CSC4005 – Distributed and Parallel Computing

Prof. Yeh-Ching Chung

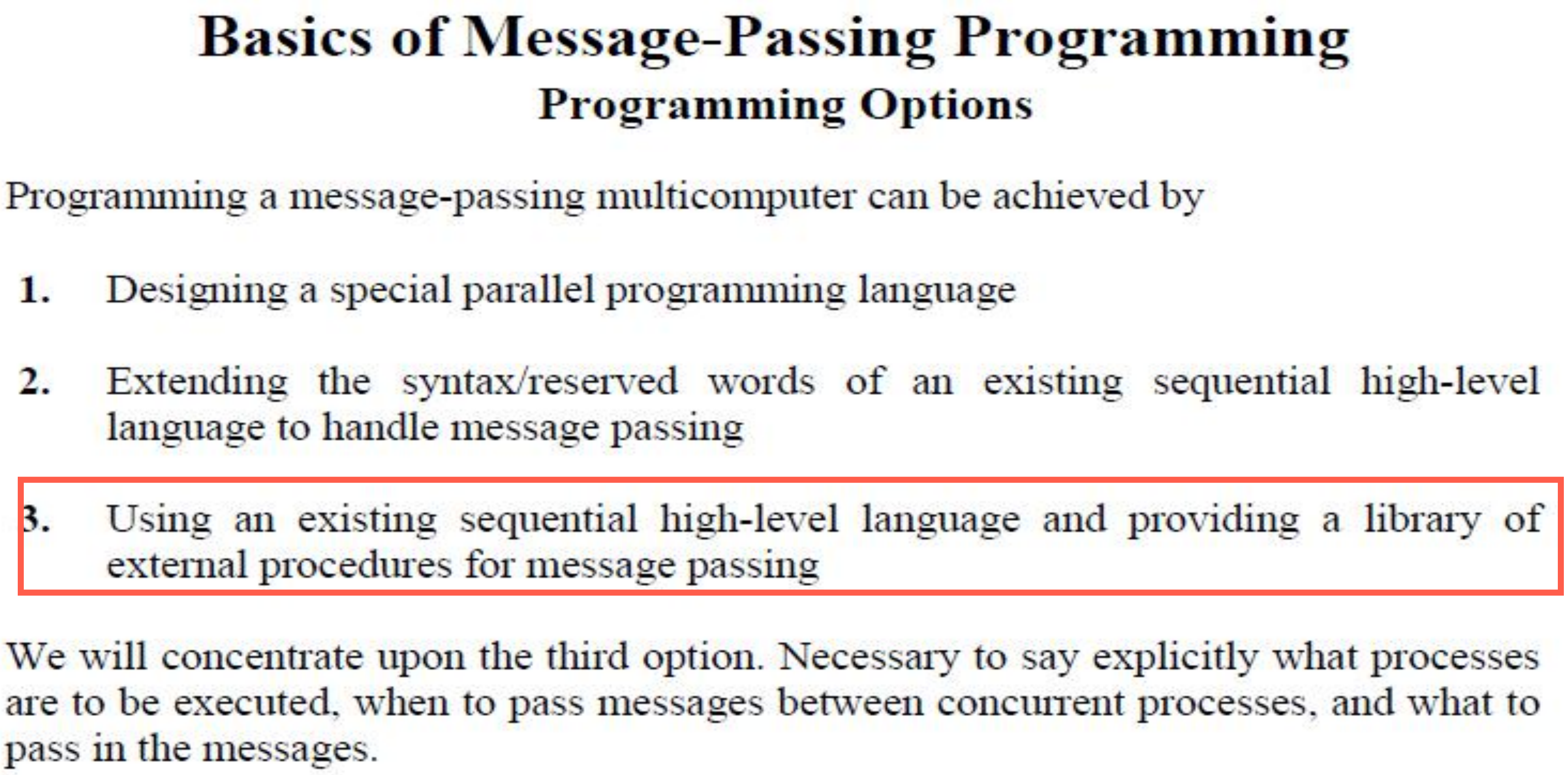
School of Data Science

Chinese University of Hong Kong, Shenzhen

Outline

- Introduction to Parallel Computers
- Message Passing Computing and Programming
- Multithreaded Programming
- CUDA Programming
- OpenMP Programming
- Embarrassingly Parallel Computations
- Partitioning and Divide–and–Conquer Strategies
- Pipelined Computations
- Synchronous Computations
- Load Balancing and Termination Detection
- Sorting Algorithms

Message-Passing Computing



Basics of Message-Passing Programming

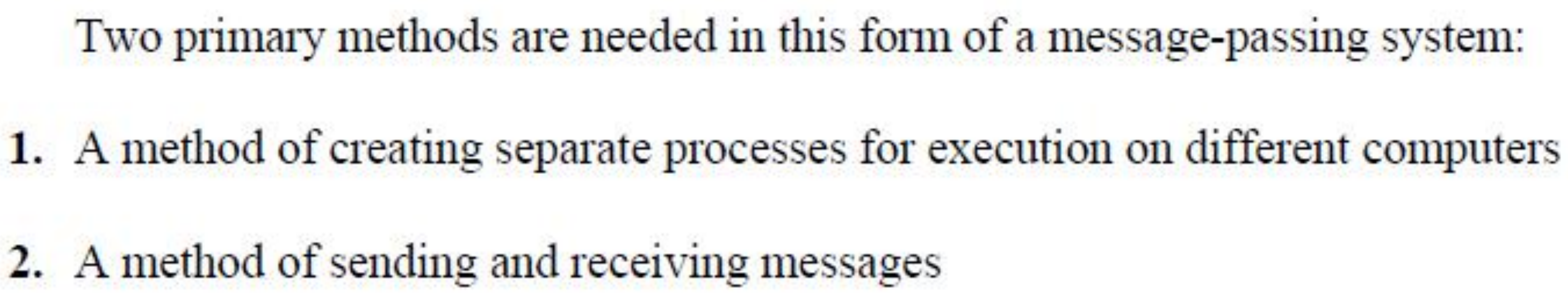
Programming Options

Programming a message-passing multicomputer can be achieved by

1. Designing a special parallel programming language
2. Extending the syntax/reserved words of an existing sequential high-level language to handle message passing
3. Using an existing sequential high-level language and providing a library of external procedures for message passing

We will concentrate upon the third option. Necessary to say explicitly what processes are to be executed, when to pass messages between concurrent processes, and what to pass in the messages.

Goals



Two primary methods are needed in this form of a message-passing system:

1. A method of creating separate processes for execution on different computers
2. A method of sending and receiving messages

Single Program Multiple Data (SPMD) Model

Different processes are merged into one program. Within the program are control statements that will customize the code; i.e. select different parts for each process.

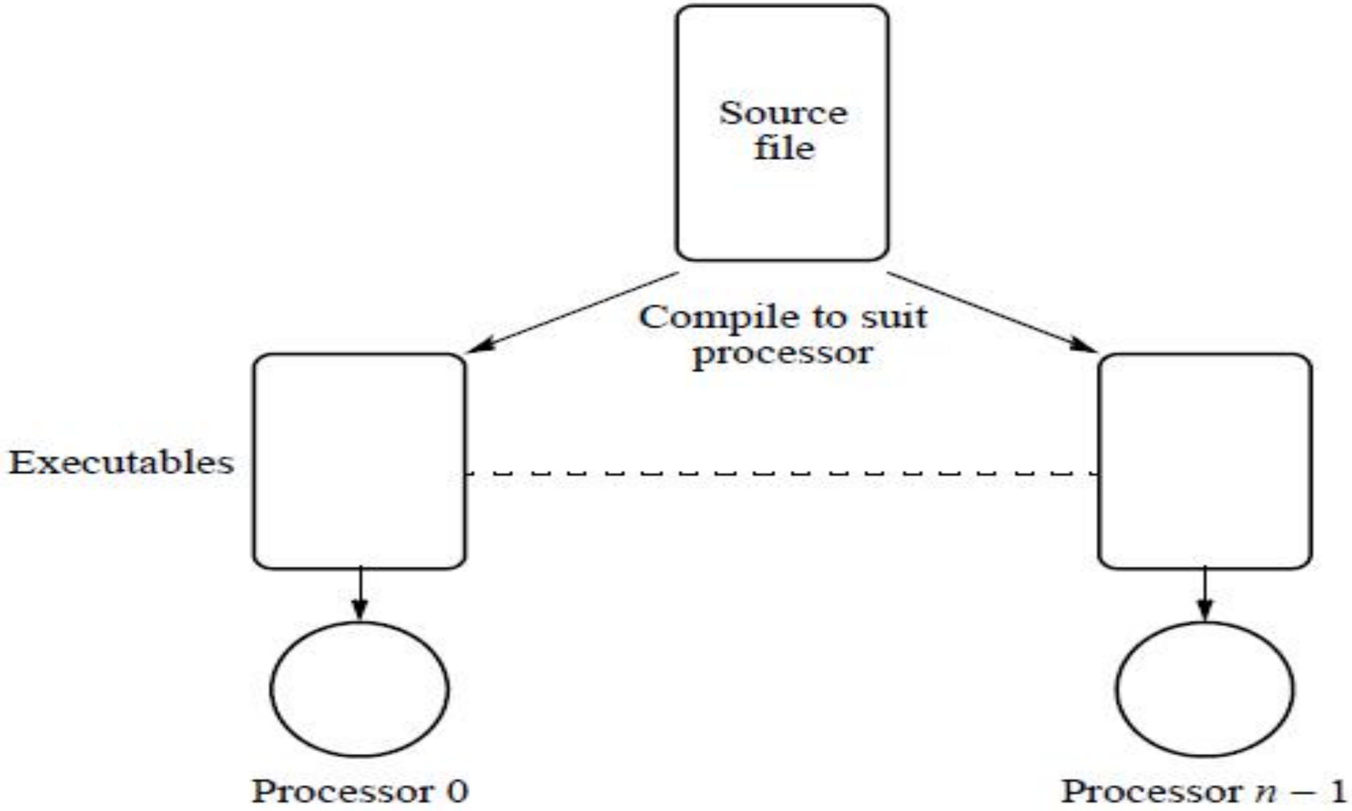
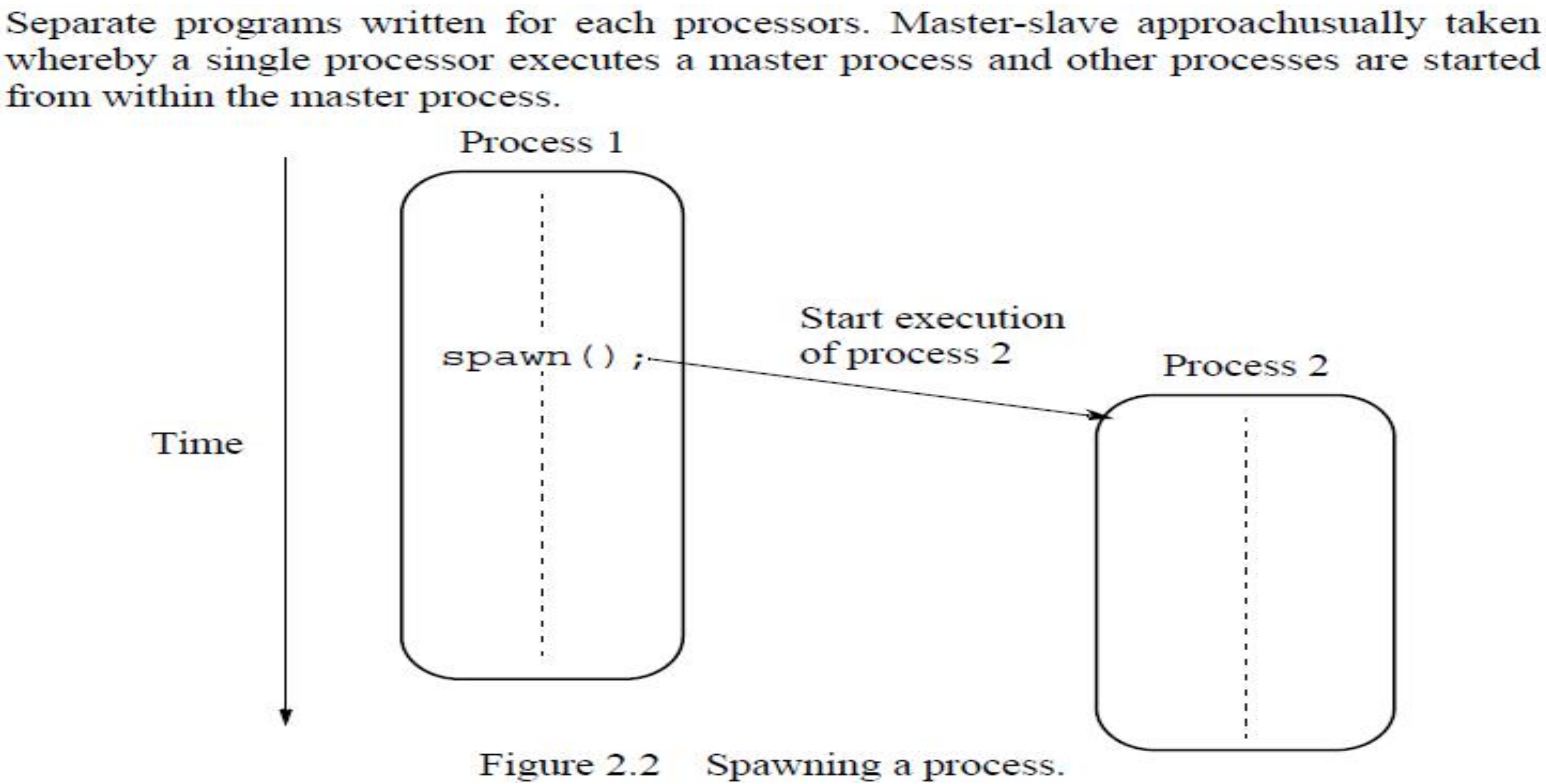


Figure 2.1 Single program, multiple data operation.

Multiple Program Multiple Data (MPMD) Model



Separate programs written for each processors. Master-slave approachusually taken whereby a single processor executes a master process and other processes are started from within the master process.

Figure 2.2 Spawning a process.

Basic Send and Receive Routines

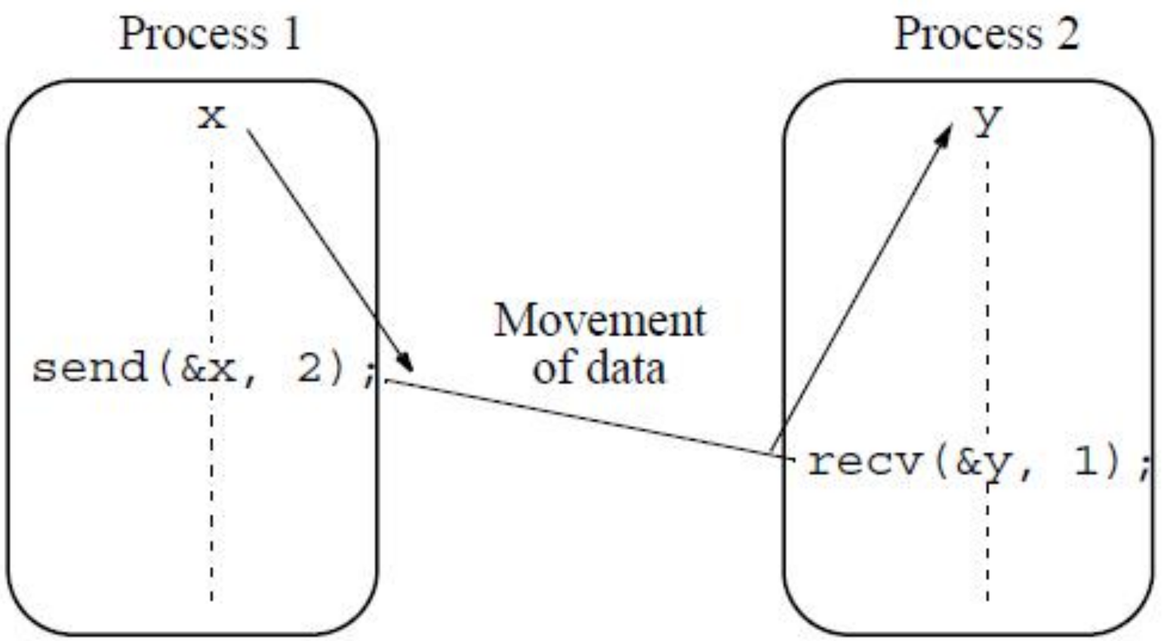
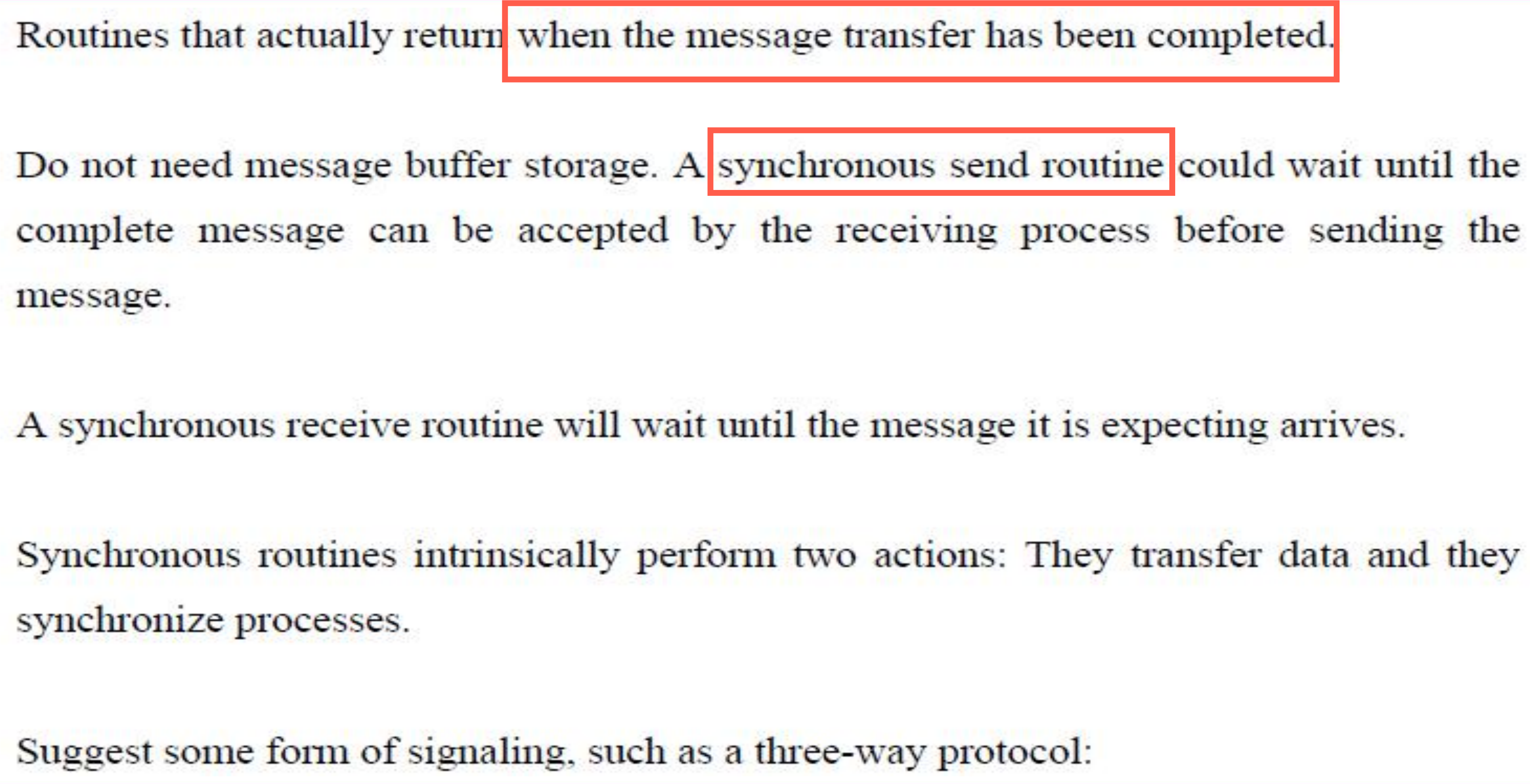


Figure 2.3 Passing a message between processes using `send()` and `recv()` library calls.

Synchronous Message Passing (1)



Routines that actually return when the message transfer has been completed.

Do not need message buffer storage. A synchronous send routine could wait until the complete message can be accepted by the receiving process before sending the message.

A synchronous receive routine will wait until the message it is expecting arrives.

Synchronous routines intrinsically perform two actions: They transfer data and they synchronize processes.

Suggest some form of signaling, such as a three-way protocol:

Synchronous Message Passing (2)

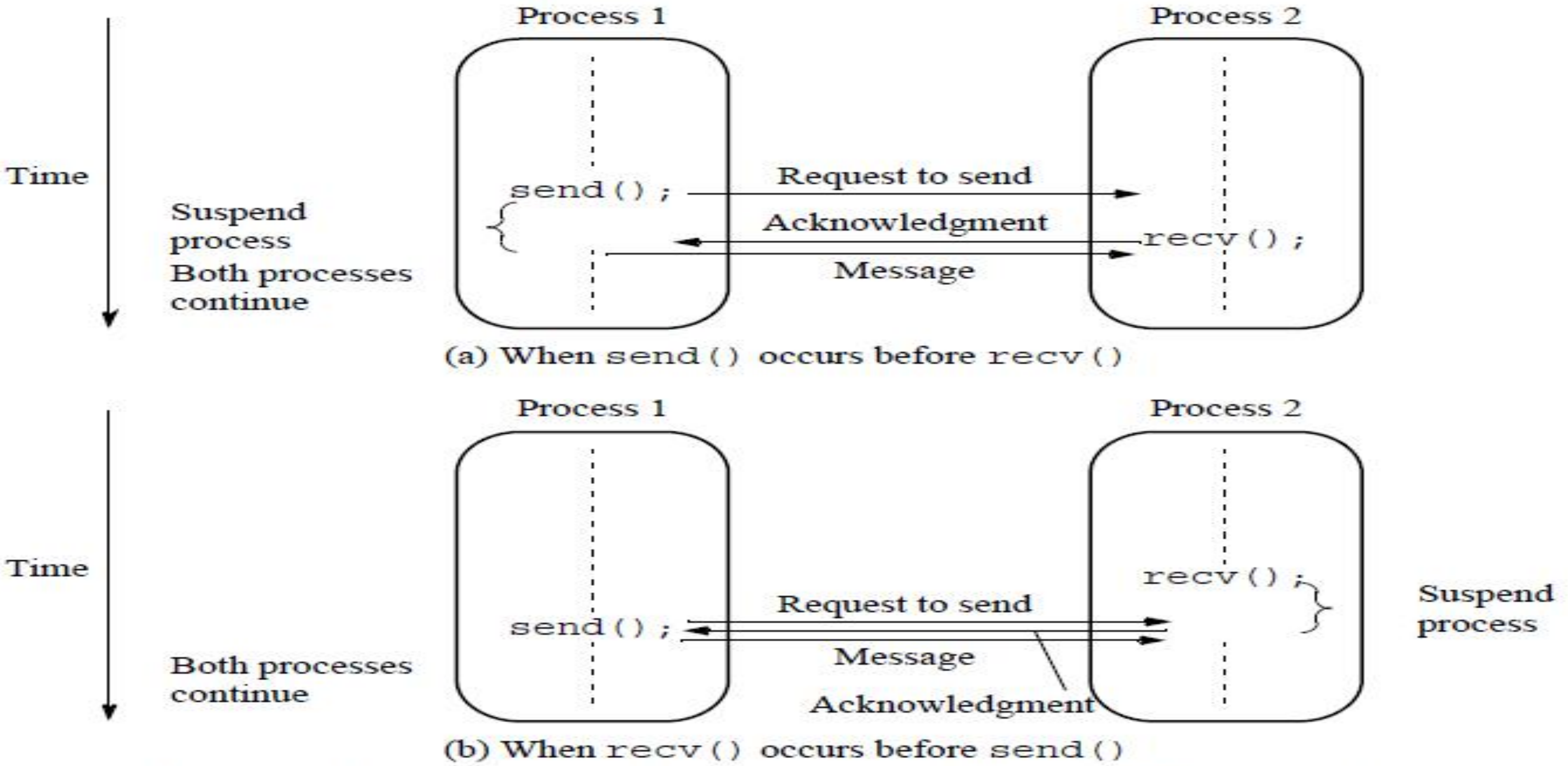
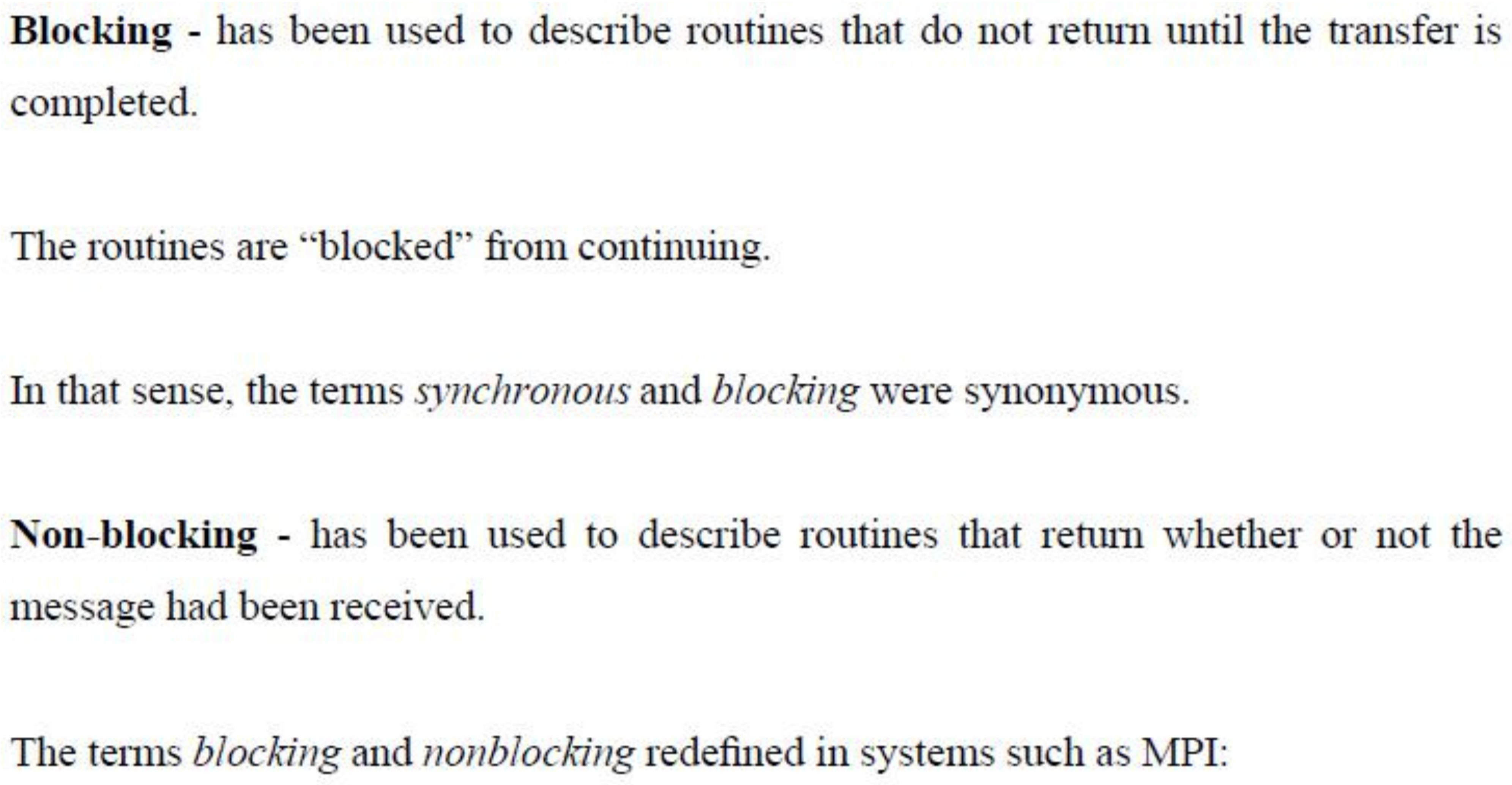


Figure 2.4 Synchronous send ( ) and recv ( ) library calls using a three-way

Blocking and Nonblocking Message Passing



Blocking - has been used to describe routines that do not return until the transfer is completed.

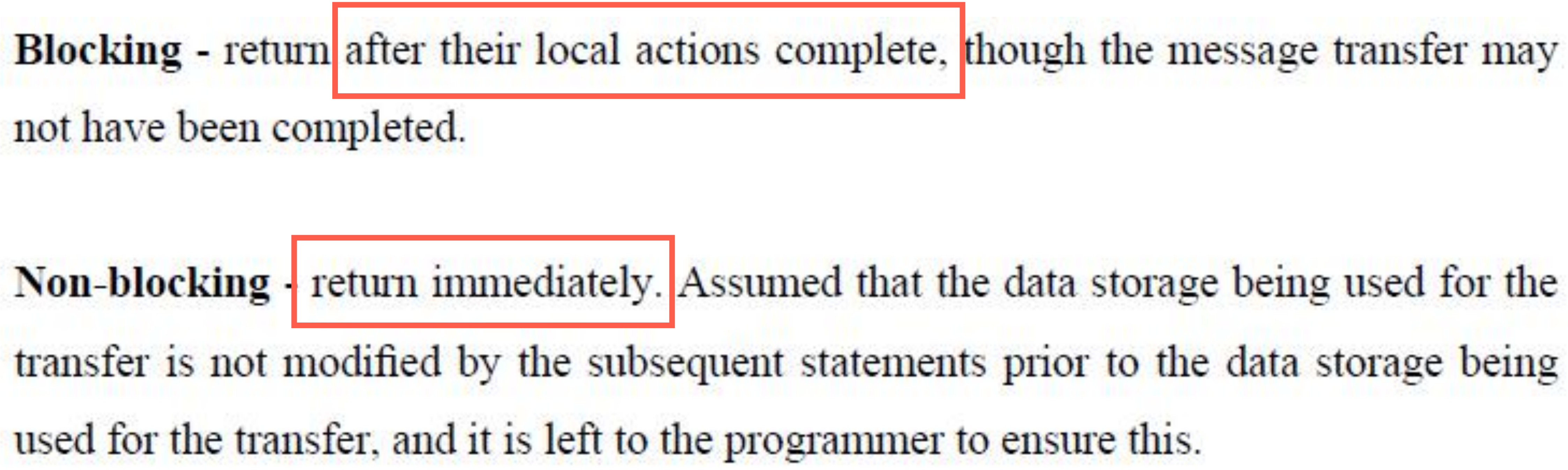
The routines are “blocked” from continuing.

In that sense, the terms *synchronous* and *blocking* were synonymous.

Non-blocking - has been used to describe routines that return whether or not the message had been received.

The terms *blocking* and *nonblocking* redefined in systems such as MPI:

MPI Definitions pf Blocking and Nonblocking



Blocking - return after their local actions complete, though the message transfer may not have been completed.

Non-blocking - return immediately. Assumed that the data storage being used for the transfer is not modified by the subsequent statements prior to the data storage being used for the transfer, and it is left to the programmer to ensure this.

How Message-Passing Routines Can Return Before the Message Transfer Has Been Completed

Generally, a *message buffer* needed between source and destination to hold message:

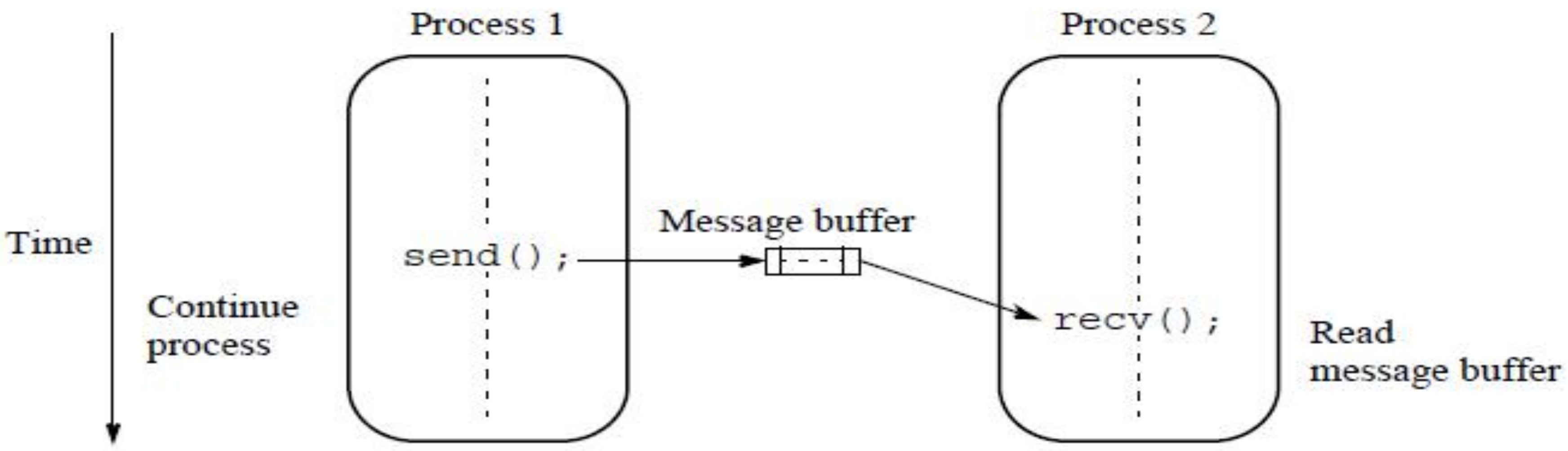
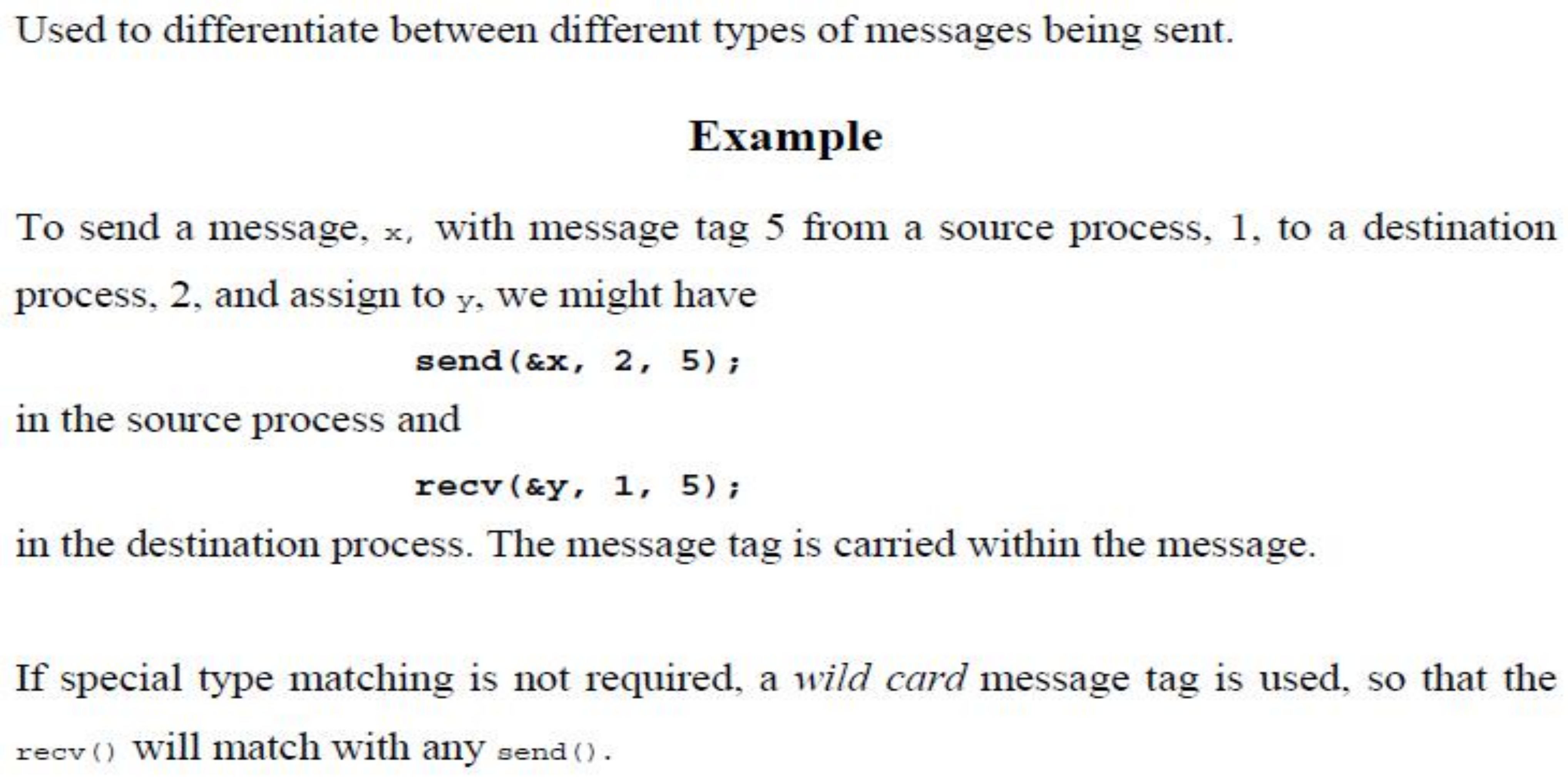


Figure 2.5 Using a message buffer.

For send routine, once the local actions have been completed and the message is safely on its way, the process can continue with subsequent work. Buffers can only be of finite length and a point could be reached when the send routine is held up because all the available buffer space has been exhausted.

Message Tag



Used to differentiate between different types of messages being sent.

Example

To send a message, `x`, with message tag 5 from a source process, 1, to a destination process, 2, and assign to `y`, we might have

```
send(&x, 2, 5);
```

in the source process and

```
recv(&y, 1, 5);
```

in the destination process. The message tag is carried within the message.

If special type matching is not required, a *wild card* message tag is used, so that the `recv()` will match with any `send()`.

Broadcast

Sending the same message to all the processes concerned with the problem.

Multicast - sending the same message to a defined group of processes.

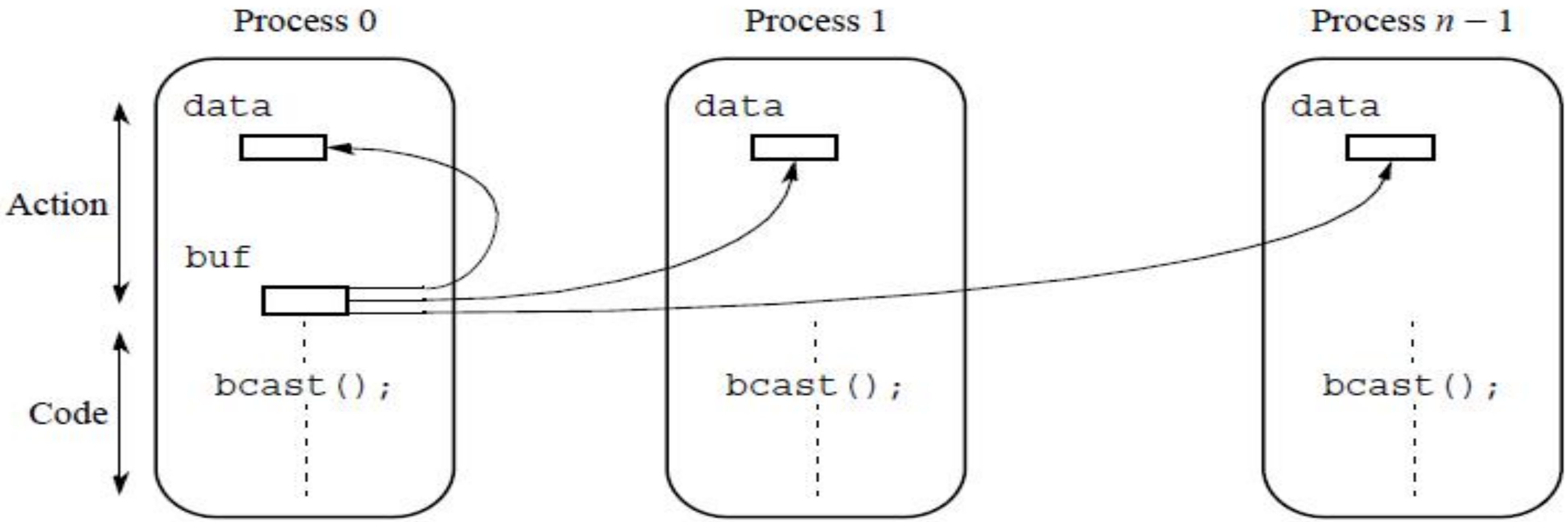


Figure 2.6 Broadcast operation.

Scatter

Sending each element of an array of data in the root to a separate process. The contents of the *i*th location of the array is sent to the *i*th process.

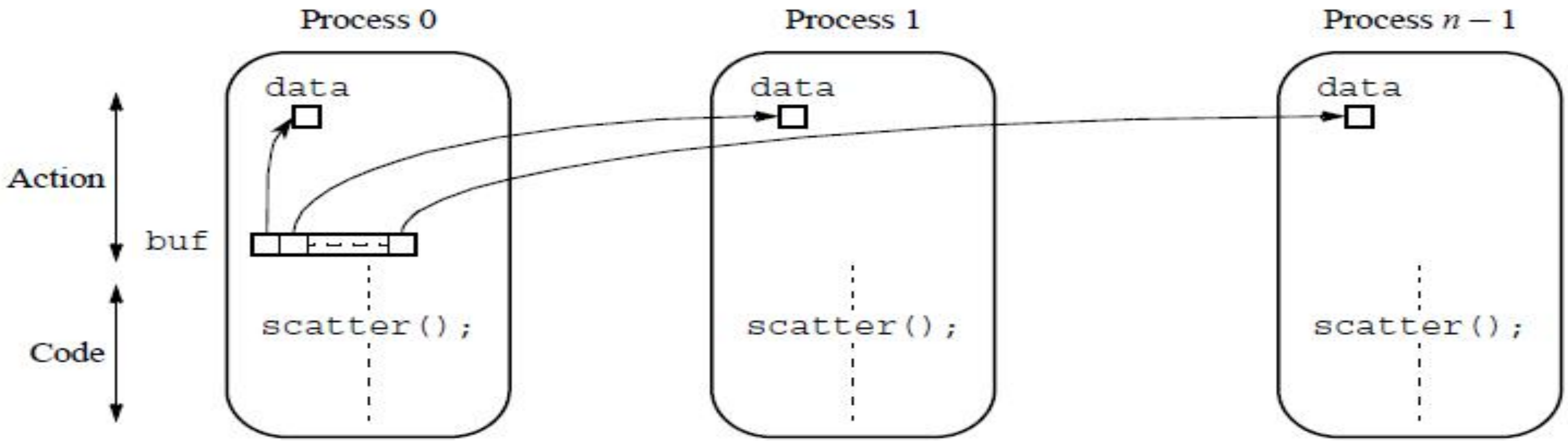


Figure 2.7 Scatter operation.

Gather

Having one process collect individual values from a set of processes.

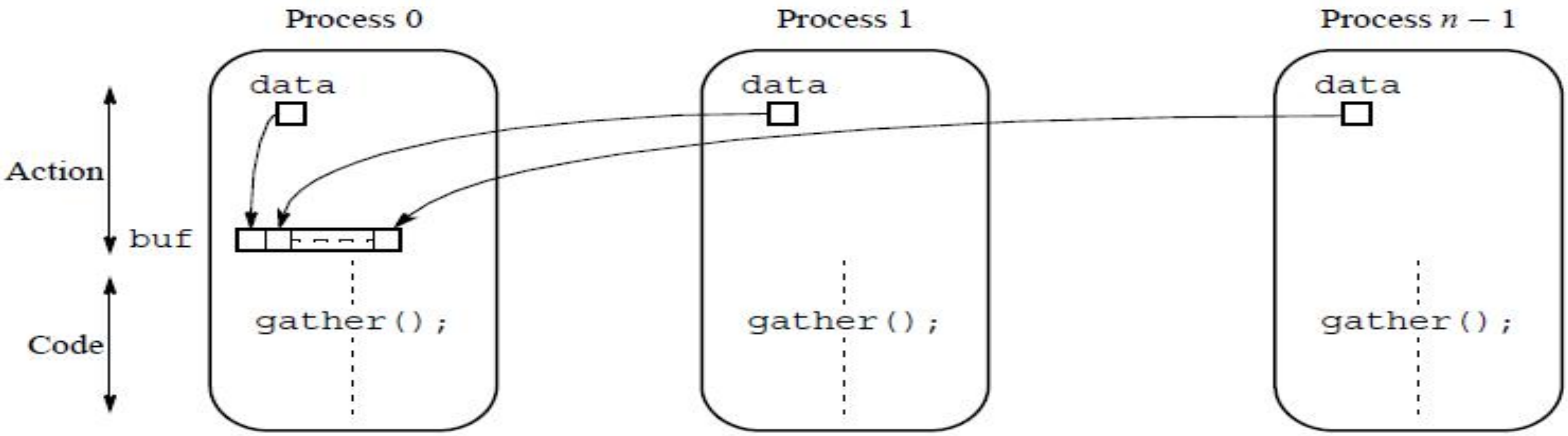


Figure 2.8 Gather operation.

Reduce

Gather operation combined with a specified arithmetic or logical operation. Example, the values could be gathered and then added together by the root:

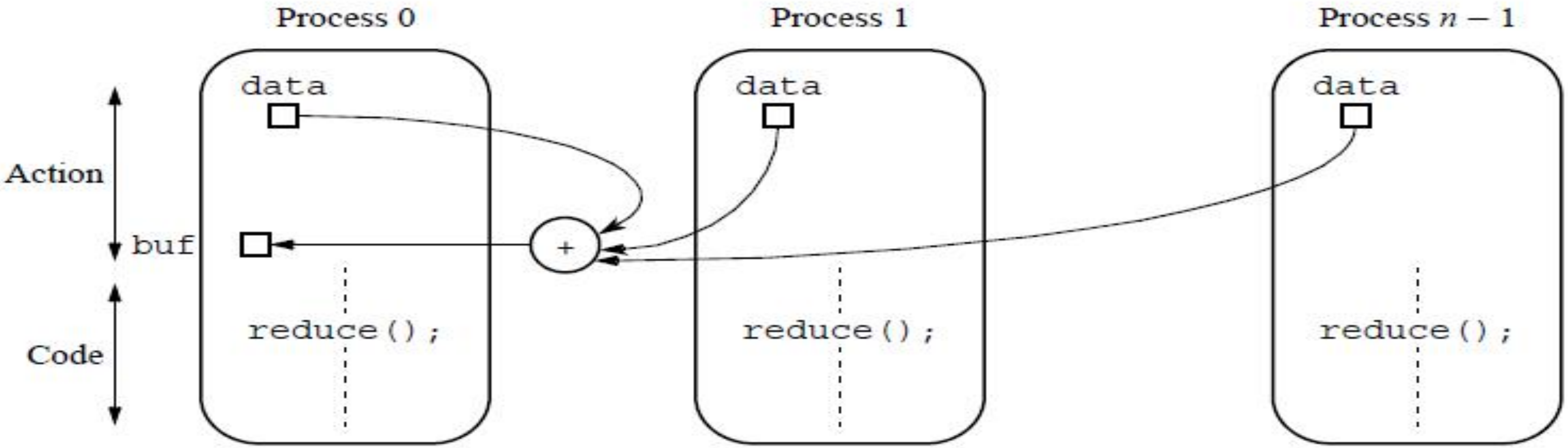
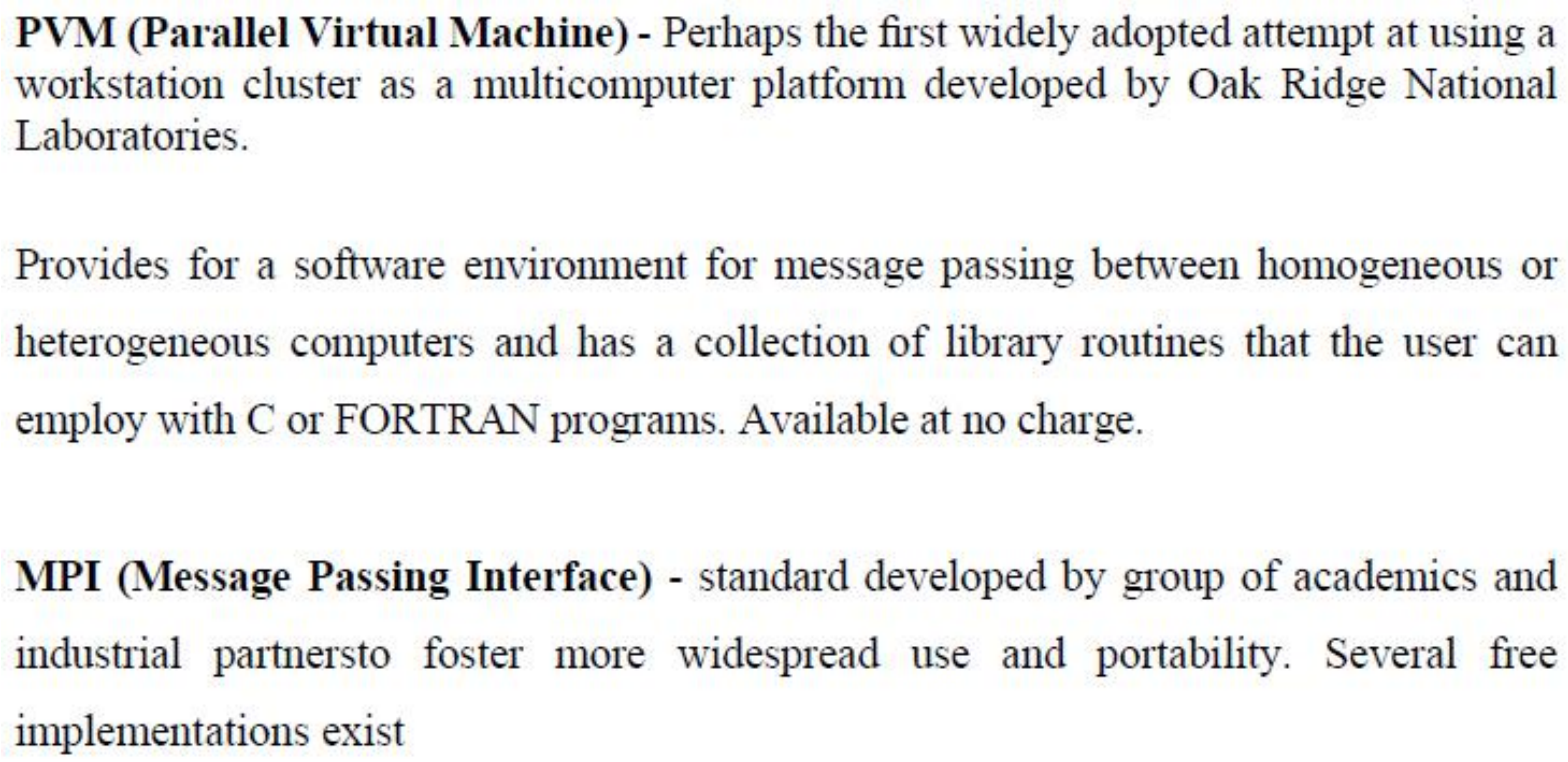


Figure 2.9 Reduce operation (addition).

Using Workstation Clusters – Software Tools

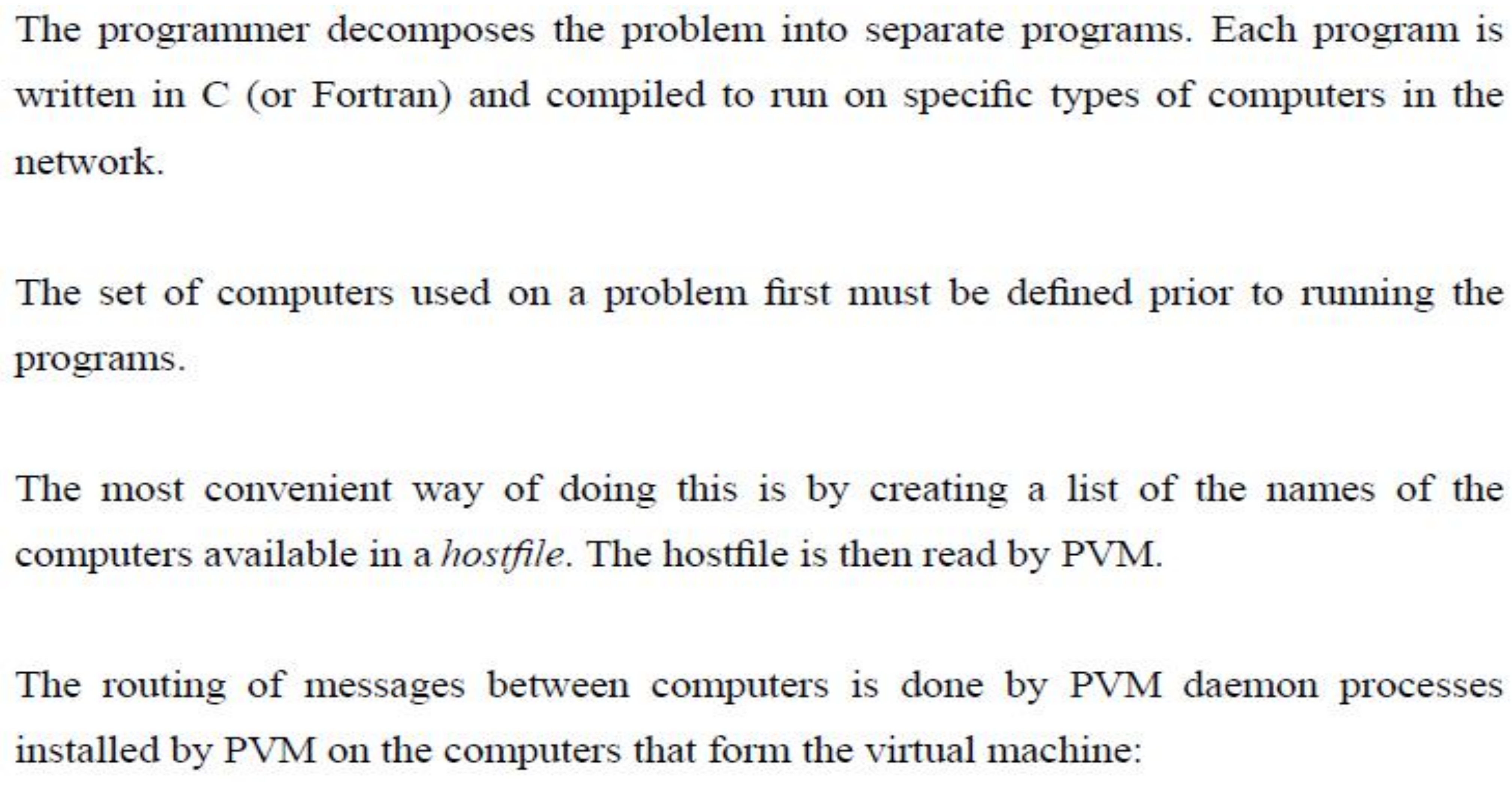


PVM (Parallel Virtual Machine) - Perhaps the first widely adopted attempt at using a workstation cluster as a multicomputer platform developed by Oak Ridge National Laboratories.

Provides for a software environment for message passing between homogeneous or heterogeneous computers and has a collection of library routines that the user can employ with C or FORTRAN programs. Available at no charge.

MPI (Message Passing Interface) - standard developed by group of academics and industrial partnersto foster more widespread use and portability. Several free implementations exist

PVM (1)



The programmer decomposes the problem into separate programs. Each program is written in C (or Fortran) and compiled to run on specific types of computers in the network.

The set of computers used on a problem first must be defined prior to running the programs.

The most convenient way of doing this is by creating a list of the names of the computers available in a *hostfile*. The hostfile is then read by PVM.

The routing of messages between computers is done by PVM daemon processes installed by PVM on the computers that form the virtual machine:

PVM (2)

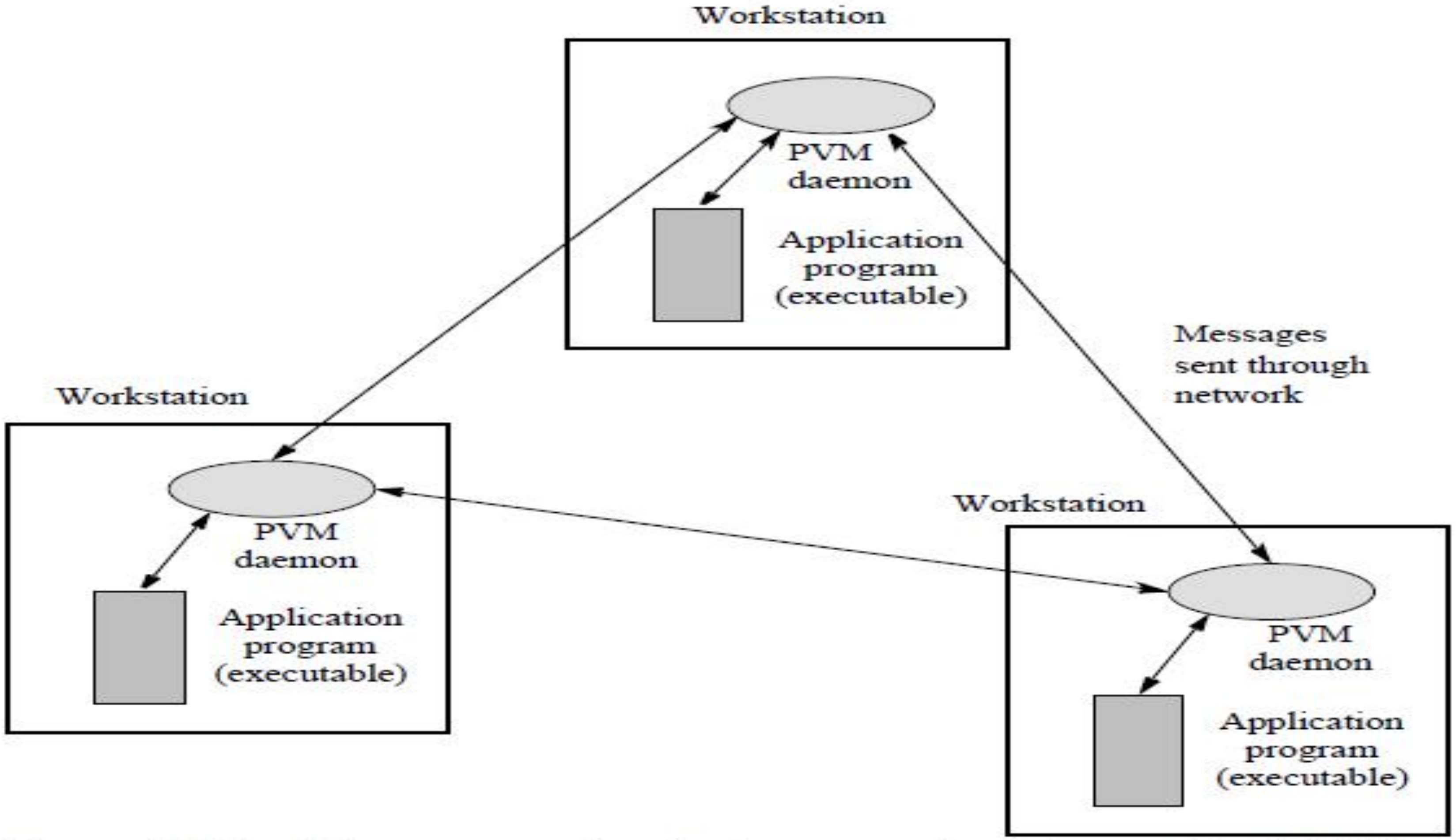

Workstation
PVM daemon
Application program (executable)
Messages sent through network
Workstation
PVM daemon
Application program (executable)
Workstation
PVM daemon
Application program (executable)

Figure 2.10 Message passing between workstations using PVM.

PVM (3)

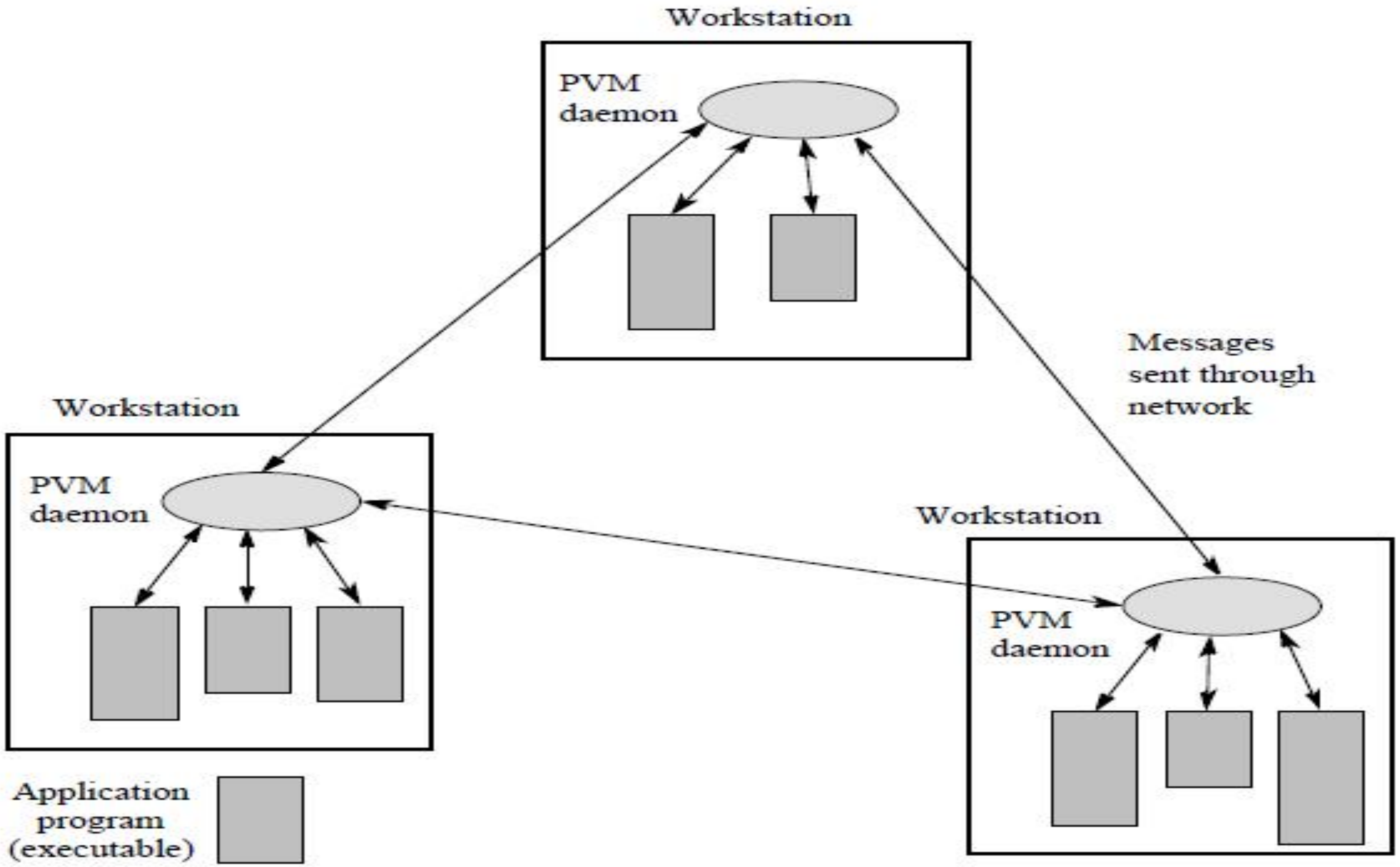
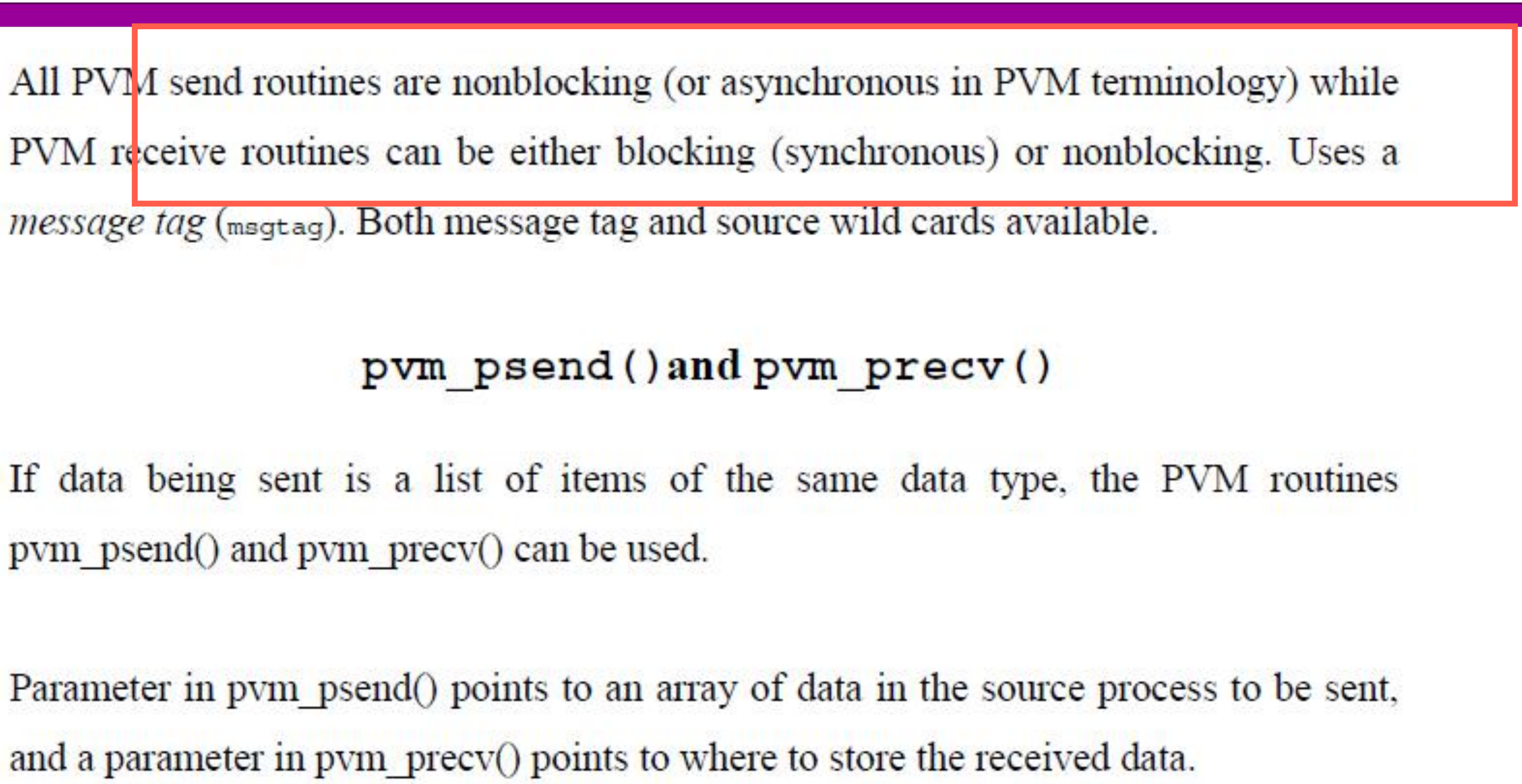


Figure 2.11 Multiple processes allocated to each processor (workstation).

Basic Message-Passing Routines (1)



All PVM send routines are nonblocking (or asynchronous in PVM terminology) while PVM receive routines can be either blocking (synchronous) or nonblocking. Uses a *message tag* (`msgtag`). Both message tag and source wild cards available.

`pvm_psend()` and `pvm_precv()`

If data being sent is a list of items of the same data type, the PVM routines pvm_psend() and pvm_precv() can be used.

Parameter in pvm_psend() points to an array of data in the source process to be sent, and a parameter in pvm_precv() points to where to store the received data.

Basic Message-Passing Routines (2)

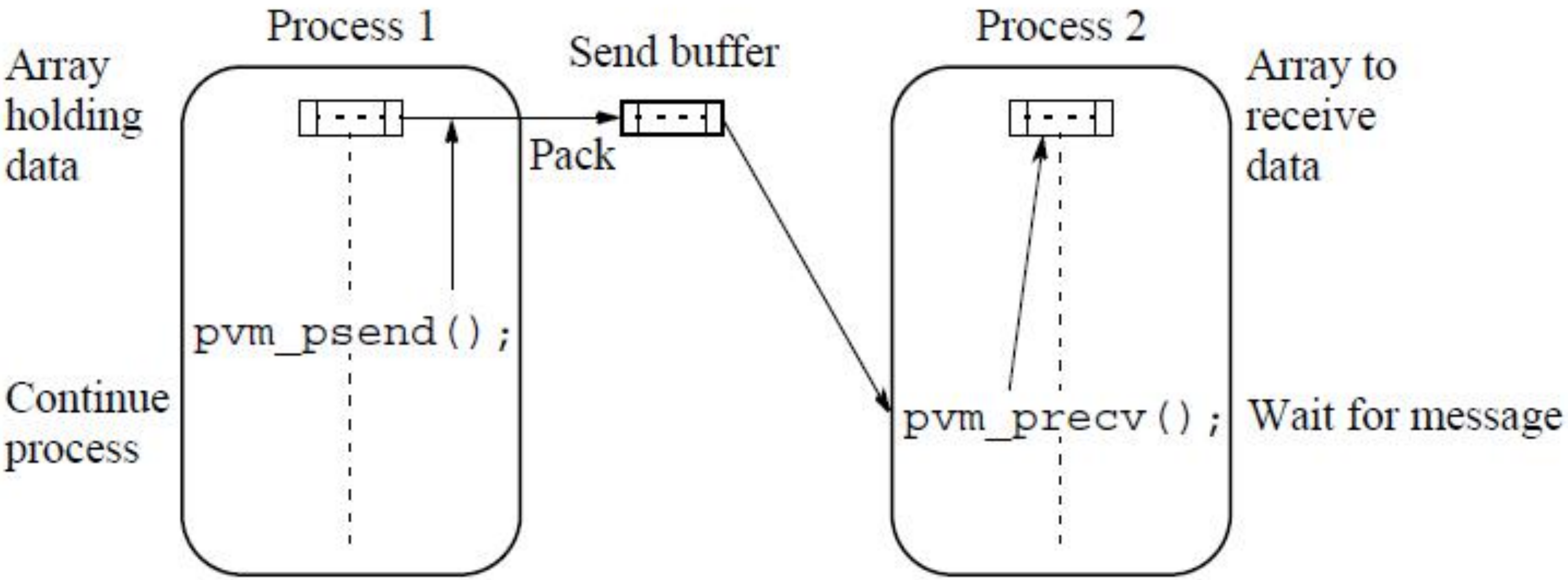
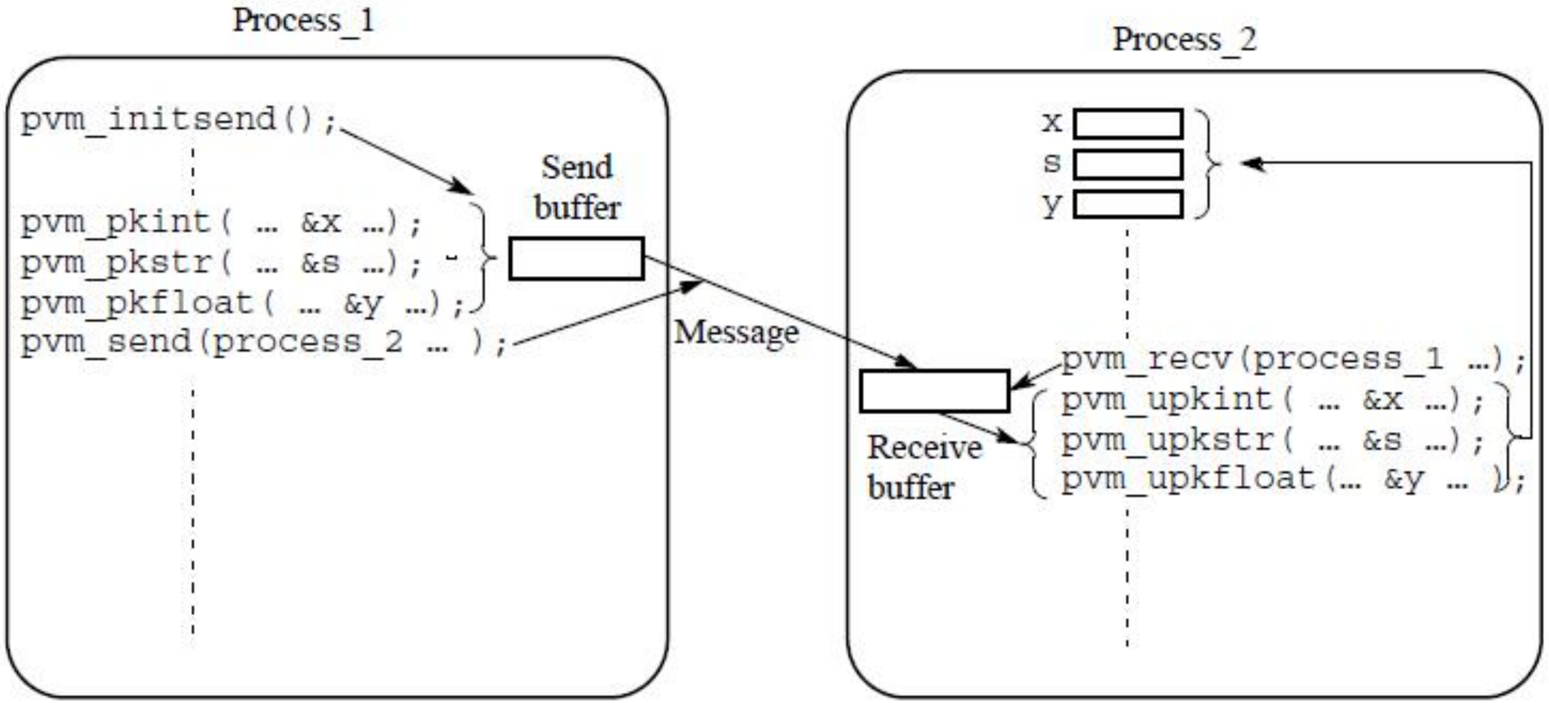


Figure 2.12 pvm_psend() and pvm_precv() system calls.

Sending Data Composed of Various Types

Data packed into a send buffer prior to sending data. Receiving process must unpack its receive buffer according to format in which it was packed. Specific packing and unpacking routines for each datatype.

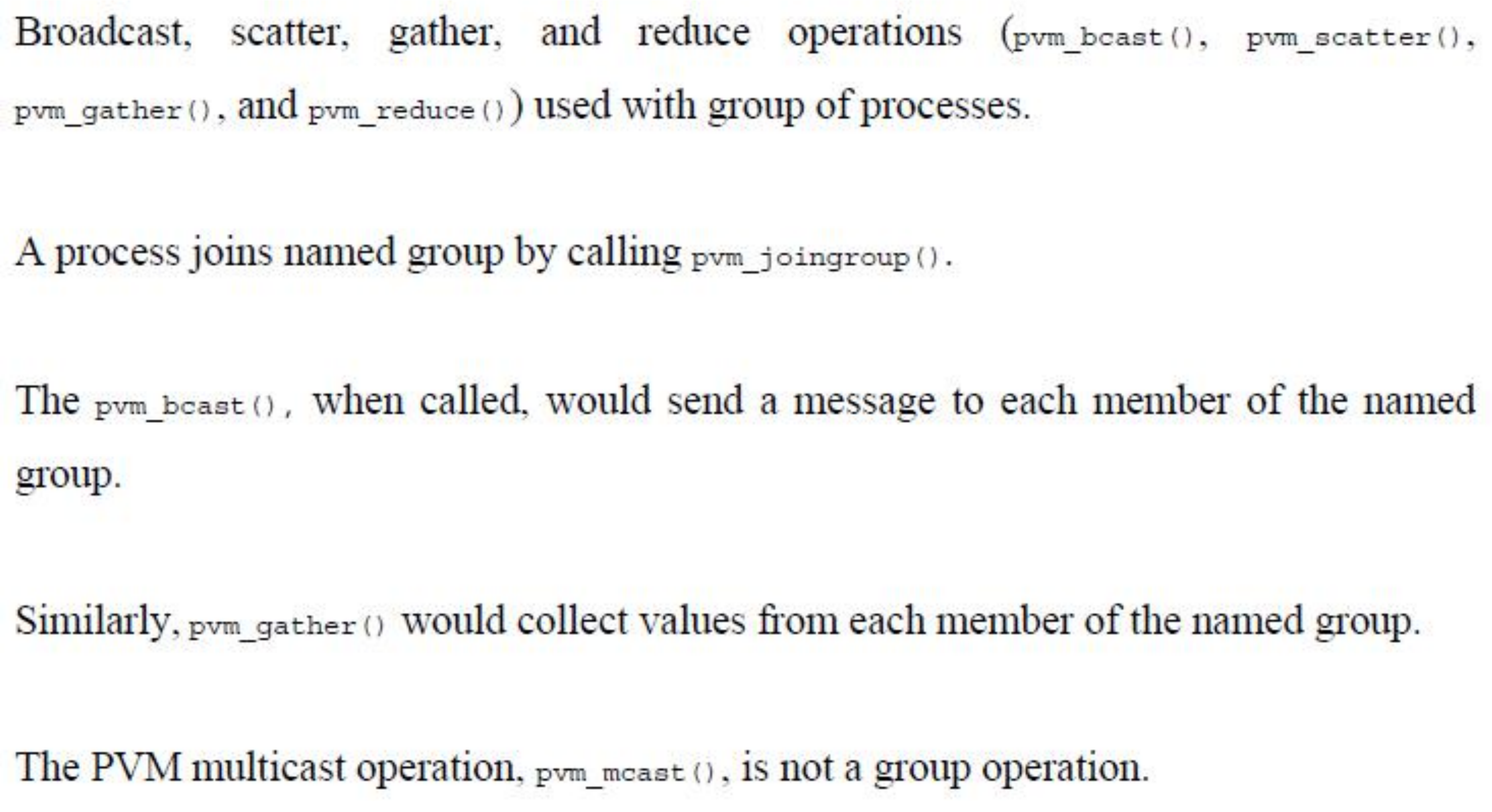


```
pvm_initsend();
pvm_pkint( … &x …);
pvm_pkstr( … &s …);
pvm_pkfloat( … &y …);
pvm_send(process_2 … );
```

```
pvm_recv(process_1 …);
pvm_upkint( … &x …);
pvm_upkstr( … &s …);
pvm_upkfloat(… &y … );
```

Figure 2.13 PVM packing messages, sending, and unpacking.

Broadcast, Multicast, Scatter, Gather, and Reduce



Broadcast, scatter, gather, and reduce operations (pvm_bcast(), pvm_scatter(), pvm_gather(), and pvm_reduce()) used with group of processes.

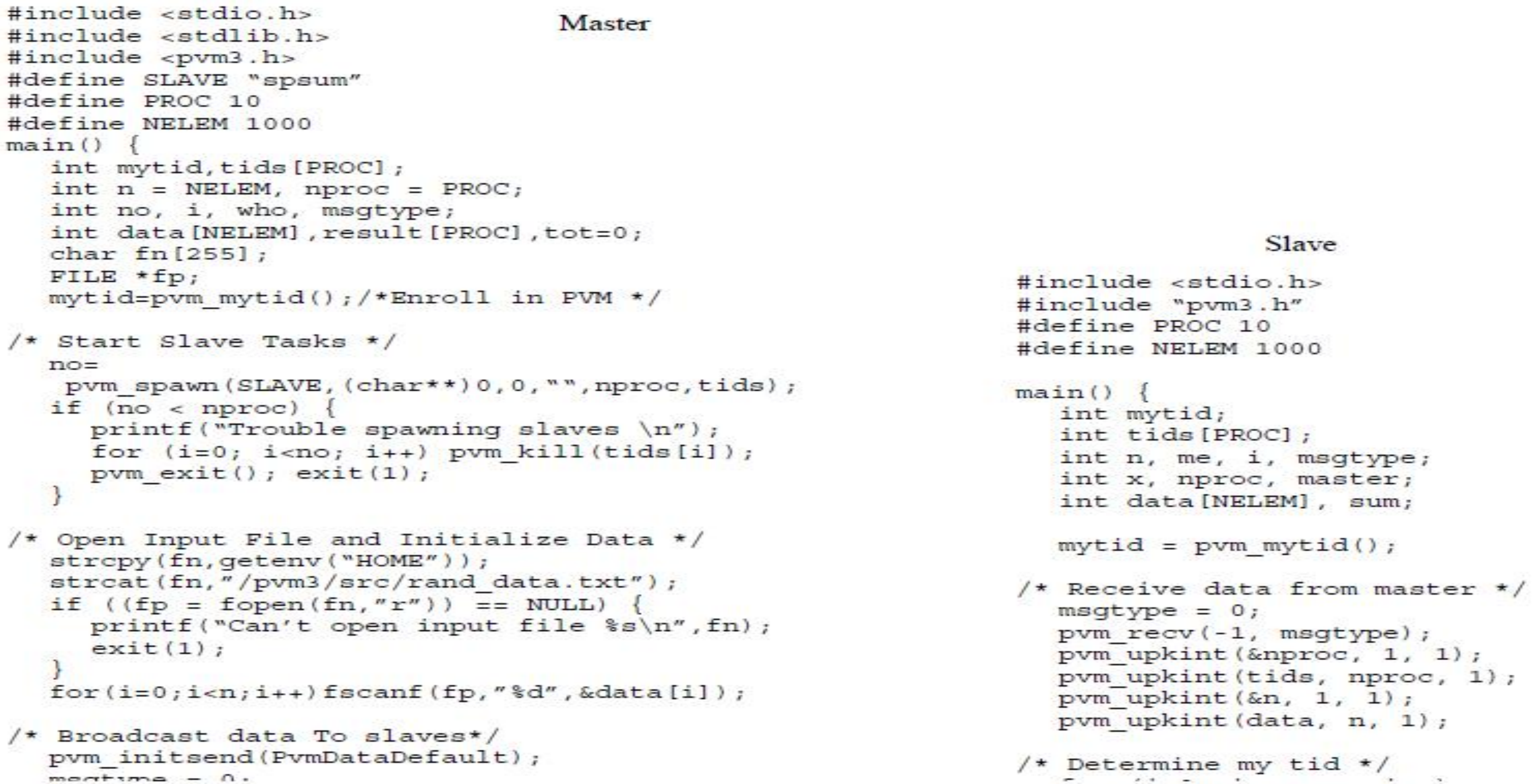
A process joins named group by calling pvm_joingroup().

The pvm_bcast(), when called, would send a message to each member of the named group.

Similarly, pvm_gather() would collect values from each member of the named group.

The PVM multicast operation, pvm_mcast(), is not a group operation.

Example - PVM Program (1)



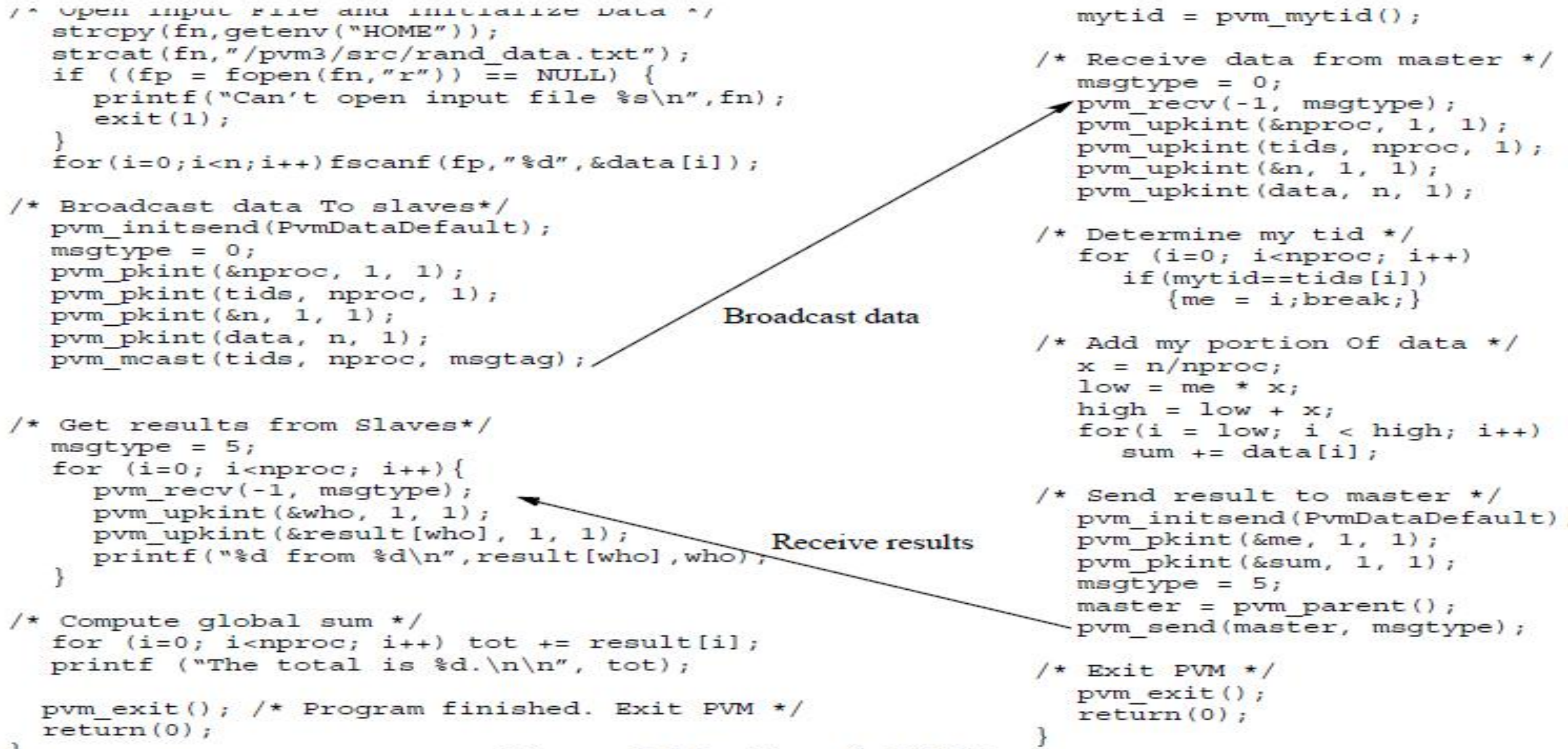
Master

```
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>
#define SLAVE “spsum”
#define PROC 10
#define NELEM 1000
main() {
   int mytid,tids[PROC];
   int n = NELEM, nproc = PROC;
   int no, i, who, msgtype;
   int data[NELEM],result[PROC],tot=0;
   char fn[255];
   FILE *fp;
   mytid=pvm_mytid();/*Enroll in PVM */

/* Start Slave Tasks */
   no=
    pvm_spawn(SLAVE,(char**)0,0,“”,nproc,tids);
   if (no < nproc) {
      printf(“Trouble spawning slaves \n”);
      for (i=0; i<no; i++) pvm_kill(tids[i]);
      pvm_exit(); exit(1);
   }

/* Open Input File and Initialize Data */
   strcpy(fn,getenv(“HOME”));
   strcat(fn,”/pvm3/src/rand_data.txt”);
   if ((fp = fopen(fn,”r”)) == NULL) {
      printf(“Can’t open input file %s\n”,fn);
      exit(1);
   }
   for(i=0;i<n;i++)fscanf(fp,”%d”,&data[i]);

/* Broadcast data To slaves*/
   pvm_initsend(PvmDataDefault);
   msgtype = 0;
```

Slave

```
#include <stdio.h>
#include “pvm3.h”
#define PROC 10
#define NELEM 1000

main() {
   int mytid;
   int tids[PROC];
   int n, me, i, msgtype;
   int x, nproc, master;
   int data[NELEM], sum;

   mytid = pvm_mytid();

/* Receive data from master */
   msgtype = 0;
   pvm_recv(-1, msgtype);
   pvm_upkint(&nproc, 1, 1);
   pvm_upkint(tids, nproc, 1);
   pvm_upkint(&n, 1, 1);
   pvm_upkint(data, n, 1);

/* Determine my tid */
```

Example - PVM Program (2)

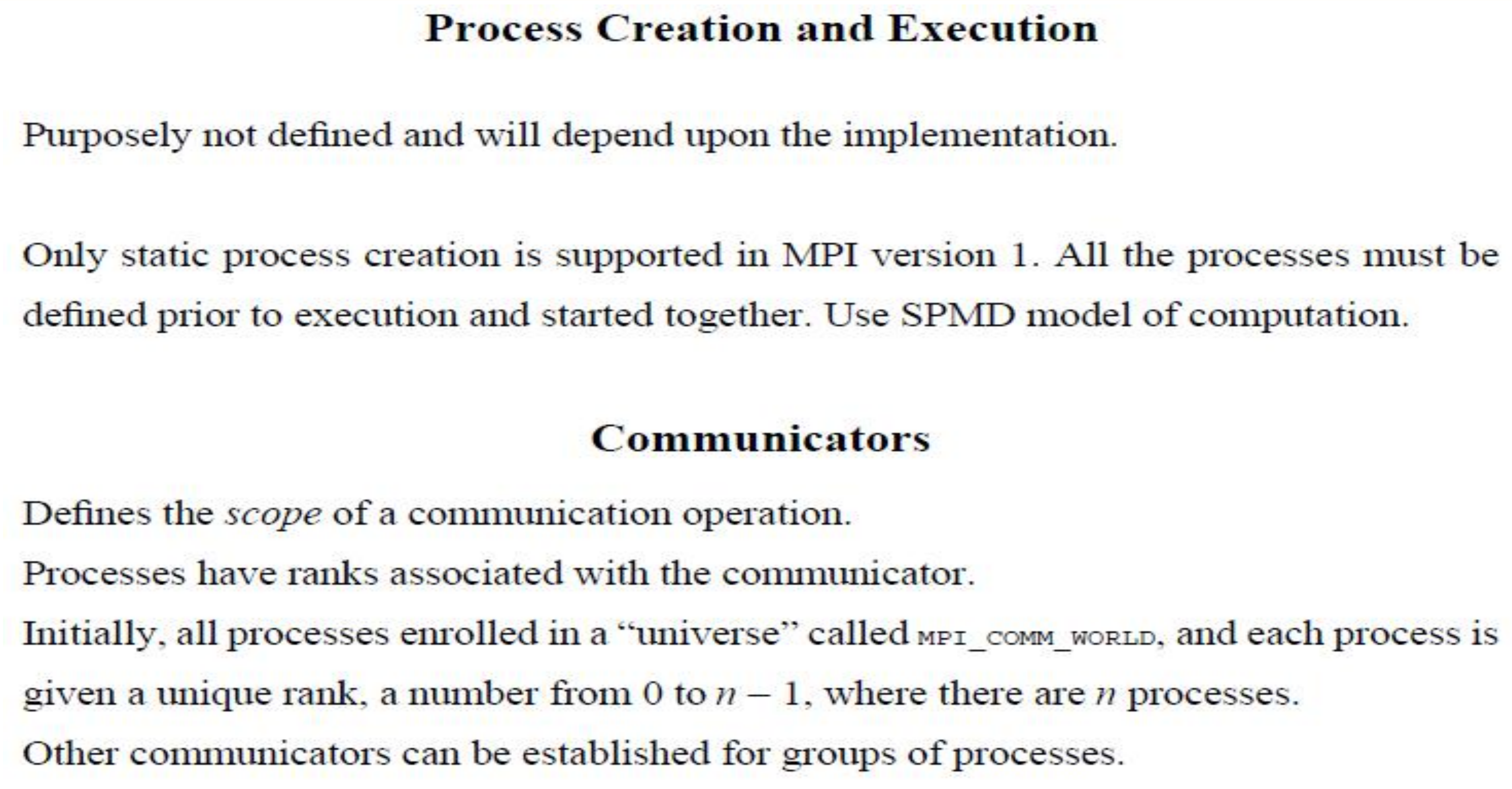


```
/* Open input File and Initialize Data */
   strcpy(fn,getenv("HOME"));
   strcat(fn,"/pvm3/src/rand_data.txt");
   if ((fp = fopen(fn,"r")) == NULL) {
      printf("Can't open input file %s\n",fn);
      exit(1);
   }
   for(i=0;i<n;i++)fscanf(fp,"%d",&data[i]);

/* Broadcast data To slaves*/
   pvm_initsend(PvmDataDefault);
   msgtype = 0;
   pvm_pkint(&nproc, 1, 1);
   pvm_pkint(tids, nproc, 1);
   pvm_pkint(&n, 1, 1);
   pvm_pkint(data, n, 1);
   pvm_mcast(tids, nproc, msgtag);


/* Get results from Slaves*/
   msgtype = 5;
   for (i=0; i<nproc; i++){
      pvm_recv(-1, msgtype);
      pvm_upkint(&who, 1, 1);
      pvm_upkint(&result[who], 1, 1);
      printf("%d from %d\n",result[who],who);
   }

/* Compute global sum */
   for (i=0; i<nproc; i++) tot += result[i];
   printf ("The total is %d.\n\n", tot);

  pvm_exit(); /* Program finished. Exit PVM */
  return(0);
}
```

```
   mytid = pvm_mytid();

/* Receive data from master */
   msgtype = 0;
   pvm_recv(-1, msgtype);
   pvm_upkint(&nproc, 1, 1);
   pvm_upkint(tids, nproc, 1);
   pvm_upkint(&n, 1, 1);
   pvm_upkint(data, n, 1);

/* Determine my tid */
   for (i=0; i<nproc; i++)
      if(mytid==tids[i])
         {me = i;break;}

/* Add my portion Of data */
   x = n/nproc;
   low = me * x;
   high = low + x;
   for(i = low; i < high; i++)
      sum += data[i];

/* Send result to master */
   pvm_initsend(PvmDataDefault);
   pvm_pkint(&me, 1, 1);
   pvm_pkint(&sum, 1, 1);
   msgtype = 5;
   master = pvm_parent();
   pvm_send(master, msgtype);

/* Exit PVM */
   pvm_exit();
   return(0);
}
```

Figure 2.15 Sample PVM program.

MPI



Process Creation and Execution

Purposely not defined and will depend upon the implementation.

Only static process creation is supported in MPI version 1. All the processes must be defined prior to execution and started together. Use SPMD model of computation.

Communicators

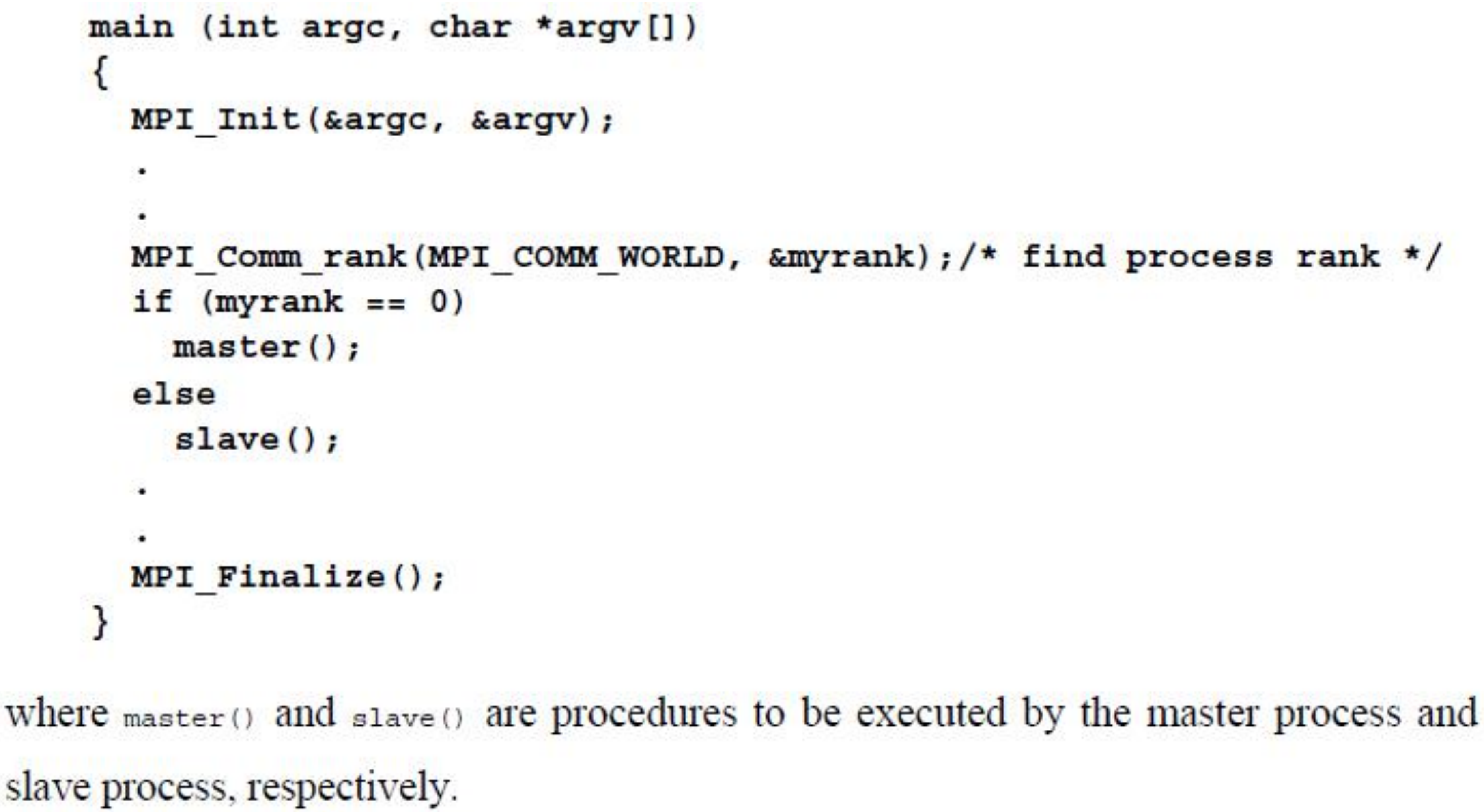
Defines the *scope* of a communication operation.

Processes have ranks associated with the communicator.

Initially, all processes enrolled in a “universe” called `MPI_COMM_WORLD`, and each process is given a unique rank, a number from 0 to $n - 1$, where there are n processes.

Other communicators can be established for groups of processes.

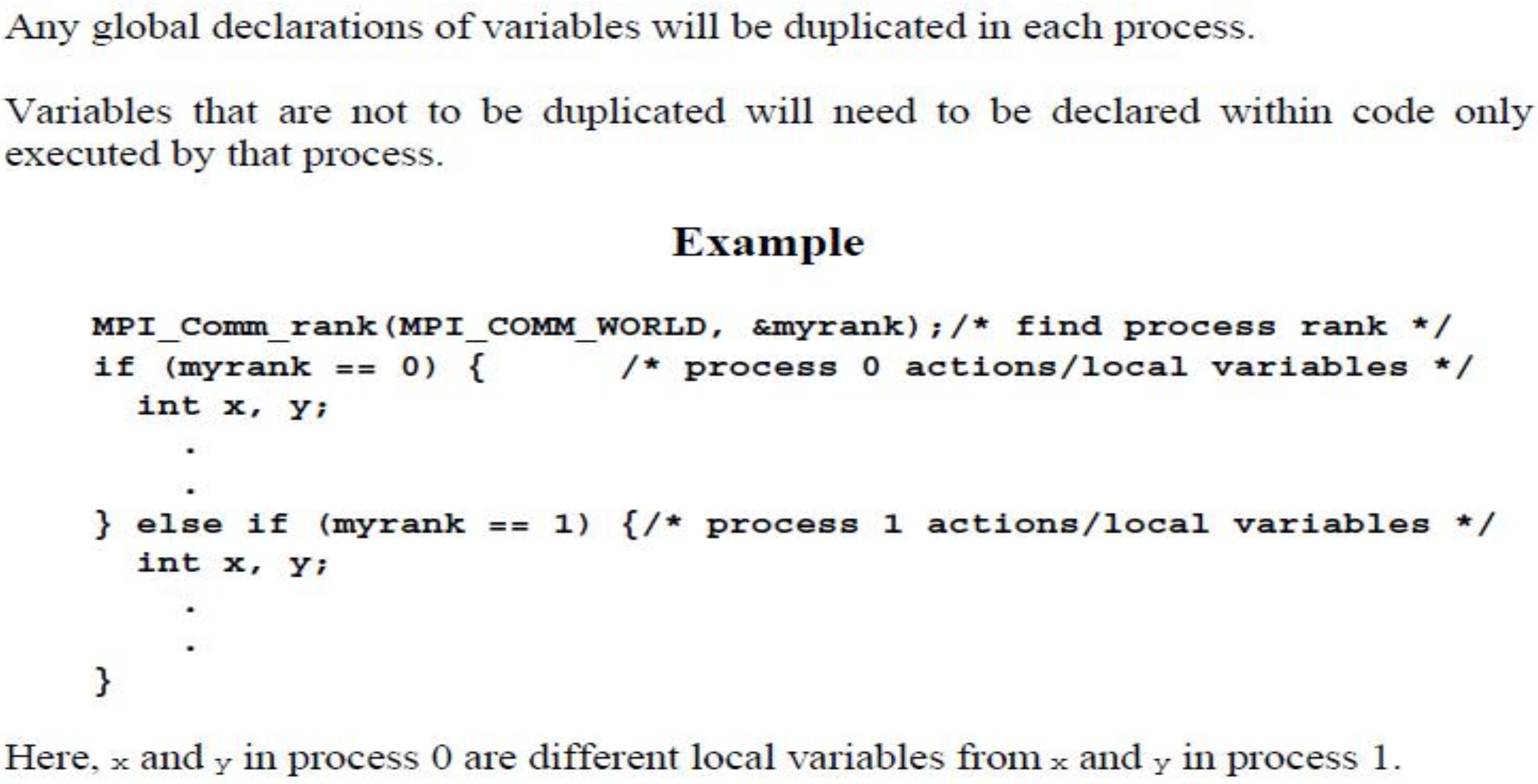
Using the SPMD Computational Model



```
main (int argc, char *argv[])
{
  MPI_Init(&argc, &argv);
  .
  .
  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find process rank */
  if (myrank == 0)
    master();
  else
    slave();
  .
  .
  MPI_Finalize();
}
```

where `master()` and `slave()` are procedures to be executed by the master process and slave process, respectively.

Global and Local Variables



Any global declarations of variables will be duplicated in each process.

Variables that are not to be duplicated will need to be declared within code only executed by that process.

Example

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find process rank */
if (myrank == 0) {        /* process 0 actions/local variables */
  int x, y;
      .
      .
} else if (myrank == 1) {/* process 1 actions/local variables */
  int x, y;
      .
      .
}
```

Here, x and y in process 0 are different local variables from x and y in process 1.

Unsafe Message Passing with Library

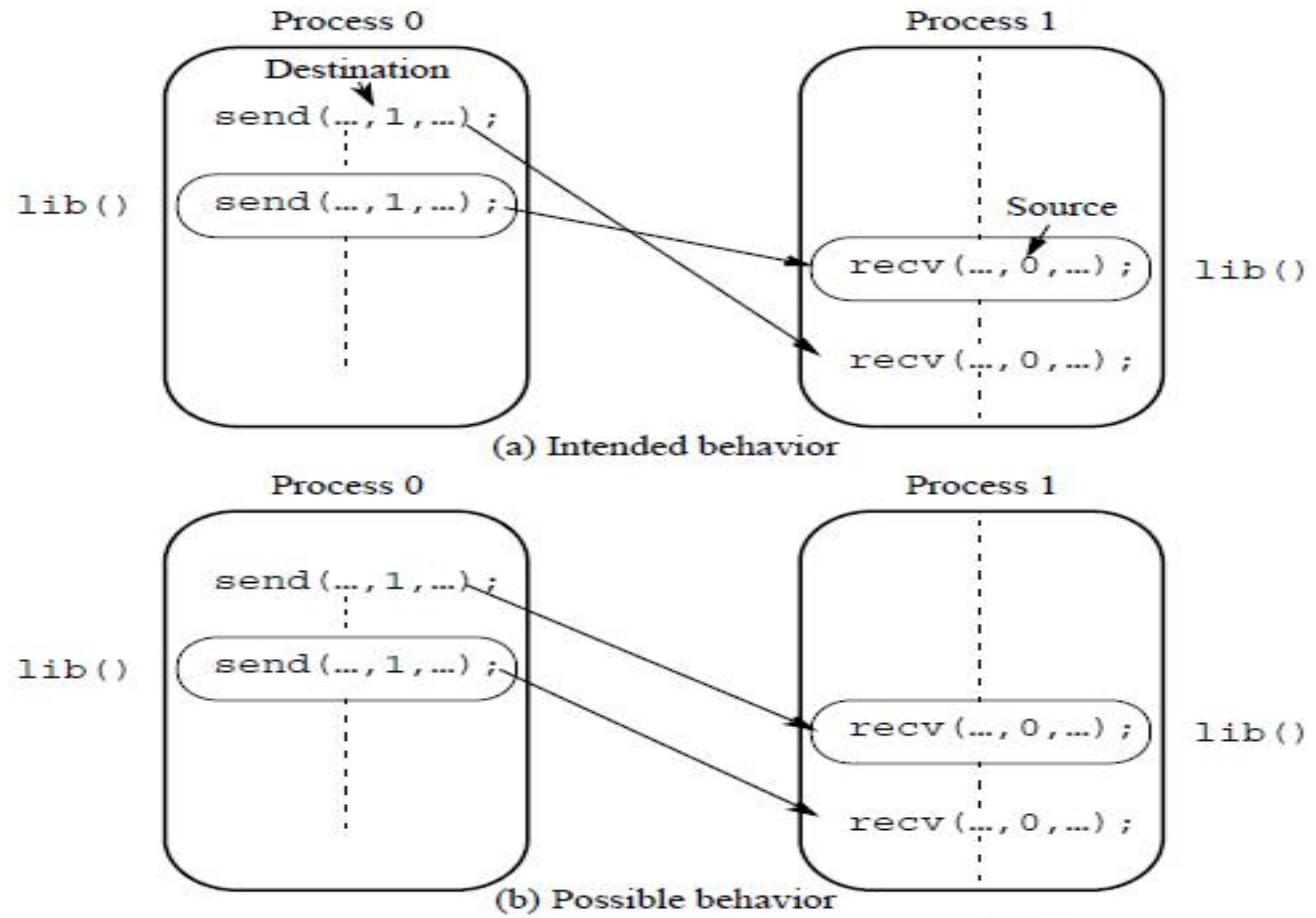
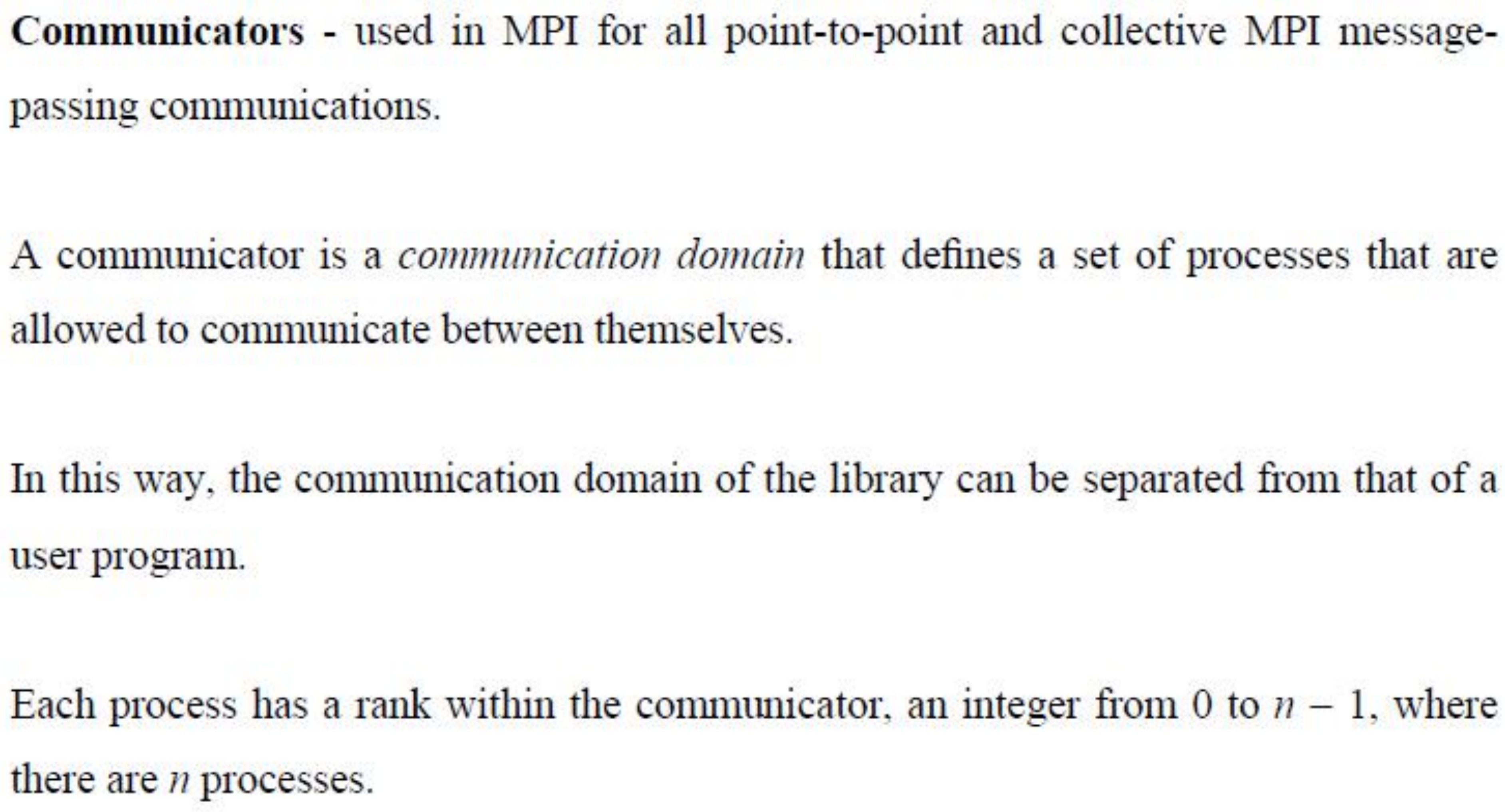


Figure 2.16 Unsafe message passing with libraries.

MPI Solution



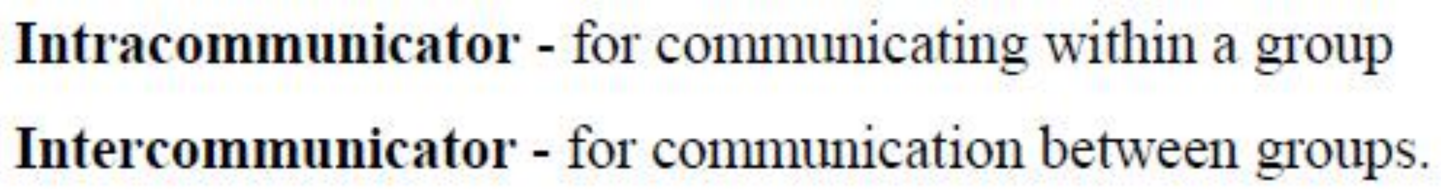
Communicators - used in MPI for all point-to-point and collective MPI message-passing communications.

A communicator is a *communication domain* that defines a set of processes that are allowed to communicate between themselves.

In this way, the communication domain of the library can be separated from that of a user program.

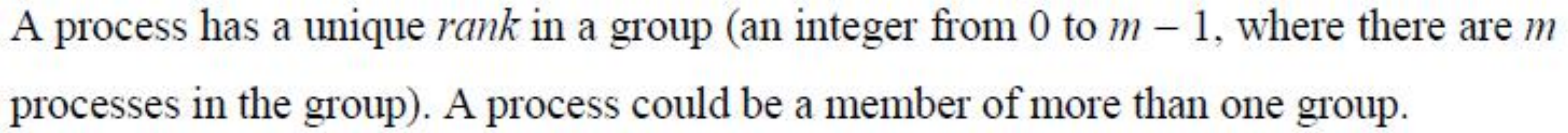
Each process has a rank within the communicator, an integer from 0 to $n - 1$, where there are n processes.

Communication Types

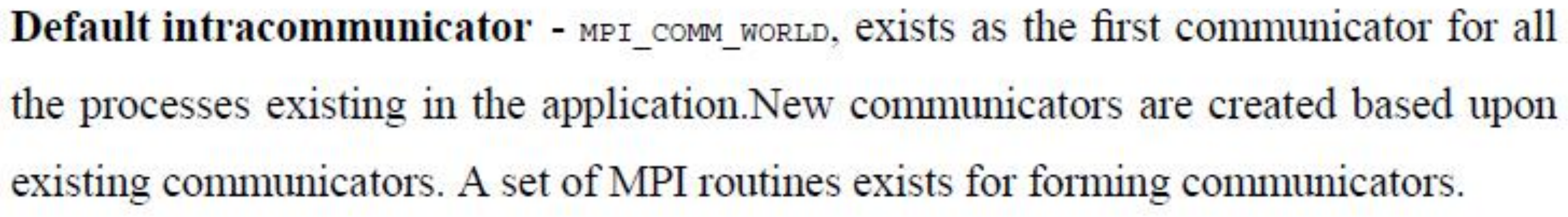


Intracommunicator - for communicating within a group

Intercommunicator - for communication between groups.

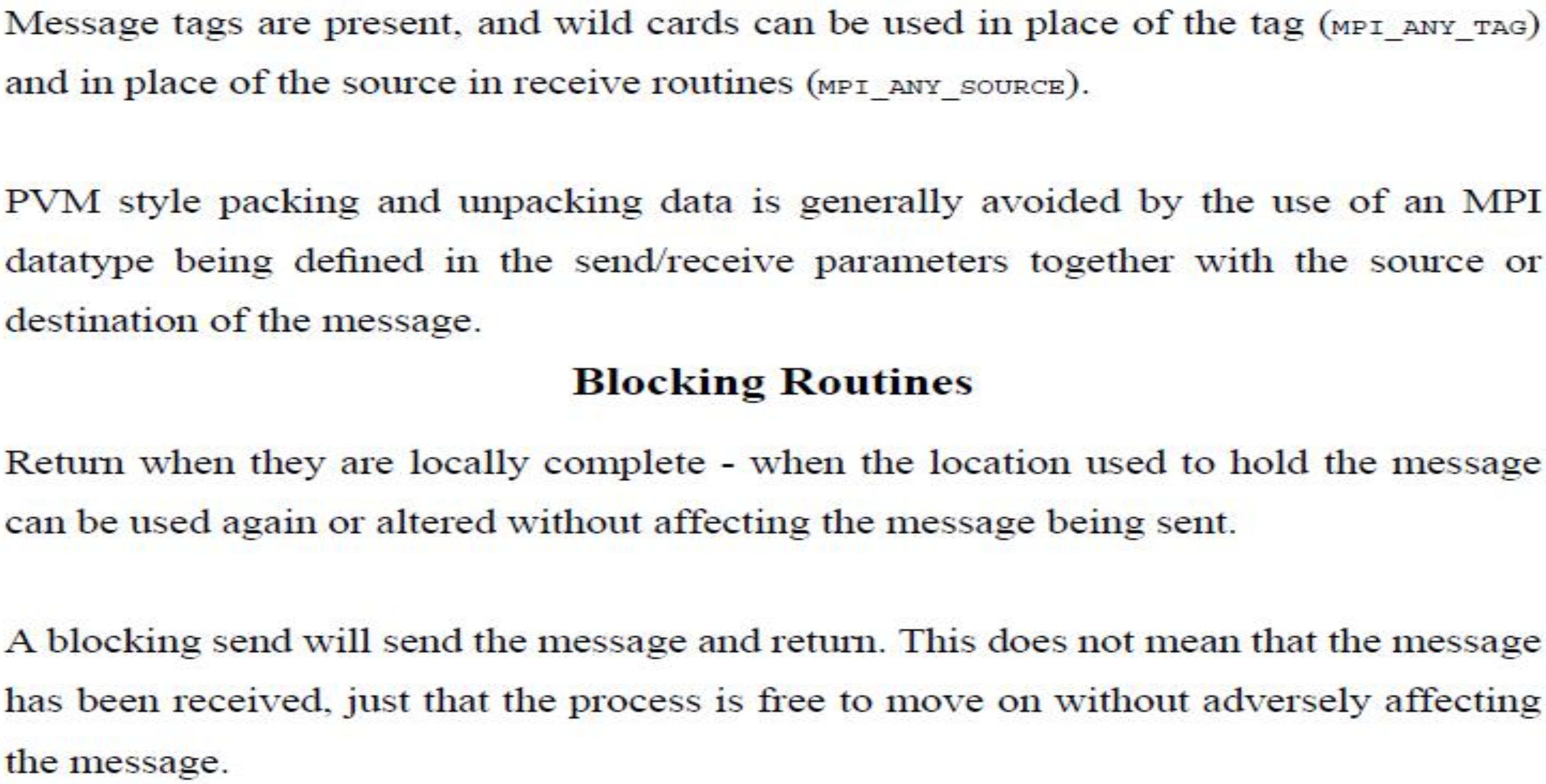


A process has a unique *rank* in a group (an integer from 0 to $m - 1$, where there are m processes in the group). A process could be a member of more than one group.



Default intracommunicator - `MPI_COMM_WORLD`, exists as the first communicator for all the processes existing in the application.New communicators are created based upon existing communicators. A set of MPI routines exists for forming communicators.

Point-to-Point Communication



Message tags are present, and wild cards can be used in place of the tag (MPI_ANY_TAG) and in place of the source in receive routines (MPI_ANY_SOURCE).

PVM style packing and unpacking data is generally avoided by the use of an MPI datatype being defined in the send/receive parameters together with the source or destination of the message.

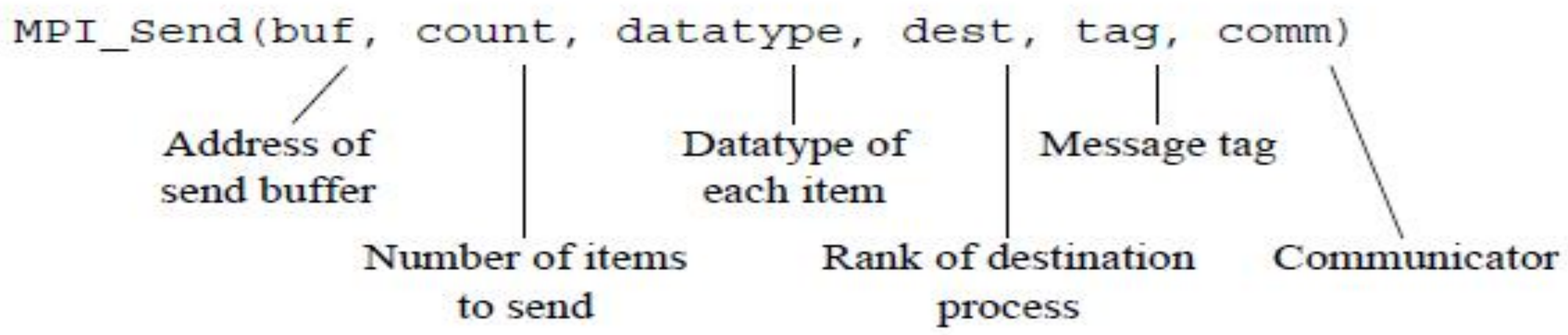
Blocking Routines

Return when they are locally complete - when the location used to hold the message can be used again or altered without affecting the message being sent.

A blocking send will send the message and return. This does not mean that the message has been received, just that the process is free to move on without adversely affecting the message.

MPI Blocking Send and Receive

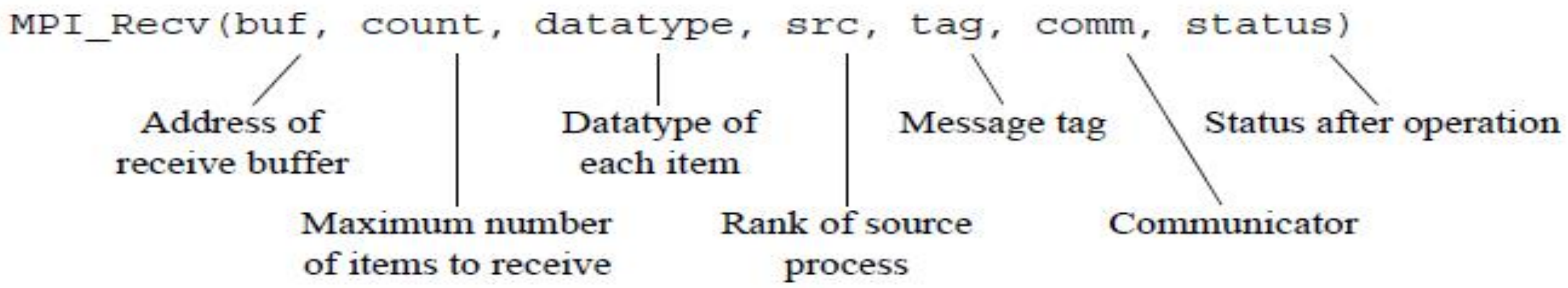
The general format of parameters of the blocking send is



```
MPI_Send(buf, count, datatype, dest, tag, comm)
```

- buf: Address of send buffer
- count: Number of items to send
- datatype: Datatype of each item
- dest: Rank of destination process
- tag: Message tag
- comm: Communicator

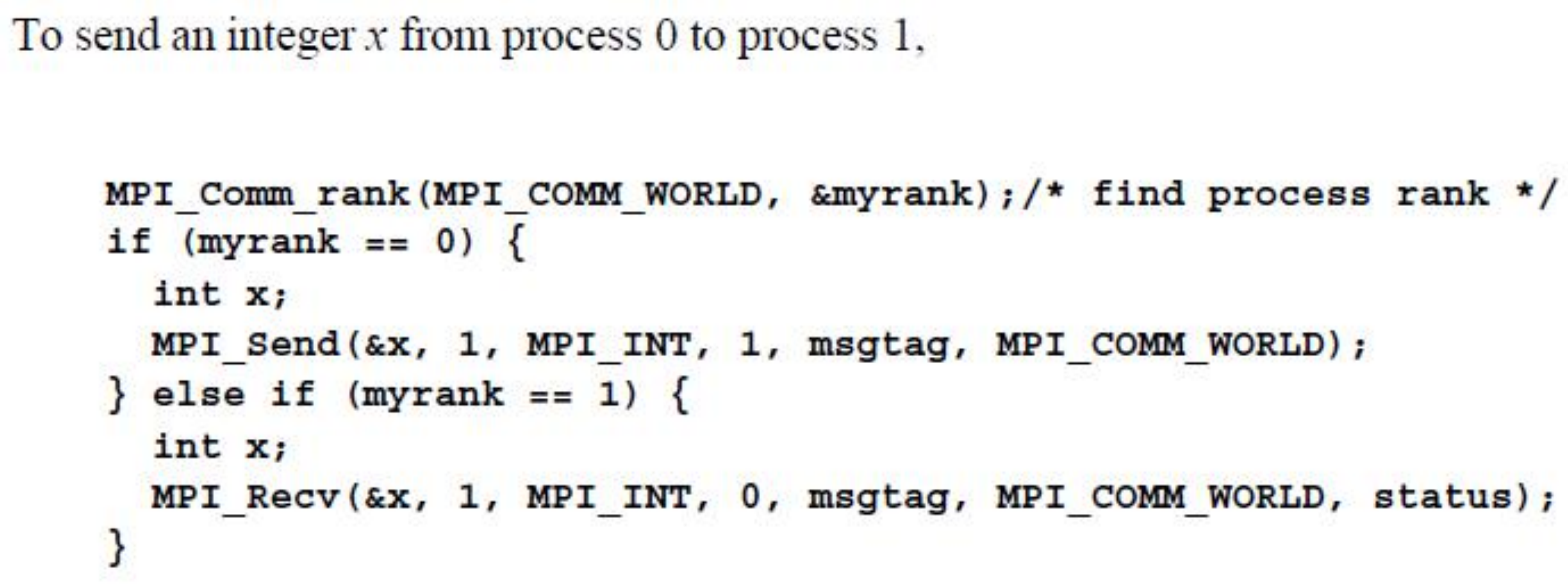
The general format of parameters of the blocking receive is



```
MPI_Recv(buf, count, datatype, src, tag, comm, status)
```

- buf: Address of receive buffer
- count: Maximum number of items to receive
- datatype: Datatype of each item
- src: Rank of source process
- tag: Message tag
- comm: Communicator
- status: Status after operation

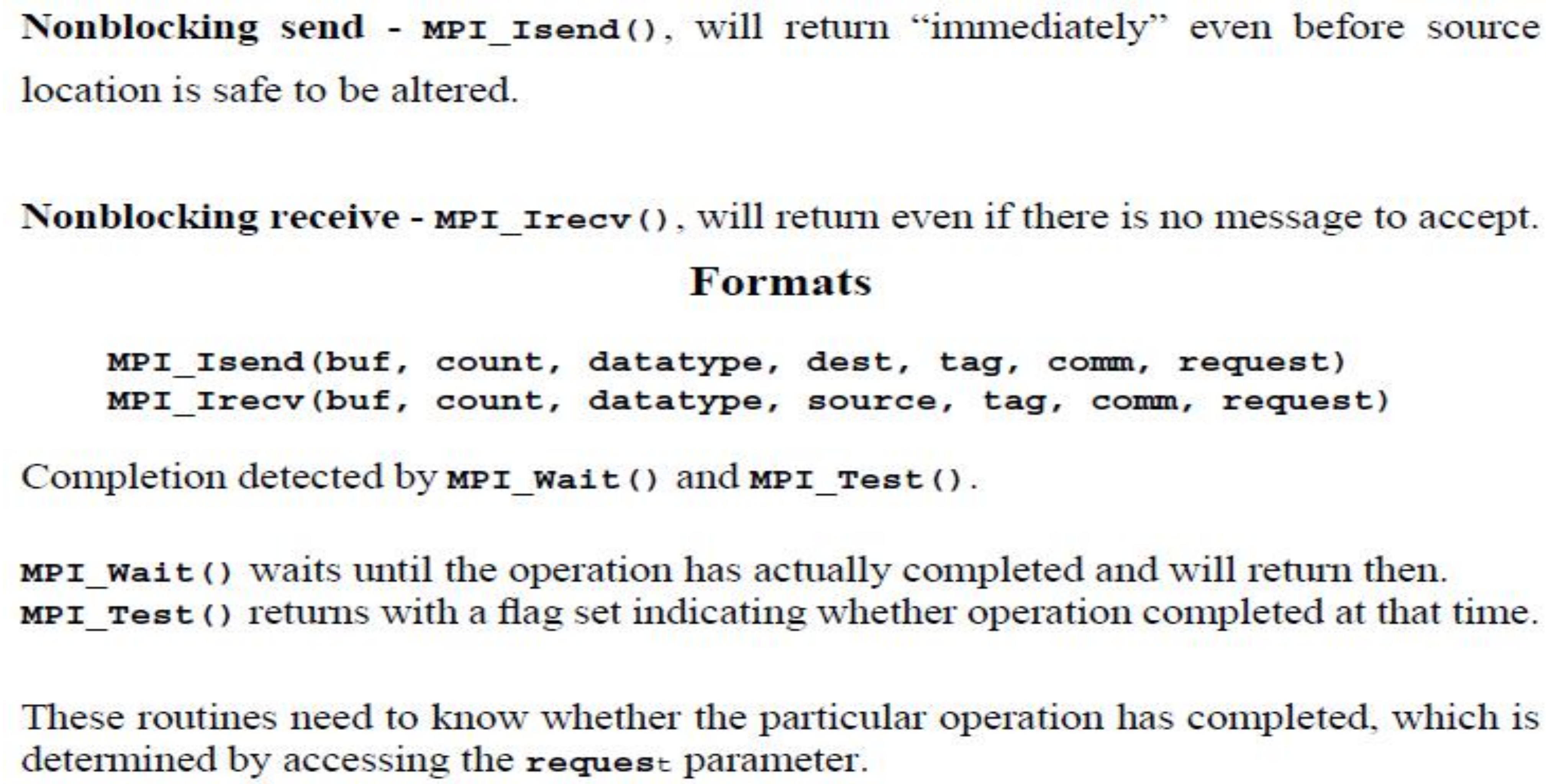
Example



To send an integer *x* from process 0 to process 1,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find process rank */
if (myrank == 0) {
  int x;
  MPI_Send(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD);
} else if (myrank == 1) {
  int x;
  MPI_Recv(&x, 1, MPI_INT, 0, msgtag, MPI_COMM_WORLD, status);
}
```

Nonblocking Routines



Nonblocking send - `MPI_Isend()`, will return "immediately" even before source location is safe to be altered.

Nonblocking receive - `MPI_Irecv()`, will return even if there is no message to accept.

Formats

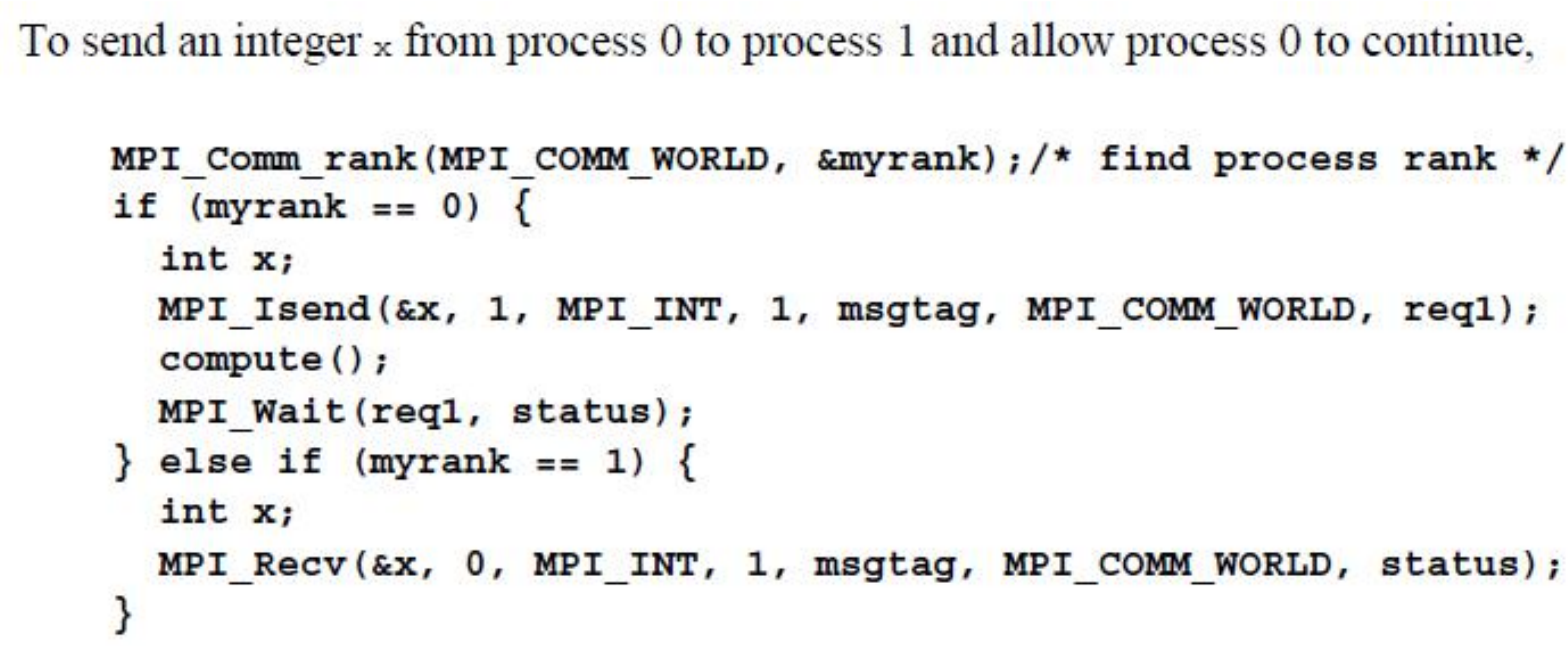
```
MPI_Isend(buf, count, datatype, dest, tag, comm, request)
MPI_Irecv(buf, count, datatype, source, tag, comm, request)
```

Completion detected by `MPI_Wait()` and `MPI_Test()`.

`MPI_Wait()` waits until the operation has actually completed and will return then.
`MPI_Test()` returns with a flag set indicating whether operation completed at that time.

These routines need to know whether the particular operation has completed, which is determined by accessing the `request` parameter.

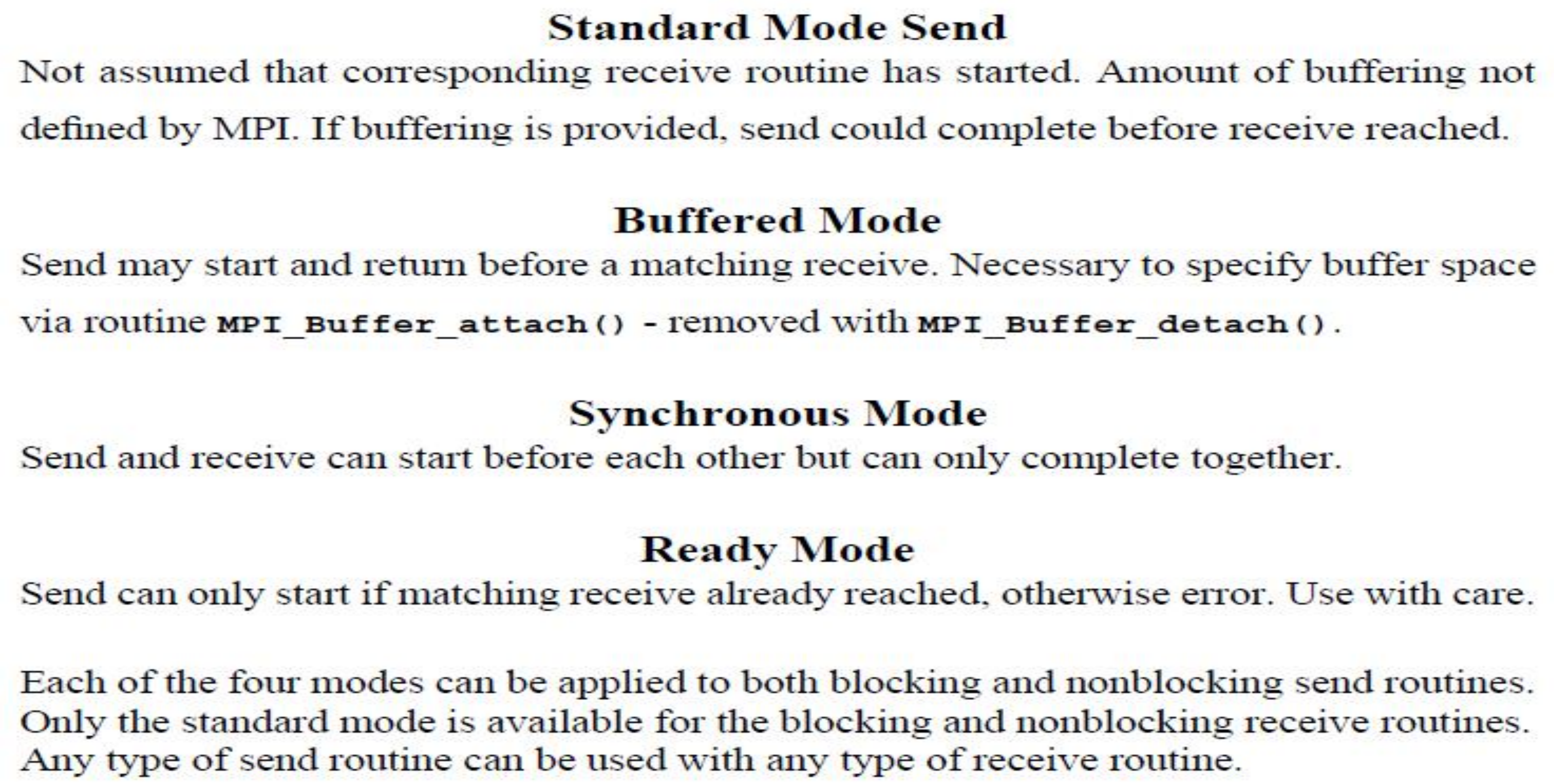
Example



To send an integer x from process 0 to process 1 and allow process 0 to continue,

```
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);/* find process rank */
if (myrank == 0) {
   int x;
   MPI_Isend(&x, 1, MPI_INT, 1, msgtag, MPI_COMM_WORLD, req1);
   compute();
   MPI_Wait(req1, status);
} else if (myrank == 1) {
   int x;
   MPI_Recv(&x, 0, MPI_INT, 1, msgtag, MPI_COMM_WORLD, status);
}
```

Send Communication Modes



Standard Mode Send

Not assumed that corresponding receive routine has started. Amount of buffering not defined by MPI. If buffering is provided, send could complete before receive reached.

Buffered Mode

Send may start and return before a matching receive. Necessary to specify buffer space via routine `MPI_Buffer_attach()` - removed with `MPI_Buffer_detach()`.

Synchronous Mode

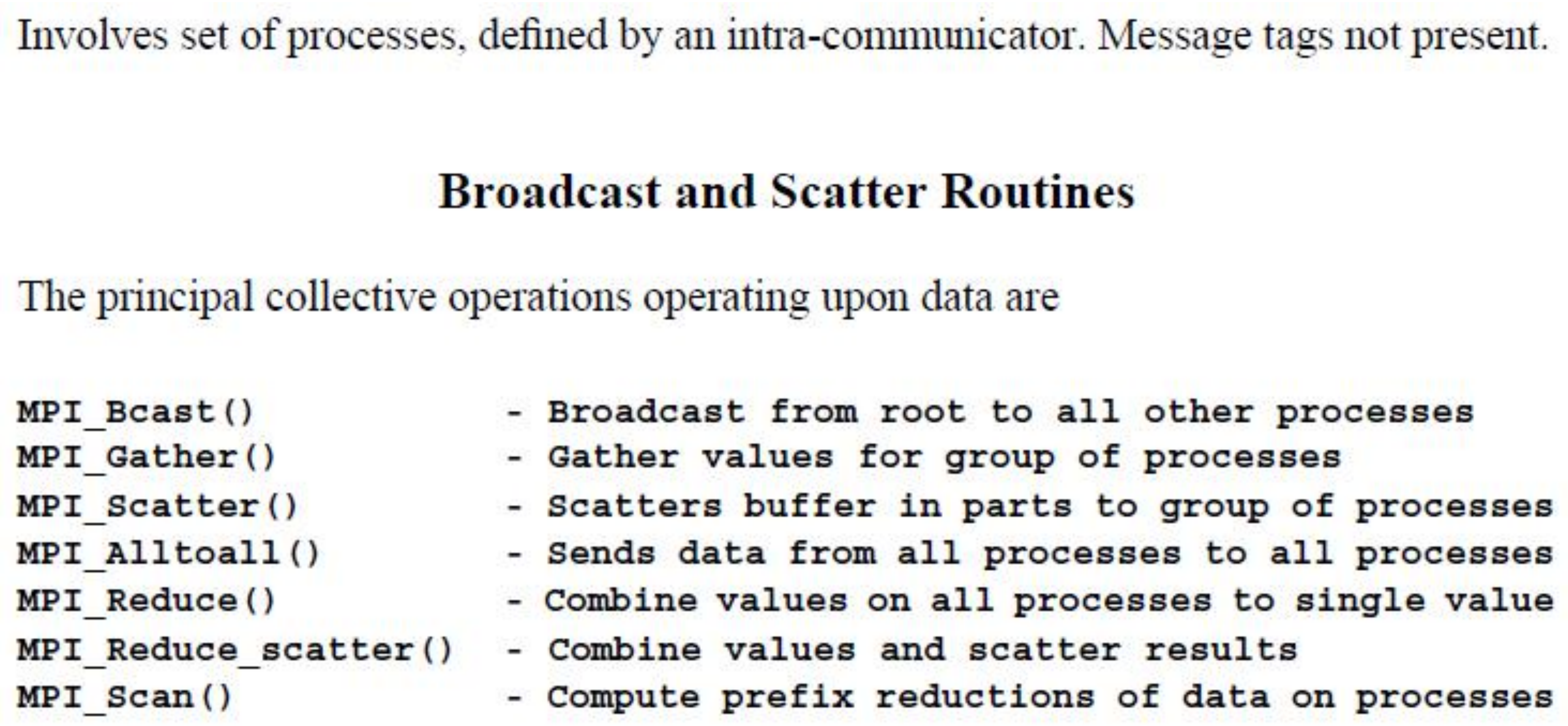
Send and receive can start before each other but can only complete together.

Ready Mode

Send can only start if matching receive already reached, otherwise error. Use with care.

Each of the four modes can be applied to both blocking and nonblocking send routines.
Only the standard mode is available for the blocking and nonblocking receive routines.
Any type of send routine can be used with any type of receive routine.

Collective Communication



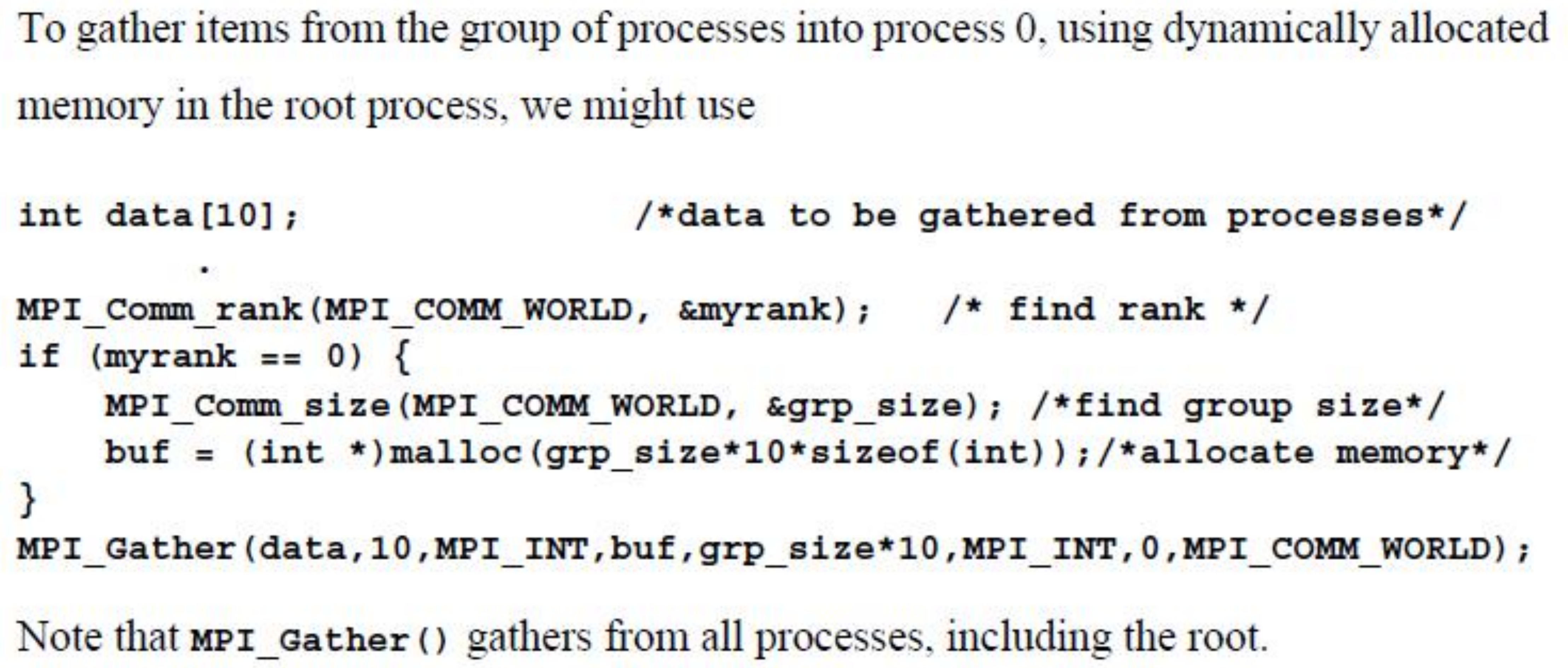
Involves set of processes, defined by an intra-communicator. Message tags not present.

Broadcast and Scatter Routines

The principal collective operations operating upon data are

`MPI_Bcast()`	- Broadcast from root to all other processes
`MPI_Gather()`	- Gather values for group of processes
`MPI_Scatter()`	- Scatters buffer in parts to group of processes
`MPI_Alltoall()`	- Sends data from all processes to all processes
`MPI_Reduce()`	- Combine values on all processes to single value
`MPI_Reduce_scatter()`	- Combine values and scatter results
`MPI_Scan()`	- Compute prefix reductions of data on processes

Example

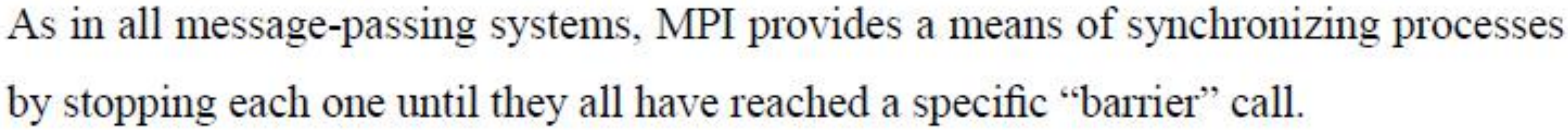


To gather items from the group of processes into process 0, using dynamically allocated memory in the root process, we might use

```
int data[10];                /*data to be gathered from processes*/
           .
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);   /* find rank */
if (myrank == 0) {
    MPI_Comm_size(MPI_COMM_WORLD, &grp_size); /*find group size*/
    buf = (int *)malloc(grp_size*10*sizeof(int));/*allocate memory*/
}
MPI_Gather(data,10,MPI_INT,buf,grp_size*10,MPI_INT,0,MPI_COMM_WORLD);
```

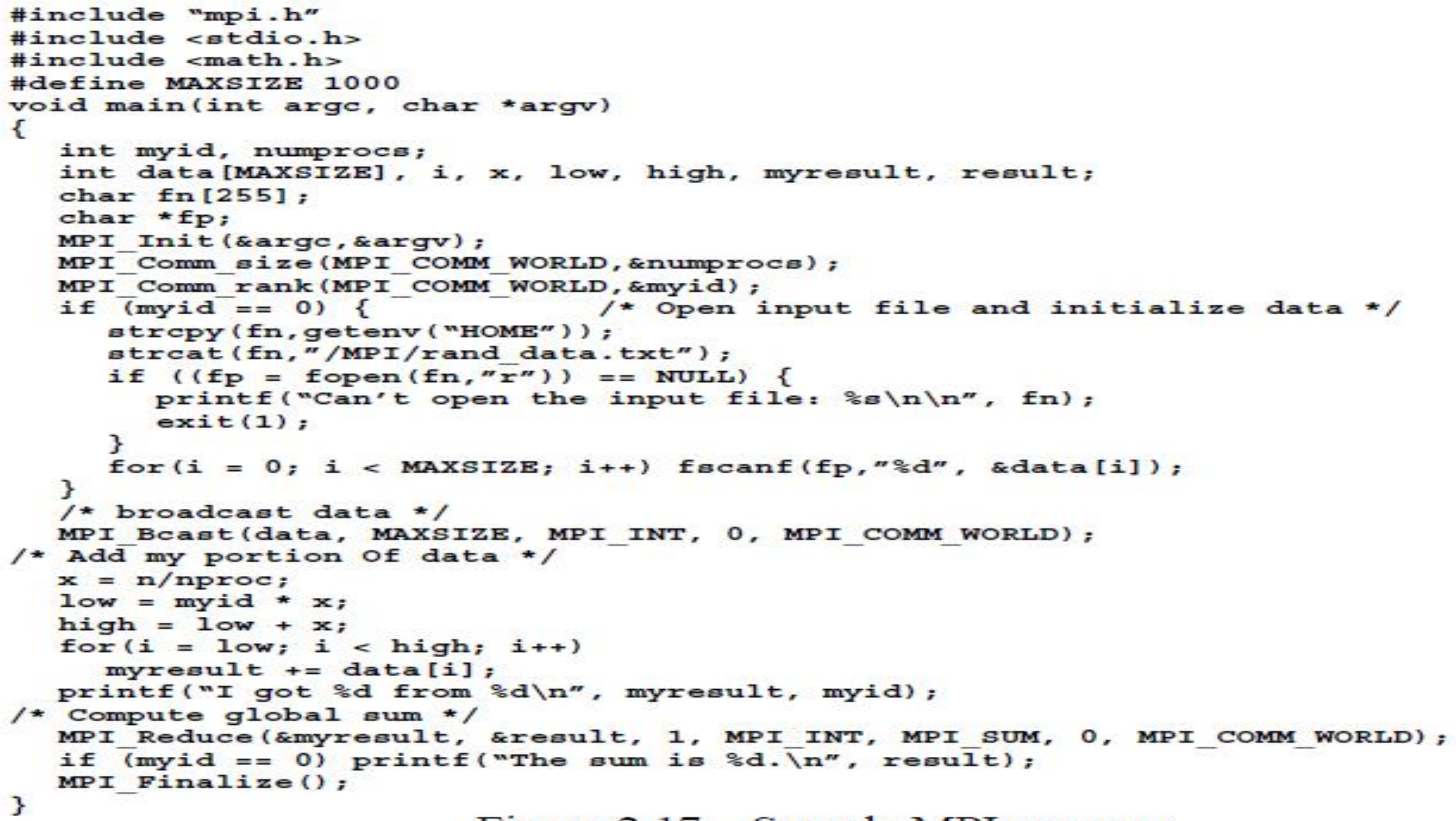
Note that `MPI_Gather()` gathers from all processes, including the root.

Barrier



As in all message-passing systems, MPI provides a means of synchronizing processes by stopping each one until they all have reached a specific “barrier” call.

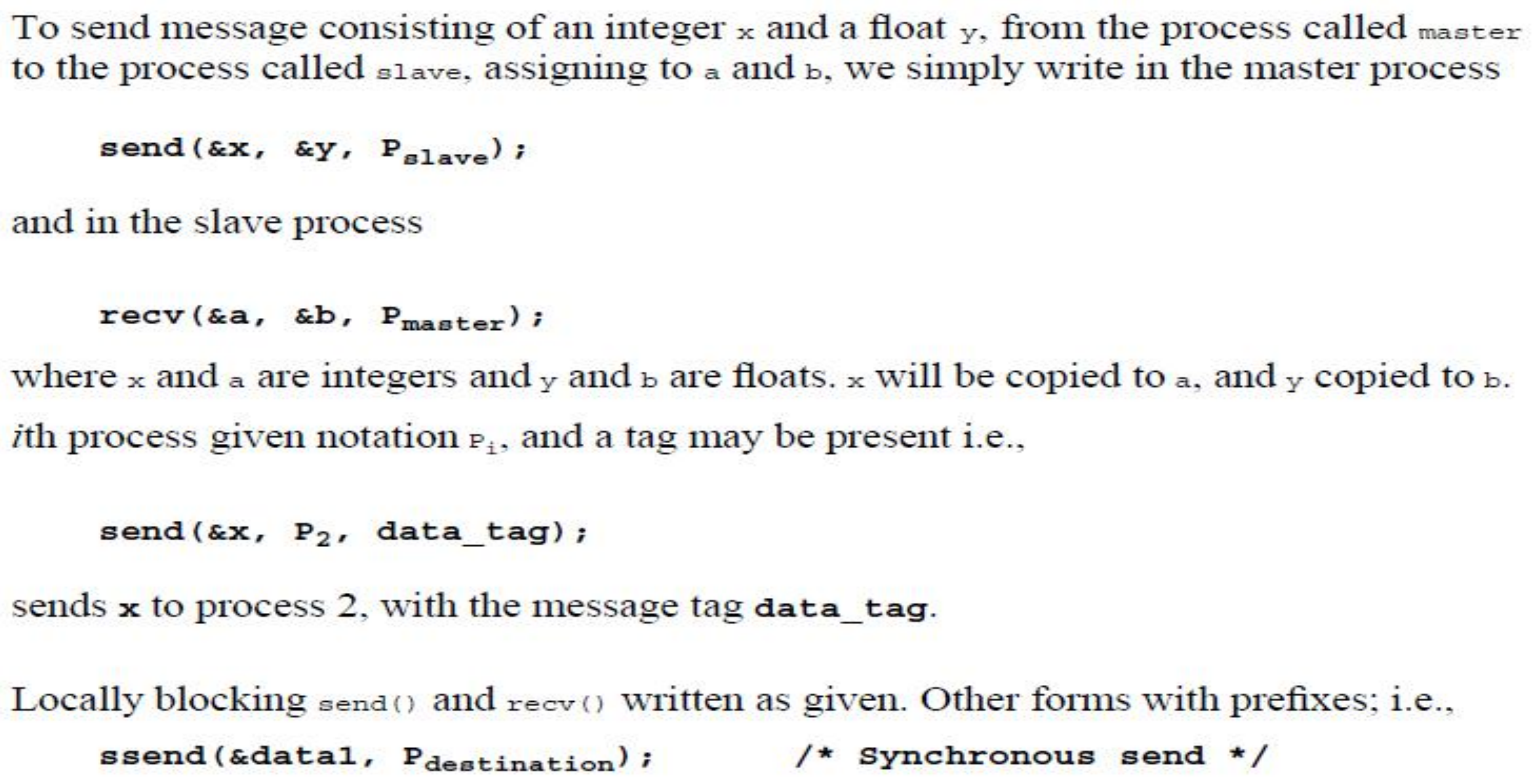
Example



```
#include “mpi.h”
#include <stdio.h>
#include <math.h>
#define MAXSIZE 1000
void main(int argc, char *argv)
{
   int myid, numprocs;
   int data[MAXSIZE], i, x, low, high, myresult, result;
   char fn[255];
   char *fp;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   if (myid == 0) {              /* Open input file and initialize data */
      strcpy(fn,getenv(“HOME”));
      strcat(fn,”/MPI/rand_data.txt”);
      if ((fp = fopen(fn,”r”)) == NULL) {
         printf(“Can’t open the input file: %s\n\n”, fn);
         exit(1);
      }
      for(i = 0; i < MAXSIZE; i++) fscanf(fp,”%d”, &data[i]);
   }
   /* broadcast data */
   MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);
/* Add my portion Of data */
   x = n/nproc;
   low = myid * x;
   high = low + x;
   for(i = low; i < high; i++)
      myresult += data[i];
   printf(“I got %d from %d\n”, myresult, myid);
/* Compute global sum */
   MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
   if (myid == 0) printf(“The sum is %d.\n”, result);
   MPI_Finalize();
}
```

Figure 2.17 Sample MPI program.

Pseudocode Constructs



To send message consisting of an integer x and a float y, from the process called master to the process called slave, assigning to a and b, we simply write in the master process

```
send(&x, &y, P_slave);
```

and in the slave process

```
recv(&a, &b, P_master);
```

where x and a are integers and y and b are floats. x will be copied to a, and y copied to b.

*i*th process given notation P_i, and a tag may be present i.e.,

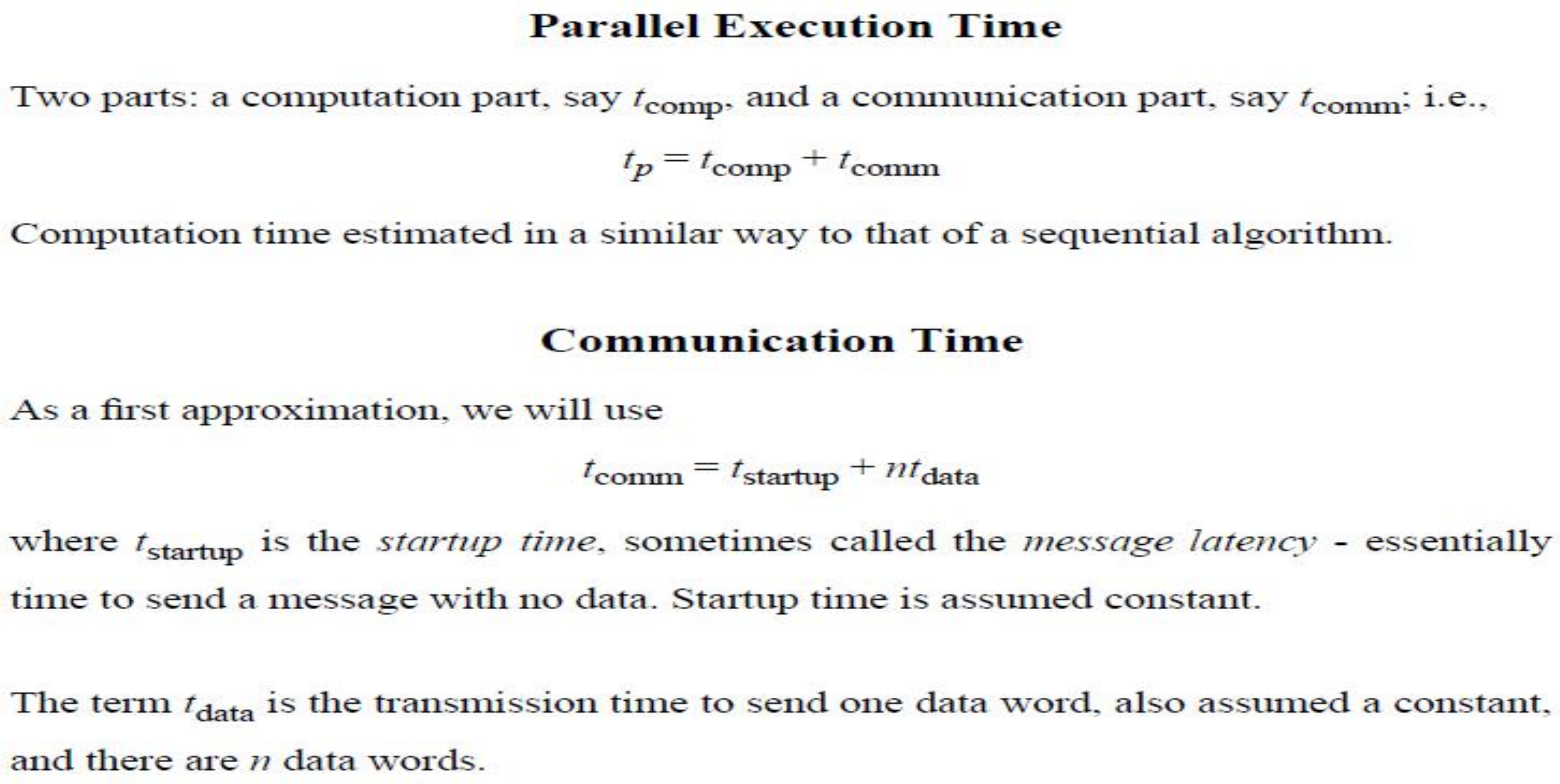
```
send(&x, P_2, data_tag);
```

sends **x** to process 2, with the message tag **data_tag**.

Locally blocking send() and recv() written as given. Other forms with prefixes; i.e.,

```
ssend(&data1, P_destination);        /* Synchronous send */
```

Time Complexity of MPI Program



Parallel Execution Time

Two parts: a computation part, say t_{comp}, and a communication part, say t_{comm}; i.e.,

$$t_p = t_{\text{comp}} + t_{\text{comm}}$$

Computation time estimated in a similar way to that of a sequential algorithm.

Communication Time

As a first approximation, we will use

$$t_{\text{comm}} = t_{\text{startup}} + n t_{\text{data}}$$

where t_{startup} is the *startup time*, sometimes called the *message latency* - essentially time to send a message with no data. Startup time is assumed constant.

The term t_{data} is the transmission time to send one data word, also assumed a constant, and there are n data words.

Communication Time

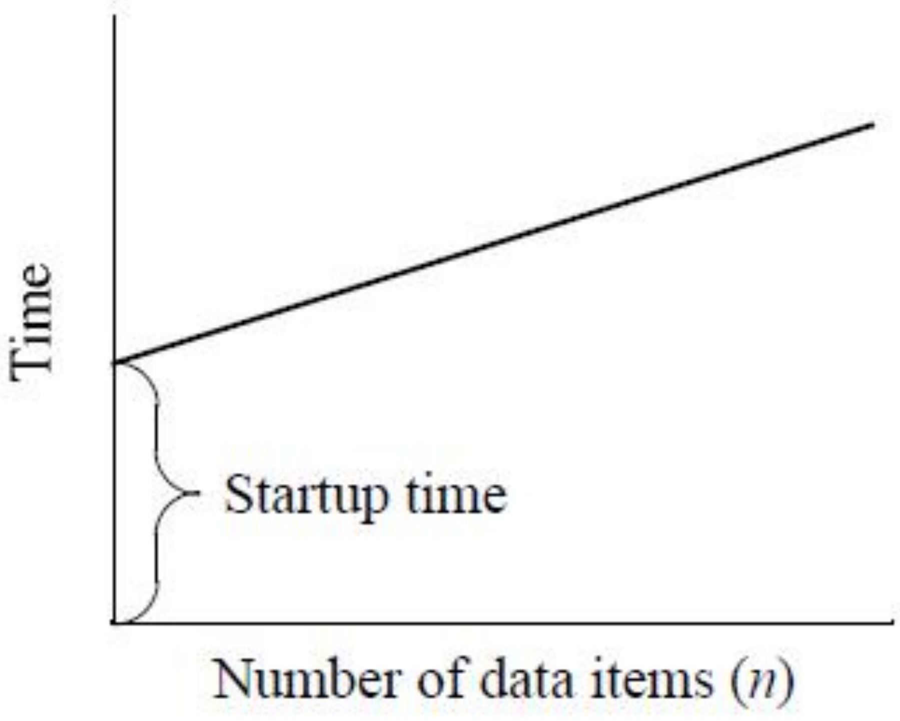
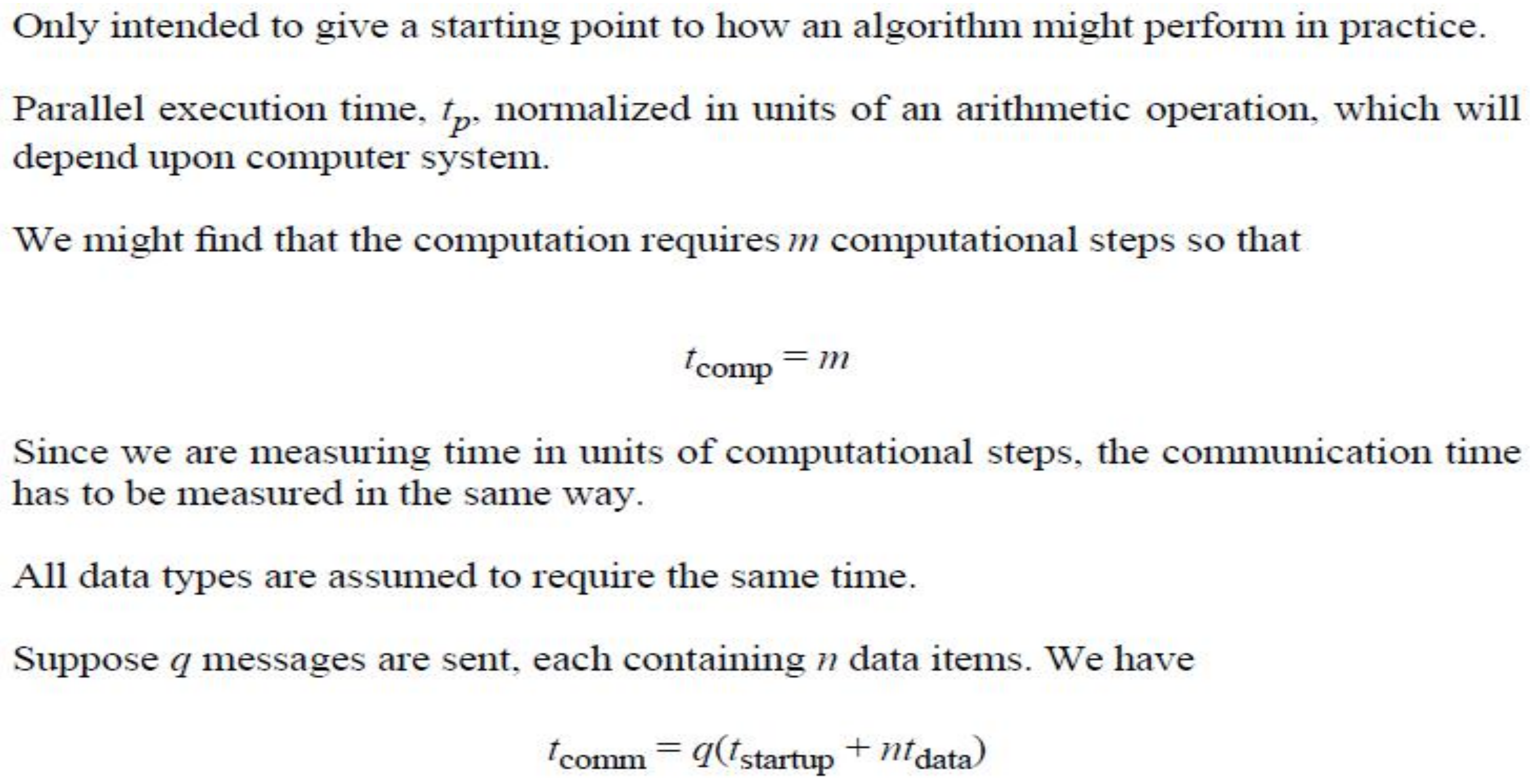


Figure 2.18 Theoretical communication time.

Important Note on Interpretation of Equations



Only intended to give a starting point to how an algorithm might perform in practice.

Parallel execution time, t_p, normalized in units of an arithmetic operation, which will depend upon computer system.

We might find that the computation requires m computational steps so that

$$t_{\mathrm{comp}} = m$$

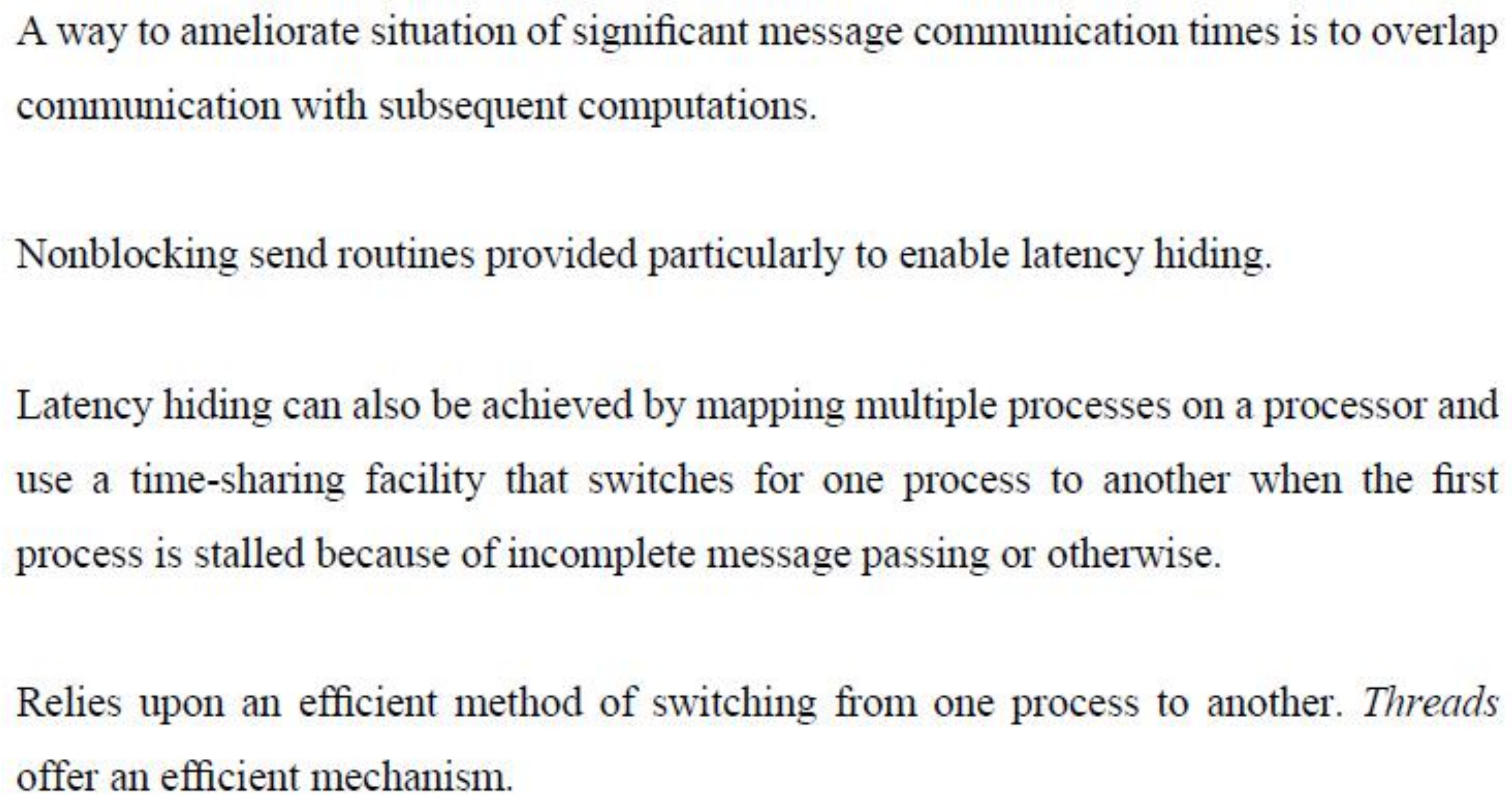
Since we are measuring time in units of computational steps, the communication time has to be measured in the same way.

All data types are assumed to require the same time.

Suppose q messages are sent, each containing n data items. We have

$$t_{\mathrm{comm}} = q(t_{\mathrm{startup}} + nt_{\mathrm{data}})$$

Latency Hidden



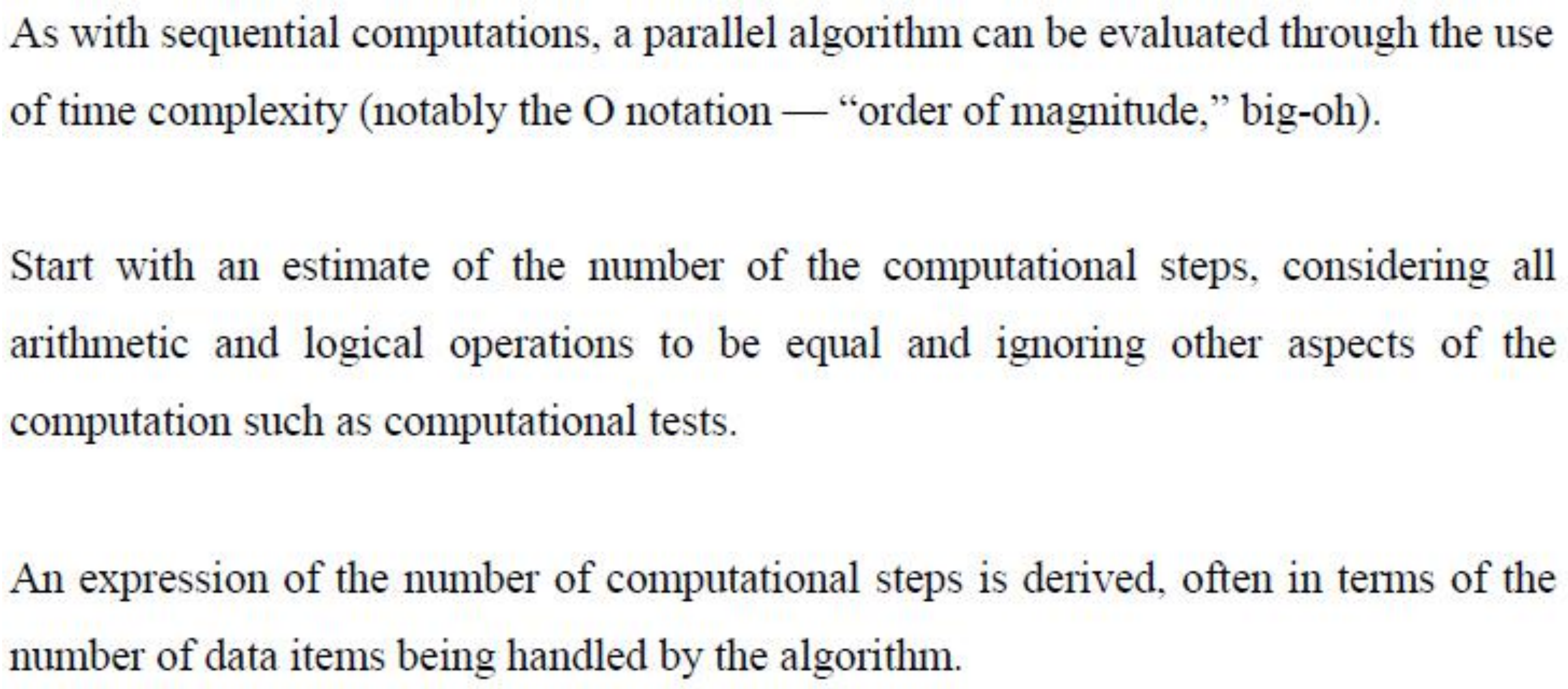
A way to ameliorate situation of significant message communication times is to overlap communication with subsequent computations.

Nonblocking send routines provided particularly to enable latency hiding.

Latency hiding can also be achieved by mapping multiple processes on a processor and use a time-sharing facility that switches for one process to another when the first process is stalled because of incomplete message passing or otherwise.

Relies upon an efficient method of switching from one process to another. *Threads* offer an efficient mechanism.

Time Complexity (1)

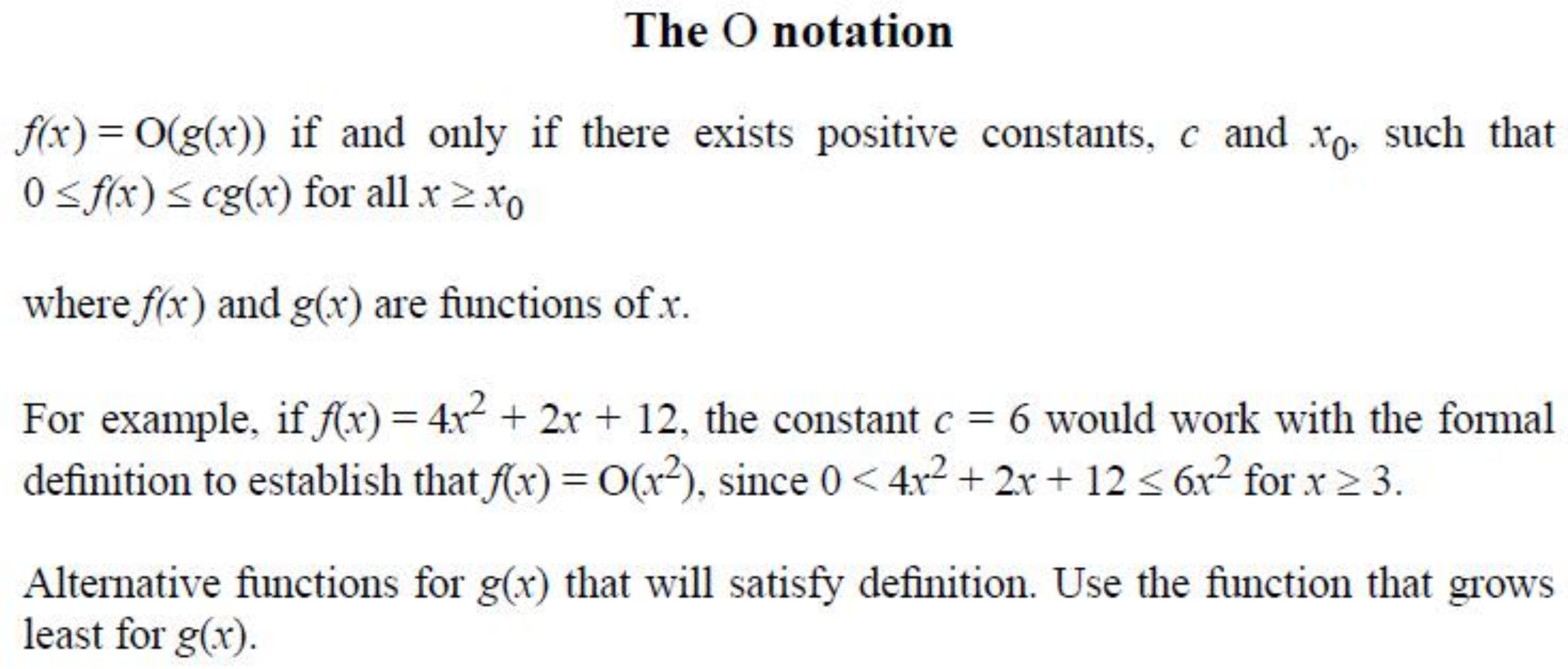


As with sequential computations, a parallel algorithm can be evaluated through the use of time complexity (notably the O notation — “order of magnitude,” big-oh).

Start with an estimate of the number of the computational steps, considering all arithmetic and logical operations to be equal and ignoring other aspects of the computation such as computational tests.

An expression of the number of computational steps is derived, often in terms of the number of data items being handled by the algorithm.

Time Complexity (2)



The O notation

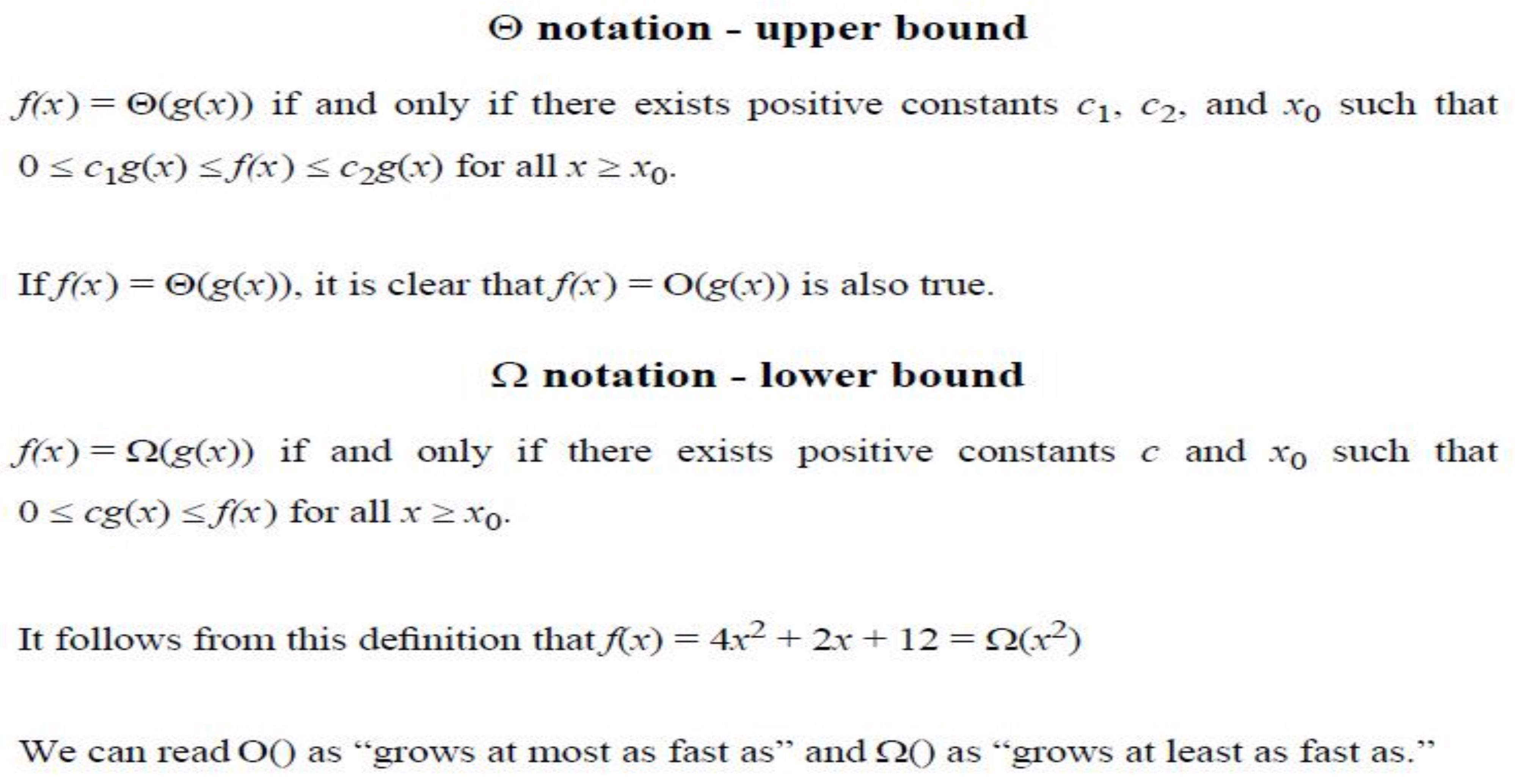
$f(x) = \mathrm{O}(g(x))$ if and only if there exists positive constants, c and x_0, such that $0 \leq f(x) \leq cg(x)$ for all $x \geq x_0$

where $f(x)$ and $g(x)$ are functions of x.

For example, if $f(x) = 4x^2 + 2x + 12$, the constant $c = 6$ would work with the formal definition to establish that $f(x) = \mathrm{O}(x^2)$, since $0 < 4x^2 + 2x + 12 \leq 6x^2$ for $x \geq 3$.

Alternative functions for $g(x)$ that will satisfy definition. Use the function that grows least for $g(x)$.

Time Complexity (3)



Θ notation - upper bound

$f(x) = \Theta(g(x))$ if and only if there exists positive constants c_1, c_2, and x_0 such that $0 \le c_1 g(x) \le f(x) \le c_2 g(x)$ for all $x \ge x_0$.

If $f(x) = \Theta(g(x))$, it is clear that $f(x) = O(g(x))$ is also true.

Ω notation - lower bound

$f(x) = \Omega(g(x))$ if and only if there exists positive constants c and x_0 such that $0 \le cg(x) \le f(x)$ for all $x \ge x_0$.

It follows from this definition that $f(x) = 4x^2 + 2x + 12 = \Omega(x^2)$

We can read O() as "grows at most as fast as" and Ω() as "grows at least as fast as."

Example

The execution time of a sorting algorithm often depends upon the original order of the numbers to be sorted. It may be that it requires at least $n \log n$ steps, but could require n^2 steps for n numbers depending upon the order of the numbers. This would be indicated by a time complexity of $\Omega(n \log n)$ and $O(n^2)$.

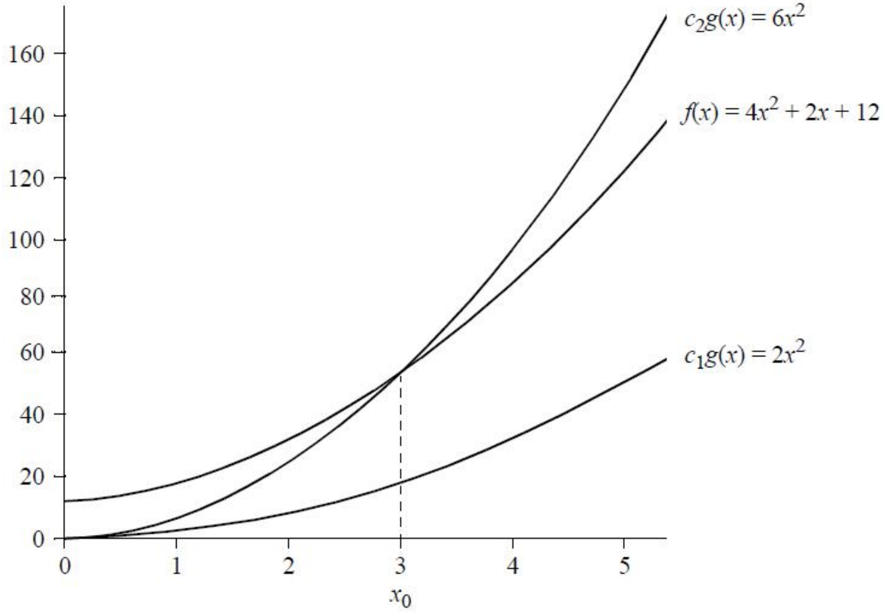
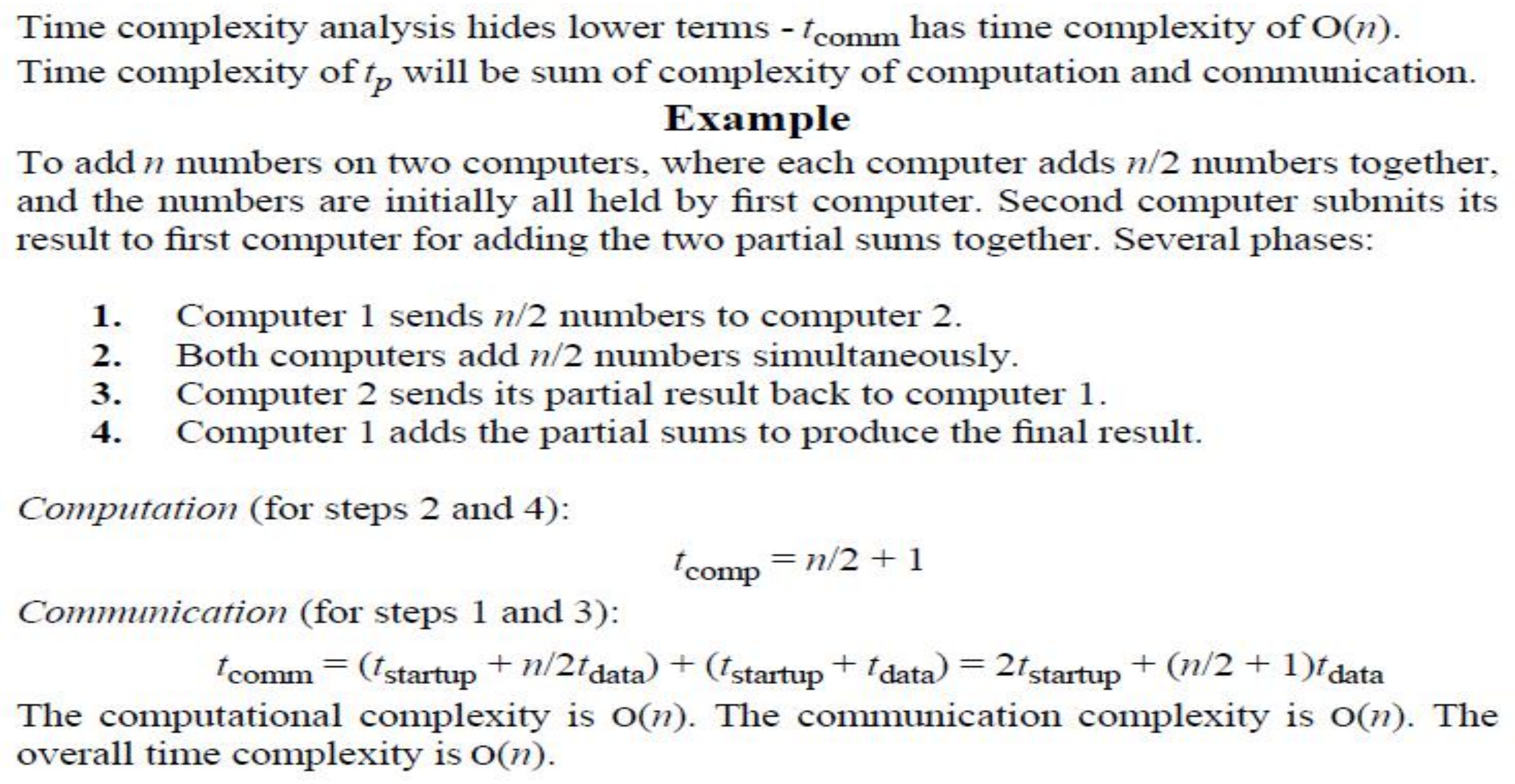


Figure 2.19 Growth of function $f(x) = 4x^2 + 2x + 12$.

Time Complexity of a Parallel Algorithm



Time complexity analysis hides lower terms - t_{comm} has time complexity of O(n).

Time complexity of t_p will be sum of complexity of computation and communication.

Example

To add n numbers on two computers, where each computer adds $n/2$ numbers together, and the numbers are initially all held by first computer. Second computer submits its result to first computer for adding the two partial sums together. Several phases:

1. Computer 1 sends $n/2$ numbers to computer 2.
2. Both computers add $n/2$ numbers simultaneously.
3. Computer 2 sends its partial result back to computer 1.
4. Computer 1 adds the partial sums to produce the final result.

Computation (for steps 2 and 4):

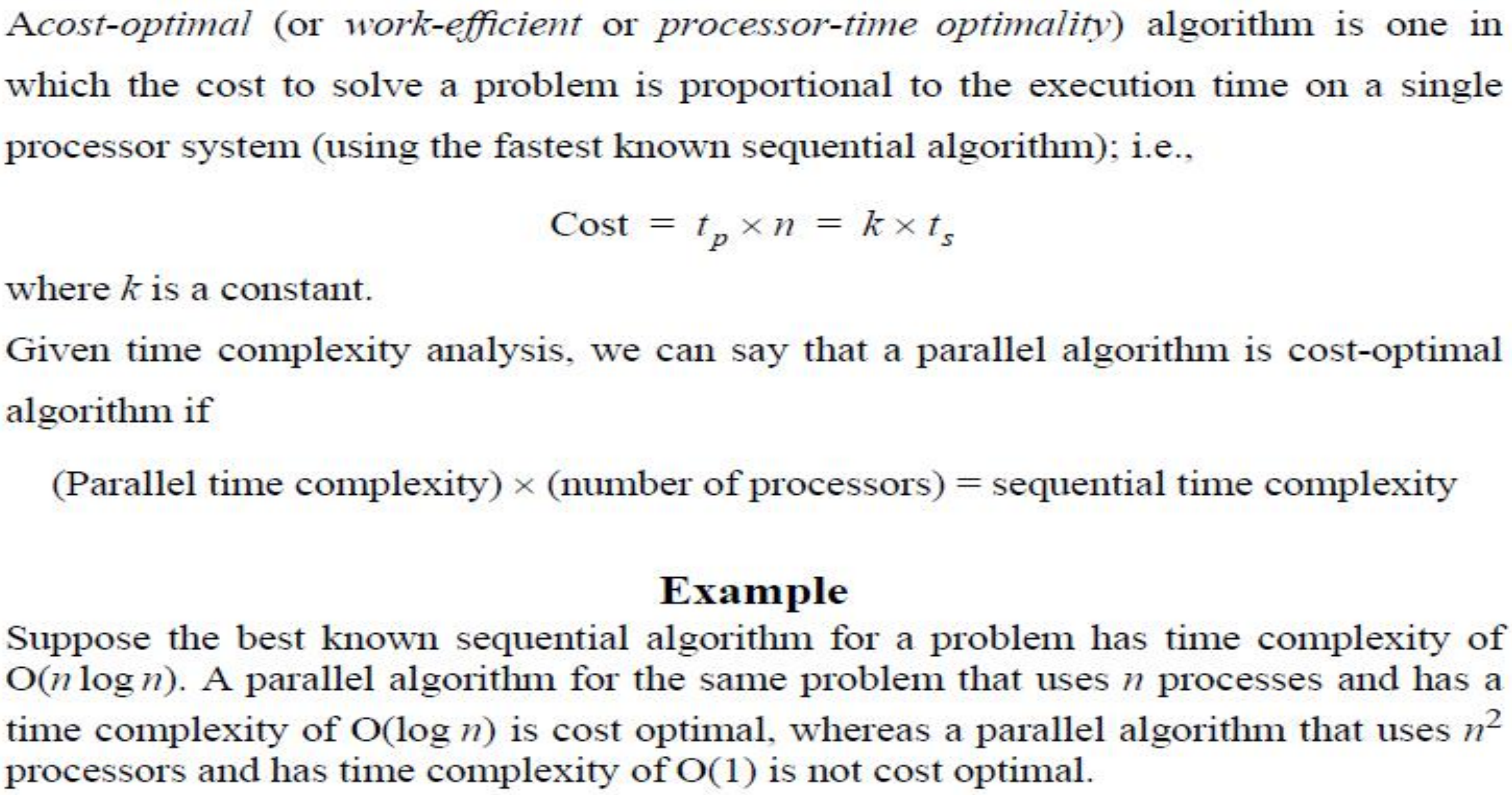
$$t_{\text{comp}} = n/2 + 1$$

Communication (for steps 1 and 3):

$$t_{\text{comm}} = (t_{\text{startup}} + n/2t_{\text{data}}) + (t_{\text{startup}} + t_{\text{data}}) = 2t_{\text{startup}} + (n/2 + 1)t_{\text{data}}$$

The computational complexity is O(n). The communication complexity is O(n). The overall time complexity is O(n).

Cost Optimal Algorithms



A*cost-optimal* (or *work-efficient* or *processor-time optimality*) algorithm is one in which the cost to solve a problem is proportional to the execution time on a single processor system (using the fastest known sequential algorithm); i.e.,

$$\text{Cost} = t_p \times n = k \times t_s$$

where k is a constant.

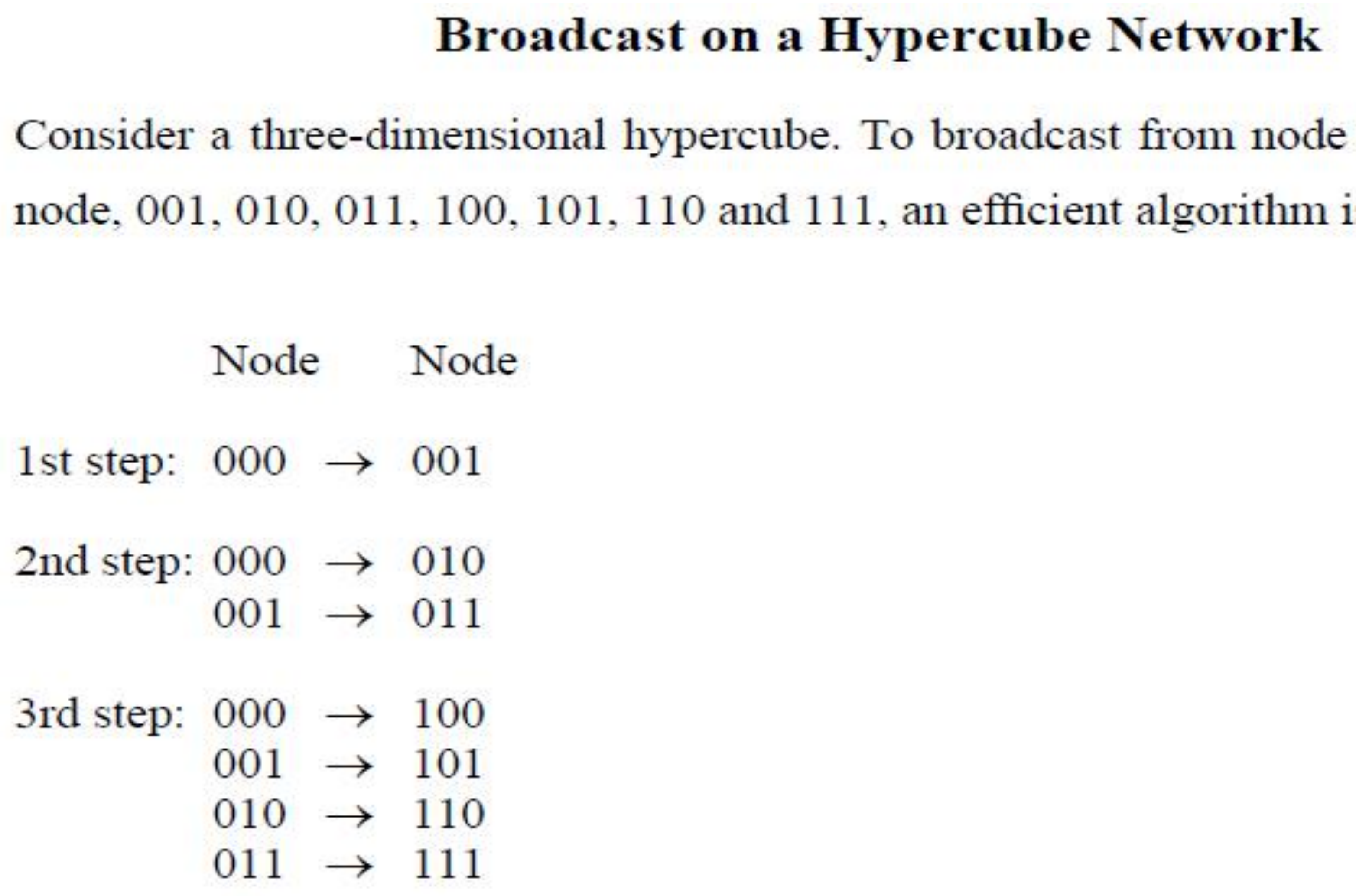
Given time complexity analysis, we can say that a parallel algorithm is cost-optimal algorithm if

(Parallel time complexity) × (number of processors) = sequential time complexity

Example

Suppose the best known sequential algorithm for a problem has time complexity of $O(n \log n)$. A parallel algorithm for the same problem that uses n processes and has a time complexity of $O(\log n)$ is cost optimal, whereas a parallel algorithm that uses n^2 processors and has time complexity of $O(1)$ is not cost optimal.

Time Complexity of Broadcast (1)



Broadcast on a Hypercube Network

Consider a three-dimensional hypercube. To broadcast from node 000 to every other node, 001, 010, 011, 100, 101, 110 and 111, an efficient algorithm is

	Node		Node
1st step:	000	→	001
2nd step:	000	→	010
	001	→	011
3rd step:	000	→	100
	001	→	101
	010	→	110
	011	→	111

Time Complexity of Broadcast (2)

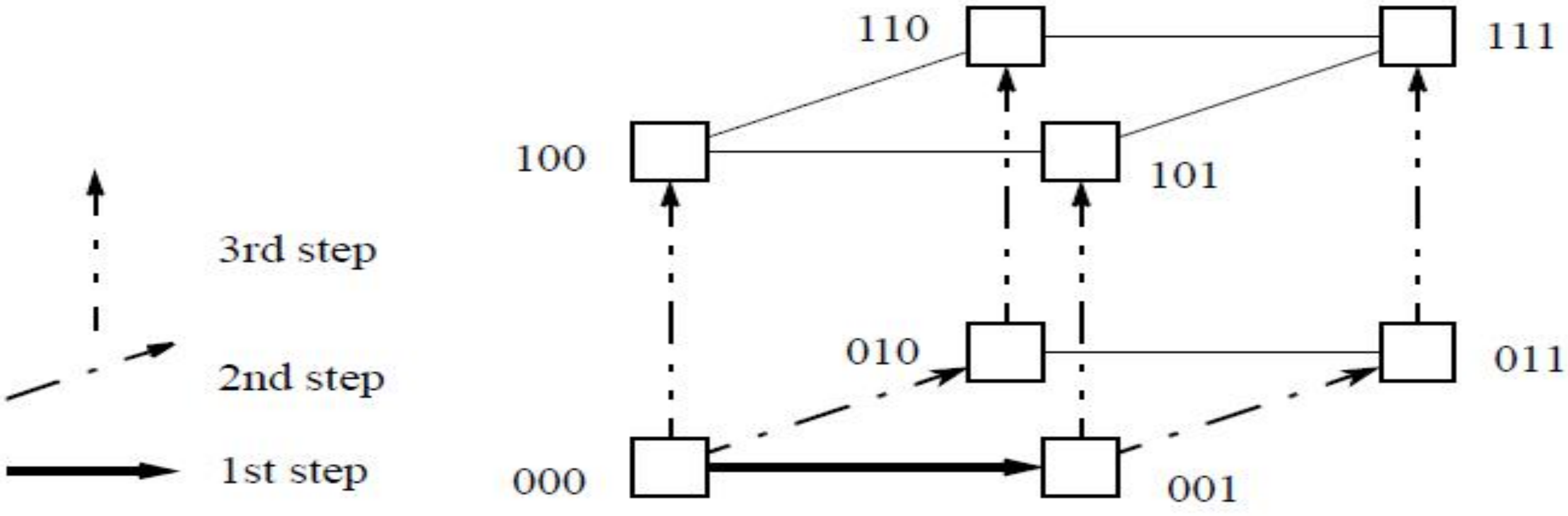


Figure 2.20 Broadcast in a three-dimensional hypercube.

The time complexity for a hypercube system will be $O(\log n)$, using this algorithm, which is optimal because the *diameter* of a hypercube network is $\log n$. It is necessary at least to use this number of links in the broadcast to reach the furthest node.

Time Complexity of Broadcast (3)

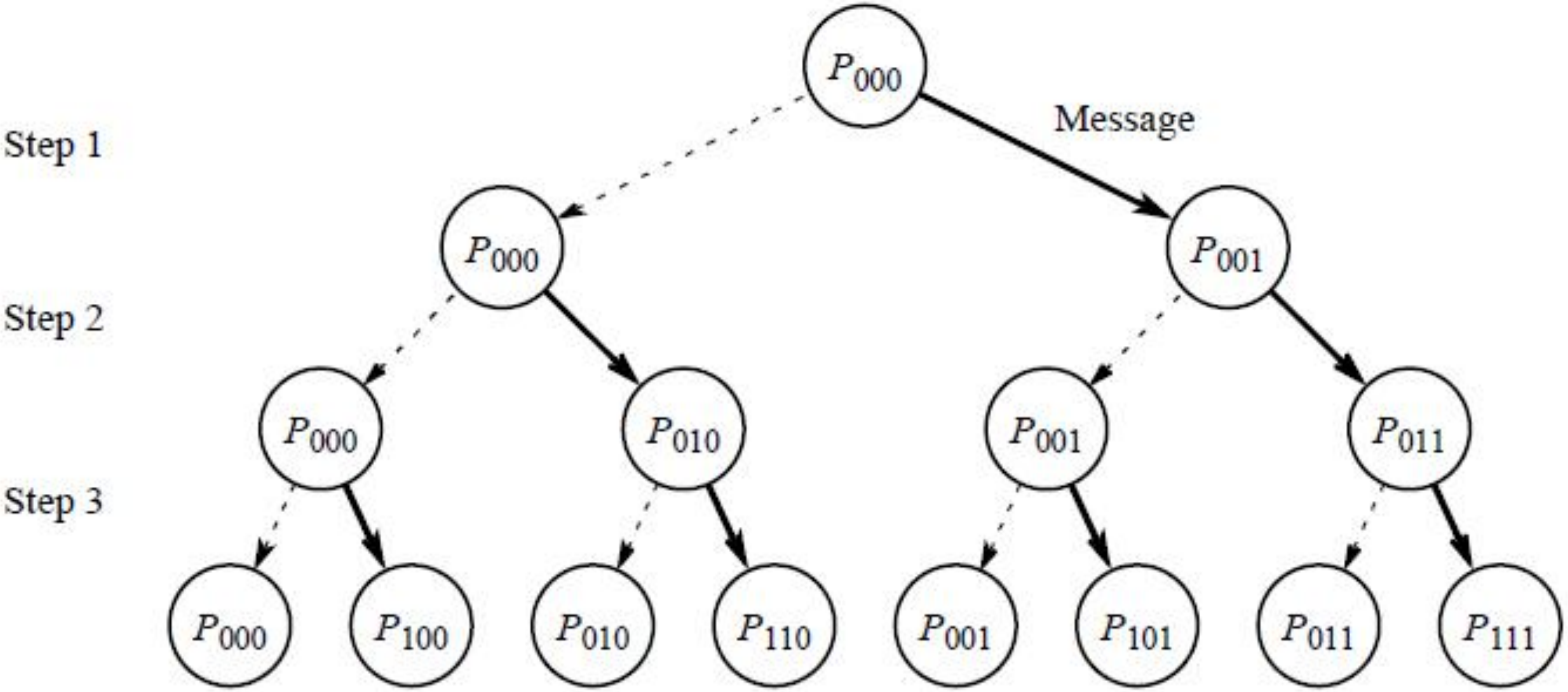
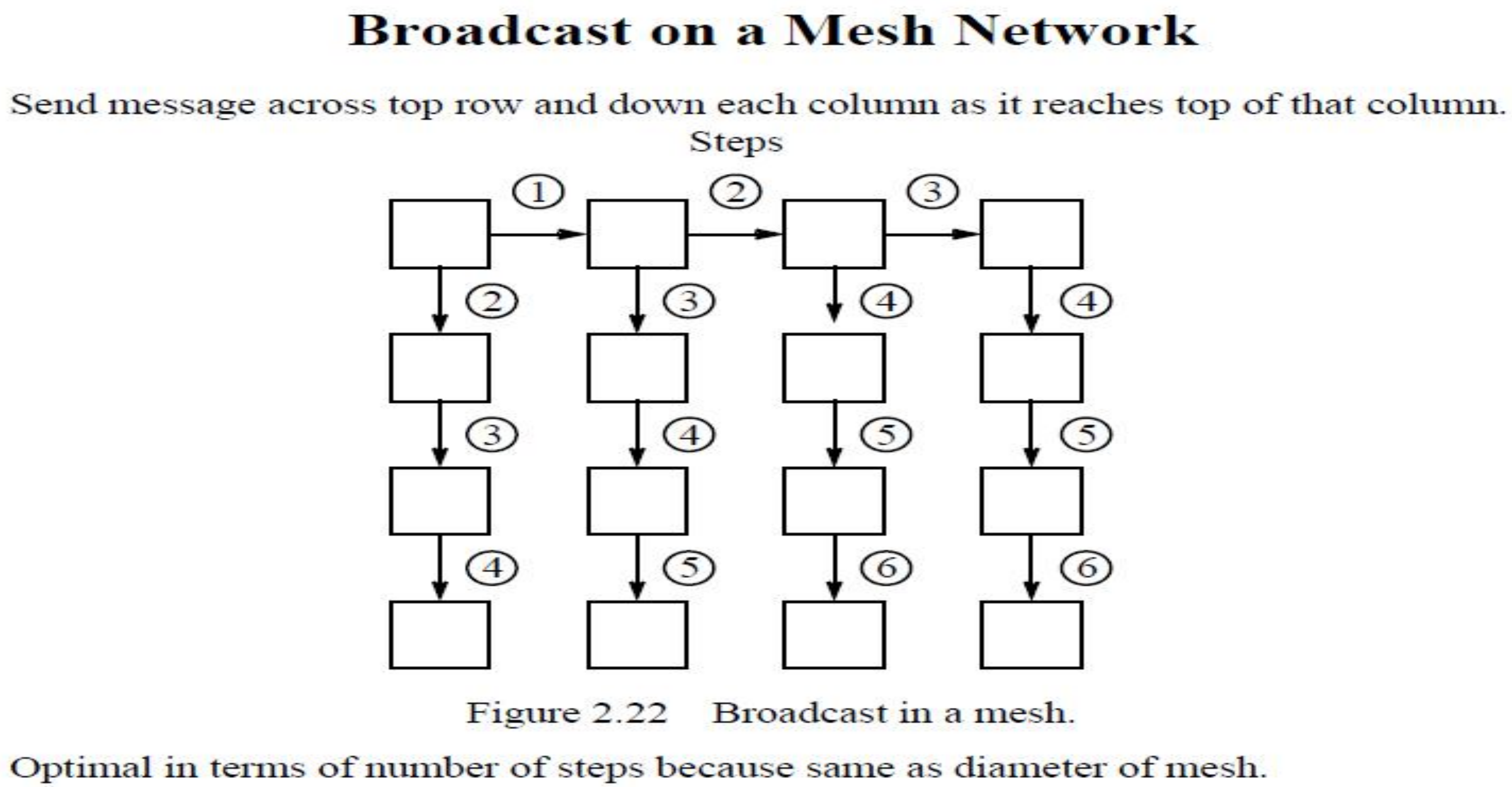


Figure 2.21 Broadcast as a tree construction.

Time Complexity of Broadcast (4)



Broadcast on a Mesh Network

Send message across top row and down each column as it reaches top of that column.

Steps
①
②
③
②
③
④
④
③
④
⑤
⑤
④
⑤
⑥
⑥

Figure 2.22 Broadcast in a mesh.

Optimal in terms of number of steps because same as diameter of mesh.

Broadcast on a Workstation Cluster

Broadcast on a single Ethernet connection can be done using a single message that is read by all the destinations on the network simultaneously

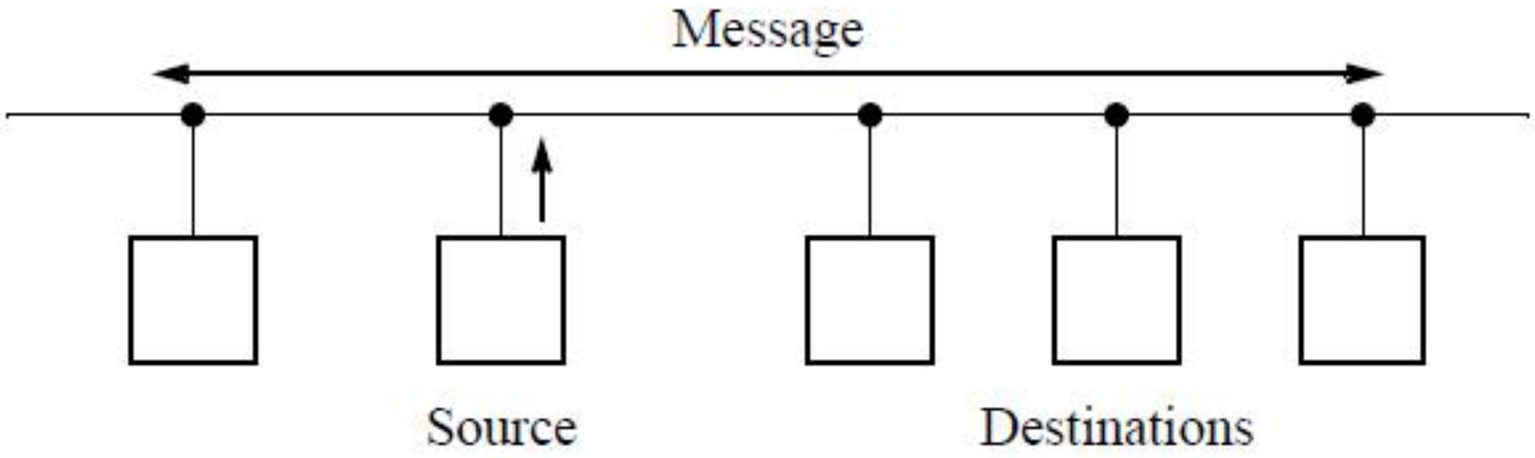


Figure 2.23 Broadcast on an Ethernet network.

1-to-*N* Fan-Out Broadcast

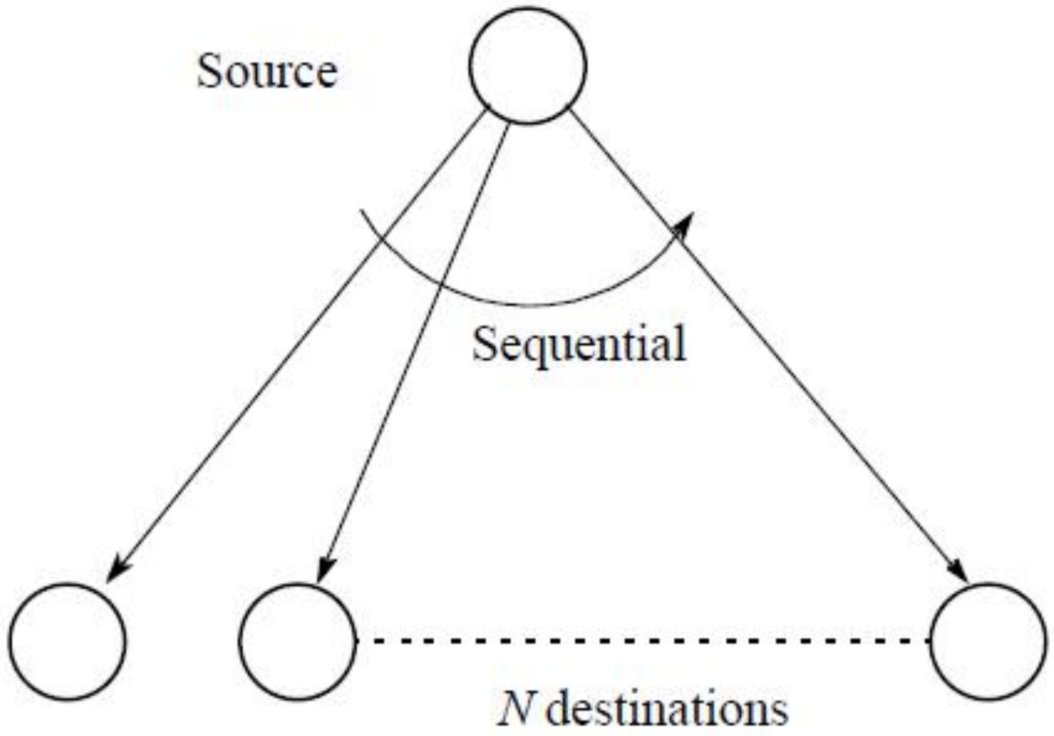


Figure 2.24 1-to-*N* fan-out broadcast.

1-to-*N* Fan-Out Broadcast on a Tree

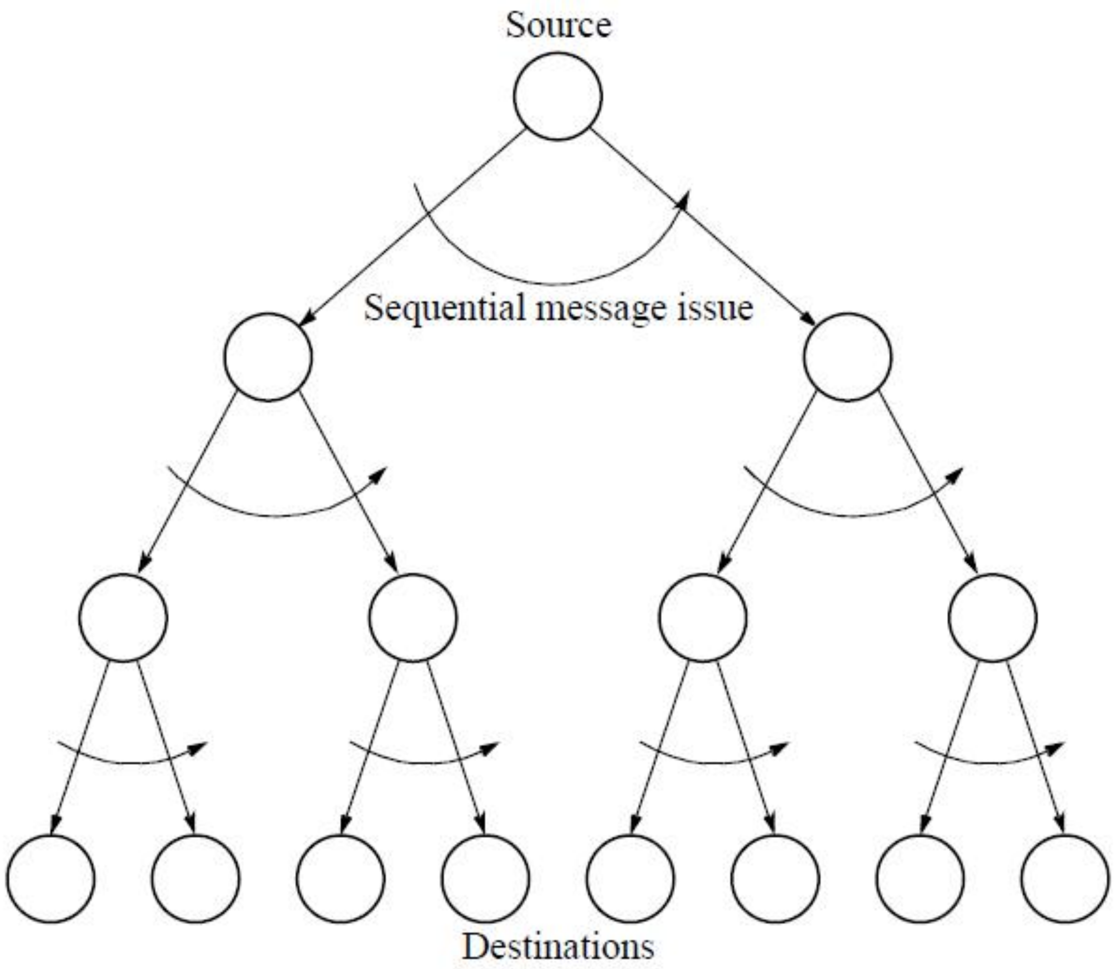
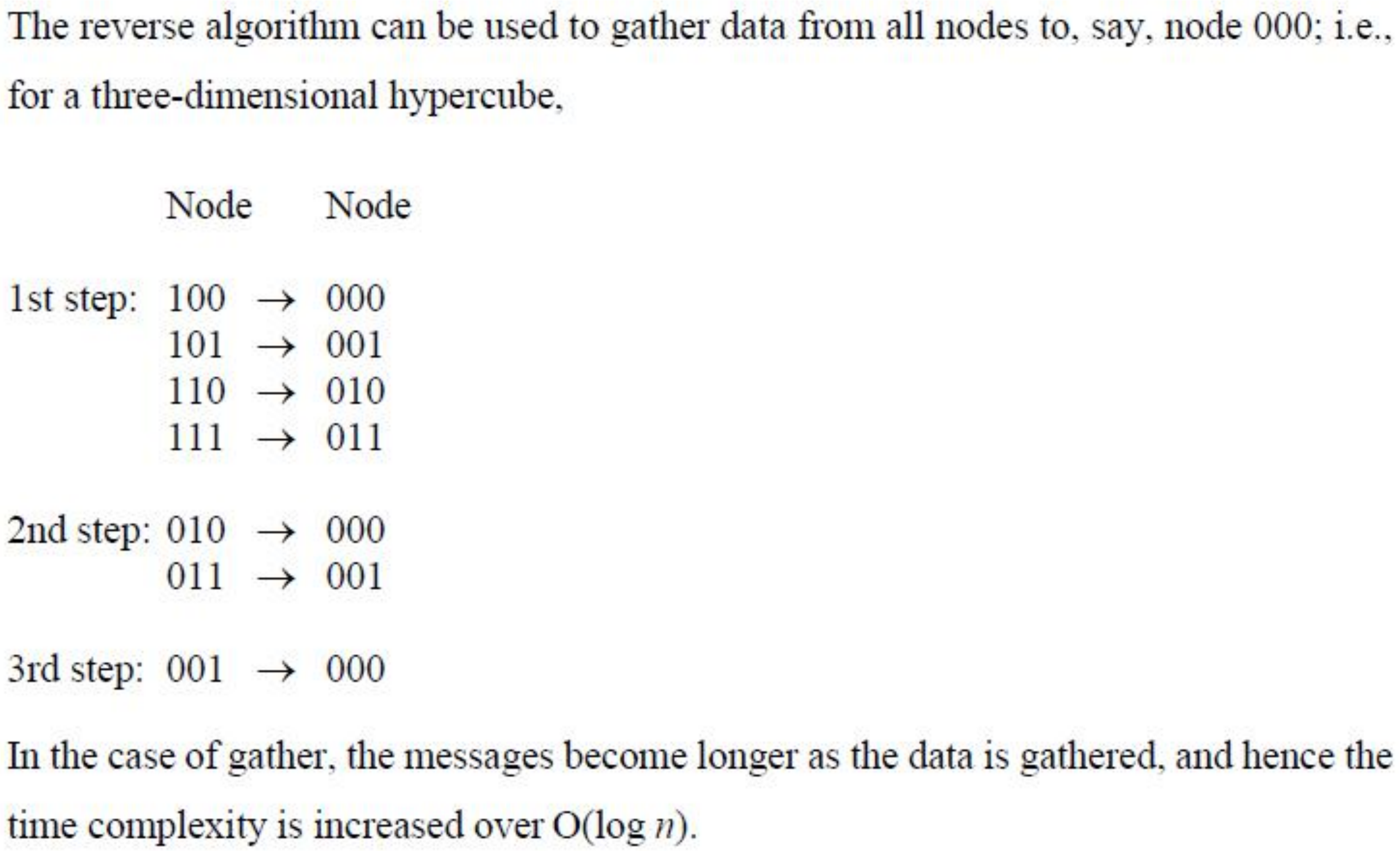


Figure 2.25 1-to-*N* fan-out broadcast on a tree structure.

Gather on a Hypercube Network



The reverse algorithm can be used to gather data from all nodes to, say, node 000; i.e., for a three-dimensional hypercube,

	Node		Node
1st step:	100	→	000
	101	→	001
	110	→	010
	111	→	011
2nd step:	010	→	000
	011	→	001
3rd step:	001	→	000

In the case of gather, the messages become longer as the data is gathered, and hence the time complexity is increased over O(log n).

Debugging and Evaluating Parallel Programs

Visualization Tools

Programs can be watched as they are executed in a *space-time diagram* (or *process-time diagram*):

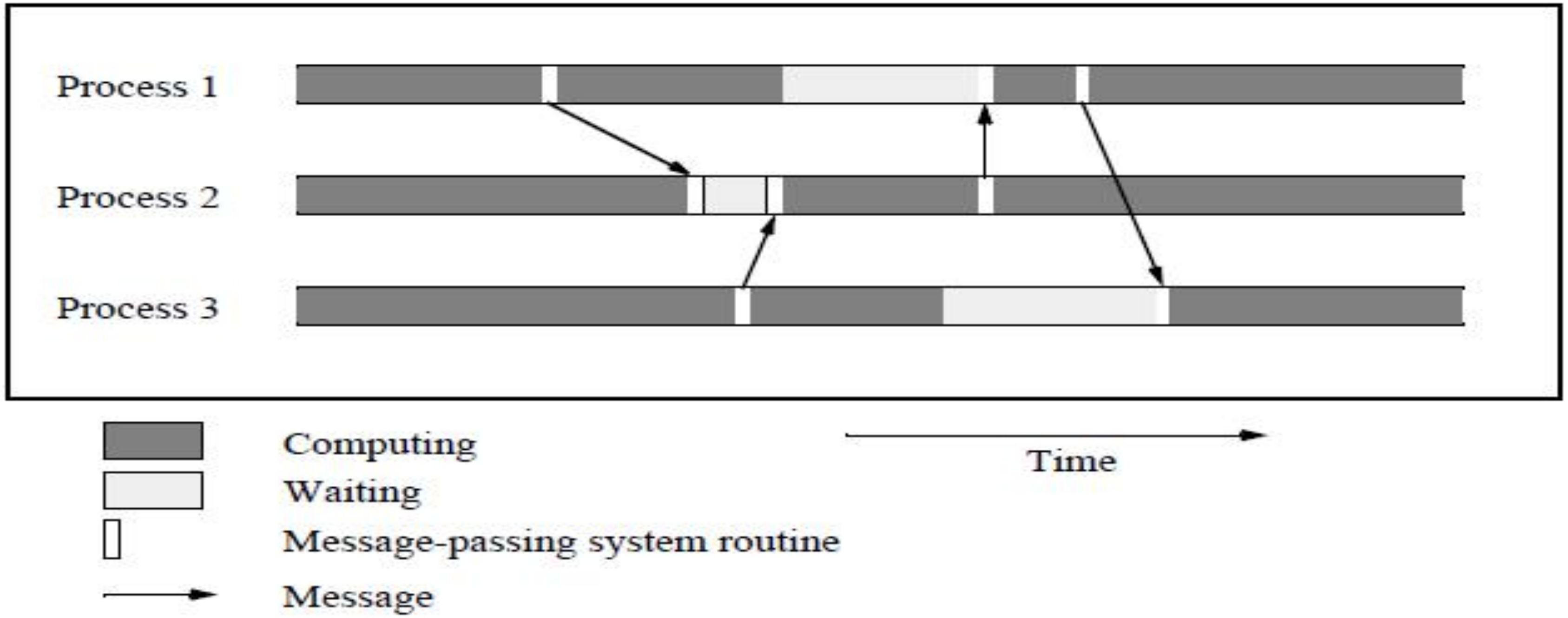
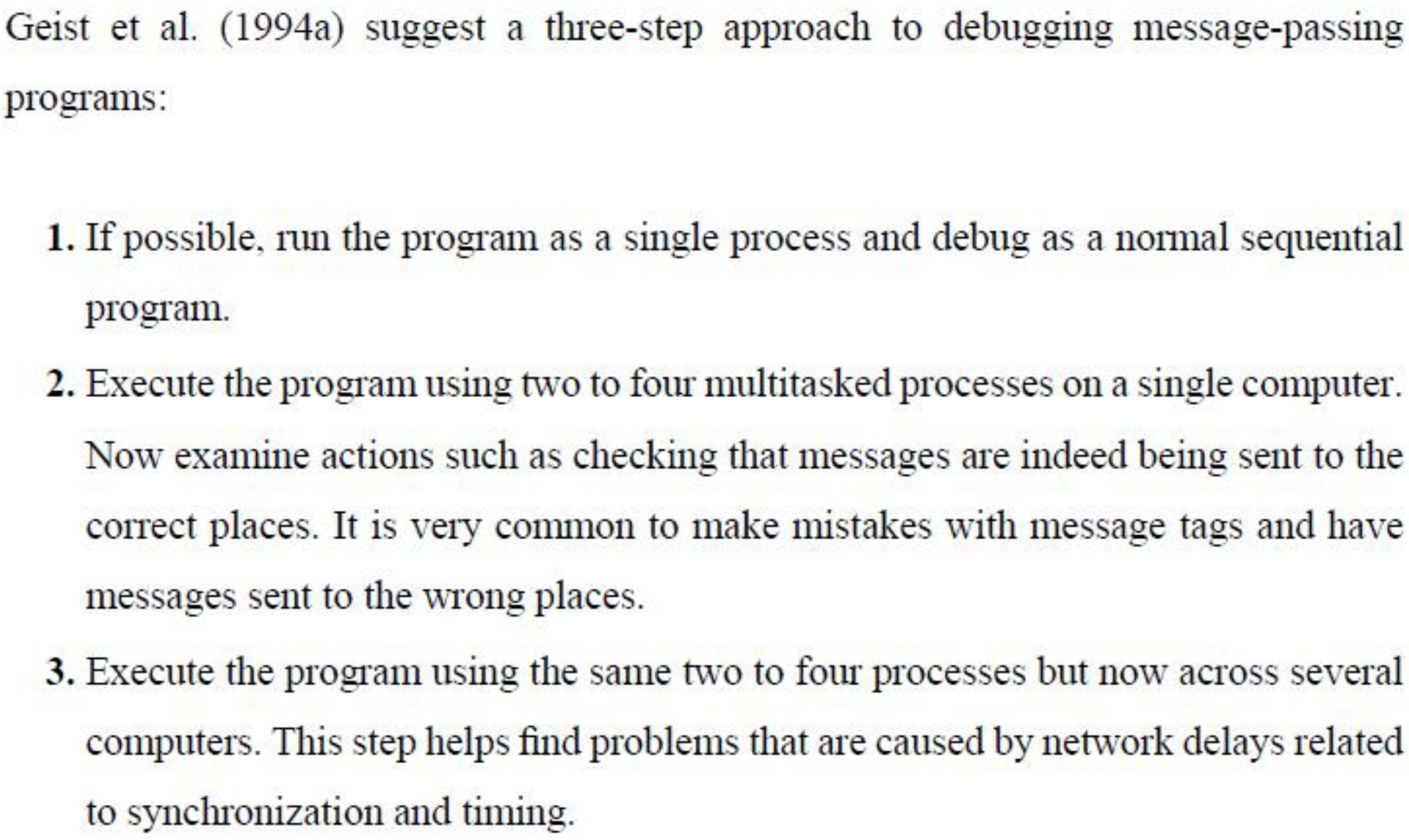


Figure 2.26 Space-time diagram of a parallel program.

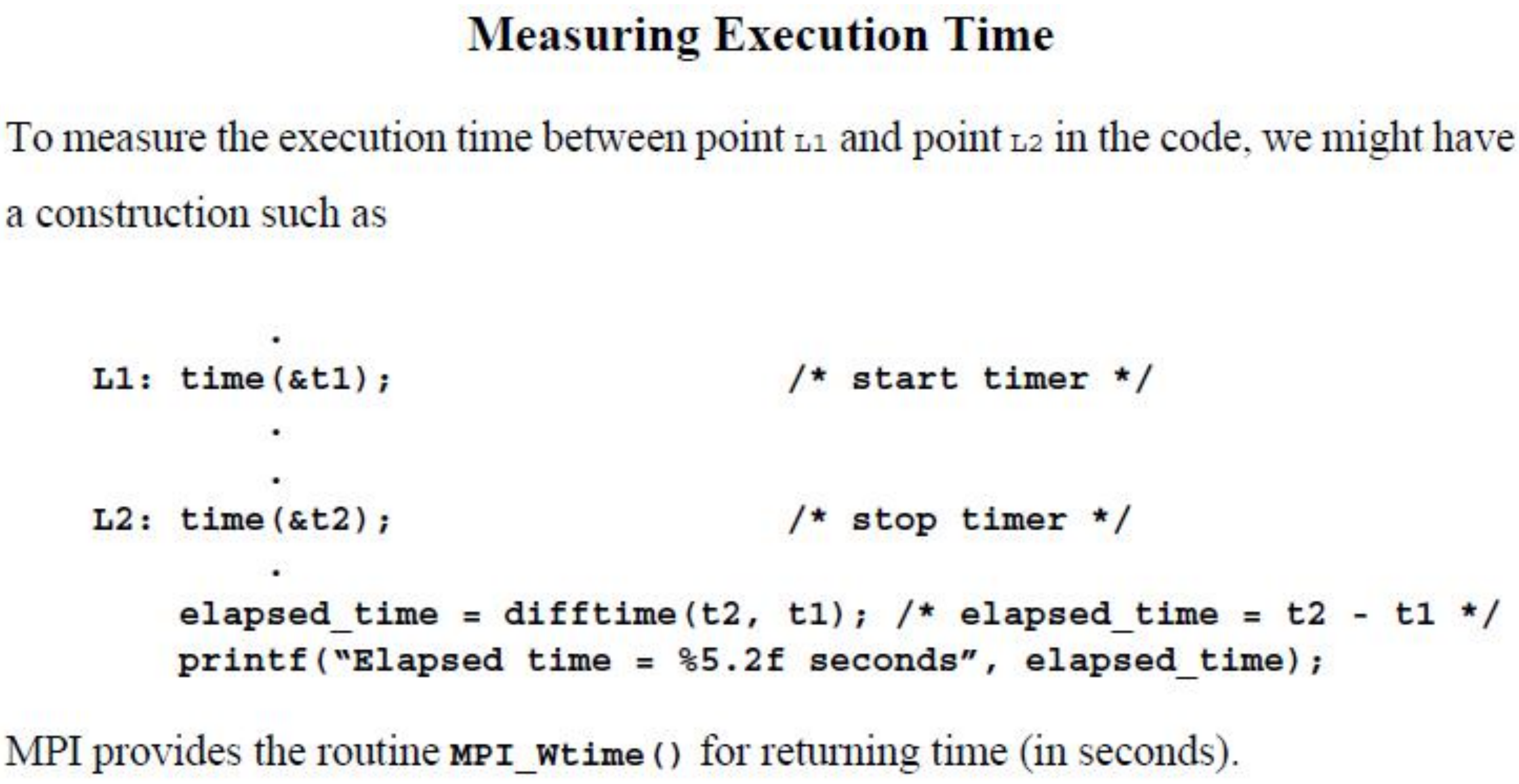
Debugging Strategies



Geist et al. (1994a) suggest a three-step approach to debugging message-passing programs:

1. If possible, run the program as a single process and debug as a normal sequential program.
2. Execute the program using two to four multitasked processes on a single computer. Now examine actions such as checking that messages are indeed being sent to the correct places. It is very common to make mistakes with message tags and have messages sent to the wrong places.
3. Execute the program using the same two to four processes but now across several computers. This step helps find problems that are caused by network delays related to synchronization and timing.

Evaluating Programs Empirically



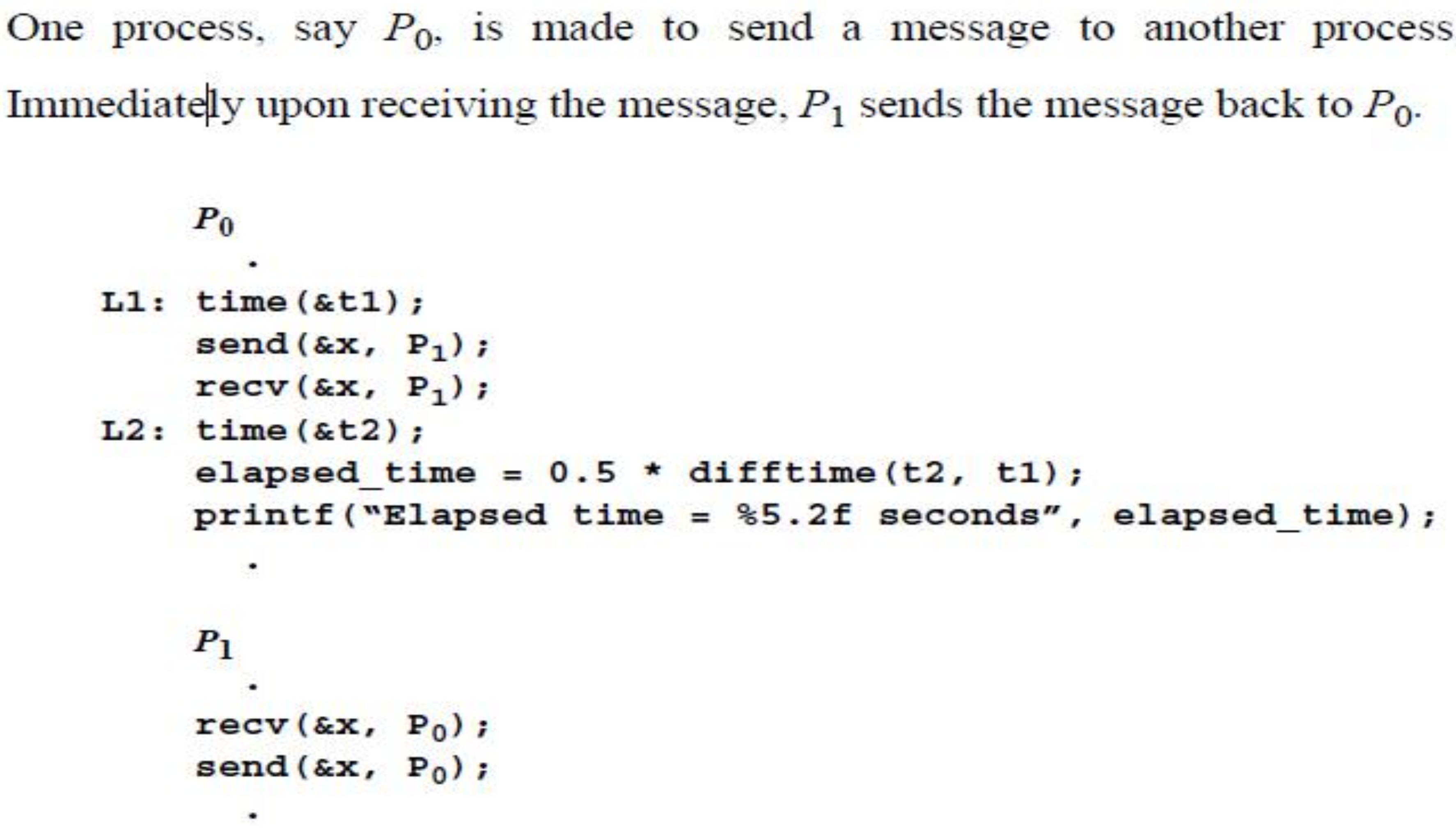
Measuring Execution Time

To measure the execution time between point L1 and point L2 in the code, we might have a construction such as

```
        .
L1: time(&t1);                  /* start timer */
        .
        .
L2: time(&t2);                  /* stop timer */
        .
    elapsed_time = difftime(t2, t1); /* elapsed_time = t2 - t1 */
    printf(“Elapsed time = %5.2f seconds”, elapsed_time);
```

MPI provides the routine `MPI_Wtime()` for returning time (in seconds).

Communication Time by the Ping-Pong Method



One process, say P_0, is made to send a message to another process, say P_1.

Immediately upon receiving the message, P_1 sends the message back to P_0.

P_0

```
     .
L1: time(&t1);
    send(&x, P1);
    recv(&x, P1);
L2: time(&t2);
    elapsed_time = 0.5 * difftime(t2, t1);
    printf("Elapsed time = %5.2f seconds", elapsed_time);
     .
```

P_1

```
     .
    recv(&x, P0);
    send(&x, P0);
     .
```

Profiling

A *profile* is a histogram or graph showing time spent on different parts of program:

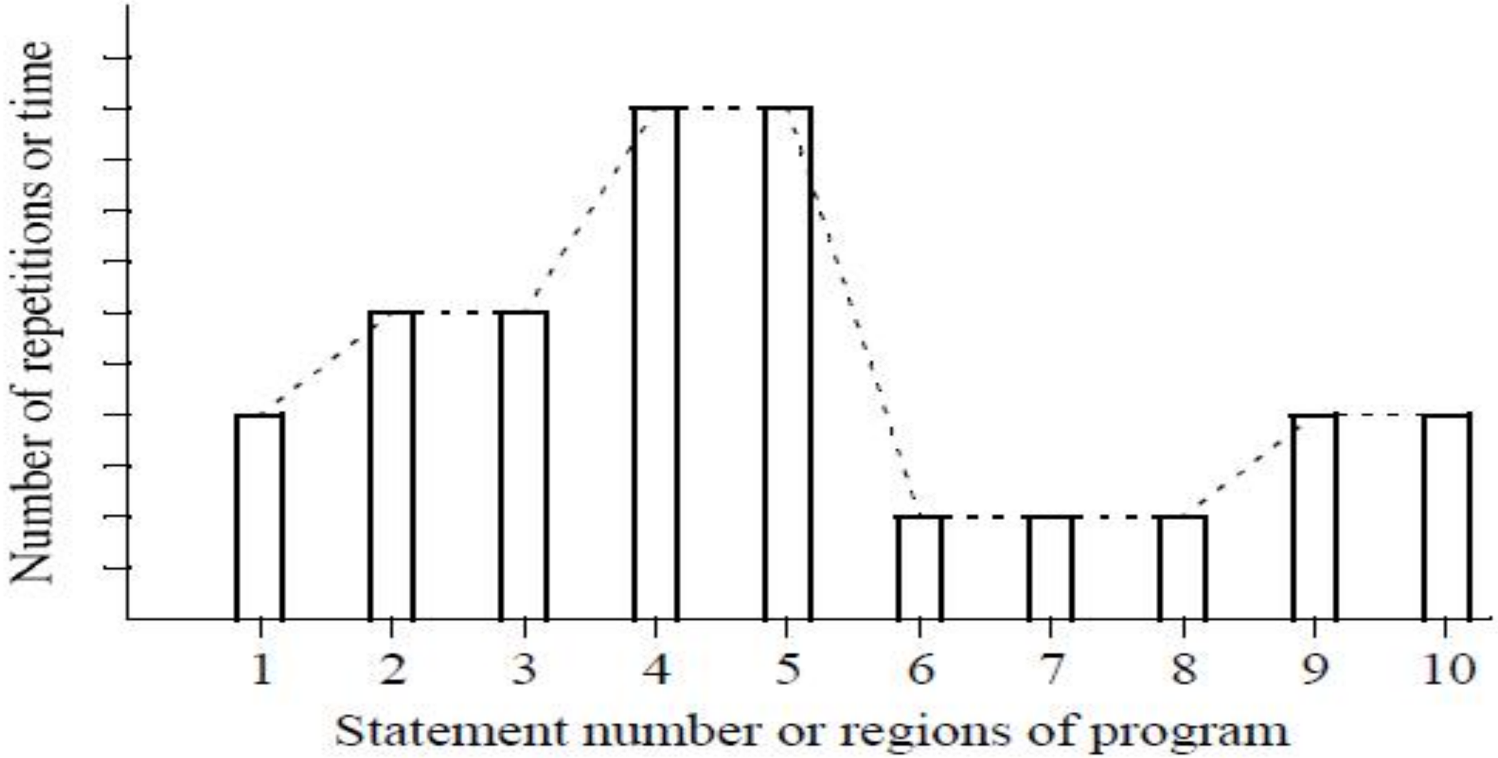


Figure 2.27 Program profile.