*

Message-Passing Computing

*Material based on B. Wilkinson et al., "PARALLEL PROGRAMMING. Techniques and Applications Using Networked Workstations and Parallel Computers"

Basics of Message-Passing Computing

We will use high-level language C and message-passing library calls to handle message passing between processes.

We must specify explicitly what processes to run, when to pass messages, and what to pass in messages.

To do this we need:

- Means to create processes to run on separate computers.

- A means to send/receive messages.

Process Creation

Process: independent parallelizable subparts of a problem.

Usually, one per processor.

Two methods of creating processes: static and dynamic

Static Process Creation: The number is specified before execution begins and doesn't increase.

Programmer specifies programs to run and number at command line.

Usually, there is one *master process*, that controls the overall behavior. The remaining processes are *slave* or *worker processes.*

SPMD Model

In the *single program multiple data* (SPMD) model, the processes are merged into one program.

Control sections in the program select portions of the program to be executed for each process.

Executable code compiled for each processor (may be different type/OS). Each process loads its version.

SPMD commonly used for MPI.

Dynamic Process Creation

This is when new processes can be created while others are already running. They can also be destroyed.

More powerful, but significant overhead to create new process.

Usually used with *multiple program multiple data* (MPMD). For MPMD, different and separate programs written for different processes.

Use master/slave method. Master process starts, and *spawns* (creates) new processes as needed. See Figure 2.2.

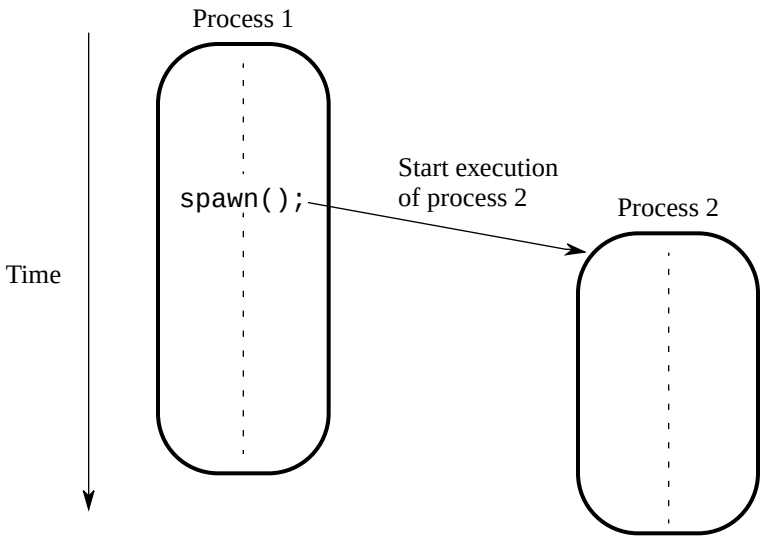


Figure 2.2 Spawning a process.

Basic Send and Receive Routines

Fundamental to be able to send/receive messages.

Simplest form would have destination ID and message, such as:

```
send(&x, destination_id); // in source process

recv(&y, source_id);  // in destination process
```

Very simple format. Doesn't even allow for multiple data types, and arrays etc.

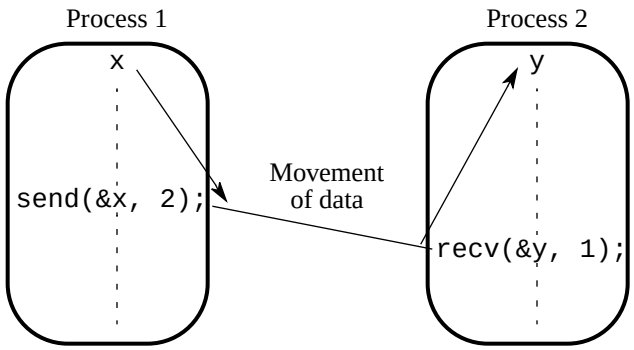


Figure 2.3 Passing a message between processes using `send()` and `recv()` library calls.

Synchronous Message Passing

Synchronous is used when routines do not return until message transfer has finished.

Routines don't require message buffers.

Two processes using synchronous routines to transfer a message will be synchronized.

Neither can proceed until the message has been sent and the other has received it.

Term *rendezvous* used to describe this synchronization.

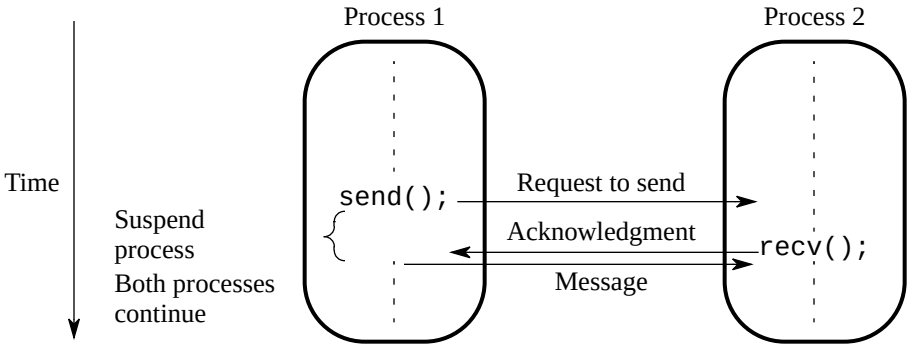
Synchronous Message Passing Cont.

Requires signalling, such as a three-way protocol:

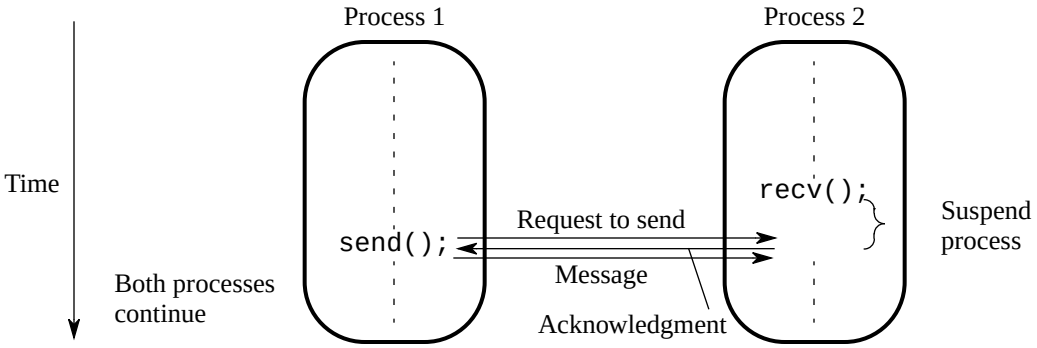
1. Source sends a "request to send" message to destination.

2. When destination ready to receive message, it sends acknowledgment.

3. When acknowledgment received, source sends actual message.



(a) When `send()` occurs before `recv()`



(b) When `recv()` occurs before `send()`

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

Blocking and Nonblocking Message Passing

For MPI, these type of routines can return before message transfer is complete.

Requires use of *message buffers* between source and destination to hold data until transfer complete. See Figure 2.5.

For a *recv()*, a *send()* must be done first else buffer empty and *recv()* must wait.

For *send()*, once message transfered to buffer, the process can move onto other tasks.

However, it may be necessary to know that message has actually been received.

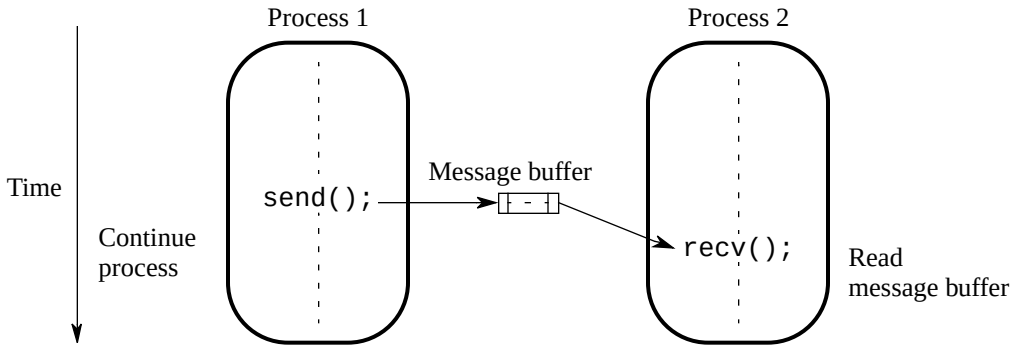


Figure 2.5 Using a message buffer.

Blocking and Nonblocking Message Passing Cont

MPI uses the following definitions:

Blocking Routines: Routines that use message buffer and return after local actions have finished (message transfer may not yet have completed). Also called *locally blocking.*

Nonblocking: Routines that return immediately.

For MPI nonblocking, data storage used in transfer must not be locally modified until transfer complete.

Message Selection

Previously, the destination process only accepted messages from the process whose ID was specified in the *recv()* function call.

Also useful to be able to specify a *wild card* address so the destination process will accept a message from any process.

For more flexibility, can also select a message by an attached *message tag.*

Differentiates between different types of messages.

Can also have *wild card* message tags.

Still need more powerful selection mechanism to differentiate between messages sent between library routines and user processes.

Broadcast, and Scatter

Broadcast: When one processes sends a message to multiple destination processes at once. See Figure 2.6.

Must first identify which processes that will be involved in broadcast.

Scatter: Send each element of an array of data at the root process to a different process.

Need to define group and root process. See Figure 2.7.

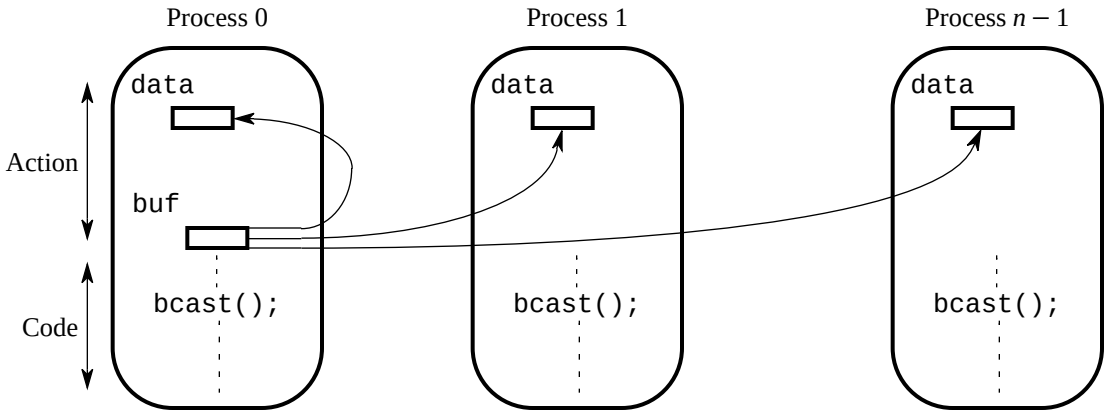


Figure 2.6 Broadcast operation.

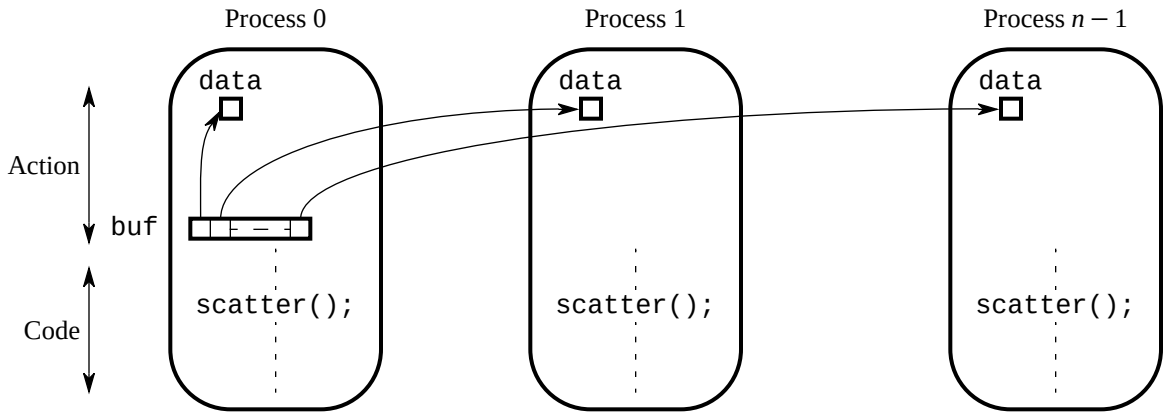


Figure 2.7 Scatter operation.

Gather and Reduce

Gather: When one process (root) collects individual data from a set of processes. Again, have to define a group and a root process.

Typically done after computation to collect results from each process. See Figure 2.8.

Reduce: When gather operation is combined with an arithmetic or logical operation, called *reduce* operation. See Figure 2.9.

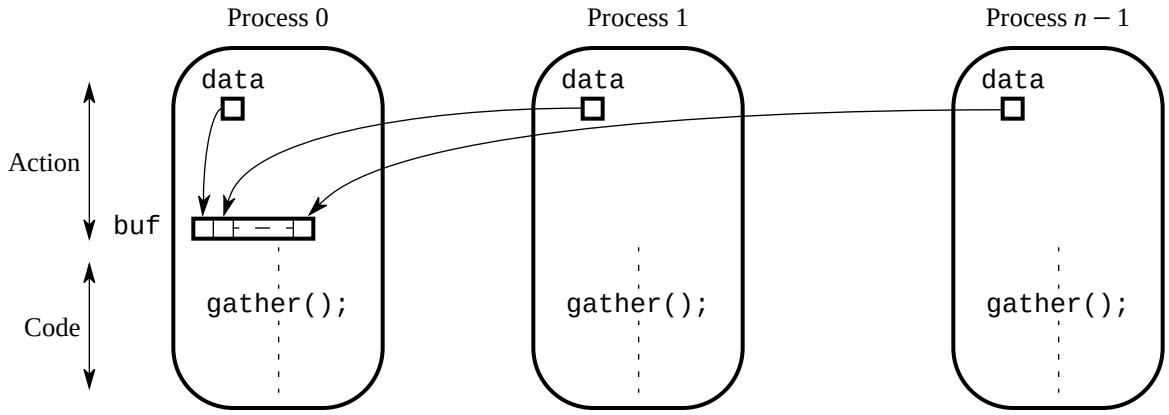


Figure 2.8 Gather operation.

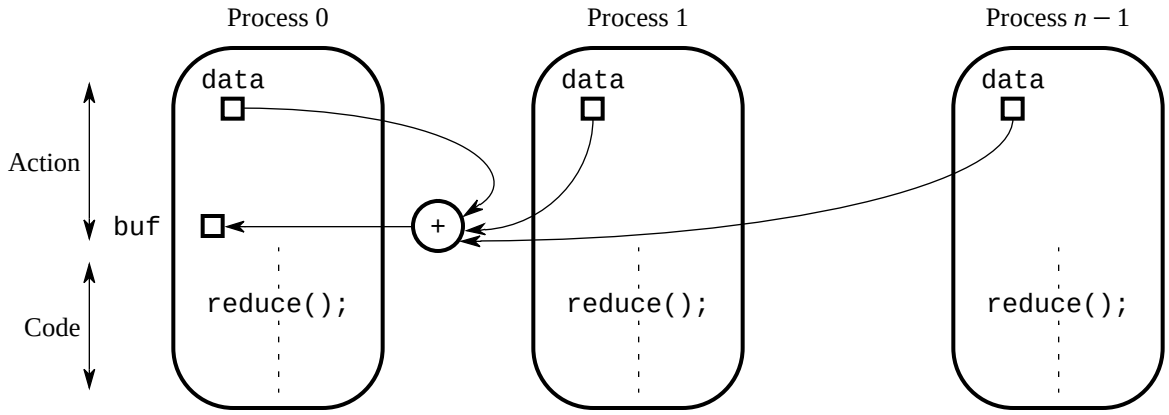


Figure 2.9 Reduce operation (addition).

Message Passing Interface (MPI)

Provides library routines for message passing and associated operations.

Excellent MPI resource:

http://www-unix.mcs.anl.gov/mpi/

Click on: "MPI Standard 1.1." Scroll down to contents for detailed description of MPI and routines. At bottom, see: "MPI 1.1 Standard Index" for easy look up of specific functions.

MPI on CAS Network

For linux cluster, log onto penguin.cas.mcmaster.ca, and use cnode1 to cnode6. For suns, birkhoff, wolf01 to wolf22. Easiest to use all linux or all suns.

Create ~/.rhosts file with names of these machines.

ie.

For cluster:

```
penguin
cnode1
cnode2
cnode3
```

For suns:

```
wolf01.cas.mcmaster.ca
wolf02.cas.mcmaster.ca
wolf03.cas.mcmaster.ca
wolf04.cas.mcmaster.ca
```

Compiling and Running MPI Programs

To compile, log onto penguin or one of the fox machines (depends on which you will use) and type:

```
mpicc prog.c -o prog
```

To run program, type:

```
mpirun -np 4 -machinefile machines prog
```

Option "-np" is the number of processes to run for computation. Here, 4 processes.

File "machines" contains the names of the machines to run processes on. Names should also be in .rhosts file!

ie. for "machines"

```
penguin
cnode1
cnode2
cnode3
```

Initializing Program

All MPI programs must contain the following:

```
main (int argc,char *argv[])
{
MPI_Init(&argc, &argv);      /* Initialize MPI */

.
.
.

MPI_Finalize();              /* terminate MPI */

}
```

Communicators

Communicators define the scope of a communication operation. It provides both group and context information.

It identifies a set of processes that are allowed to communicate with each other.

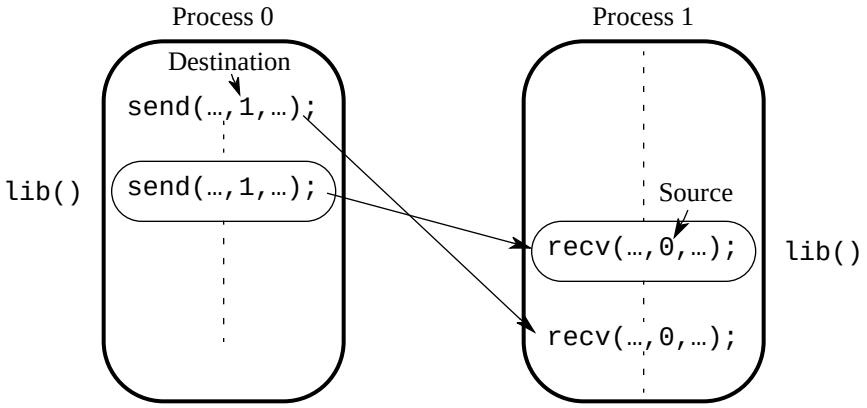
The context creates a "safe universe" for processes to communicate.

Allows a means to separate communication from library from user code. See Figure 2.15.

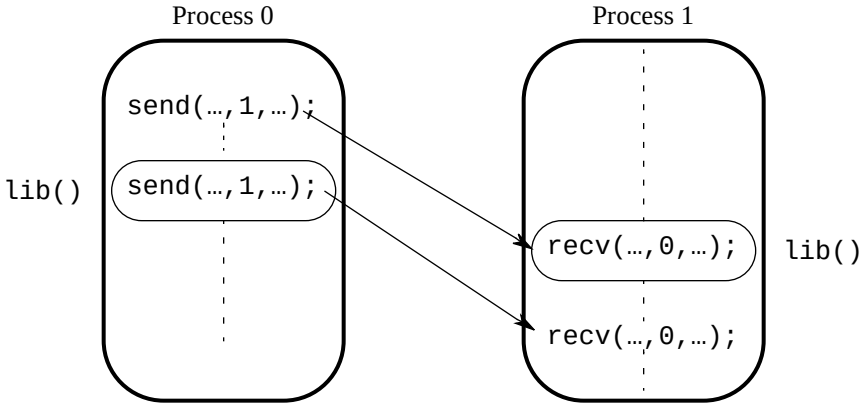
Each process has a rank (ID) for a communicator.

For communicator of size n, ranks are sequential from 0 to $n-1$.

Can create new communicator from old. Same context, but you get to specify subset of group members.



(a) Intended behavior



(b) Possible behavior

Figure 2.15 Unsafe message passing with libraries.

Communicators Cont

Communicator is used to define group and context for all point-to-point and collective communication operations. The rank (ID) parameter is relative to the communicator.

At start of program, have default communicator called *MPI_COMM_WORLD* that all processes belong to. Context is user program.

For many applications, this is sufficient.

Getting One's Bearings

A process needs to be able to determine its rank for a given communicator. This is done as follows:

```
int myrank;

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
```

One often needs to know how many processes in a communicator. The size of a group attached to a communicator can be determined as follows:

```
int nprocs;

MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
```

Point-to-Point Communication

MPI send and receive routines use message tags.

Can use the wild cards *MPI_ANY_TAG* and *MPI_ANY_SOURCE* for the tag and source parameters for receive routines.

Specify data type by using one of the standard MPI datatypes (see Table 5.1) or a user defined type.

MPI has several versions of send and receive routines.

Distinguished by concepts of *locally complete* and *globally complete.*

Locally complete means routine has complete at least its part of the operation.

Globally complete means everyone involved in operation have completed their part.

Data Types

MPI Datatype	C Datatype
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_LONG_INT	long long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_SHORT	unsigned short

Table 5.1
Basic (predefined) MPI datatypes for C

*

*W. Gropp, E. Lusk, and A. Skjellum, *Using MPI. Portable Parallel Programming with the Message-Passing Interface, 2nd Ed.*

MPI Blocking Routines

These routines return when they are locally complete.

For blocking send routine, this means when data has been copied from the send buffer, and the buffer can now be safely reused.

For receive buffer, this is when data has been copied to the receive buffer and is available to be read.

Blocking send command:

```
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
```

MPI Blocking Routines Cont

Blocking receive command:

```
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
```

The status variable contains info about message received. Actual message tag is status.MPI_TAG. Rank of sending process is status.MPI_SOURCE.

Can determine number of datatype elements actually received with following command:

```
MPI_Get_count(&status, datatype, &nelements);
```

Program ezstarto.c

```
/*
 * Copyright 1998-2001, University of Notre Dame.
 * Authors: Jeffrey M. Squyres, Arun Rodrigues, and Brian Barrett
 *          with Kinis L. Meyer, M. D. McNally, and Andrew Lumsdaine
 *
 * This file is part of the Notre Dame LAM implementation of MPI.
*
 *  NOTE: This example has unnecessary use of printf.  They are only
 *        to give the new user some feedback.
 *
 */


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

#define WORKTAG 1
#define DIETAG 2
#define NUM_WORK_REQS 100

/*
 * Local functions
 */
static void master(void);
static void slave(void);
```

Main function

```
/*
 * main
 * This program is really MIMD, bit is written SPMD for
 * simplicity in launching the application.
 */
int
main(int argc, char* argv[])
{
int myrank;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD,/* group of everybody */
&myrank); /* 0 thru N-1 */

if (myrank == 0) {
master();
} else {
slave();
}


MPI_Finalize();
return(0);
}
```

Control Function for Master

```
/*
 * master
 * The master process sends work requests to the slaves
 * and collects results.
 */
static void
master(void)
{
int ntasks, rank, work;
double result;
MPI_Status status;

MPI_Comm_size(MPI_COMM_WORLD,
&ntasks); /* #processes in app */
/*
 * Seed the slaves.
 */
work = NUM_WORK_REQS; /* simulated work */

        printf("Master started.\n\n");

for (rank = 1; rank < ntasks; ++rank) {

MPI_Send(&work,/* message buffer */
1,/* one data item */
MPI_INT,/* of this type */
rank,/* to this rank */
WORKTAG,/* a work message */
MPI_COMM_WORLD);/* always use this */
work--;

printf("Seeding Machine %d.\n\n",rank);
}
```

Control Function for Master Cont.

```
/*
 * Receive a result from any slave and dispatch a new work
 * request until work requests have been exhausted.
 */
while (work > 0) {

MPI_Recv(&result,/* message buffer */
1,/* one data item */
MPI_DOUBLE,/* of this type */
MPI_ANY_SOURCE,/* from anybody */
MPI_ANY_TAG,/* any message */
MPI_COMM_WORLD,/* communicator */
&status); /* recv'd msg info */

MPI_Send(&work, 1, MPI_INT, status.MPI_SOURCE,
WORKTAG, MPI_COMM_WORLD);

work--; /* simulated work */

printf("Feeding machine %d.\n\n",status.MPI_SOURCE);
}
/*
 * Receive results for outstanding work requests.
 */

printf("Receiving outstanding work.\n\n");

for (rank = 1; rank < ntasks; ++rank) {
MPI_Recv(&result, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &status);
}
```

Control Function for Master Cont. II

```
/*
 * Tell all the slaves to exit.
 */
for (rank = 1; rank < ntasks; ++rank) {
MPI_Send(0, 0, MPI_INT, rank, DIETAG, MPI_COMM_WORLD);
}


printf("Master exiting.\n\n");

}
```

Control Function for Slave

```
/*
 * slave
 * Each slave process accepts work requests and returns
 * results until a special termination request is received.
 */
static void
slave(void)
{
double result;
int work;
MPI_Status status;

for (;;) {
MPI_Recv(&work, 1, MPI_INT, 0, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
/*
 * Check the tag of the received message.
 */
if (status.MPI_TAG == DIETAG) {
return;
}

sleep(1);
result = 6.0; /* simulated result */

MPI_Send(&result, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
}
```

MPI Nonblocking Routines

A nonblocking routine returns immediately, whether or not it is locally complete.

Allows interspersion of communication and computation operations. Important for slow communication links.

Nonblocking send command:

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

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
request - ID for given send operation.
```

MPI Nonblocking Routines Cont.

Nonblocking receive command:

```
MPI_Irecv(buf, count, datatype, src, tag, comm, request);

  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
  request - ID for given receive operation.
```

Need to be able to test for completion:

```
MPI_Test(request,flag, status);

  request - ID for given receive operation.
  flag - true if operation complete
  status - status of operation.

MPI_Wait(request,status);

  request - ID for given receive operation.
  status - status of operation.
```

MPI Nonblocking Routines Example

```
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);
}
```

*

*B. Wilkinson, and M. Allen, *Parallel Programming. Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice-Hall, 1999.

MPI Collective Communication

MPI versions of broadcast, scatter, gather, and reduce.

The communicator parameter defines the group to be used.

Broadcast command:

```
MPI_Bcast(buf, count, datatype, root, comm);

  buf - Address of send/receive buffer
  count- Number of entries buffer can hold/contains
  datatype - Datatype of each item
  root - Rank of broadcast root
  comm - Communicator
```

MPI Scatter Operation

Scatter command:

```
MPI_Scatter(sendbuf, sendcount, sendtype, recvbuf,
            recvcount, recvtype, root, comm);

  sendbuf - Address of send buffer (only significant
            at root).
  sendcount- Number of elements to send to each process
             (only significant at root).
  sendtype - Datatype of each item (only significant
                 at root).
  recvbuf - Address of receive buffer
  recvcount- Number of elements to put in recv buffer
  recvtype - Datatype of each item in recv buffer
  root - Rank of sending process
  comm - Communicator
```

Root process sends *sendcount* elements to each process in *comm.* Process 0 gets the first *sendcount* elements, process 1 the next and so on.

MPI Gather Operation

Gather command:

```
MPI_Gather(sendbuf, sendcount, sendtype, recvbuf,
           recvcount, recvtype, root, comm);

  sendbuf - Address of send buffer
  sendcount- Number of elements in send buffer
  sendtype - Datatype of send buffer
  recvbuf - Address of receive buffer (only significant
            at root).
  recvcount- Number of elements for a single receive (only
             significant at root).
  recvtype - Datatype of recv buffer elements (only significar
            at root).
  root - Rank of receiving process
  comm - Communicator
```

Root process receives *sendcount* elements from each process in *comm.* Process 0 sends the first *recvcount* elements (goes at start of *recvbuf*), process 1 sends the next and so on (in *recvbuf*, goes after first *recvcount* elements).

MPI Reduce Operation

Reduce command:

```
MPI_Reduce(sendbuf, recvbuf, count, datatype,
           op, root, comm);

  sendbuf - Address of send buffer.
  recvbuf - Address of receive buffer (only
            significant at root).
  count- Number of elements to put in send buffer.
  datatype - Datatype of each item in send buffer.
  op - reduce operation.
  root - Rank of root process.
  comm - Communicator.
```

MPI has several predefined operations such as MPI_MAX, MPI_MIN, MPI_SUM, and MPI_PROD. Can also have user-defined operations.

If each process provides more than one element, then reduce operation performed element-wise.

Program mpidemo.c

```
/*************************************************
 * this is a mpi demo program for course 4f03
 * what it does is to add 1000 random numbers
 * in parallel on several machines and return
 * the results to the root process
 *
 * Based on example in Parallel Programming. Techniques and
 * Applications Using Networked Workstations and Parallel Computers,
 * Prentice-Hall, 1999.
*/

#include "mpi.h"
#include <stdio.h>
#include <math.h>
#include <sys/utsname.h>

#define MAXSIZE 1000
#define FILE_NAME  "rand_data.txt"


int main(int argc, char *argv[])
{
    int myid, numprocs;
    int data[MAXSIZE], i, x, low, high, myresult=0, result=0;
    FILE *fp;
    struct utsname my_name;
    char name[SYS_NMLN];
    MPI_Status  status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    /*get my name*/
    uname(&my_name);
```

```
if(myid == 0)  {
    printf("\n\nMaster (rank %d): %s started\n\n\n",
           myid, my_name.nodename);

    for(i=1; i<numprocs; i++){
      MPI_Recv(name, SYS_NMLN, MPI_CHAR, i, 0, MPI_COMM_WORLD,
               &status);
      printf("Slave (rank %d): %s started\n\n",
             status.MPI_SOURCE, name);
    }
}
else {
    MPI_Send(my_name.nodename, SYS_NMLN, MPI_CHAR, 0, 0,
             MPI_COMM_WORLD);  }


if (myid == 0)
{
    printf("\n\nPreparing data...\n\n");

    /* Open Input File and Initialize Data */
    if ((fp = fopen(FILE_NAME,"r")) == NULL)    {
        printf("Can't open the input file: %s\n\n", FILE_NAME);
        exit(1);
    }

    /*read data from file*/
    for(i=0; i<MAXSIZE; i++)
        fscanf(fp,"%d", &data[i]);
}
```

```
    /*broadcast data to everyone*/
    MPI_Bcast(data, MAXSIZE, MPI_INT, 0, MPI_COMM_WORLD);


    /*job partitioning*/
    x = MAXSIZE/numprocs;
    low = myid * x;
    high = low + x;

    if (myid == (numprocs -1))
        high = MAXSIZE;

    for(i=low; i<high; i++)
        myresult += data[i];


    MPI_Reduce(&myresult, &result, 1, MPI_INT, MPI_SUM,
               0, MPI_COMM_WORLD);


    if (myid == 0)
       printf("The sum is %d, calculation is done!\n\n",
               result);


    MPI_Finalize();

    return 0;
}
```

Pseudocode Constructs

To specify MPI code, need to keep track of numerous parameters and details.

This makes examples more complicated and less readable.

For teaching parallel programming, much of this detail is unnecessary.

Will instead use pseudocode for describing algorithms. See notation used in Section 2.2.3.

Parallel Execution Time

In later chapters, we will discuss methods to achieve parallelism. We need means to evaluate them.

First: How fast is parallel implementation?

Need to be able to determine number of computation steps as well as estimate communication overhead.

Parallel execution time (t_p) has two parts: computation time (t_{comp}) and communication time (t_{comm})

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

Computation Time

Can be estimated in similar manner as for a sequential algorithm.

Difference: when processes are being executed simultaneously, we only have to analyze steps for most complex process.

Usually assume all processors are the same (same speed, memory etc).

Different types of computers will be handled by using algorithms that spread load across available computers (*load balancing*).

Communication Time

Depends on message size, interconnection structure, and transfer method (ie. circuit switching, store-and-forward, and wormhole routing).

For cluster of workstations, will have to consider network structure as well as contention. No simple equation feasible!

First approximation:

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

Communication Time Cont.

$t_{\textbf{startup}}$**:** This is the message *startup time.* Equivalent to time to send a message with no data.

Assumed to be a constant.

$t_{\textbf{data}}$**:** Time to send one data word. Assumed to be constant. There are n data words to send.

Transmission rate is in bits/sec. Would thus be: b/t_{data} bits/sec where b is the number of bits in a data word.

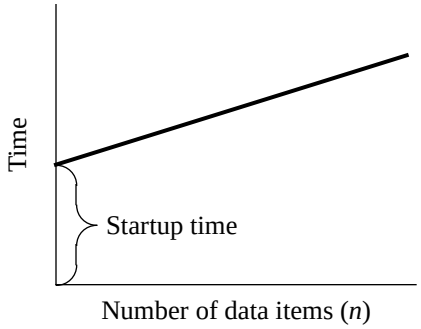


Figure 2.17 Theoretical communication time.

Interpretation of Equations

Analysis in following chapters make many assumptions.

Intended to give a starting point for how algorithm may actually perform.

Will normalize parallel execution time, t_p. Will be measured in units of some arithmetic operation.

Assuming homogeneous system and that all arithmetic operations take the same time (ie. addition takes same time as division).

Interpretation of Equations Cont.

If all processors performing same operation, then simply count the number of steps on one processor.

Take max number steps of all sequences running concurrently.

If computation has m steps, we get: $t_{\text{comp}} = m$

Have to measure communication time as well in units of computation steps.

Will assume all data formats take same time to send (ie. an eight bit char takes same time as a 64 bit float).

Interpretation of Equations: Communication

Suppose q messages sent, each containing n data elements. Gives:

$$t_{\text{comm}} = q(t_{\text{startup}} + n\, t_{\text{data}})$$

Both t_{startup} and t_{data} given in terms of computation steps so we can add t_{comp} and t_{comm}.

Startup and data transfer time dependent on actual system.

t_{startup} is often 1-2 orders of magnitude greater than t_{data} which is in turn larger than time for an arithmetic operation.

In practice, startup time dominates communication time unless n is large.

Latency Hiding

For a 200 MFLOPS machine with a startup time of 1us, the computer could execute 200 floating point operations in the time required for message startup.

Would have to perform 200 floating point operations between each message just so time computing would equal startup time.

Shareed memory manufacturers point to this as the weakness of message-passing computers.

Can handle this by overlapping computation and communication.

Latency hiding is the process of keeping a processor busy with needed work while waiting for the communication to finish.

Latency Hiding Cont.

Nonblocking routines are particularly good for this, but even MPI blocking routines allow a good amount of overlap.

Can also achieve latency hiding by mapping multiple processes to single process.

While one process is waiting for communication to end, the other process is doing useful work. These processes are sometimes referred to as *Virtual processors.*

For an m-process(or) algorithm running on a machine with n processors ($n < m$), we say the machine has *parallel slackness* of m/n.

To use parallel slackness to hide latency requires an efficient means of swapping processes such as threads.

Time Complexity

Can evaluate parallel algorithms using time complexity, in particular the $\mathbf{O}$ notation ("order of magnitude" or "big-oh" notation).

Captures characteristics of algorithm as some variable (usually data size) goes to infinity.

For algorithms, useful in comparing execution time (*time complexity*), memory requirements (*space complexity*). Also speed-up and efficiency.

Using notation for time, first need to estimate # of computation steps.

Take all arithmetic and logical operations to take same time; ignore all other operations.

Time Complexity Cont.

Derive expression for total number of steps in terms of useful variable (normally # of data items).

ie. Algorithm A1 requires $4x^2 + 2x + 12$ steps, where x is # of data elements. Growth function is *polynomial.*

ie. Algorithm A2 with function $5 \log x + 200$. This function is *logarithmic.* Note: unless specified otherwise, log will always be base 2.

O Notation: Function $f(x)$ is $O(g(x))$ if there exists positive constants c and x_0 s.t. $0 \leq f(x) \leq cg(x)$ for $x \geq x_0$.

Θ Notation

Problem with $\mathbf{O}$ Notation is more than one function satisfies. ie. Alg A1 is also $O(x^3)$. Choose function that grows the slowest.

Often, our function is within a constant value of another function. In this case, we can use the Θ notation.

Θ Notation: Function $f(x)$ is $\Theta(g(x))$ if there exists positive constants c_1, c_2, and x_0 s.t. $c_1 g(x) \leq f(x) \leq c_2 g(x)$ for $x \geq x_0$.

If $f(x)$ is $\Theta(g(x))$ then $f(x)$ is also $O(g(x))$.

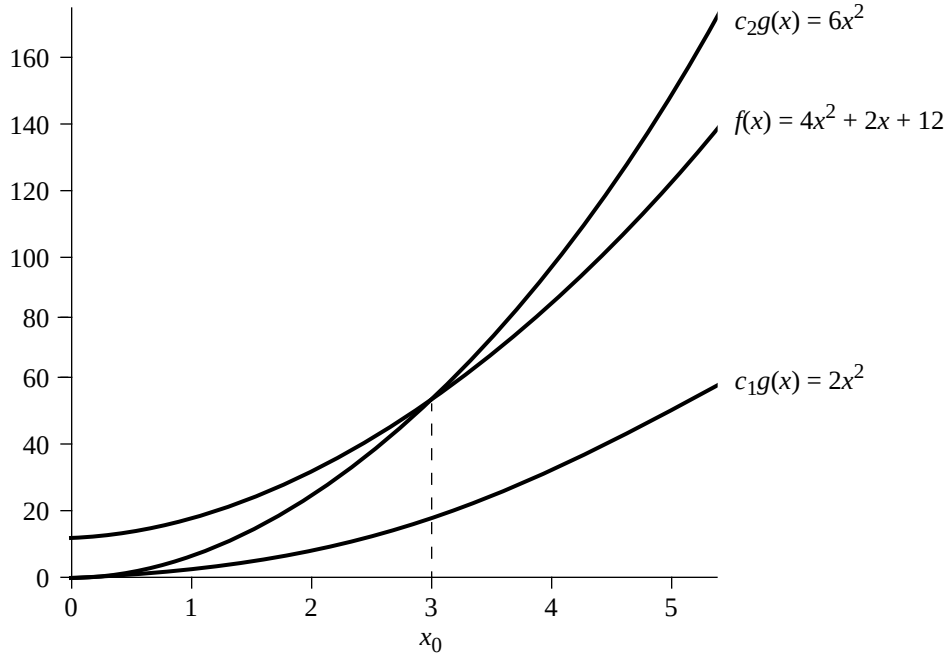


Figure 2.18 Growth of function $f(x) = 4x^2 + 2x + 12$.

Ω Notation

Ω notation is used to describe a lower bound on growth.

Ω Notation: Function $f(x)$ is $\Omega(g(x))$ if there exists positive constants c and x_0 s.t. $0 \leq cg(x) \leq f(x)$ for $x \geq x_0$.

From Figure 2.18, we have that $f(x) = 4x^2 + 2x + 12$ is $\Omega(x^2)$.

Can interpret $O()$ as "grows at most as fast as," and $\Omega()$ as "grows at least as fast as."

Finally, a function $f(x)$ is $\Theta(g(x))$ if and only if $f(x)$ is $O(g(x))$ and $\Omega(g(x))$.

Time Complexity of Parallel Algorithms

Analyzing $t_{\text{comm}} = t_{\text{startup}} + n\, t_{\text{data}}$ for time complexity, we see that is is $O(n)$.

For t_p, the time complexity will be sum of that for t_{comp} and t_{comm}.

Time Complexity eg.

Want to add n numbers on two computers (comp1 and comp2). Each adds $n/2$ numbers, and originally only comp1 has all n numbers. Comp2 sends its resul to comp1 which then adds the two partial sums.

This problem has four main steps:

1. Comp1 sends $n/2$ to comp2.

2. Both add $n/2$ in parallel

3. Comp2 sends its partial sum to comp1

4. comp1 adds the partial sums to produce final result.

Time Complexity eg. cont.

Computation steps 2 and 4:

$$t_{\text{comp}} = (\frac{n}{2} - 1) + 1 = \frac{n}{2} \qquad t_{\text{comp}} \text{ is } O(n)$$

Communication steps 1 and 3:

$$\begin{aligned} t_{\text{comm}} &= (t_{\text{startup}} + \frac{n}{2} t_{\text{data}}) + (t_{\text{startup}} + t_{\text{data}}) \\ &= 2t_{\text{startup}} + (\frac{n}{2} + 1) t_{\text{data}} \end{aligned}$$

t_{comm} is $O(n)$

$$\begin{aligned} t_p &= 2t_{\text{startup}} + (\frac{n}{2} + 1) t_{\text{data}} + \frac{n}{2} \\ &= 2t_{\text{startup}} + t_{\text{data}} + n(\frac{t_{\text{data}}}{2} + \frac{1}{2}) \end{aligned}$$

t_{p} is $O(n)$

Computation/Communication Ratio

Communication is generally costly.

If t_{comp} and t_{comm} have same time complexity, then increasing problem size probably won't improve performance.

Want complexity of t_{comp} to be greater.

For example if t_{comp} is $O(n^2)$ and t_{comm} is $O(n)$, we should be able to find a n so that t_{comp} will dominate.

Cost Optimal Algorithms

A *cost optimal* algorithm is when the cost of solving a problem using a parallel algorithm is proportional to execution time on one processor. For n processors and constant k:

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

For time complexity analysis, we can say that our parallel algorithm is cost-optimal if:

parallel time complexity $\times$ # of processors = sequential time complexity

Comments on Asymptotic Analysis

Time complexity widely used to analyze sequential programs and for theoretical analysis of parallel programs.

Not as useful to determine possible performance of parallel programs.

Notations rely on asymptotic behavior. Letting some variable (perhaps # data elements or processors) tend to infinity may not be relevant.

Number processors available constrained by cost and technology.

Also, want manageable data sizes.

Comments on Asymptotic Analysis Cont.

Analysis also ignores lower order terms that may be significant for real values of the variable.

An example is $t_{\text{comm}} = t_{\text{startup}} + n\, t_{\text{data}}$. It is $O(n)$. Ignores fact that for reasonable values of n, term t_{startup} will dominate t_{comm}.

Also ignores factors such as network contention.

Shared Memory Programs

So far, we've concentrated on message-passing programs.

For shared memory programs, communication time is zero, so time complexity is that of the computation.

This makes time complexity analysis more applicable.

However, in shared memory programs, ensuring orderly access to shared data adds additional delays.

Time Complexity of Broadcast on Hypercube Network

Most problems require data to be broadcast and then gathered.

Many software environments provide this functionality, but actual algorithm used is a function of actual architecture to system.

For a d-dimensional hypercube, we have $n = 2^d$ nodes.

For $d = 3$, to broadcast from node $r = 000$ to all other nodes, we can use the following efficient algorithm.

1. Message is sent to node whose address differs in the least significant bit (right-most bit: bit 0) from r.

2. The two nodes with message send message to nodes that differ from their address by bit 1.

3. The four nodes with message send message to nodes whose address differs in the most significant bit.

See Figure 2.19.

Message can be broadcast to all n nodes in $\log n$ steps. Complexity thus $O(\log n)$.

Optimal as diameter of hypercube is $\log n = d$.

Can express algorithm as a tree. See Figure 2.20.

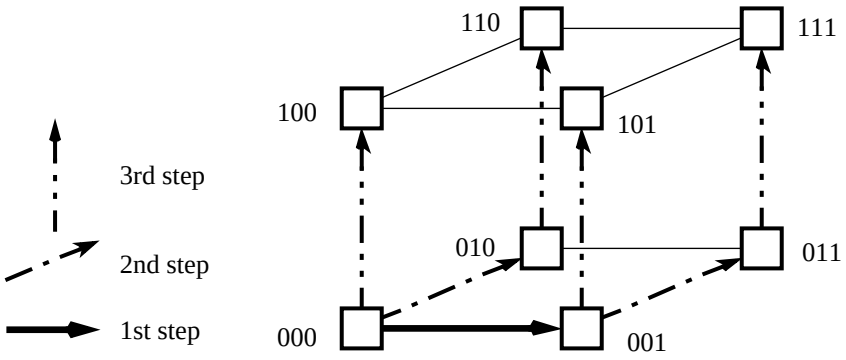


Figure 2.19 Broadcast in a three-dimensional hypercube.

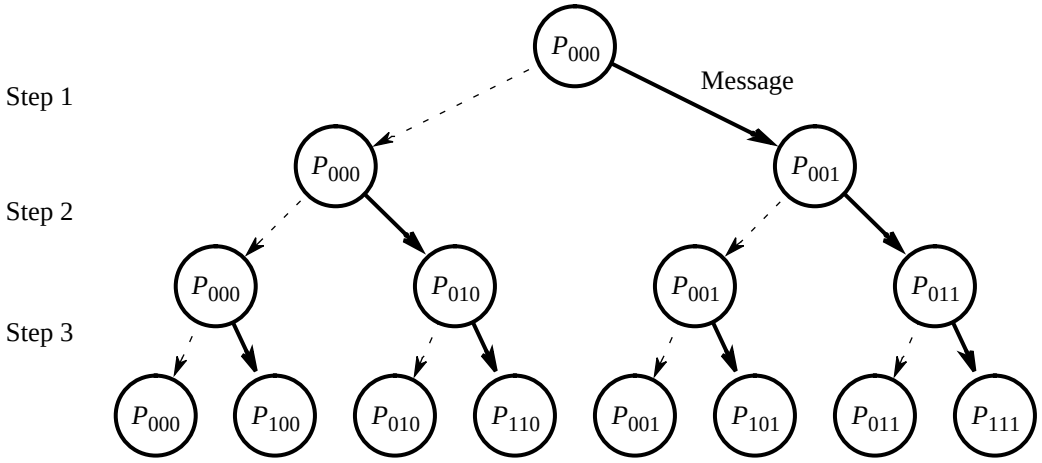


Figure 2.20 Broadcast as a tree construction.

Gather on Hypercube Network

Can use reverse of broadcast algorithm to gather data to root node, say node 000.

```
    Node          Node
1)  100     -->   000
    101     -->   001
    110     -->   010
    111     -->   011

2)  010     -->   000
    011     -->   001

3)  001     -->   000
```

Because message length increases at each step, complexity more than $O(\log n)$.

Broadcast on Mesh Network

Broadcast can be performed (ie. from top-left corner) by sending message across top row, and then down the column once received.

See Figure 2.21.

On an $n \times n$ mesh, algorithm takes $2(n-1)$ steps and is thus $O(n)$.

Optimal as diameter is $2(n-1)$.

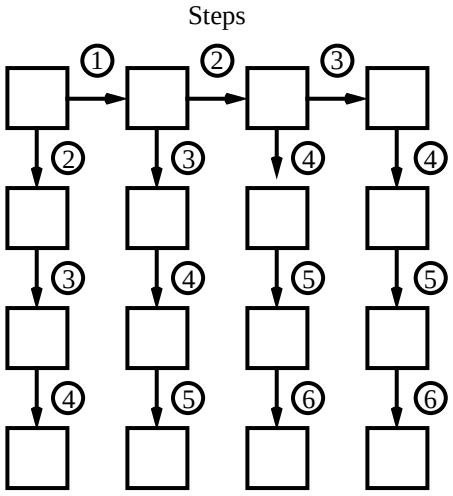


Figure 2.21 Broadcast in a mesh.

Broadcast on Workstation Cluster

In text, we will concentrate on communication with workstation clusters.

Broadcast on single ethernet connection done by sending one message that is read by all destinations on network at once. See Figure 2.22.

Time complexity is $O(1)$ for 1 data element to N computers. For n data elements is $O(n)$.

Unfortunately, not all computers have same network structure, so message-passing libraries can't rely on this.

PVM uses a 1-to-N fan-out broadcast (N is number of computers). MPI probably uses a similar method, but this is implementation specific. The same message is sent to each of the destination in turn. See Figure 2.23.

Time complexity is $O(N)$.

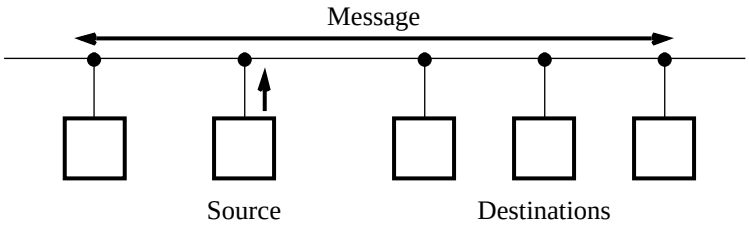


Figure 2.22 Broadcast on an Ethernet network.

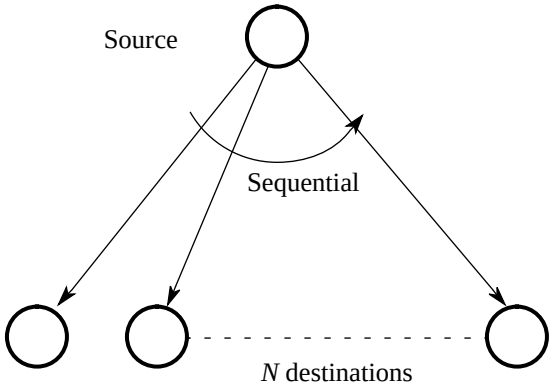


Figure 2.23 1-to-*N* fan-out broadcast.

Broadcast on Tree

Figure 2.24 shows 1-to-N fan-out broadcast on a complete tree structure.

Complexity depends on the number of nodes at each level and the number of levels.

For binary tree, our fan out for each node is $N = 2$.

If there are p final destinations, we have $\log p$ levels.

We thus have:

$$t_{\text{comm}} = 2(\log p)(t_{\text{startup}} + w\, t_{\text{data}})$$

Where we are sending w words of data.

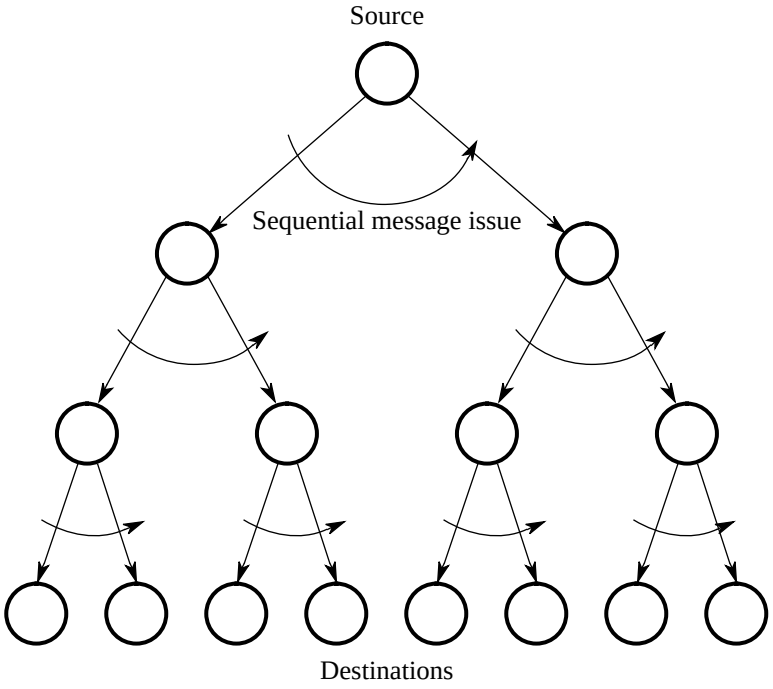


Figure 2.24 1-to-*N* fan-out broadcast on a tree structure.

Low-Level Debugging

First step: Debug program to get to work correctly

2nd: Evaluate to see how fast it runs.

3rd: Try to make it even faster!

Often hard to get a parallel program to work correctly.

Helpful to start by getting a sequential version of parallel algorithm working.

Common practice to debug sequential code is to *instrument* it. Instrumenting code means adding code to output intermediate values.

Can use similar approach with parallel code, but has important consequences.

Low-Level Debugging Cont.

In a sequential program, this will slow it down but it will still perform deterministically and give you the correct answer.

Not necessarily the case for parallel program.

Heisenberg (creater of same "uncertainty principle"): "We have to remember that what we observe is not nature itself, but nature exposed to our method of questioning."

Instrumenting a parallel program will also slow it down, but the behavior (and thus final answer) of the program might also change.

Could change the interleaved order instructions are executed in since each processor would normally be affected differently by the instrumenting code.

More Low-Level Debugging

Processes are usually running on multiple computers.

All output appears in starting terminal, but interleaved, and not necessarily in proper time ordering. *beware* Label output by rank!

Lowest level of debugging is to use a sequential debugger like gdb. Can help find error in sequential logic, but not much use for multiple processes.

These debuggers can be used to examine registers and set break-points, but can't give information about order instructions executed in concurrent processes.

More Low-Level Debugging Cont.

Sequential debuggers can't capture timing of events. Can't recognize events such as "message sent."

Parallel debuggers do exist. See MPI web site (tools(not free)) or http://www.lam-mpi.org/software/xmpi/ (XMPI for LAM version of MPI).

Visualization Tools

Parallel programs are naturals for visualization of their actions.

Version of MPI on suns comes with visualization tool called upshot (see: http://www-fp.mcs.anl.gov/~lusk/upshot/).

Program execution can be watched in a *space-time diagram.* Also called a *process-time diagram*. See Figure 2.25.

Wait period represent process idle. Visualizations can help spot incorrect behavior.

Events can be captured so can be displayed later.

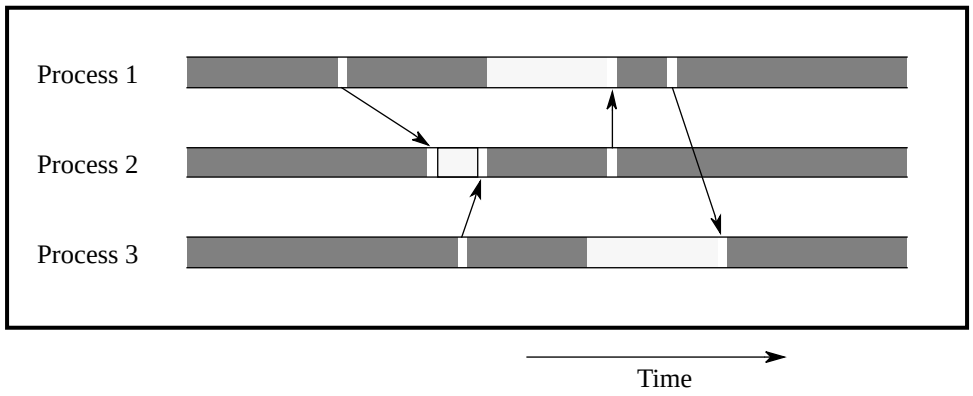


Figure 2.25 Space-time diagram of a parallel program.

Utilization-Time Diagram

A *utilization-time diagram* shows time each process spends on waiting, communication and message-passing library routines.

Helps with debugging as well as gives information on efficiency of computation.

Also, animation of processes can be used.

Each process shown, and current state info.

State changes over time animated.

Debugging Strategies For message-Passing Programs

1. If you can, run program with 1 process and debug as sequential program

2. Run program as 2-4 processes on same processor. Examine tasks such as messages being sent to correct place. Common error to screw up message tags (etc.) resulting in message going to wrong place.

3. Run same 2-4 processes using several computers. Helps to find problems due to network delay with respect to timing and synchronization.

Measuring Execution Time Empirically

Can evaluate time comlexity of algorithm, but doesn't mean a program will run fast on your computer.

Only way to know performance for sure is to code program, run it, and measure execution time!

Can instrument code to measure execution time:

See C library function: man gettimeofday

Returns elapsed time in seconds and microseconds since 00:00 Universal Coordinated Time.

Measuring Time Empirically Cont.

Can also use the *time()* system call, but only returns time in seconds. Can use *difftime()* routine with results from *time()* to automatically calculate difference.

Call once and record. Call later, and elapsed time is difference.

Warning: clocks across computers are not syncronized.

Elapsed time includes time waiting for messages and assumes processor not running other programs concurrently.

Measuring Communication Time

The *ping-pong method* can be used to measure point-point communication between two processes.

1. First process, P_0, records current time, t_1.

2. Process P_0 sends message to second process, P_1.

3. As soon as process P_1 receives message, it immediately sends it back to process P_0.

4. Process P_0 receives message and records time, t_2.

5. Communication time is $(t_2 - t_1)/2$

Profiling

A histogram or graph showing the amount of time a program spends in different parts of the program is called a *profile.* See Figure 2.26.

The *profiler* extracting the timing information from running program will alter runtime of program.

Program usually probed or sampled at intervals to produce statistical data.

Used to identify areas of the program that are visited often and/or the program spends a large percentage of its execution time in.

These are good candidates for optimization (large percentage) and/or procedure inlining (called often but small).

Primarily useful for examining code sequentially.

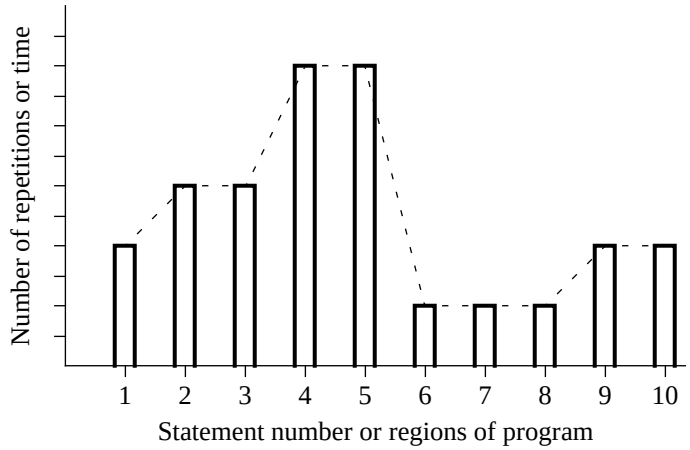


Figure 2.26 Program profile.

Profiling with gprof.

A commonly used UNIX profiler is *gprof.*

Gprof not only provides statistics about the time the program spends in each function, but provides a *call graph.*

A call graph provides information about which function calls which, as well as how much the called function contributes to the runtime of the calling function.

To use gprof, you must compile the program, say loops.c, with the "-pg" flag so that the profiling routines are linked in.

```
gcc loops.c -pg -o loops
```

When "loops" is executed, it will store the profiling data in binary form in a file called gmon.out. To view in a human-readable format, use gprof.

```
gprof loops >loops.gprof
```

loops.c

```
main() {
  int i;

  for (i=0;i<30000;i++) {
    if (i == 2*(i/2))
      foo();
    bar();
    baz();   }
}

foo() {
  int j,junk1;

  for (j = 0; j <500;j++)
     junk1 = 3456;
}

bar() {
  int k,junk2;

  for (k = 0; k <500;k++)
     junk2 = 3456;
}

baz() {
  int l,junk3;

  for (l = 0; l <900;l++)
     junk3 = 3456;
}
```

Output of gprof for loops.c

```
Flat profile:  Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ns/call  ns/call  name
 41.67      0.10     0.10    30000  3333.33  3333.33  baz
 33.33      0.18     0.08    30000  2666.67  2666.67  bar
 25.00      0.24     0.06    15000  4000.00  4000.00  foo

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.

calls      the number of times this function was invoked.

 self      the average number of milliseconds spent in this
ms/call    function per call.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call.

name       the name of the function.
```

Call Graph for loops.c

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.24                 main [1]
                0.10    0.00   30000/30000       baz [2]
                0.08    0.00   30000/30000       bar [3]
                0.06    0.00   15000/15000       foo [4]
-----------------------------------------------
                0.10    0.00   30000/30000       main [1]
[2]     41.7    0.10    0.00   30000         baz [2]
-----------------------------------------------
                0.08    0.00   30000/30000       main [1]
[3]     33.3    0.08    0.00   30000         bar [3]
-----------------------------------------------
                0.06    0.00   15000/15000       main [1]
[4]     25.0    0.06    0.00   15000         foo [4]
-----------------------------------------------
index     A unique number given to each element of the
           table. Index numbers are sorted numerically.

% time    This is the percentage of the ‘total’ time that
  was spent in this function and its children.

self   Total amount of time spent in function.

children  This is the total amount of time propagated into thi
  this function by its children.

called    This is the number of times the function was called.

name   The name of the current function.
```