): summation

```
for (i = 0; i < 1000; i++)  
    S += A[i]
```

Serial computation



Parallel computation



```
#include <mpi.h>
```

```
double A[1000 / N_PE];
```

```
int main(int argc, char *argv[])  
{
```

```
    double sum, mysum;
```

```
    MPI_Init(&argc,&argv);
```

```
    mysum = 0.0;
```

```
    for (i = 0; i < 1000 / N_PE; i++)
```

```
        mysum += A[i];
```

```
    MPI_Reduce(&mysum, &sum, 1, MPI_DOUBLE,  
             MPI_SUM, 0, MPI_COMM_WORLD);
```

```
    MPI_Finalize();
```

```
    return (0);
```

```
}
```

Explanation

- Data A is allocated in each process
 - Each process allocates enough size (the total size divided by the number of processes)
- Computation and communication
 - Each process computes a partial sum, and communicates with all processes to sum it up by collective communication
 - **MPI_Reduce**(&*mysum*, &*sum*, 1, **MPI_DOUBLE**,
 - **MPI_SUM**, 0, **MPI_COMM_WORLD**);
 - Combines *mysum* (an array of MPI_DOUBLE with size 1) using MPI_SUM, and returns the combined value *sum* of the root process (rank 0)

Sample program (3): Cpi

- Calculate the PI by the integral calculus
- Test program of MPICH
 - Riemann Sum
 - Broadcast n (number of divided parts)
 - Reduce the partial sum
 - The partial sum is computed in cyclic manner

$$\pi = \int_0^1 \frac{4}{1+t^2} dt$$

...

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

```
h = 1.0 / n;
```

```
sum = 0.0;
```

```
for (i = myid + 1; i <= n; i += numprocs){
```

```
    x = h * (i - 0.5);
```

```
    sum += f(x);
```

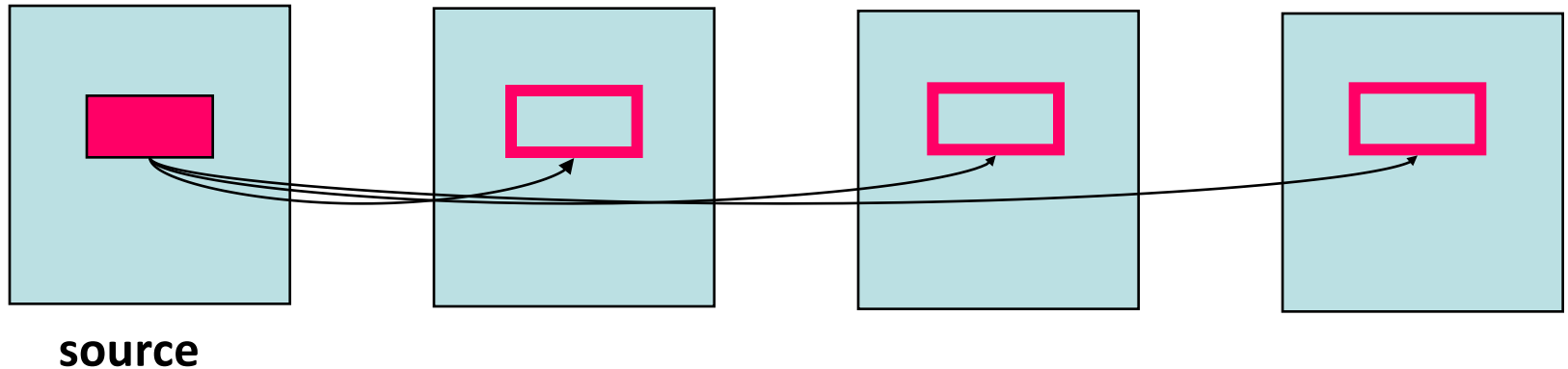
```
}
```

```
mypi = h * sum;
```

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE,  
            MPI_SUM, 0, MPI_COMM_WORLD);
```


broadcast

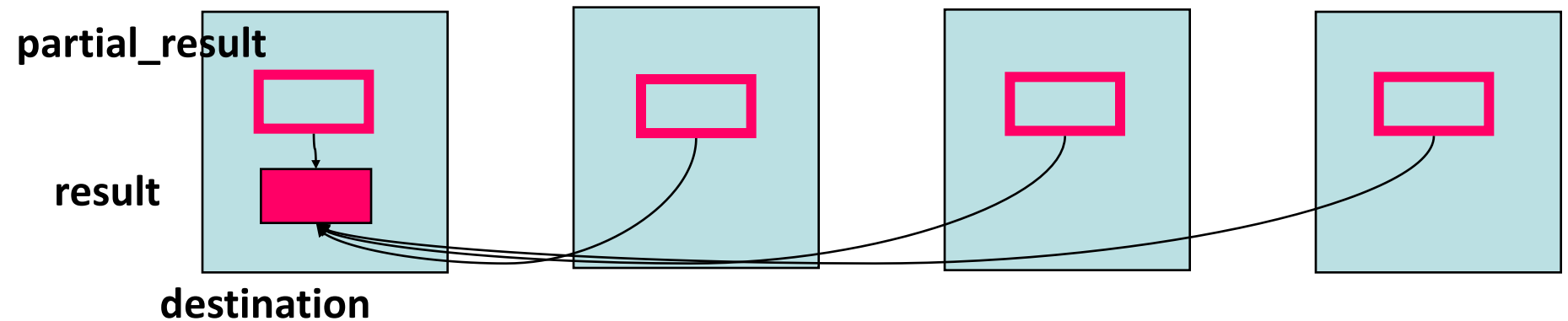
```
MPI_Bcast(  
  void    *data_buffer, // address of source and destination buffer of data  
  int     count,        // data counts  
  MPI_Datatype data_type, // data type  
  int     source,        // source process rank  
  MPI_Comm communicator // communicator  
);
```



It should be executed on all processes in the communicator

Reduction

```
MPI_Reduce(  
    void    *partial_result,    // address of input data  
    void    *result,            // address of output data  
    int     count,              // data count  
    MPI_Datatype data_type,      // data type  
    MPI_Op   operator,          // reduce operation  
    int     destination,        // destination process rank  
    MPI_Comm communicator       // communicator  
);
```



It should be executed on all processes in the communicator

MPI_Allreduce returns the result on all processes

```
/* cpi mpi version */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

double
f(double a)
{
    return (4.0 / (1.0 + a * a));
}

int
main(int argc, char *argv[])
{
    int n = 0, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    double startwtime = 0.0, endwtime;
    int namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
```

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Get_processor_name(processor_name, &namelen);
fprintf(stderr, "Process %d on %s¥n", myid, processor_name);

if (argc > 1)
    n = atoi(argv[1]);
startwtime = MPI_Wtime();
/* broadcast 'n' */
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n <= 0) {
    fprintf(stderr, "usage: %s #partition¥n", *argv);
    MPI_Abort(MPI_COMM_WORLD, 1);
}
```

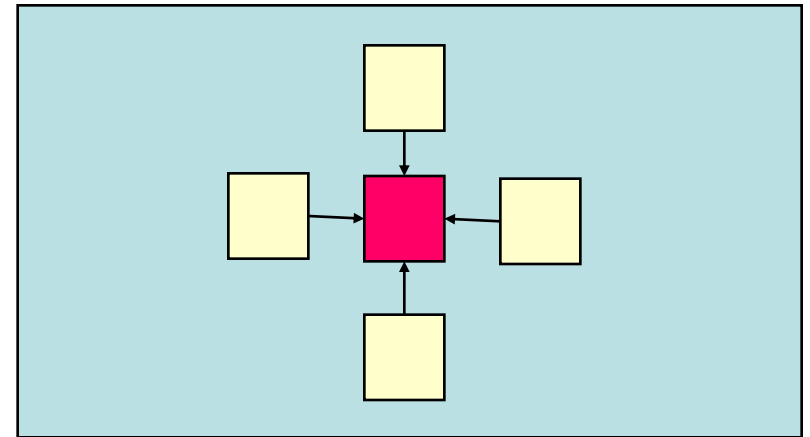
```

/* calculate each part of pi */
h = 1.0 / n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs){
    x = h * (i - 0.5);
    sum += f(x);
}
mypi = h * sum;
/* sum up each part of pi */
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (myid == 0) {
    printf("pi is approximately %.16f, Error is %.16f¥n",
        pi, fabs(pi - PI25DT));
    endwtime = MPI_Wtime();
    printf("wall clock time = %f¥n",
        endwtime - startwtime);
}
MPI_Finalize();
return (0);
}

```

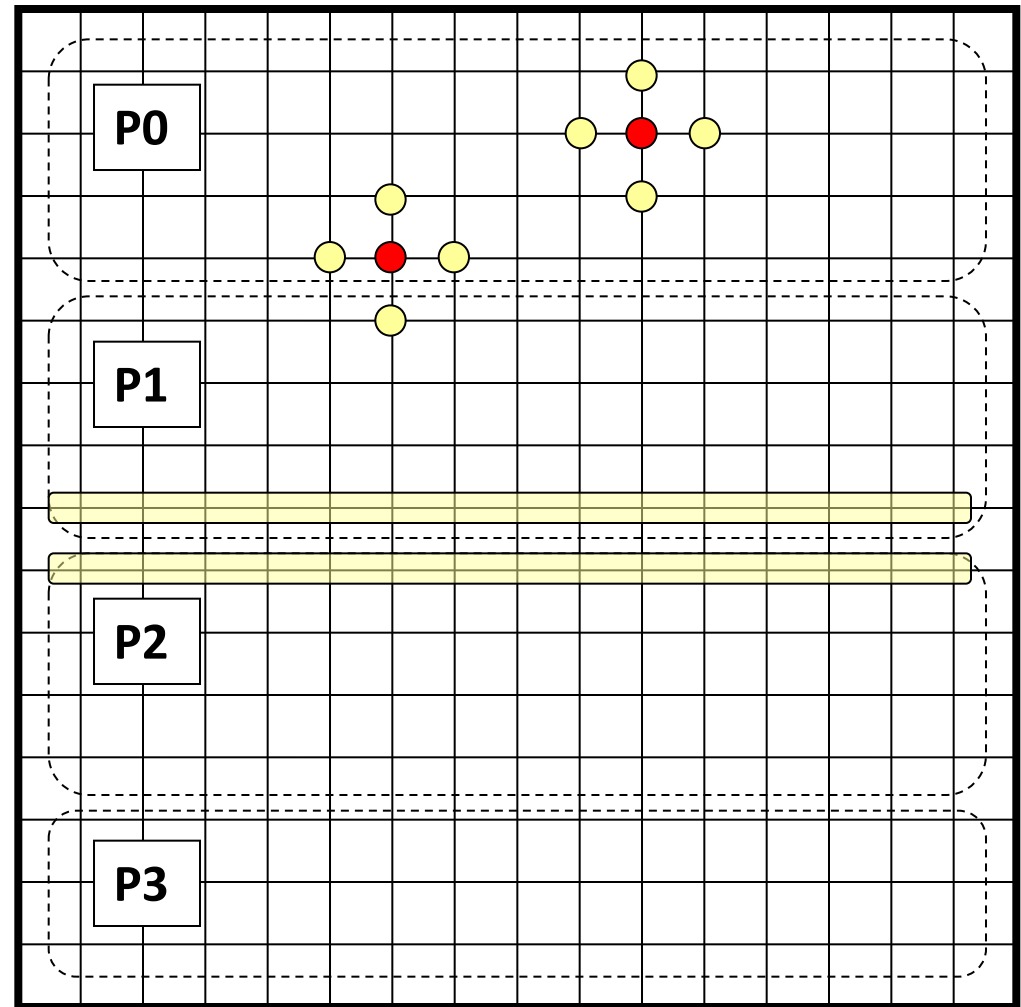
Sample program (4): laplace

- Explicit solution of Laplace equation
 - Update by averaging data of up, down, left, right four points
 - Prepare two arrays old and new to keep the old (previous) value
 - Region segmentation, region division
 - Compute the residual to check the convergence



Matrix decomposition and nearest neighbor communication

- Block distribution of 2D region
- To update boundary elements, boundary elements of neighbors are required
- Data exchange of boundary elements



Blocking point-to-point communication

- Send/Receive

```
MPI_Send(  
    void            *send_data_buffer, // address of input data  
    int             count,              // data count  
    MPI_Datatype    data_type,          // data type  
    int             destination,         // destination process rank  
    int             tag,                // message tag  
    MPI_Comm        communicator        // communicator  
);
```

```
MPI_Recv(  
    void            *recv_data_buffer, // address of receive data  
    int             count,              // data count  
    MPI_Datatype    data_type,          // data type  
    int             source,             // source process rank  
    int             tag,                // message tag  
    MPI_Comm        communicator,       // communicator  
    MPI_Status      *status             // status information  
);
```


Point-to-point communication

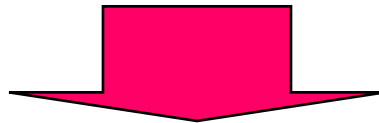
- Message is specified by address and size
 - Typed: MPI_INT, MPI_DOUBLE, ...
 - Binary data can be specified by MPI_BYTE with message size in byte
- Source/destination is specified by process rank and message tag
 - MPI_ANY_SOURCE for any source process rank
 - MPI_ANY_TAG for any message tag
- Status information includes the source rank, size, tag of the received message

Nonblocking point-to-point communication

- MPI_Isend/Irecv initiates the communication, MPI_Wait waits for the completion in semantics of blocking communication
 - Computation and communication can be overlapped if the communication can be executed in the background

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

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



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

Process topology

- int **MPI_Cart_create**(MPI_Comm *comm_old*, int *ndims*, int **dims*, int **periods*, int *reorder*, MPI_Comm **comm_cart*);
 - Creates *comm_cart* with *ndims* dimensional hypercube topology
 - Process size of each dimension is specified by *dims*
 - *Periods* specified whether each dimension is periodical or not
 - *Reorder* specifies whether it allows renumbering of ranks between old and new communicators

Source/destination of shift communication

- int **MPI_Cart_shift**(MPI_Comm *comm*, int *direction*, int *disp*, int **rank_source*, int **rank_dest*);
 - *Direction* specified the dimension of shift communication
 - It is 0 to ndims-1 in ndims dimension case
 - *Disp* is a displacement of shift communication
 - It returns *rank_source* as a source rank and *rank_dest* as a destination rank
 - If the boundary is not periodical, it returns MPI_PROC_NULL if it exceeds the boundary

```
/* calculate process ranks for 'down' and 'up' */  
MPI_Cart_shift(comm, 0, 1, &down, &up);  
  
/* recv from down */  
MPI_Irecv(&uu[x_start-1][1], YSIZE, MPI_DOUBLE, down, TAG_1,  
           comm, &req1);  
/* recv from up */  
MPI_Irecv(&uu[x_end][1], YSIZE, MPI_DOUBLE, up, TAG_2,  
          comm, &req2);  
  
/* send to down */  
MPI_Send(&u[x_start][1], YSIZE, MPI_DOUBLE, down, TAG_2, comm);  
/* send to up */  
MPI_Send(&u[x_end-1][1], YSIZE, MPI_DOUBLE, up, TAG_1, comm);  
  
MPI_Wait(&req1, &status1);  
MPI_Wait(&req2, &status2);
```

In a process of rank 0 and numprocs-1, MPI_Cart_shift returns **MPI_PROC_NULL**
No need to treat specially. MPI_Send and Irecv do not do anything if
MPI_PROC_NULL is specified

```
/*  
 * Laplace equation with explicit method  
 */  
#include <stdio.h>  
#include <stdlib.h>  
#include <math.h>  
#include <mpi.h>  
  
/* square region */  
#define XSIZE 256  
#define YSIZE 256  
#define PI 3.1415927  
#define NITER 10000  
double u[XSIZE + 2][YSIZE + 2], uu[XSIZE + 2][YSIZE + 2];  
double time1, time2;  
void lap_solve(MPI_Comm);  
int myid, numprocs;  
int namelen;  
char processor_name[MPI_MAX_PROCESSOR_NAME];  
int xsize;
```

2D target domain
Uu is for new values

```
void
initialize()
{
    int x, y;

    /* initialization */
    for (x = 1; x < XSIZE + 1; x++)
        for (y = 1; y < YSIZE + 1; y++)
            u[x][y] = sin((x - 1.0) / XSIZE * PI) +
                cos((y - 1.0) / YSIZE * PI);
    /* zero clear in the boundary */
    for (x = 0; x < XSIZE + 2; x++) {
        u [x][0] = u [x][YSIZE + 1] = 0.0;
        uu[x][0] = uu[x][YSIZE + 1] = 0.0;
    }
    for (y = 0; y < YSIZE + 2; y++) {
        u [0][y] = u [XSIZE + 1][y] = 0.0;
        uu[0][y] = uu[XSIZE + 1][y] = 0.0;
    }
}
```

```
#define TAG_1 100
#define TAG_2 101

#ifdef FALSE
#define FALSE 0
#endif

void lap_solve(MPI_Comm comm)
{
    int x, y, k;
    double sum;
    double t_sum;
    int x_start, x_end;
    MPI_Request req1, req2;
    MPI_Status status1, status2;
    MPI_Comm comm1d;
    int down, up;
    int periods[1] = { FALSE };
```



```
/*  
 * Create one dimensional cartesian topology with  
 * nonperiodical boundary  
 */  
MPI_Cart_create(comm, 1, &numprocs, periods, FALSE, &comm1d);  
/* calculate process ranks for 'down' and 'up' */  
MPI_Cart_shift(comm1d, 0, 1, &down, &up);  
  
x_start = 1 + xsize * myid;  
x_end = 1 + xsize * (myid + 1);
```

- Create *comm1d* with one dimensional topology
 - The boundary is not periodical
- Obtain the *up* and *down* process rank
 - The boundary process may obtain MPI_PROC_NULL

```

for (k = 0; k < NITER; k++){
    /* old <- new */
    for (x = x_start; x < x_end; x++)
        for (y = 1; y < YSIZE + 1; y++)
            uu[x][y] = u[x][y];

    /* recv from down */
    MPI_Irecv(&uu[x_start - 1][1], YSIZE, MPI_DOUBLE,
              down, TAG_1, comm1d, &req1);
    /* recv from up */
    MPI_Irecv(&uu[x_end][1], YSIZE, MPI_DOUBLE,
              up, TAG_2, comm1d, &req2);
    /* send to down */
    MPI_Send(&u[x_start][1], YSIZE, MPI_DOUBLE,
              down, TAG_2, comm1d);
    /* send to up */
    MPI_Send(&u[x_end - 1][1], YSIZE, MPI_DOUBLE,
              up, TAG_1, comm1d);

    MPI_Wait(&req1, &status1);
    MPI_Wait(&req2, &status2);

```

```

    /* update */
    for (x = x_start; x < x_end; x++)
        for (y = 1; y < YSIZE + 1; y++)
            u[x][y] = .25 * (uu[x - 1][y] + uu[x + 1][y] +
                            uu[x][y - 1] + uu[x][y + 1]);
}
/* check sum */
sum = 0.0;
for (x = x_start; x < x_end; x++)
    for (y = 1; y < YSIZE + 1; y++)
        sum += uu[x][y] - u[x][y];
MPI_Reduce(&sum, &t_sum, 1, MPI_DOUBLE, MPI_SUM, 0, comm1d);
if (myid == 0)
    printf("sum = %g¥n", t_sum);
MPI_Comm_free(&comm1d);
}

```

```
int
main(int argc, char *argv[])
{
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Get_processor_name(processor_name, &namelen);
    fprintf(stderr, "Process %d on %s¥n", myid, processor_name);

    xsize = XSIZE / numprocs;
    if ((XSIZE % numprocs) != 0)
        MPI_Abort(MPI_COMM_WORLD, 1);
    initialize();
    MPI_Barrier(MPI_COMM_WORLD);
    time1 = MPI_Wtime();
    lap_solve(MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    time2 = MPI_Wtime();
    if (myid == 0)
        printf("time = %g¥n", time2 - time1);
    MPI_Finalize();
    return (0);
}
```

Things to improve

- This program allocates the whole array although it is not necessary
 - When the partial array is allocated, the index of array should be computed from global index to local index
 - This is essential to solve large-scale problem using distributed memory machine
- Two dimensional distribution of 2D array is more efficient than one dimensional distribution
 - Reduce the communication size
 - Can be parallelize by more number of processors

Summary

- Parallel computing is indispensable to achieve high performance
- OpenMP is easy, but may be efficient up to 16 processors
- For more number of processors, MPI is only the choice
 - Tradeoff between the programming cost and execution time
 - We expect a new programming language that replaces MPI for longer term
- It is not **too** difficult to parallelize scientific applications
 - It has inherent data parallelism
 - Most scientific applications can be parallelized in some “parallelization patterns”

Parallel computing In coins

- `viola0[1-6].coins.tsukuba.ac.jp` (6 node cluster)
 - 8 cores / node
 - 2.93GHz Nehalem x 2 sockets
 - 12GB memory / node
 - 1333MHz 2GB DDR3 x 3 channels x 2
 - Network bandwidth 4GB/s
 - 4x QDR Infiniband
 - Software
 - CentOS5.4
 - OpenMPI*, MVAPICH1, MVAPICH2
 - System default is OpenMPI
 - It can be selected by `mpi-selector-menu`
 - gcc, gfortran, Sun JDK6
 - BLAS, LAPACK, ScaLAPACK

Initial setting

- Ssh authentication setting

```
% ssh-keygen -t rsa  
% cat .ssh/id_rsa.pub >> .ssh/authorized_keys
```

- Create known hosts file (also for hostnames of IB interface such as viola01-ib0)

```
% echo StrictHostKeyChecking no >> .ssh/config  
% ssh viola01-ib0 hostname  
viola01.coins.tsukuba.ac.jp  
% ssh viola02-ib0 hostname  
viola02.coins.tsukuba.ac.jp  
...  
% ssh viola06-ib0 hostname  
viola06.coins.tsukuba.ac.jp
```


Selection of MPI

\$ mpi-selector-menu

Current system default: openmpi-1.3.2-gcc-x86_64

Current user default: <none>

"u" and "s" modifiers can be added to numeric and "U" commands to specify "user" or "system-wide".

1. mvapich-1.1.0-gcc-x86_64
 2. mvapich2-1.2-gcc-x86_64
 3. openmpi-1.3.2-gcc-i386
 4. openmpi-1.3.2-gcc-x86_64
- U. Unset default
Q. Quit

Selection (1-4[us], U[us], Q): **2u**

System default is OpenMPI, no user default specified

Select MVAPICH2

Compilation

- How to compile an MPI program

```
% mpicc -O2 a.c
```

- If you select MPI, you need to re-compile
 - Not binary compatible among OpenMPI and MVAPICH

How to execute MPI program in OpenMPI

- Create a host file

```
% cat hosts-openmpi  
viola01-ib0 slots=8  
viola02-ib0 slots=8  
...  
viola06-ib0 slots=8
```

- Execute an MPI program (OpenMPI)

```
% mpirun -hostfile hosts-openmpi -np 48 a.out
```

How to execute MPI program in MVAPICH

- Create a host file

```
% cat hosts  
viola01-ib0  
viola02-ib0  
...  
viola06-ib0  
% cat hosts hosts hosts hosts hosts hosts hosts hosts >  
hosts-mvapich
```

- Execute an MPI program

```
% mpirun_rsh -hostfile hosts-mvapich -np 48 a.out
```

- Note that the execution command in MVAPICH is mpirun_rsh

Open source implementation

OpenMP

- GNU GCC 4.2 or later
 - % cc -fopenmp . . .
- Omni OpenMP Compiler
 - <http://phase.hpcc.jp/Omni/>

MPI

- OpenMPI
 - <http://www.open-mpi.org/>
- MPICH2
 - <http://www-unix.mcs.anl.gov/mpi/mpich2/>
- YAMPII
 - <http://www.il.is.s.u-tokyo.ac.jp/yampii/>