



Parallel Programming

Part 1

SPMD vs. SIMD

- SPMD means “Single Program, Multiple Data”.
- The same program is run locally on each processor, but there is no master control as in SIMD.
- The program can do different things on different processors by conditional branching.

Why SPMD?

- SPMD allows us to take advantage of generic processors and compilers.
- Special hardware for parallel computing is not necessary.

Application Granularity Considerations

- Two kinds of granularity:
 - **Load-balancing** granularity: ratio of size of parallel work units to overall work
 - **Communication granularity**: ratio of communication intervals to computation intervals

Load-Balancing Granularity

- Finer granularity is better, since it provides more ways to distribute the work.
- Imagine that the computation work load is a 10 kg. of material:
 - Sand = fine-grain
 - Cinder blocks = coarse grain
- Which is easier to distribute?

Communication Granularity

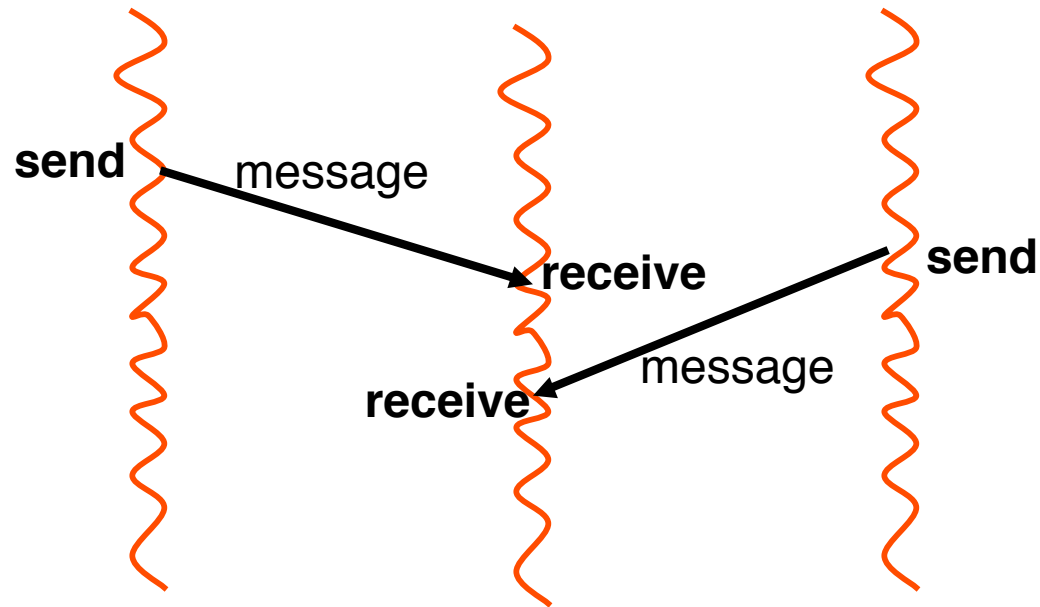
- Parallelism with fine-grain requires relatively **frequent communication** compared to the computation interval.
- If the communication is not very fast, the process' waiting time will **absorb** the speedup from parallel execution.
- Consequently, fine-grain is more suited to shared memory than to distributed memory. Conversely, distributed memory requires relatively coarse grain to be effective.
- Because SIMD has less synchronization overhead, very-fine grain is more suited to SIMD than to MIMD.

Message-Passing Paradigm

- Message-passing is the programming paradigm most closely associated with distributed memory.
- However, it can also be used in a shared memory system if the problem permits.
- It is more effective for **coarser granularity**, due to the overhead in passing messages.

Message-Passing (2)

threads/processes on different processors

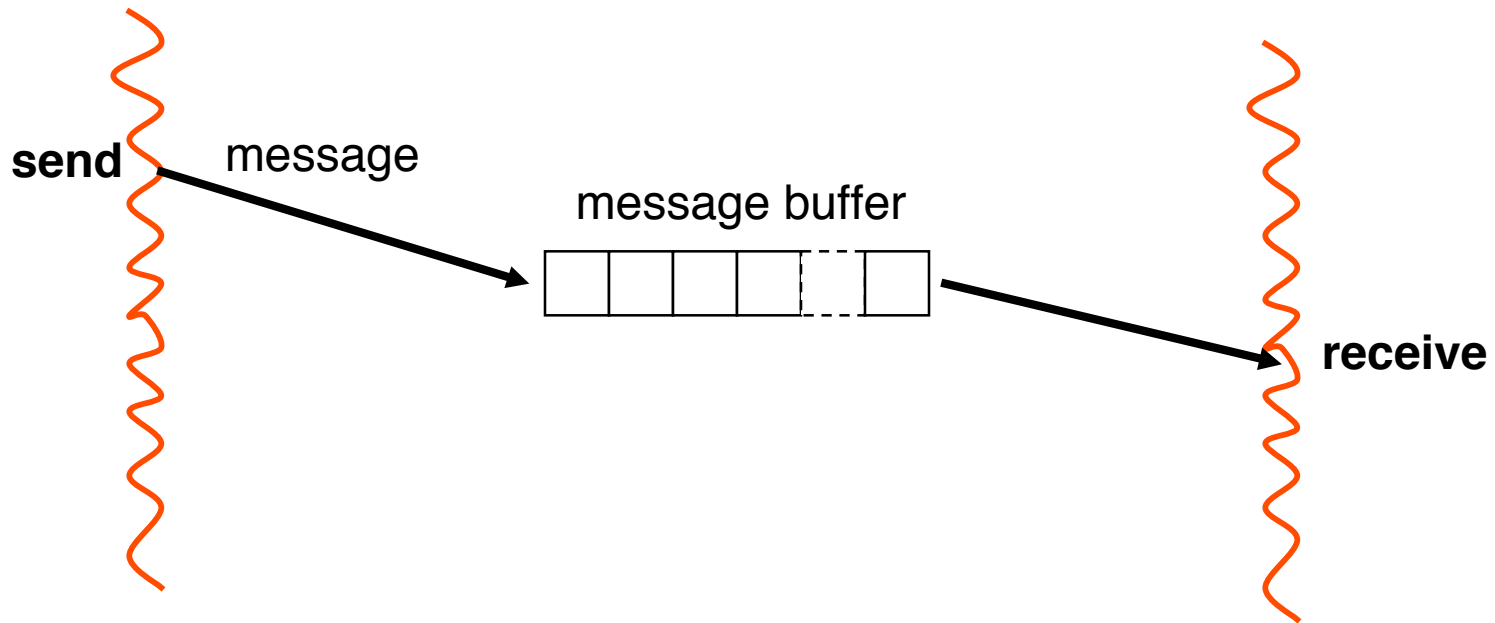


Message-Passing (3)

- Two varieties of send:
 - **Blocking send:** The sending process waits for the message to be received before proceeding.
 - **Non-blocking send:** The sending process can proceed immediately. (The message may be buffered pending receipt.)

Message Buffering

can be used to avoid blocking sender



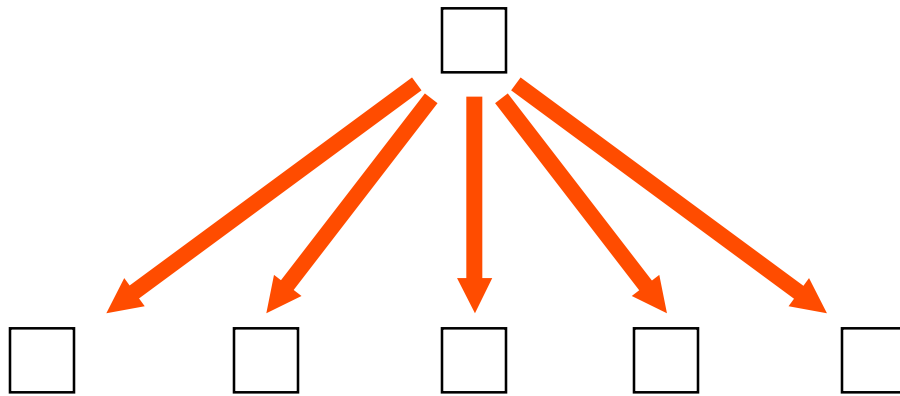
Message-Passing (4)

- Two varieties of receive:
 - **Blocking receive** (most common): The receiving process **waits until** there is a message.
 - **Non-blocking receive**: The receiving process can **check whether** there is a message to be received, and continue if not.

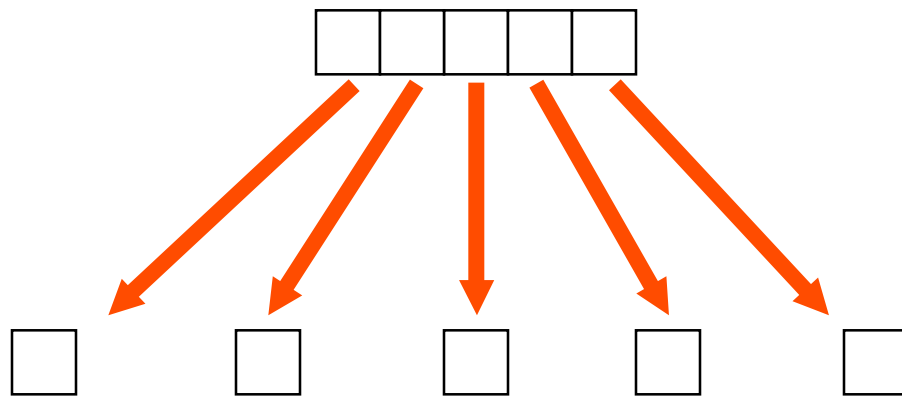
Multi-cast, Scatter, Gather, Reduce, Map

- **Multi-cast** is the equivalent of a *send* of a **single message** to **each of a set** of processes (broadcast means to *all* processes).
- **Scatter** means to send **different** elements of an array to different processes.
- **Gather** means to **collect elements** from different processes into a single array.
- **Reduce** means to form a single element using a specified binary operation.
- **Map** means to apply a single operation unary operation to all elements of an array.

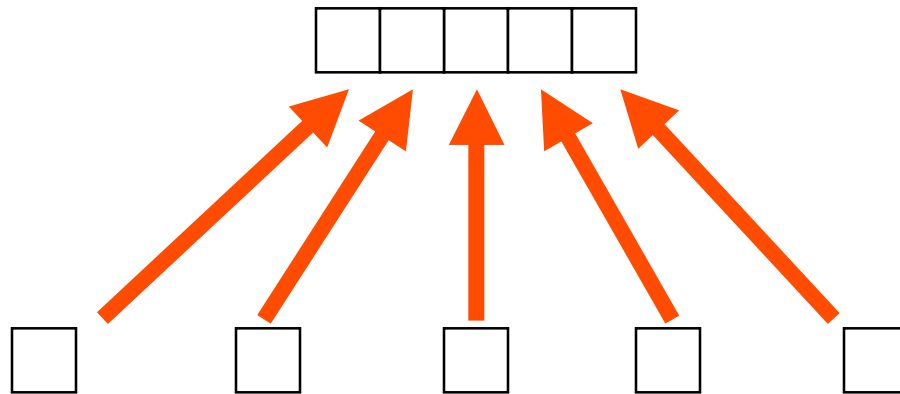
Multi-cast



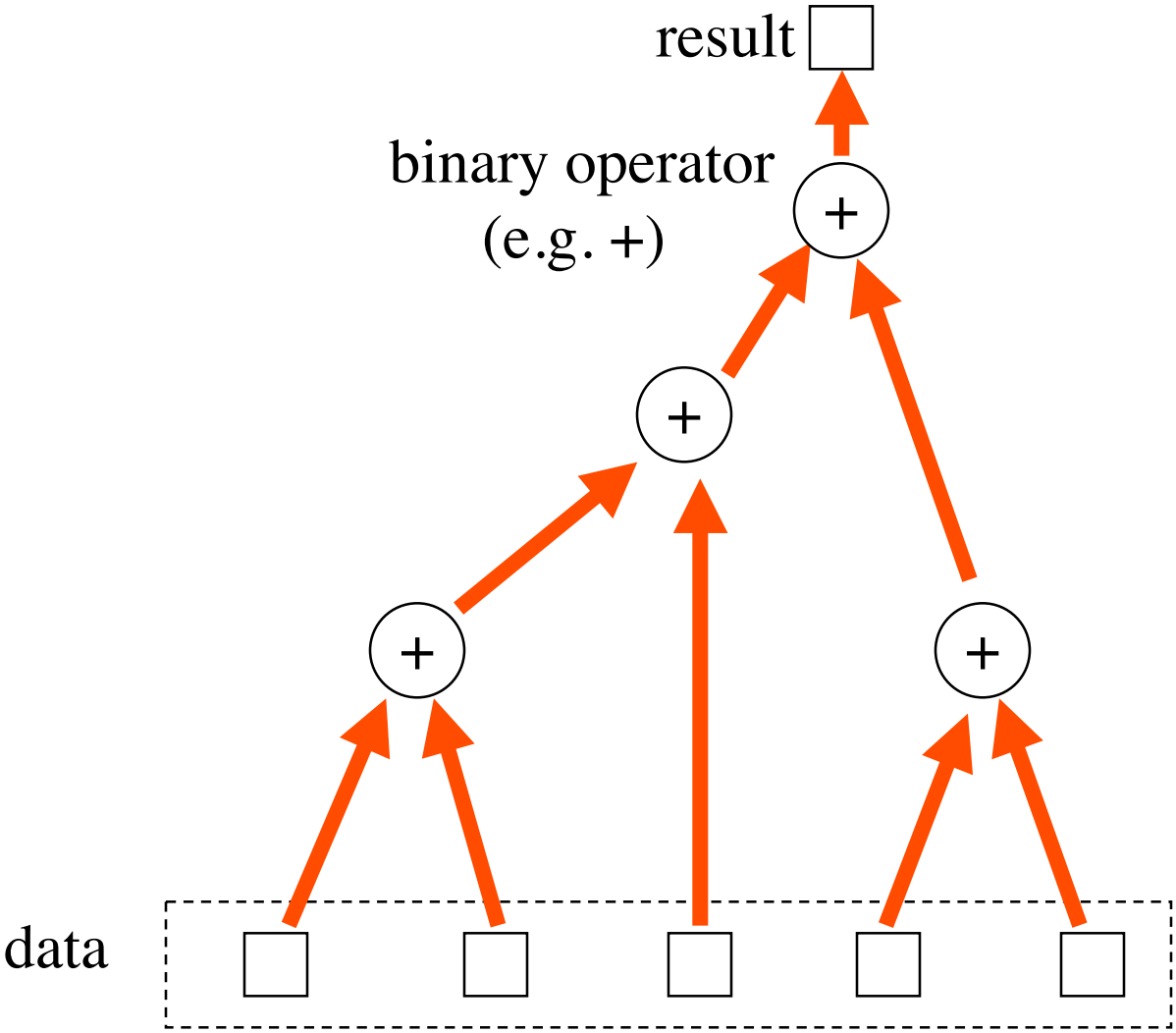
Scatter



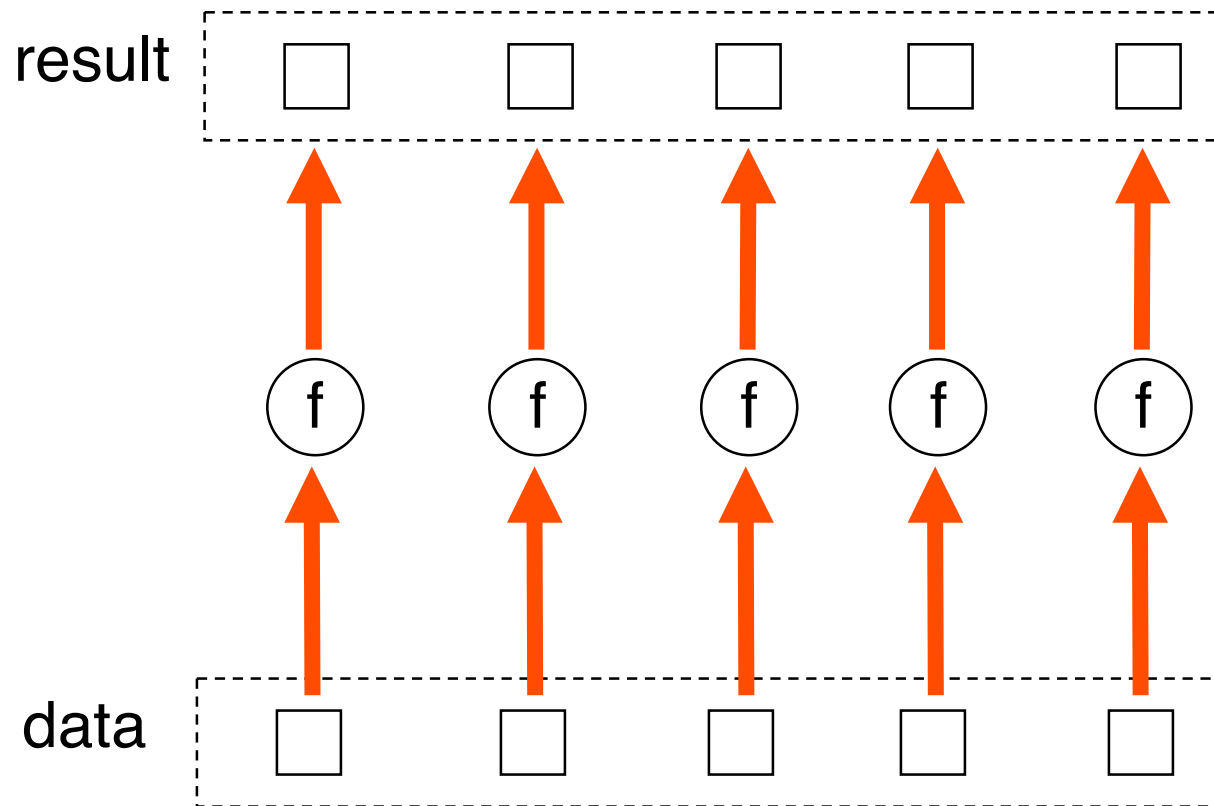
Gather



Reduce



Map



MPI Library

(Message-Passing Interface, Lusk et al.)

- Based on the SPMD (Single Program, Multiple Data Stream) idea.
- All processes run the same program, but
- Processes can differentiate themselves using assigned ID's (called the *rank* of the process), so the code actually executed can be **different** in different processes.
- Processes are divided into *groups* and the rank (0, 1, 2, ...) applies within the group.



Ewing Lusk

MPI

- Based on processes, rather than processors.
- There can be one process per processor, or more than one.
- It is up to the programmer to decide how many.

MPI

- MPI standard library is defined for:
 - Fortran
 - C
 - C++
 - Java
 - Python
 - Perl
 - Ocaml

MPI Communicators

- Communication between or within a group is defined by an abstraction called a ***Communicator*** (type MPI_Comm).
- A common pre-defined communicator is

MPI_COMM_WORLD

MPI Invocation

- The number of processes is defined on the command line:

mpirun -np *Number-of-processes Executable Args*

- The program initializes using (C syntax):

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

where `argc` and `argv` are from *Args* on the command line.

MPI Finalization

- Always terminate execution with:

`MPI_Finalize();`

MPI

- The program can find out the **number of processes**:

(C syntax)

```
MPI_Comm_size(Communicator, &nprocs);
```

e.g.

```
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

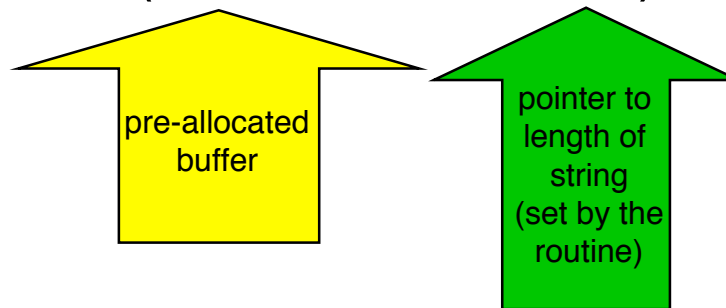
MPI Process Identification

- A process can determine its own **rank**:

```
MPI_Comm_rank(Communicator, &id);
```

- and the name of its processor:

```
MPI_Get_processor_name(name, &namelen);
```



MPI Barrier

- A process can **join** a **barrier** within its group:

```
MPI_Barrier(Communicator);
```

Master and Slaves

- It is common to declare one process (usually the one with id 0) as the master and others as slaves.
- A process can then can execute code **conditioned** upon whether it is master or slave (by checking its own id).
- The **slaves** do the main work in parallel.
- The **master** is in charge of initial setup and later direction, although it could also do work.

Sending Messages

```
int MPI_Send(  
    void* buf,                // address of buffer  
    int count,                // number of items  
    MPI_Datatype datatype,    // type of each item  
    int dest,                 // rank of destination  
    int tag,                  // tag value of message  
    MPI_Comm comm)           // communicator
```

The return value indicates a success code, which will be `MPI_SUCCESS` if the operation is successful.

Receiving Messages

```
int MPI_Recv(  
    void* buf,                // address of buffer  
    int count,                // maximum number of items  
    MPI_Datatype datatype,    // type of each item  
    int source,               // rank of source  
    int tag,                  // tag value of message  
    MPI_Comm comm,           // communicator  
    MPI_Status *status)      // status indicator
```

The status indicator gives information about what was received.

Send-Receive Matching

- The purpose of the **tag** argument is to allow a single receive operation to **discriminate** among different tags of messages that might be sent.
- For a message to be received from a sender, both the tag **and** the **source** must match the sender values in the receive statement.

Wild Cards

- Wild cards can also be used to designate receiving from *any* source:

MPI_ANY_SOURCE

- The tag value can also be a wild-card:

MPI_ANY_TAG

MPI Datatypes

(most correspond to C datatypes of a similar name)

- MPI_CHAR
 - MPI_SHORT
 - MPI_INT
 - MPI_LONG
 - MPI_FLOAT
 - MPI_DOUBLE
 - MPI_LONG_DOUBLE
 - MPI_UNSIGNED
 - MPI_UNSIGNED_SHORT
 - MPI_UNSIGNED_LONG
 - MPI_UNSIGNED_CHAR
- These do not correspond to any C dataype:
- MPI_PACKED
 - MPI_BYTE

Status indicator

- is a struct containing three fields:
 - MPI_SOURCE
 - MPI_TAG
 - MPI_ERROR

indicating the corresponding information about the message received.

- It also contains the length of the message received, using a call of the form:

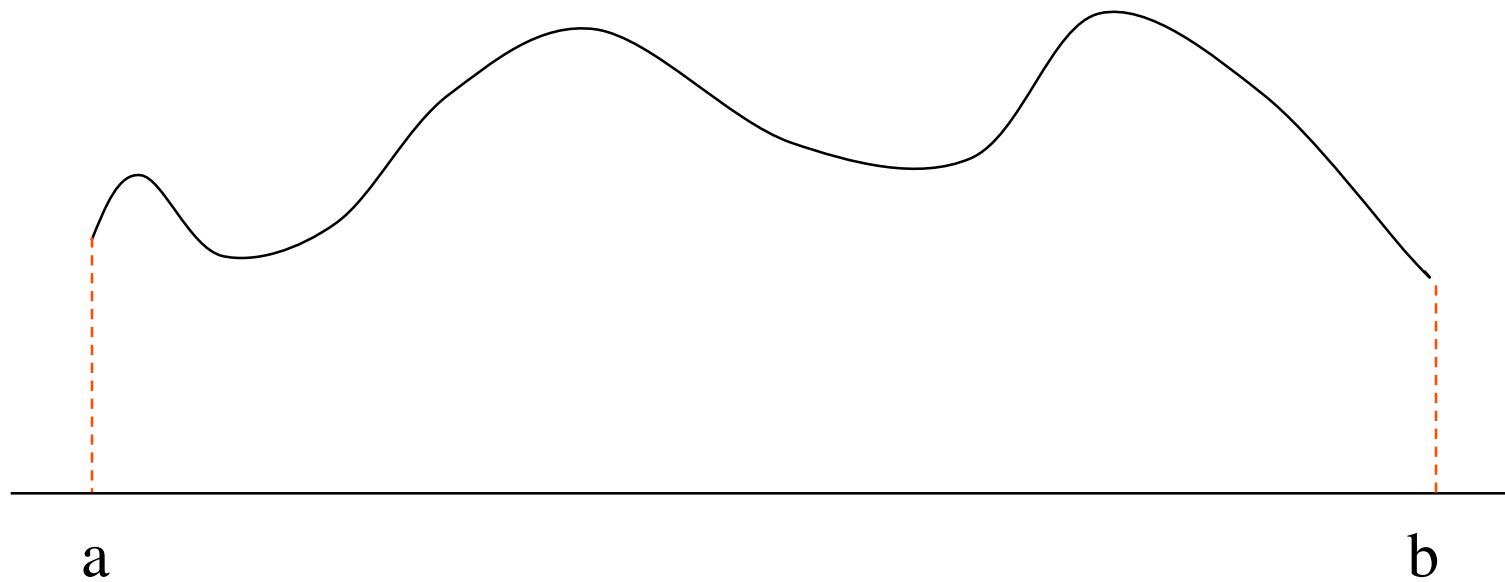
```
MPI_Get_count(MPI_Status, MPI_Datatype, int *count)
```

An Example

- **Integrate** a function of one real variable numerically.
- The function will be passed as an argument to the *integrate* function.
- Other arguments to the integrate function include:
 - The limits of integration
 - The number of sub-divisions
 - The MPI communicator to be used

Integration Example

function to be integrated

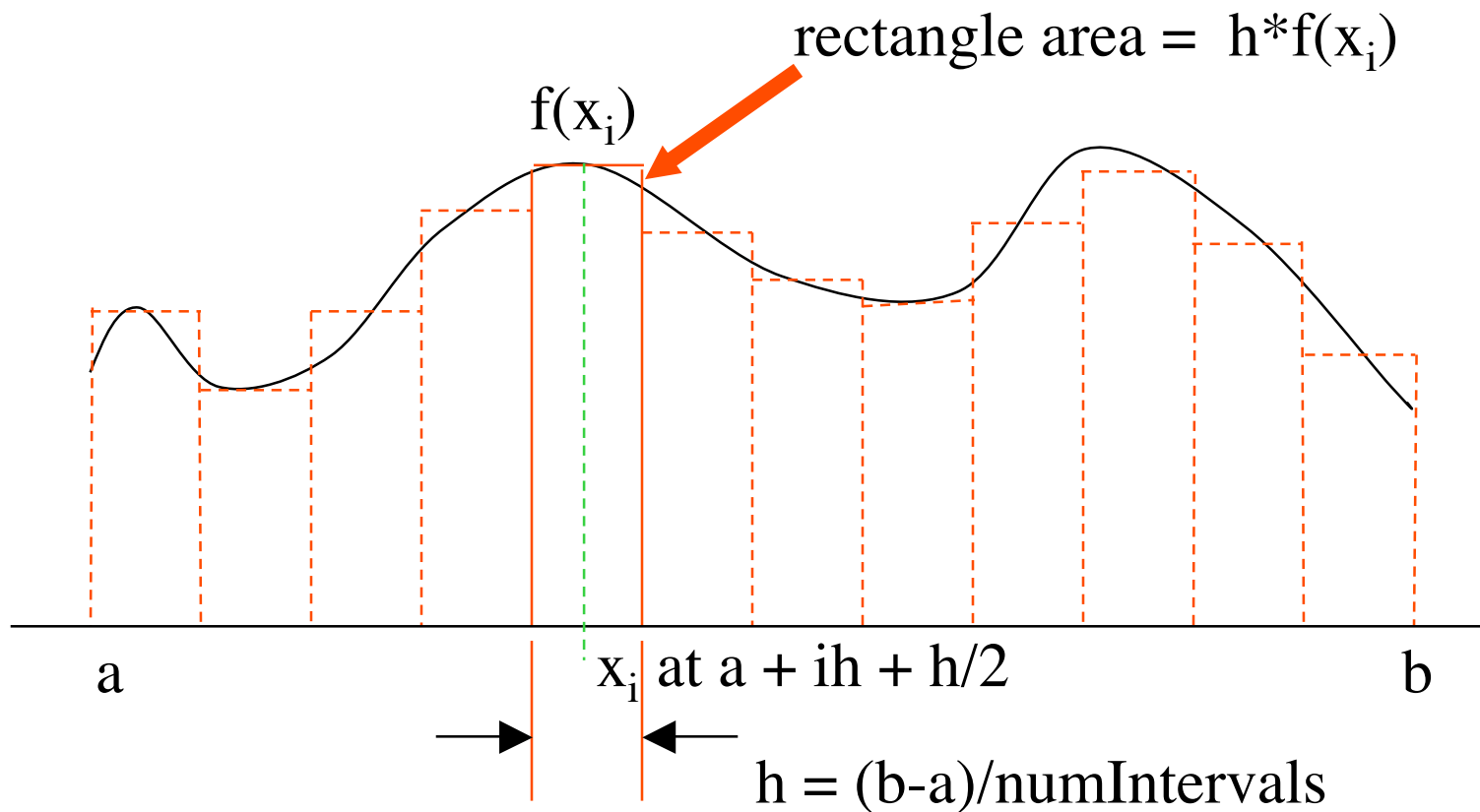


a

b

limits of integration

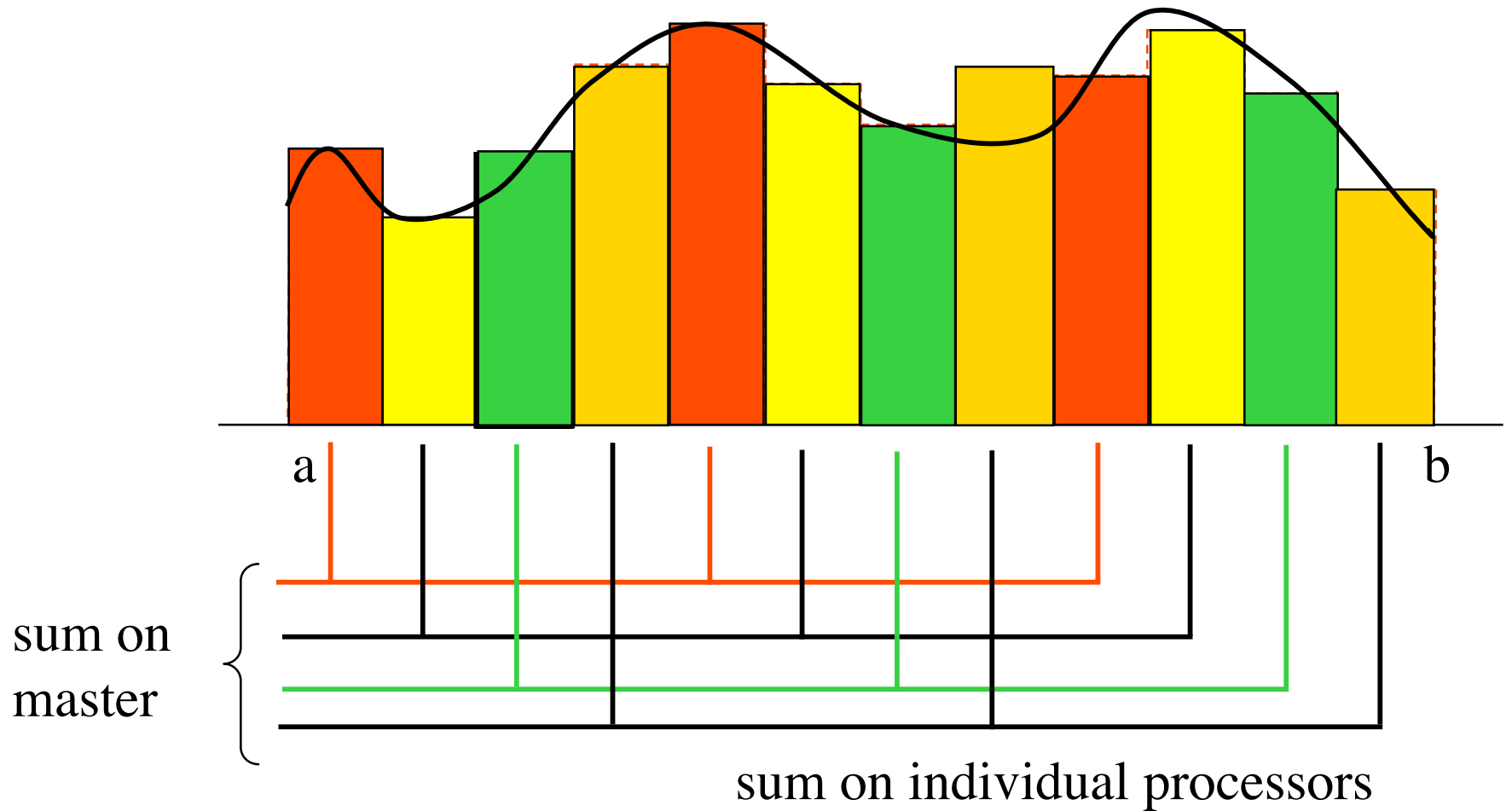
Rectangles approximate area under curve



Point-to-Point Version

- The number of processes is given on the command line.
- Process 0 will be the master.
- Each process j , *including the master*, computes the sum the rectangles (implicitly) numbered i such that $i \% \text{numProcs} == j$.
- All of the slave processes send their sum to the master, which adds them together with its own sum.

Example with 4 processes



Reading/Writing MPI Code

- can be a little tricky.
- Must keep in mind that MPI is an **SPMD** (single-program, multiple-data stream) model.
- All processes execute the **same** program.
- Some processes execute one branch or another based on value of the processes' id.

MPI Code (C syntax)

```
double integrate(  
    double f(double), /* function to integrate */  
    double low, /* lower limit of integration */  
    double high, /* upper limit of integration */  
    int numIntervals, /* number of intervals to be used */  
    MPI_Comm comm) /* MPI communicator to use */  
{  
    MPI_Status stat; /* status indicator */  
    int numProcs; /* number of processes in comm */  
    int buffsiz = 1; /* buffer size for messages */  
    int tag = 1; /* tag for messages */  
    int id; /* id of this process */  
    int master = 0; /* id of master process */  
    double h; /* width of rectangle */  
    double area; /* area of this process' rectangles */  
    double integral; /* approximation to integral */  
    int i;  
  
    MPI_Comm_size(comm, &numProcs); /* get number of processes */  
    MPI_Comm_rank(comm, &id); /* get this process' id */
```

```

h = (high - low) / numIntervals;           /* compute rectangle width    */

area = 0;                                   /* compute area of rectangles */
for( i = id ; i < numIntervals; i += numProcs )
{
    area += f( h * ((double)i + 0.5) );
}

if (id == master)
{
    /* master adds up all sums    */
    integral = area;
    for( i = 1; i < numProcs; i++ )
    {
        MPI_Recv(&area, buffsiz, MPI_DOUBLE, MPI_ANY_SOURCE, tag, comm, &stat);
        integral += area;
    }
}
else
{
    /* slave sends area to master */
    MPI_Send(&area, buffsiz, MPI_DOUBLE, master, tag, comm);
}

return h * integral;
}

```

MPI Reduce Version

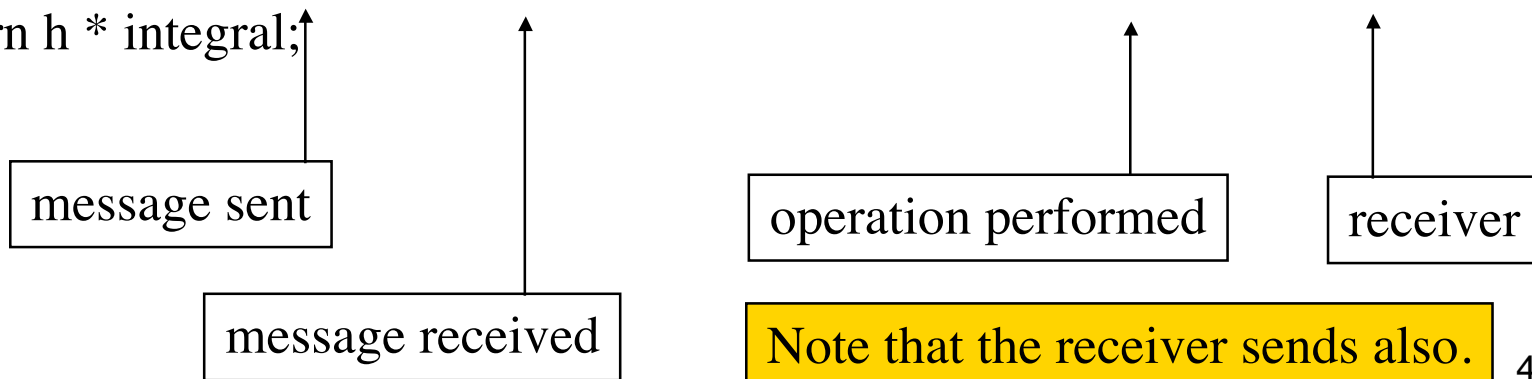
- The same basic idea as the point-to-point version, except that rather than explicitly sending and receiving messages, the **reduce** operation of MPI is used.

MPI Code for reduce version

```
h = (high - low) / numIntervals;          /* compute rectangle width    */

area = 0;                                  /* compute area of rectangles  */
for( i = id ; i < numIntervals; i += numProcs )
{
    area += f(h * ((double)i + 0.5));
}
```

```
MPI_Reduce(&area, &integral, tag, MPI_DOUBLE, MPI_SUM, master, comm);
return h * integral;
```



Note that the receiver sends also.

MPI Reduce Version

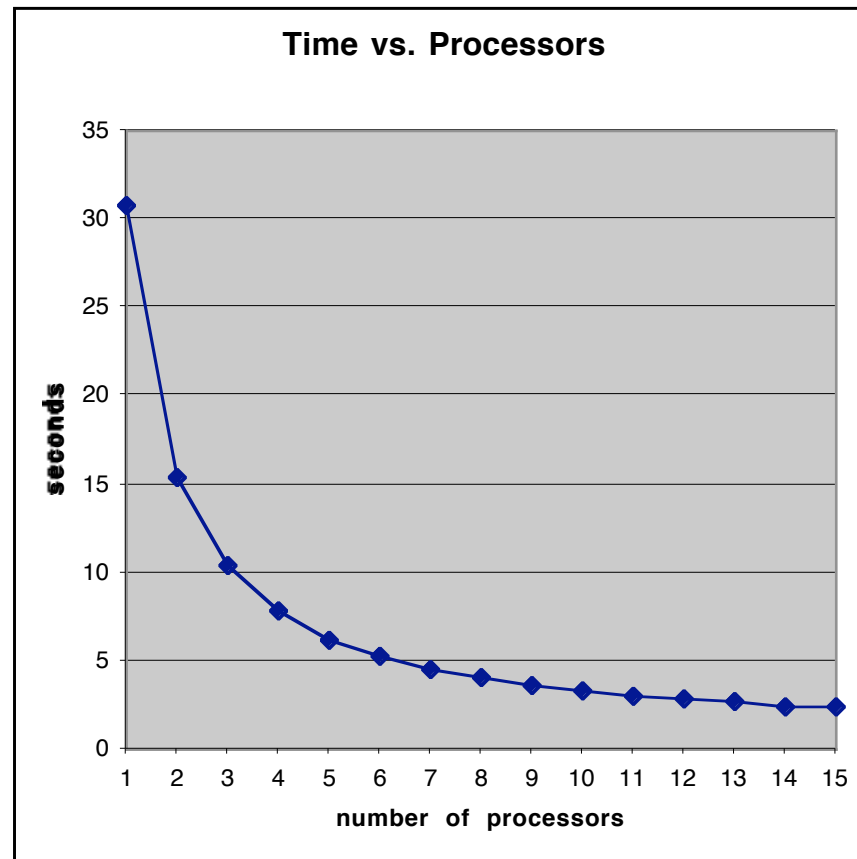
- The operator applied in reduce is limited to the repertoire provided by MPI:
 - [MPI_MAX] maximum
 - [MPI_MIN] minimum
 - [MPI_SUM] sum
 - [MPI_PROD] product
 - [MPI_LAND] logical and
 - [MPI_BAND] bit-wise and
 - [MPI_LOR] logical or
 - [MPI_BOR] bit-wise or
 - [MPI_LXOR] logical xor
 - [MPI_BXOR] bit-wise xor
 - [MPI_MAXLOC] max value **and location**
 - [MPI_MINLOC] min value **and location**

Results on HMC Math Beowulf

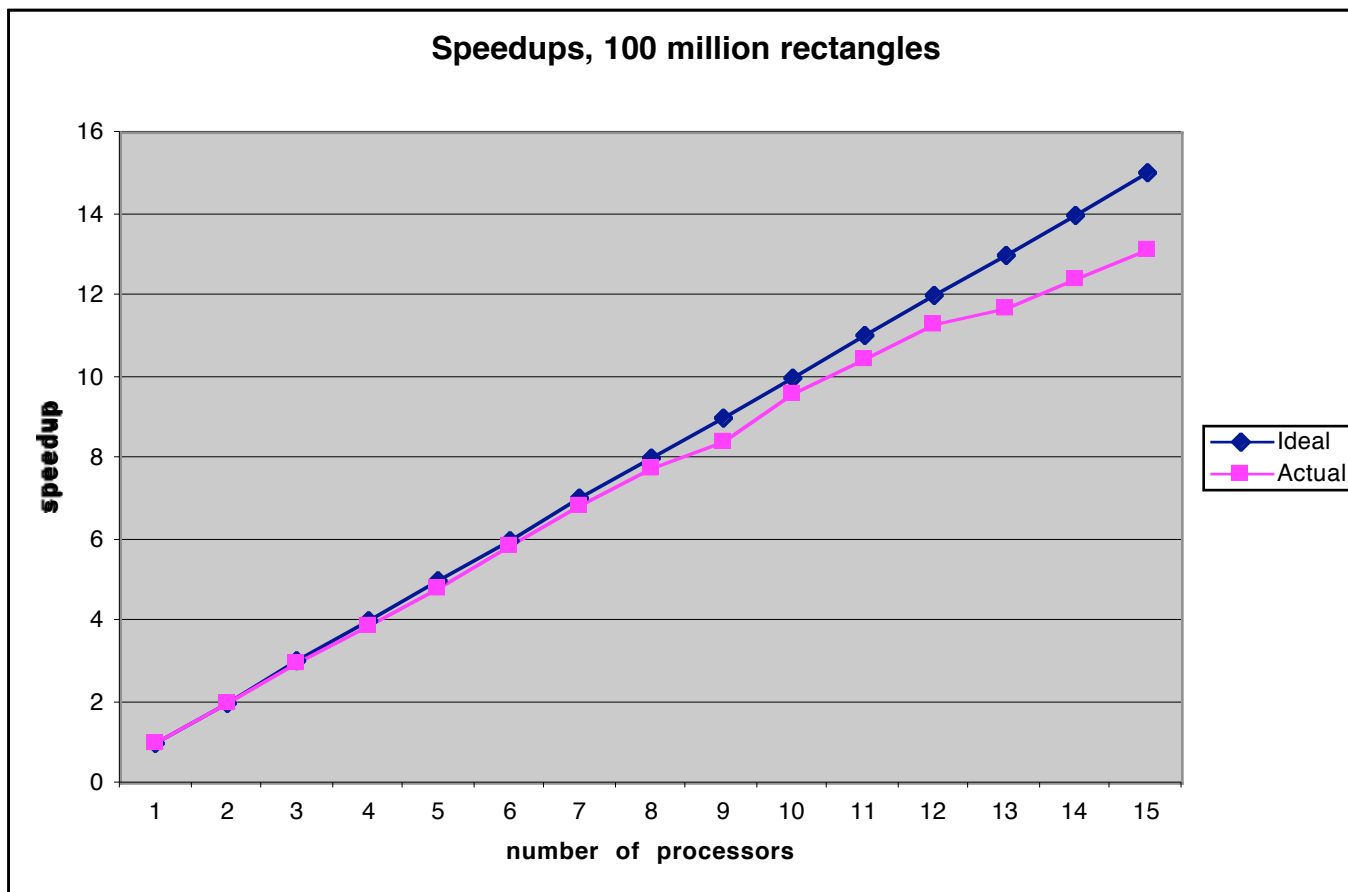
100 million rectangles

processors	result	error	time (sec)	effort	speedup
1	3.14159265359023	4.41E-13	30.73	30.73	1.00
2	3.14159265358999	2.07E-13	15.48	30.96	1.99
3	3.14159265359014	3.49E-13	10.46	31.38	2.94
4	3.14159265359019	4.01E-13	7.90	31.60	3.89
5	3.14159265358964	-1.50E-13	6.38	31.89	4.82
6	3.14159265358965	-1.37E-13	5.24	31.43	5.86
7	3.14159265358964	-1.53E-13	4.51	31.60	6.81
8	3.14159265358960	-1.85E-13	3.97	31.77	7.74
9	3.14159265358974	-4.53E-14	3.67	33.01	8.37
10	3.14159265358974	-4.84E-14	3.22	32.16	9.54
11	3.14159265358973	-5.46E-14	2.95	32.41	10.42
12	3.14159265358973	-5.51E-14	2.72	32.64	11.30
13	3.14159265358973	-5.37E-14	2.64	34.32	11.64
14	3.14159265358974	-4.71E-14	2.48	34.67	12.39
15	3.14159265358974	-4.62E-14	2.34	35.05	13.13

Time vs. Processors



Speedup vs. Processors



Perspective

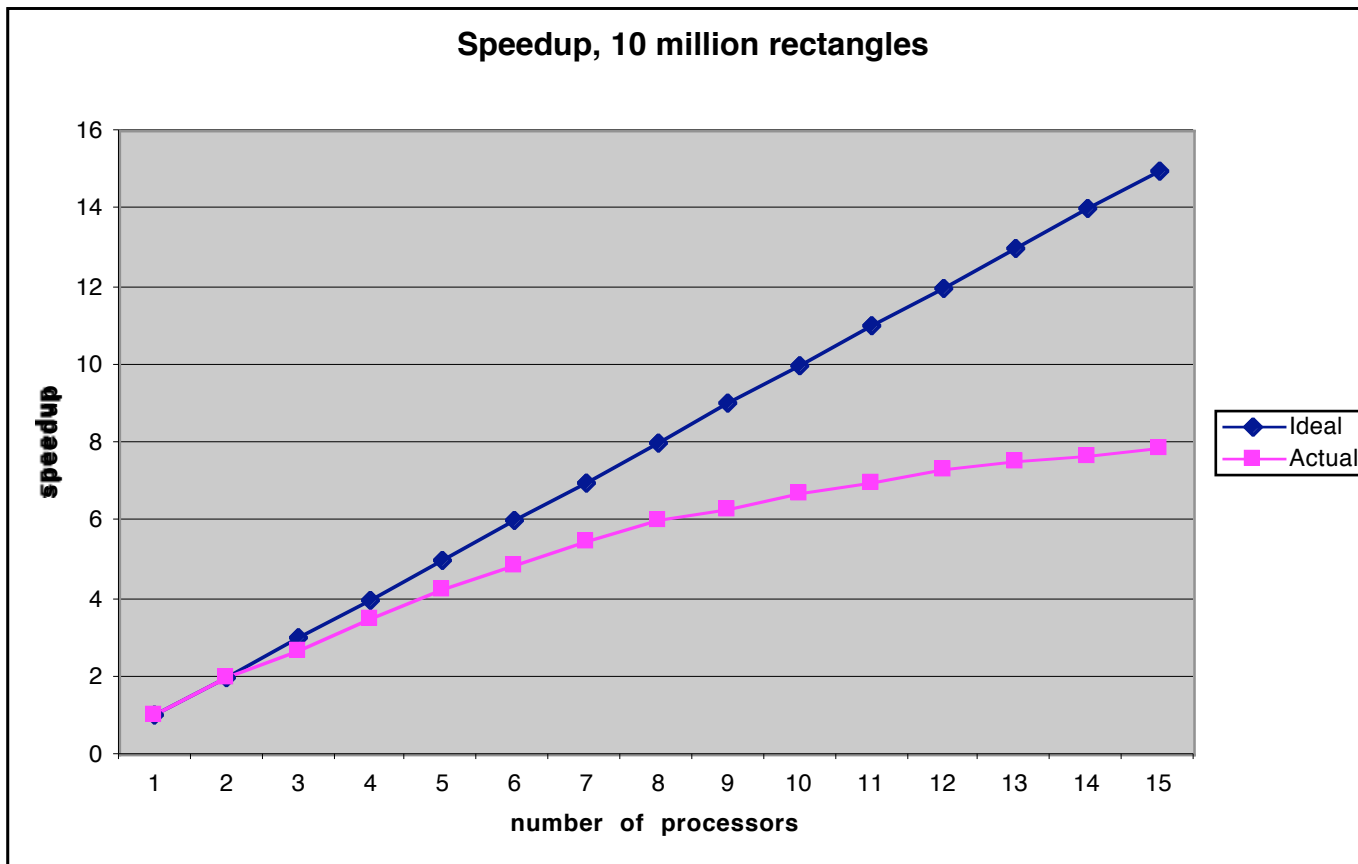
- The application seems to have good speedup.
- However, we can get the same or better accuracy with only 10 million points.
- In the latter case, the speedup is not so dramatic:

Results on HMC Beowulf

10 million rectangles

processors	result	error	time (sec)	effort	speedup
1	3.14159265358972	-6.44E-14	3.07	3.07	1.00
2	3.14159265358998	1.89E-13	1.54	3.08	1.99
3	3.14159265358990	1.12E-13	1.14	3.42	2.69
4	3.14159265358968	-1.10E-13	0.88	3.53	3.49
5	3.14159265358970	-8.44E-14	0.73	3.65	4.21
6	3.14159265358976	-2.71E-14	0.63	3.77	4.87
7	3.14159265358979	-4.44E-16	0.56	3.92	5.48
8	3.14159265358980	1.11E-14	0.51	4.04	6.02
9	3.14159265358979	1.33E-15	0.49	4.42	6.27
10	3.14159265358979	-8.88E-16	0.46	4.60	6.67
11	3.14159265358977	-1.47E-14	0.44	4.87	6.98
12	3.14159265358976	-2.62E-14	0.42	5.02	7.31
13	3.14159265358977	-2.22E-14	0.41	5.31	7.49
14	3.14159265358978	-6.66E-15	0.40	5.55	7.68
15	3.14159265358978	-7.55E-15	0.39	5.80	7.87

Speedup vs. Processors



What about 1 million rectangles?

- Describe what is going on using the vocabulary presented thus far.
- What do you predict for 1 million?

Results on HMC Beowulf

1 million rectangles

processors	result	error	time (sec)	effort	speedup
1	3.14159265358976	-2.98E-14	0.31	0.31	1.00
2	3.14159265358994	1.47E-13	0.16	0.33	1.94
3	3.14159265358990	1.16E-13	0.22	0.66	1.41
4	3.14159265358990	1.10E-13	0.19	0.78	1.63
5	3.14159265358992	1.28E-13	0.18	0.90	1.72
6	3.14159265358989	1.04E-13	0.17	1.03	1.82
7	3.14159265358990	1.12E-13	0.16	1.13	1.94
8	3.14159265358989	9.77E-14	0.16	1.27	1.94
9	3.14159265358987	8.62E-14	0.15	1.39	2.07
10	3.14159265358987	7.73E-14	0.19	1.87	1.63
11	3.14159265358986	7.37E-14	0.19	2.07	1.63
12	3.14159265358987	8.26E-14	0.08	0.90	3.88
13	3.14159265358987	8.22E-14	0.08	1.02	3.88
14	3.14159265358987	8.44E-14	0.08	1.14	3.88
15	3.14159265358987	8.53E-14	0.09	1.34	3.44



MPI Scatter/Gather

on-line at

<http://netlib2.cs.utk.edu/utk/papers/mpi-book/node1.html>



Gather:

Used to collect “rows” of an array

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

The outcome is *as if* each of the n processes in the group (including the root process) had executed a call to

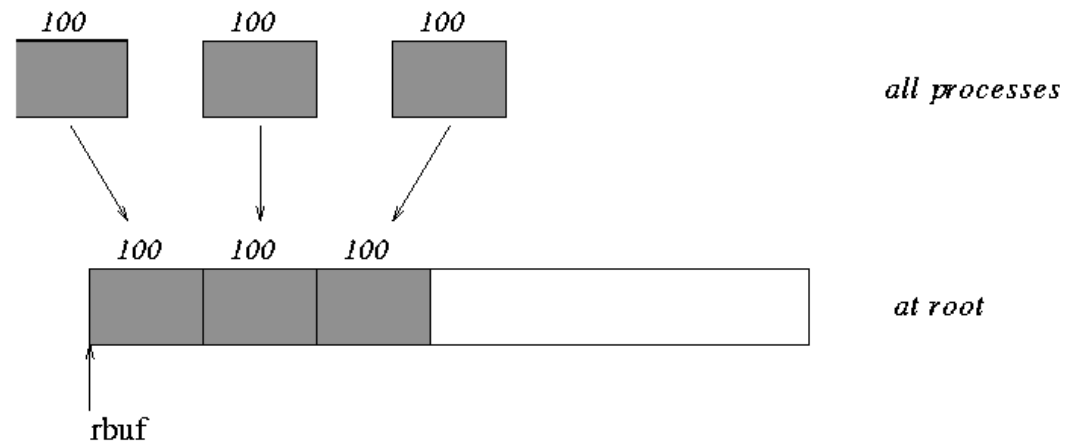
```
MPI_Send(sendbuf, sendcount, sendtype, root, ...),
```

and the root process had executed n calls to

```
MPI_Recv(recvbuf+i-recvcount-extent(recvtype), recvcount, recvtype, i, ...),
```

where `extent(recvtype)` is the type extent obtained from a call to `MPI_Type_extent()`.

Gather Example



```
MPI_Comm comm;
int gsize, sendarray[100];
int root, myrank, *rbuf;
...
MPI_Comm_rank( comm, myrank);
if ( myrank == root) {
    MPI_Comm_size( comm, &gsize);
    rbuf = (int *)malloc(gsize*100*sizeof(int));
}
MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm); 56
```

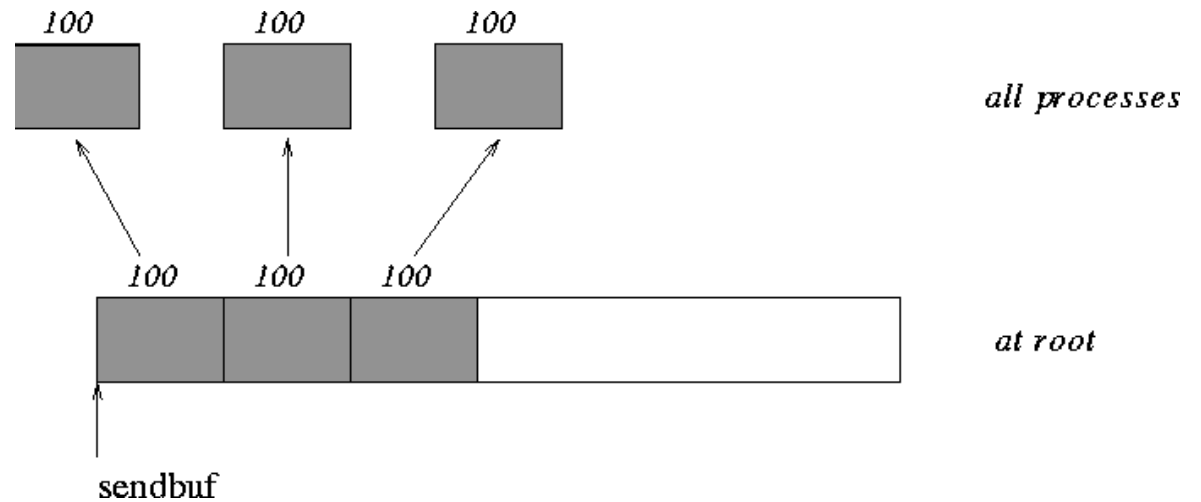
Scatter:

Used to distribute “rows” of an array

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

The outcome is *as if* the root executed n send operations,
`MPI_Send(sendbuf+i*sendcount*extent(sendtype), sendcount, sendtype, i,...)`,
 $i = 0$ to $n - 1$. and each process executed a receive,
`MPI_Recv(recvbuf, recvcount, recvtype, root,...)`.


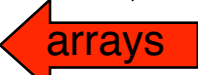
Scatter Example

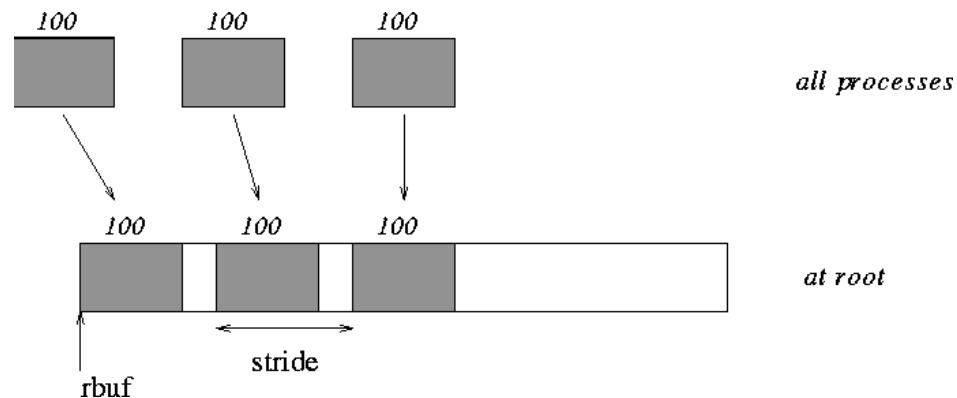


```
MPI_Comm comm;  
int gsize,*sendbuf;  
int root, rbuf[100];  
...  
MPI_Comm_size( comm, &gsize);  
sendbuf = (int *)malloc(gsize*100*sizeof(int));  
...  
MPI_Scatter( sendbuf, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);
```

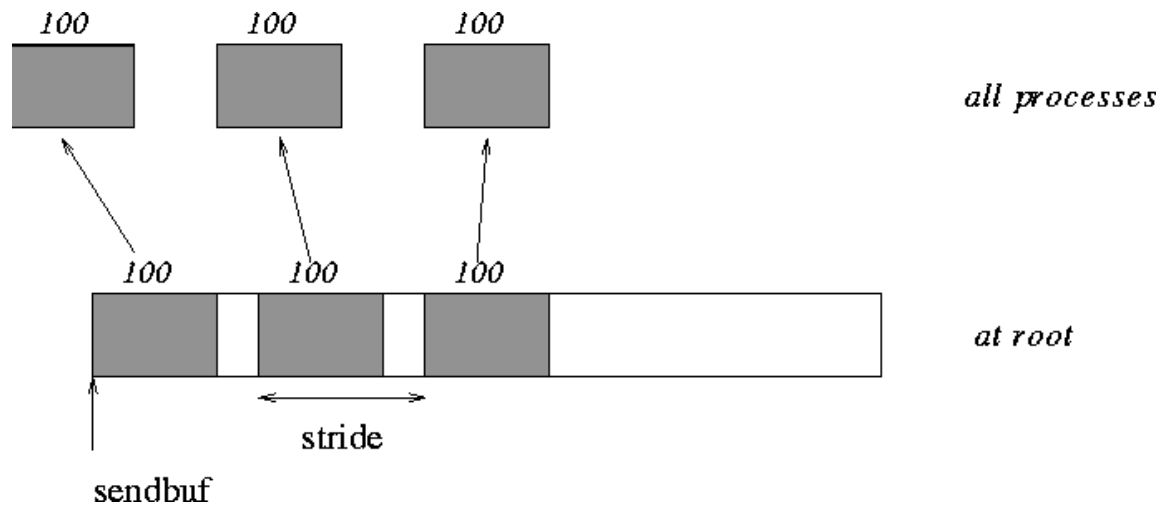
“Vector Variants”

- The **Vector Variants** allow the gathered/scattered sub-arrays to be of **different sizes**, by specifying an **array of lengths** of the sizes. They also allow there to be “gaps” (or “stride”) by making the displacements explicit.

 MPI_Gatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs,  MPI_Datatype recvttype, int root, MPI_Comm comm)



“Vector Variants”





↓
MPI_Scatterv(void* sendbuf,
int *sendcounts, int *displs, ← arrays
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm)

“All” Variants

- The **all variants** distribute the results of a gather to all processors in the communicator, rather than just one. There is a vector version analogous to gather.

MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

 MPI_Allgatherv(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int *recvcounts, int *displs,  arrays, MPI_Datatype recvtype, MPI_Comm comm)



“All-to-All” Variants

Alltoall allows everything to be scattered and gathered in one call.

Contents of distinct send buffers are sent to all receive buffers

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

```
MPI_Alltoallv(void* sendbuf, int *sendcounts,  
              int *sdispls, MPI_Datatype sendtype,  
              void* recvbuf, int *recvcounts, int *rdispls,  
              MPI_Datatype recvtype, MPI_Comm comm)
```

PVM vs. MPI

Jack Dongarra



Distinguished Professor, University of Tennessee

Distinguished Scientist, Oak Ridge National Laboratory

Also known for netlib, lapack, etc. Tutorial presentation: ⁶⁴

<http://www.netlib.org/utk/people/jd-tutorial/Presentation.html>

PVM vs. MPI

- MPI = Message-Passing Interface
 - SPMD (Single program, multiple data)
 - Each node runs the same program
 - The program “just exists”, it is not spawned explicitly
- PVM = Parallel Virtual Machine
 - “MPMD” (Multiple program, multiple data)
 - Processes are explicitly spawned
 - Processes are assigned to nodes in separate layer, possibly multiple per node

PVM

- PVM came before MPI.
- Lower level, but more flexible
- Can be used for heterogeneous network
- Arbitrary topology
- Messages can cross outside of host boundaries.
- Explicit packing and unpacking of messages required in code
- Fault tolerance features

PVM

- In PVM *daemon* processes must be resident on nodes prior to spawning PVM processes there.
- Upon command, the daemon launches the process.
- The PVM host file identifies participating nodes, or they can be added manually from the command line.
- Root process is started from pvm console command-line on one host.

PVM

- Processes explicitly spawn child processes.
- Child can determine its parent.
- Processes have their own “task id”.
- Point-to-point send/receive similar to MPI.
- Tags, wildcards similar to MPI.

Code Fragment for simple PVM process (1) (I have left out the error checking, etc.)

```
int main(int argc, char* argv[]) {  
    /* find out my task id number */  
    mytid = pvm_mytid();  
  
    /* find my parent's task id number */  
    myparent = pvm_parent();  
  
    /* if I don't have a parent then I am the parent */  
    if (myparent == PvmNoParent) {  
  
        /* spawn the child tasks */  
        info = pvm_spawn(argv[0], (char**)0, PvmTaskDefault, (char*)0,
```

Code Fragment (2)

```
/* I'm still the parent */

for (i = 0; i < ntask; i++) {
    /* receive a message from any child process */
    buf = pvm_recv(-1, JOINTAG);

    info = pvm_bufinfo(buf, &len, &tag, &tid);

    info = pvm_upkint(&mydata, 1, 1);
}
pvm_exit();
}
```

Code Fragment (3)

```
/* I'm a child */  
  
info = pvm_initsend(PvmDataDefault);  
info = pvm_pkint(&mytid, 1, 1);  
info = pvm_send(myparent, JOINTAG);  
pvm_exit();  
}
```

See <http://www.netlib.org/pvm3/> for more details.

PVM groups

- Processes explicitly join and leave groups, named symbolically.
- Multicast, gather, barriers, etc. are done relative to group, to explicit receives.
- Multicast can be into group from outside.
- Reduce operator for +, *, max, min, or *user-defined*.
- A process can be in multiple groups.