Chapter 1

Parallel Computers

Demand for Computational Speed

Continual demand for greater computational speed from a computer system than is currently possible

Areas requiring great computational speed include numerical modeling and simulation of scientific and engineering problems.

Computations must be completed within a "reasonable" time period.

Slide 15

Grand Challenge Problems

A grand challenge problem is one that cannot be solved in a reasonable amount of time with today's computers.

Obviously, an execution time of 10 years is always unreasonable.

Examples

- Modeling large DNA structures
- Global weather forecasting
- Modeling motion of astronomical bodies.

Slide 16

Weather Forecasting

Atmosphere modeled by dividing it into 3-dimensional cells. Calculations of each cell repeated many times to model passage of time.

Example

Whole global atmosphere divided into cells of size 1 mile $\times$ 1 mile $\times$ 1 mile to a height of 10 miles (10 cells high) - about 5×10^8 cells. Suppose each calculation requires 200 floating point operations. In one time step, 10^{11} floating point operations necessary.

Weather Forecasting

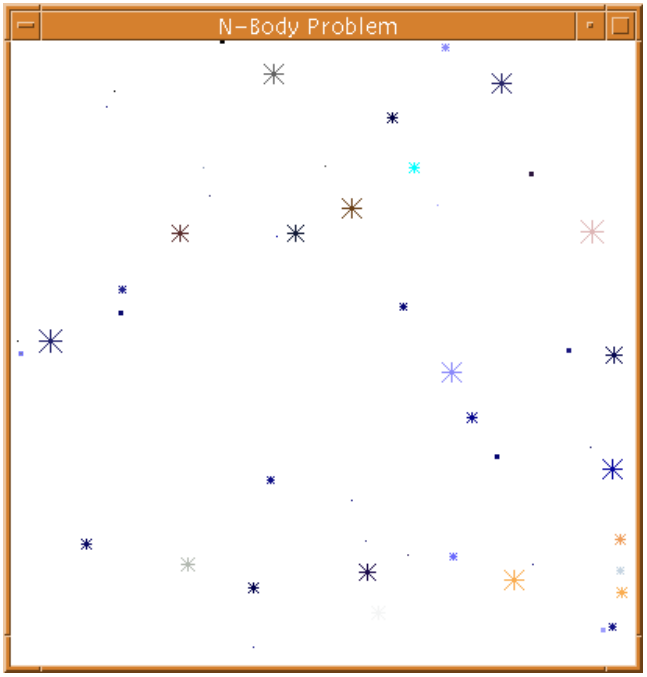
To forecast the weather over 10 days using 10-minute intervals, a computer operating at 100 Mflops (10^8 floating point operations/s) would take 10^7 seconds or over 100 days.

To perform the calculation in 10 minutes would require a computer operating at 1.7 Tflops (1.7×10^{12} floating point operations/sec).

Modeling Motion of Astronomical Bodies

Each body attracted to each other body by gravitational forces. Movement of each body predicted by calculating total force on each body. With N bodies, $N - 1$ forces to calculate for each body, or approx. N^2 calculations. ($N \log_2 N$ for an efficient approx. algorithm.) After determining new positions of bodies, calculations repeated.

A galaxy might have, say, 10^{11} stars. Even if each calculation could be done in 1 μs (an extremely optimistic figure), it would take 10^9 years for one iteration using the N^2 algorithm and almost a year for one iteration using an efficient $N \log_2 N$ approximate algorithm.



Astrophysical *N*-body simulation by Scott Linssen (undergraduate University of North Carolina at Charlotte [UNCC] student).

Parallel Computing

Using more than one computer, or a computer with more than one processor, to solve a problem.

Motives

Usually faster computation - very simple idea - that *n* computers operating simultaneously can achieve the result *n* times faster - it will not be *n* times faster for various reasons.

Other motives include: fault tolerance, larger amount of memory available, ...

Slide 21

Background

Parallel computers - computers with more than one processor - and their programming - parallel programming - has been around for more than 40 years.

Slide 22

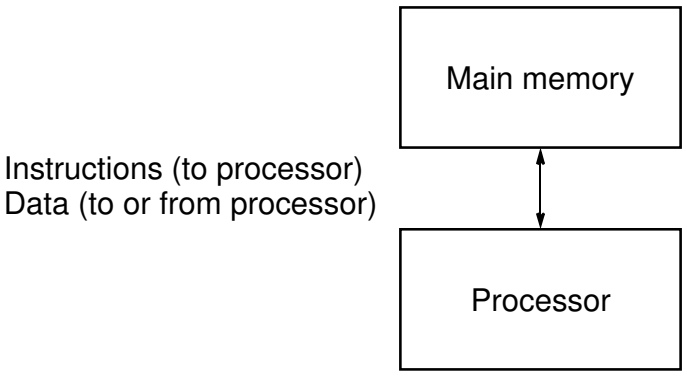
Gill writes in 1958:

"... There is therefore nothing new in the idea of parallel programming, but its application to computers. The author cannot believe that there will be any insuperable difficulty in extending it to computers. It is not to be expected that the necessary programming techniques will be worked out overnight. Much experimenting remains to be done. After all, the techniques that are commonly used in programming today were only won at the cost of considerable toil several years ago. In fact the advent of parallel programming may do something to revive the pioneering spirit in programming which seems at the present to be degenerating into a rather dull and routine occupation ..."

Gill, S. (1958), "Parallel Programming," *The Computer Journal*, vol. 1, April, pp. 2-10.

Conventional Computer

Consists of a processor executing a program stored in a (main) memory:



Each main memory location located by its *address*. Addresses start at 0 and extend to $2^n - 1$ when there are n bits (binary digits) in the address.

Types of Parallel Computers

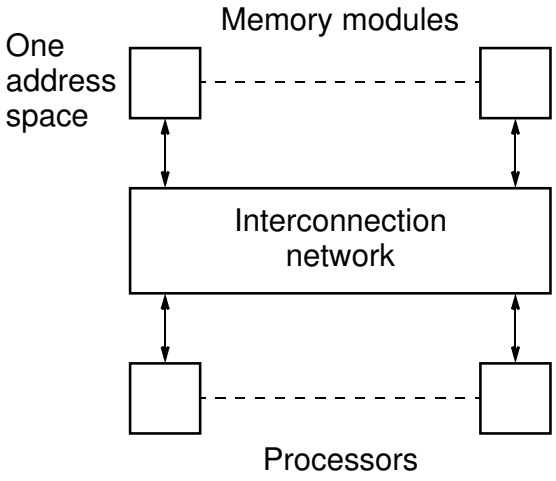
Two principal types:

- Shared memory multiprocessor
- Distributed memory multicomputer

Slide 25

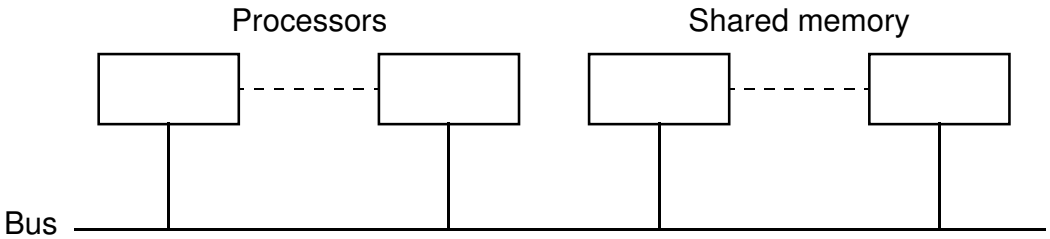
Shared Memory Multiprocessor System

Natural way to extend single processor model - have multiple processors connected to multiple memory modules, such that each processor can access any memory module - so-called *shared memory* configuration:



Slide 26

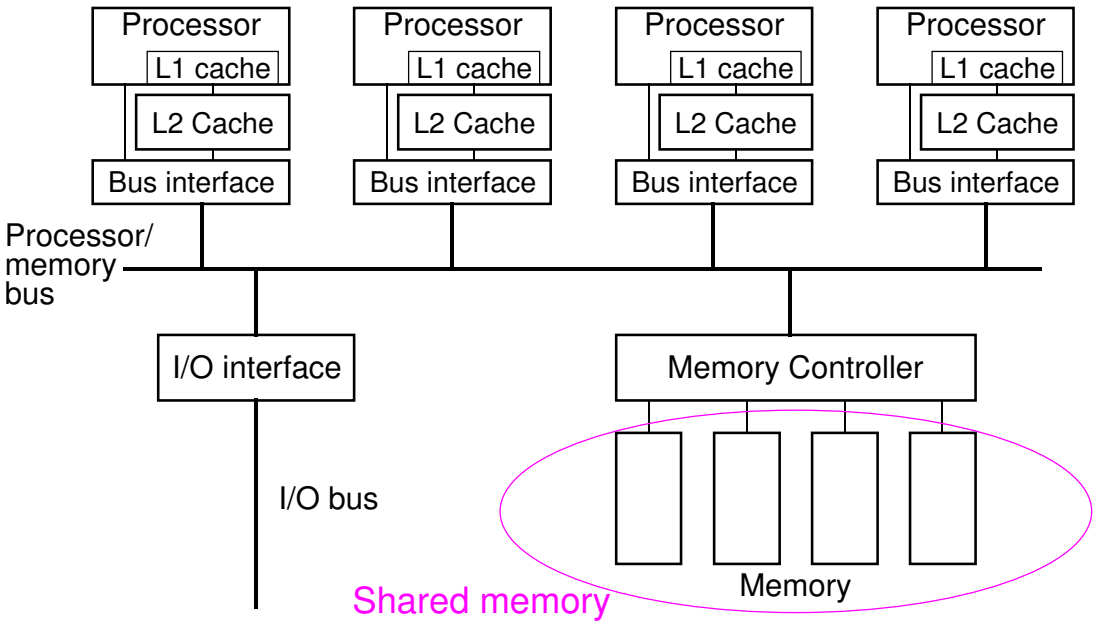
Simplistic view of a small shared memory multiprocessor



Examples:

- Dual Pentiums
- Quad Pentiums

Quad Pentium Shared Memory Multiprocessor



Shared memory multiprocessor system

Any memory location can be accessible by any of the processors.

A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.

Generally, shared memory programming more convenient although it does require access to shared data to be controlled by the programmer (using critical sections etc.)

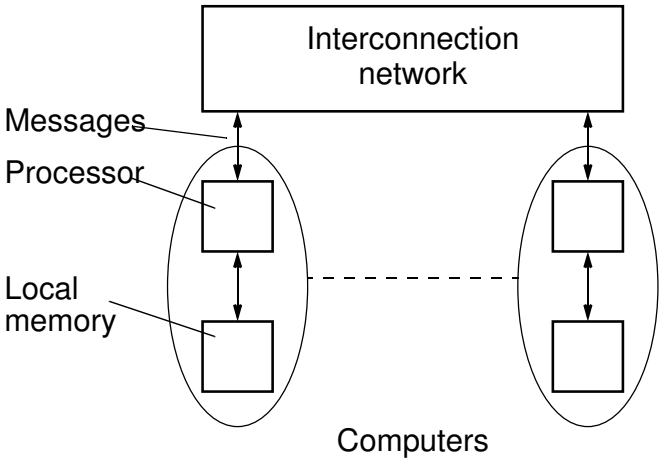
Several Alternatives for Programming Shared Memory Multiprocessors:

Using:

- Threads (Pthreads, Java, ..) in which the programmer decomposes the program into individual parallel sequences, each being thread, and each being able to access variables declared outside the threads.
- A sequential programming language with preprocessor compiler directives to declare shared variables and specify parallelism. Example OpenMP - industry standard
- A sequential programming language with user-level libraries to declare and access shared variables.
- A parallel programming language with syntax for parallelism, in which the compiler creates the appropriate executable code for each processor (not now common)
- A sequential programming language and ask a parallelizing compiler to convert it into parallel executable code. - also not now common

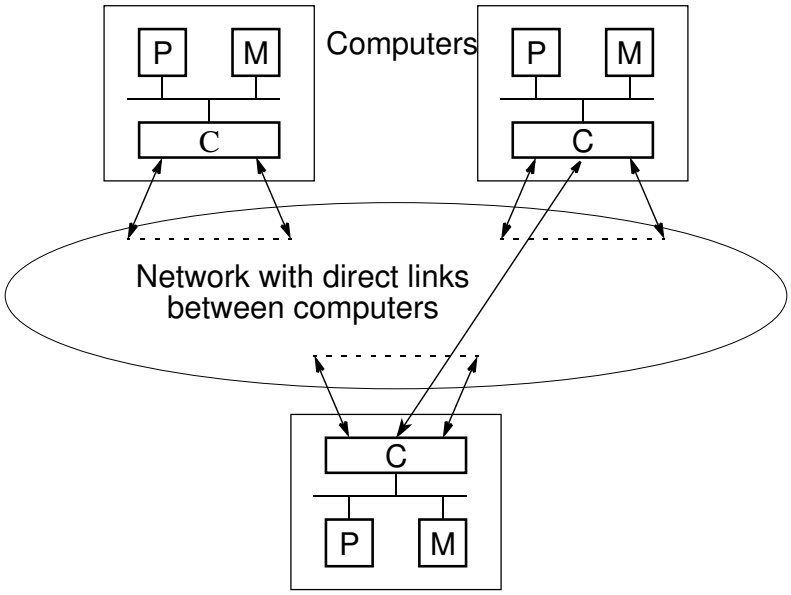
Message-Passing Multicomputer

Complete computers connected through an interconnection network:



Static Network Message-Passing Multicomputers

Computers connected by direct links:



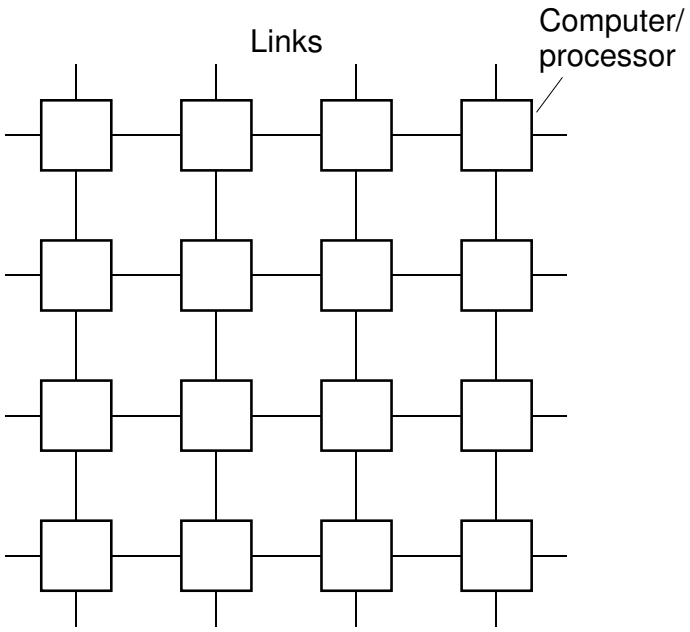
Static Link Interconnection Networks

Various:

- Ring
- Tree
- 2-D and 3-D arrays
- Hypercube

Slide 33

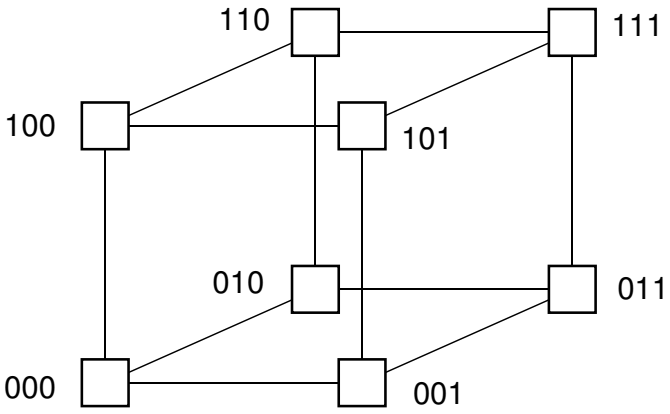
Two-dimensional array (mesh)



Also three-dimensional - used in some large high performance systems.

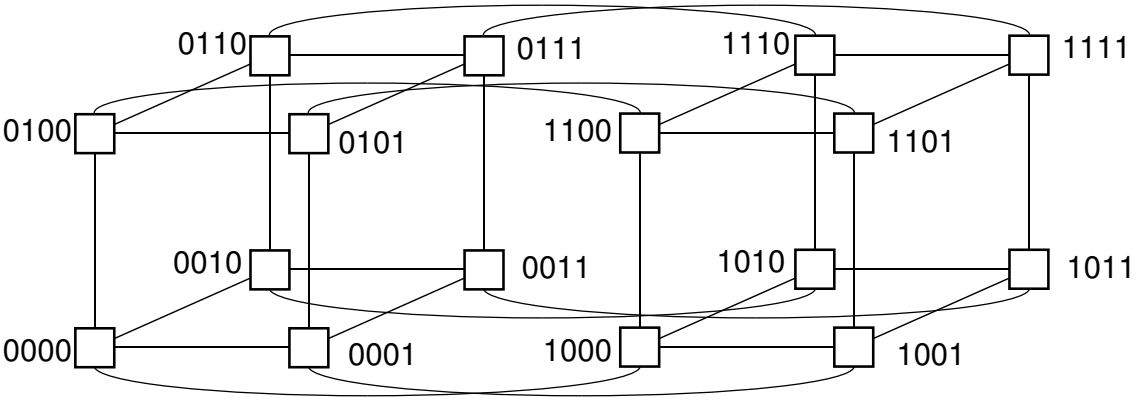
Slide 34

Three-dimensional hypercube



Slide 35

Four-dimensional hypercube



Hypercubes popular in 1980's - not now

Slide 36

Networked Computers as a Multicomputer Platform

A *network of workstations* (NOWs) became a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing in early 1990's.

Several Projects

- Berkely NOW project

Key advantages:

- Very high performance workstations and PCs readily available at low cost.
- The latest processors can easily be incorporated into the system as they become available.
- Existing software can be used or modified.

Beowulf Clusters*

A group of interconnected "commodity" computers achieving high performance with low cost.

Typically using commodity interconnects - high speed Ethernet, and Linux OS.

* Beowulf comes from name given by NASA Goddard Space Flight Center cluster project.

Slide 39

Cluster Interconnects

- Originally fast Ethernet on low cost clusters
- Gigabit Ethernet - easy upgrade path

Using Ethernet switches to connect computers

More Specialized/Higher Performance

- Myrinet - 2.4 Gbits/sec - disadvantage: single vendor
- cLan
- SCI (Scalable Coherent Interface)
- QsNet
- Infiniband - may be important as infininbnand interfaces may be intergrated on next generation PCs

See Beowulf reference book for more details.

Slide 40

Message Passing Parallel Programming Software Tools for Clusters

Parallel Virtual Machine (PVM) - developed in late 1980's. Became very popular.

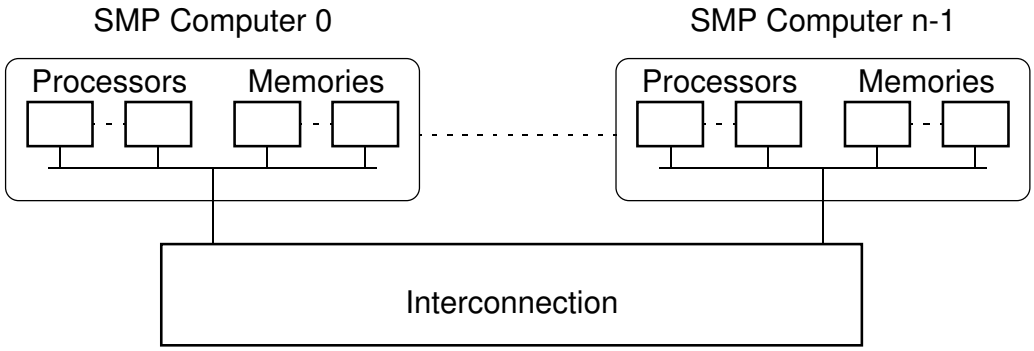
Message-Passing Interface (MPI) - standard defined in 1990s.

Both provide a set of user-level libraries for message passing. Use with regular programming languages (C, C++, ...).

Slide 41

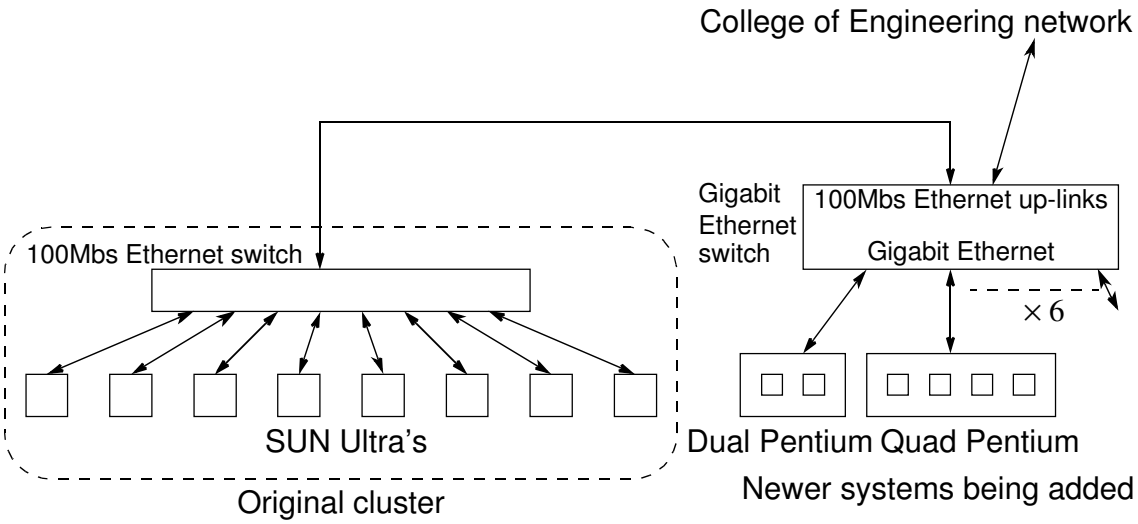
SMP Cluster

Can have a cluster of shared memory computers (symmetrical multiprocessors)



Slide 42

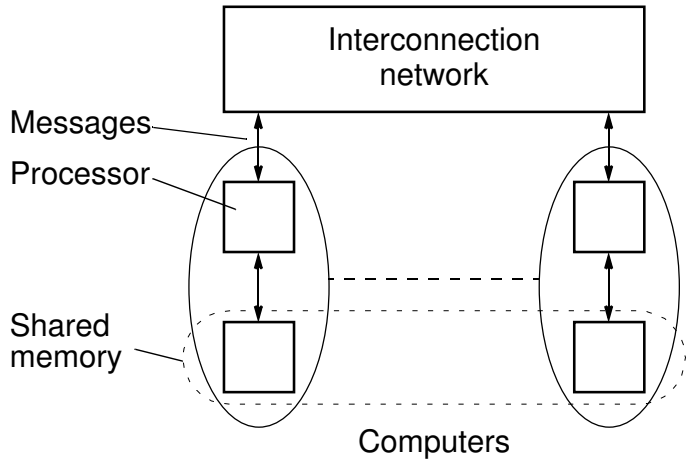
UNCC Department of Computer Science Cluster



Distributed Shared Memory

Making the main memory of a cluster of computers look as though it is a single memory with a single address space.

Then can use shared memory programming techniques.



Flynn's Classifications

Flynn (1966) created a classification for computers based upon instruction streams and data streams:

Single instruction stream-single data stream (SISD) computer

In a single processor computer, a single stream of instructions is generated from the program. The instructions operate upon a single stream of data items. Flynn called this single processor computer a s*ingle instruction stream-single data stream* (SISD) computer.

Slide 45

Multiple Instruction Stream-Multiple Data Stream (MIMD) Computer

General-purpose multiprocessor system - each processor has a separate program and one instruction stream is generated from each program for each processor. Each instruction operates upon different data.

Both the shared memory and the message-passing multiprocessors so far described are in the MIMD classification.

Slide 46

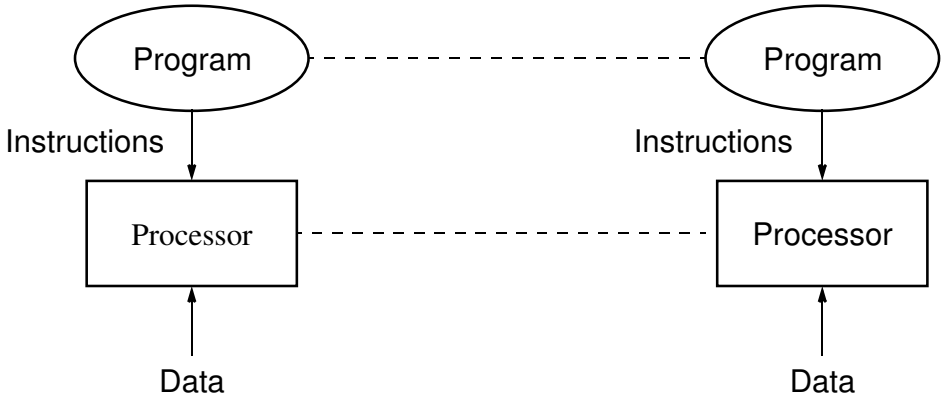
Single Instruction Stream-Multiple Data Stream (SIMD) Computer

A specially designed computer in which a single instruction stream is from a single program, but multiple data streams exist. The instructions from the program are broadcast to more than one processor. Each processor executes the same instruction in synchronism, but using different data.

Developed because there are a number of important applications that mostly operate upon arrays of data.

Multiple Program Multiple Data (MPMD) Structure

Within the MIMD classification, which we are concerned with, each processor will have its own program to execute:



Single Program Multiple Data (SPMD) Structure

Single source program is written and each processor will execute its personal copy of this program, although independently and not in synchronism.

The source program can be constructed so that parts of the program are executed by certain computers and not others depending upon the identity of the computer.

Speedup Factor

$$S(n) = \frac{\text{Execution time using one processor (single processor system)}}{\text{Execution time using a multiprocessor with n processors}} = \frac{t_s}{t_p}$$

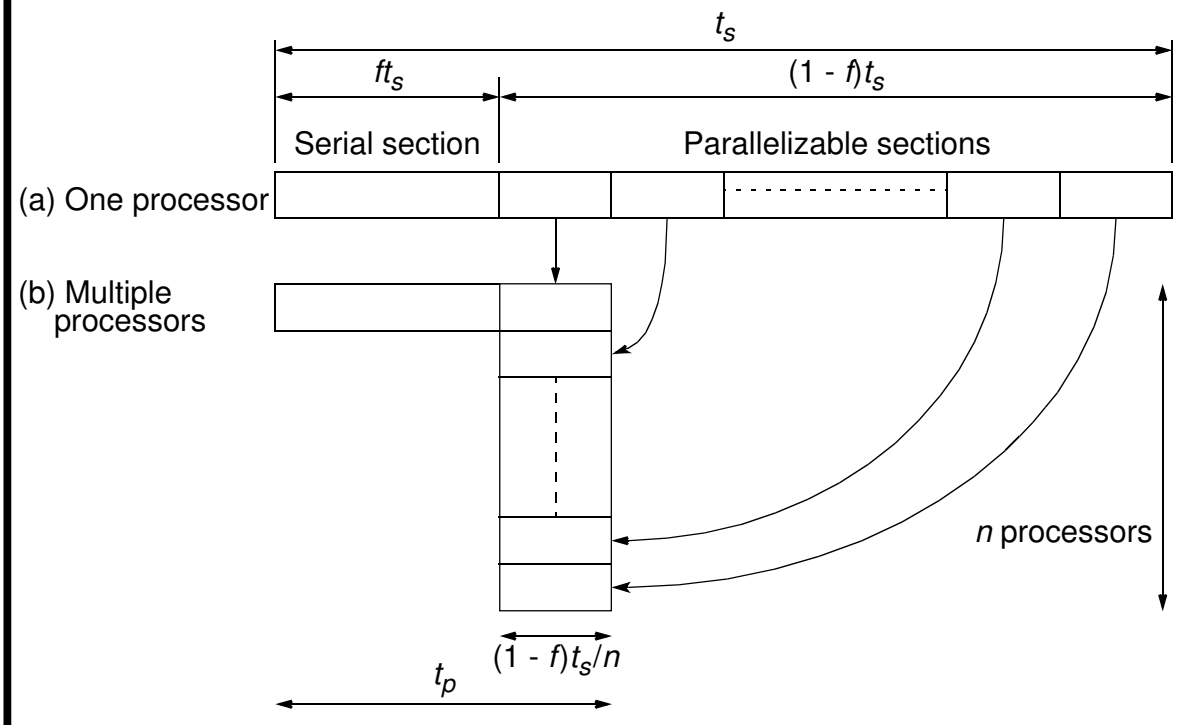
where t_s is execution time on a single processor and t_p is execution time on a multiprocessor. $S(n)$ gives increase in speed by using multiprocessor. Underlying algorithm for parallel implementation might be (and is usually) different.

Speedup factor can also be cast in terms of computational steps:

$$S(n) = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with n processors}}$$

Maximum speedup is (usually) n with n processors (*linear speedup*).

Maximum Speedup - Amdahl's law

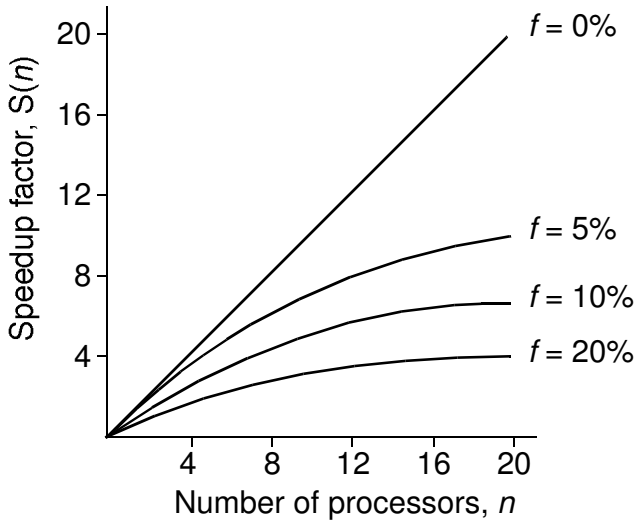


Speedup factor is given by:

$$S(n) = \frac{t_s}{ft_s + (1-f)t_s/n} = \frac{n}{1+(n-1)f}$$

This equation is known as *Amdahl's law*

Speedup against number of processors



Even with infinite number of processors, maximum speedup limited to 1/*f*.

Example: With only 5% of computation being serial, maximum speedup is 20, irrespective of number of processors.

Chapter 2

Message-Passing Computing

Basics of Message-Passing Programming using user-level message passing libraries

Two primary mechanisms needed:

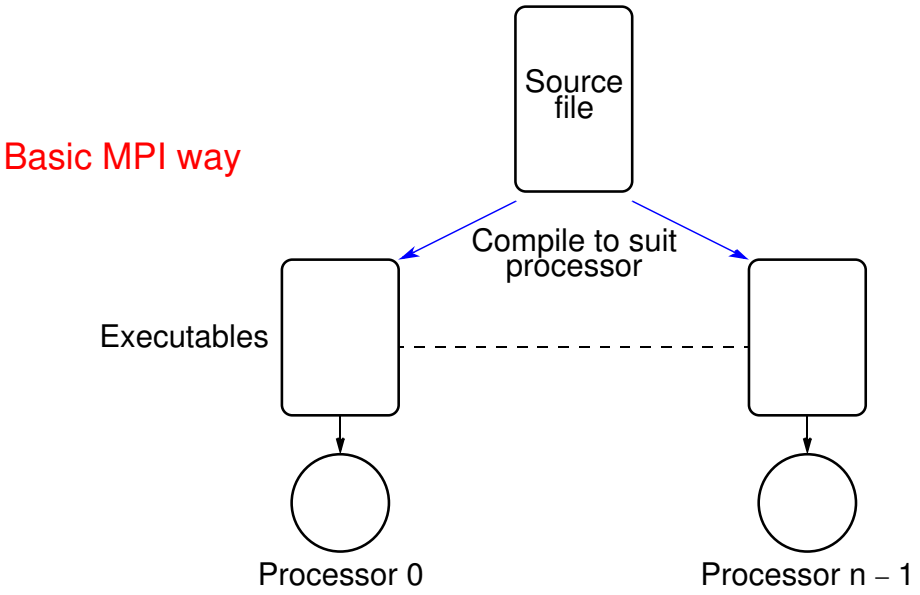
1. A method of creating separate processes for execution on different computers

2. A method of sending and receiving messages

Slide 55

Single Program Multiple Data (SPMD) model

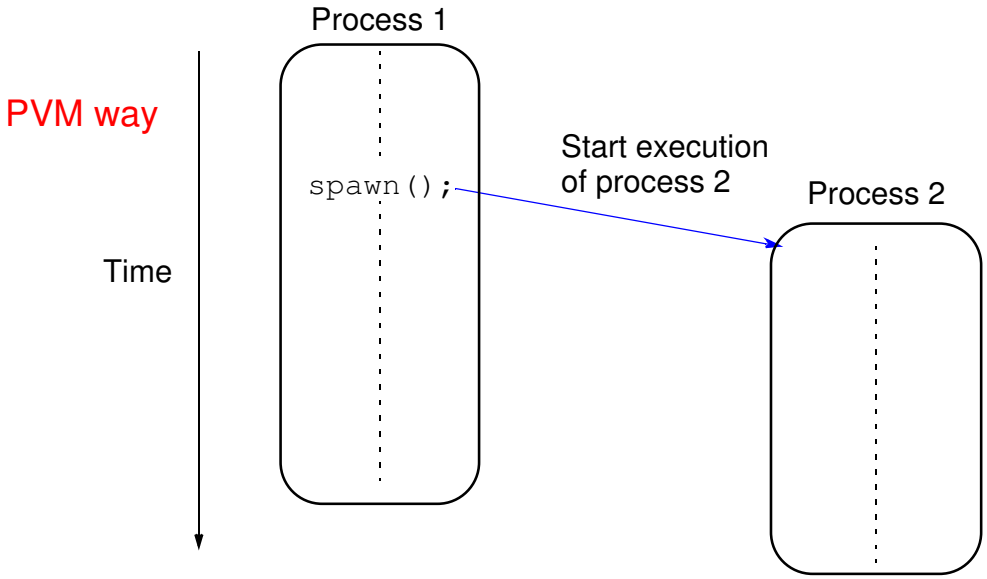
Different processes merged into one program. Within program, control statements select different parts for each processor to execute. All executables started together - static process creation.



Slide 56

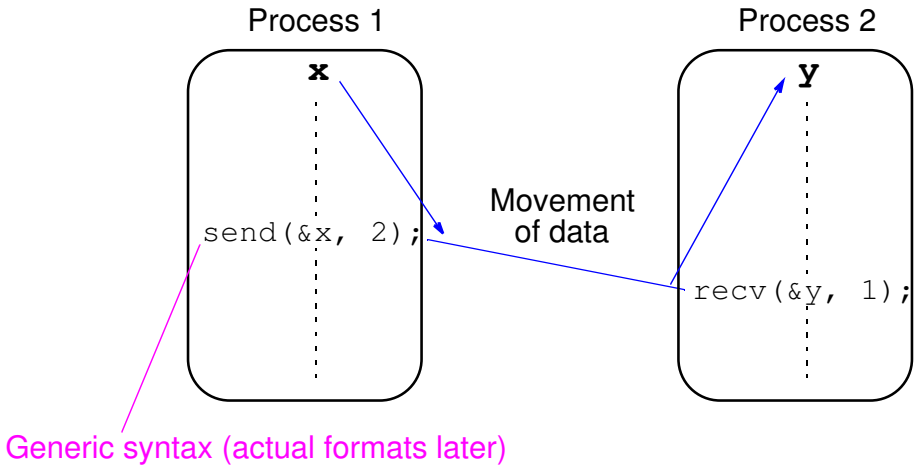
Multiple Program Multiple Data (MPMD) Model

Separate programs for each processor. Master-slave approach usually taken. One processor executes master process. Other processes started from within master process - dynamic process creation.



Basic "point-to-point" Send and Receive Routines

Passing a message between processes using `send()` and `recv()` library calls:



Synchronous Message Passing

Routines that actually return when message transfer completed.

Synchronous send routine

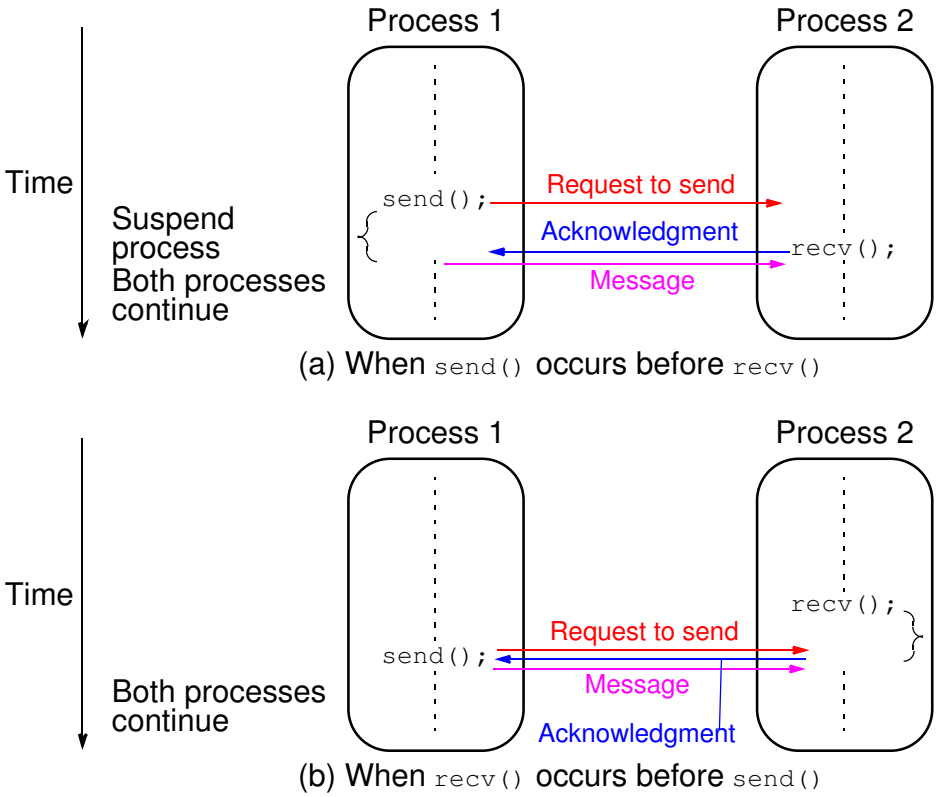
Waits until complete message can be accepted by the receiving process before sending the message.

Synchronous receive routine

Waits until the message it is expecting arrives.

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

Synchronous `send()` and `recv()` library calls using 3-way protocol



Asynchronous Message Passing

Routines that do not wait for actions to complete before returning. Usually require local storage for messages.

More than one version depending upon the actual semantics for returning.

In general, they do not synchronize processes but allow processes to move forward sooner. Must be used with care.

Slide 61

MPI Definitions of Blocking and Non-Blocking

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

Non-blocking - return immediately.

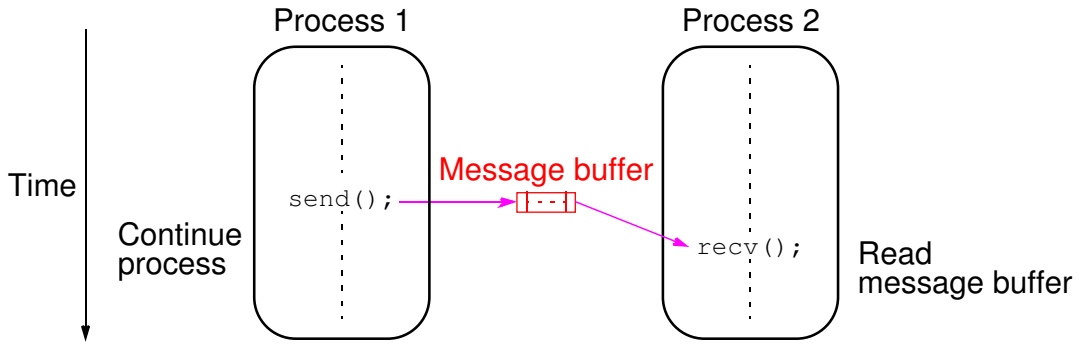
Assumes that data storage to be used for transfer not modified by subsequent statements prior to tbeing used for transfer, and it is left to the programmer to ensure this.

Notices these terms may have different interpretations in other systems.)

Slide 62

How message-passing routines can return before message transfer completed

Message buffer needed between source and destination to hold message:



Slide 63

Asynchronous (blocking) routines changing to synchronous routines

Once local actions completed and message is safely on its way, sending process can continue with subsequent work.

Buffers only of finite length and a point could be reached when send routine held up because all available buffer space exhausted.

Then, send routine will wait until storage becomes re-available - *i.e then routine behaves as a synchronous routine*.

Slide 64

Message Tag

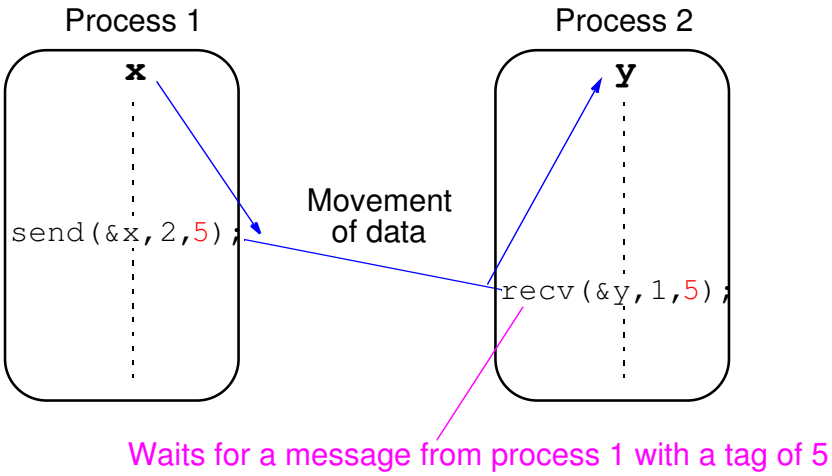
Used to differentiate between different types of messages being sent.

Message tag is carried within message.

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

Message Tag Example

To send a message, **x**, with message tag 5 from a source process, 1, to a destination process, 2, and assign to **y**:



"Group" message passing routines

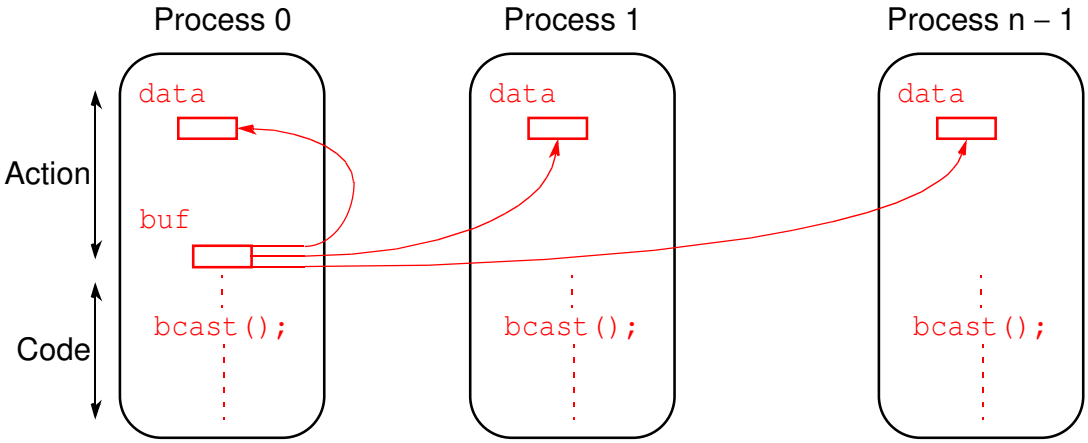
Apart from point-to-point message passing routines, have routines that send message(s) to a group of processes or receive message(s) from a group of processes - higher efficiency than separate point-to-point routines although not absolutely necessary.

Slide 67

Broadcast

Sending same message to all processes concerned with problem.

Multicast - sending same message to defined group of processes.

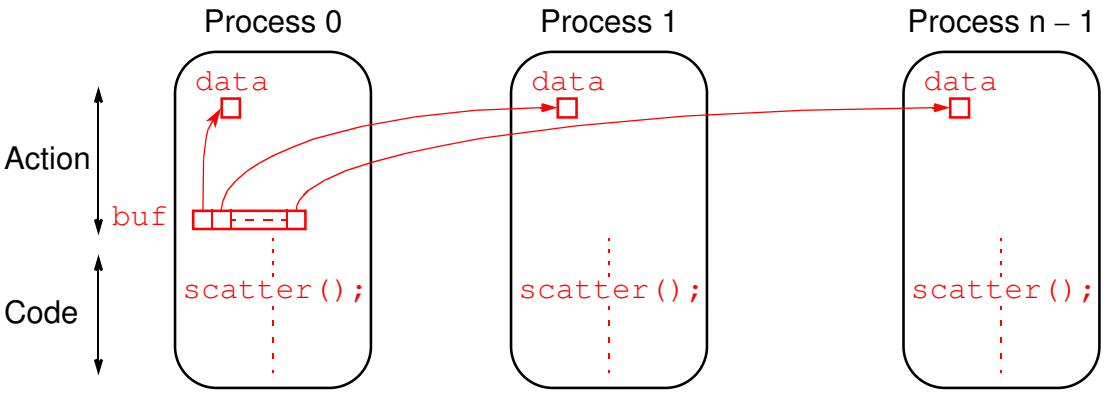


MPI form

Slide 68

Scatter

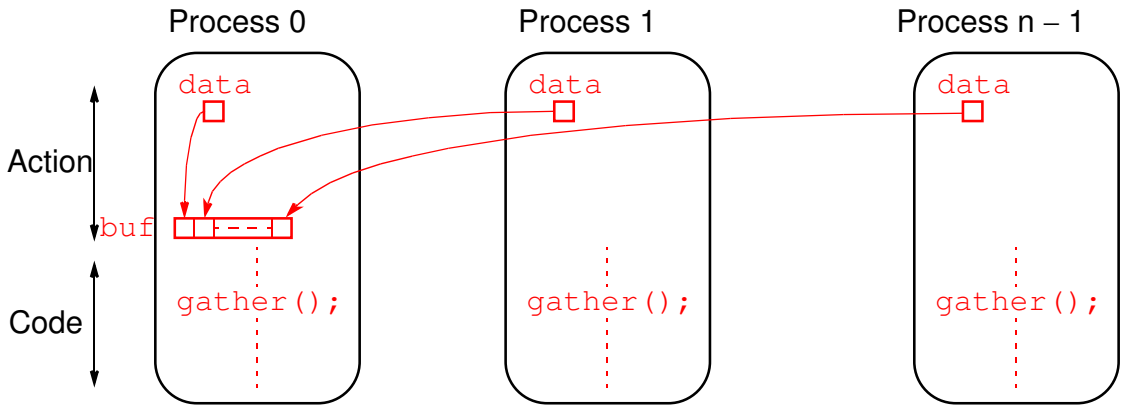
Sending each element of an array in *root* process to a separate process. Contents of *i*th location of array sent to *i*th process.



MPI form

Gather

Having one process collect individual values from set of processes.



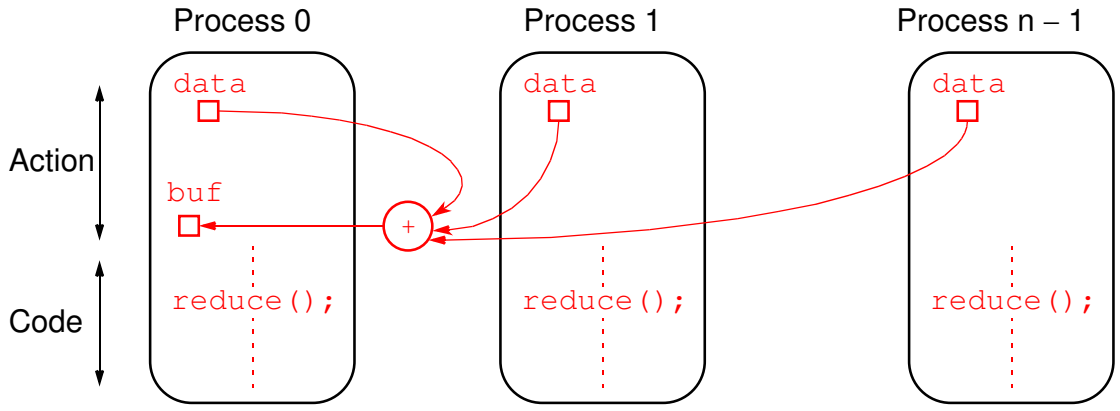
MPI form

Reduce

Gather operation combined with specified arithmetic/logical operation.

Example

Values could be gathered and then added together by root:



MPI form

PVM (Parallel Virtual Machine)

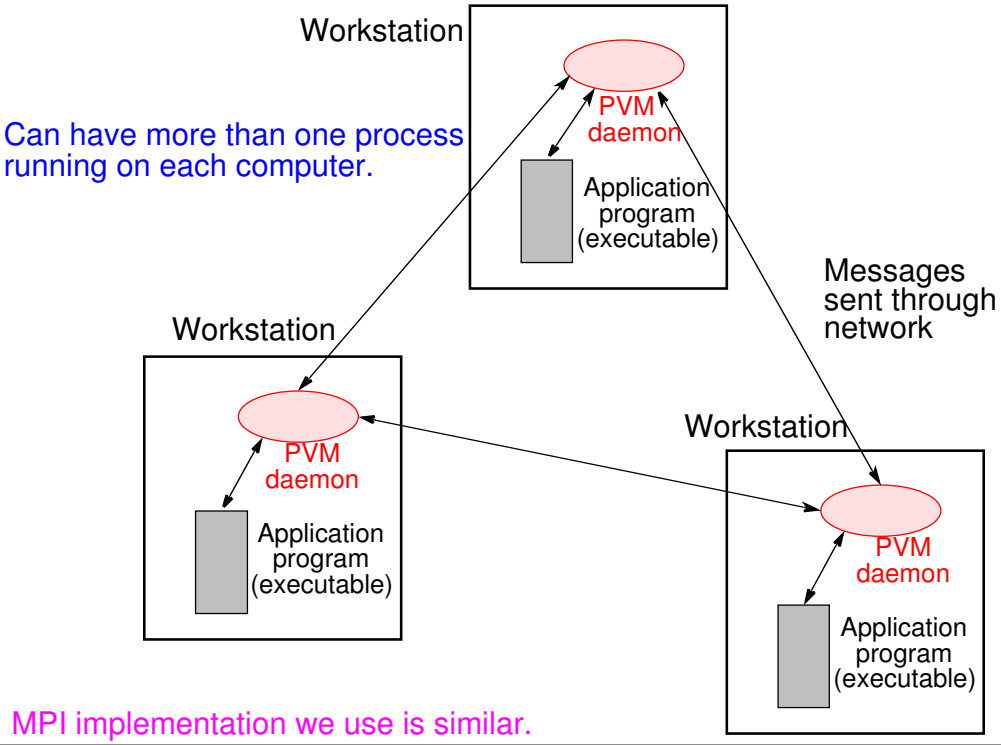
Perhaps first widely adopted attempt at using a workstation cluster as a multicomputer platform, developed by Oak Ridge National Laboratories. Available at no charge.

Programmer decomposes problem into separate programs (usually a master program and a group of identical slave programs).

Each program compiled to execute on specific types of computers.

Set of computers used on a problem first must be defined prior to executing the programs (in a hostfile).

Message routing between computers done by PVM daemon processes installed by PVM on computers that form the *virtual machine*.



MPI implementation we use is similar.

PVM Message-Passing Routines

All PVM send routines are nonblocking (or asynchronous in PVM terminology)

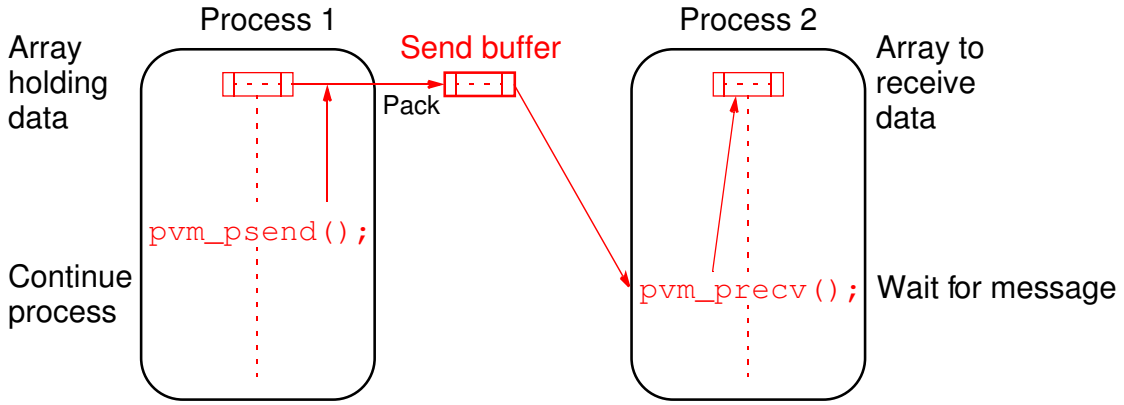
PVM receive routines can be either blocking (synchronous) or nonblocking.

Both message tag and source wild cards available.

Basic PVM Message-Passing Routines

`pvm_psend()` and `pvm_precv()` system calls.

Can be used if data being sent is a list of items of the same data type.



Full list of parameters for `pvm_psend()` and `pvm_precv()`

```
pvm_psend(int dest_tid, int msgtag, char *buf, int len, int datatype)

pvm_precv(int source_tid, int msgtag, char *buf, int len, int datatype)
```

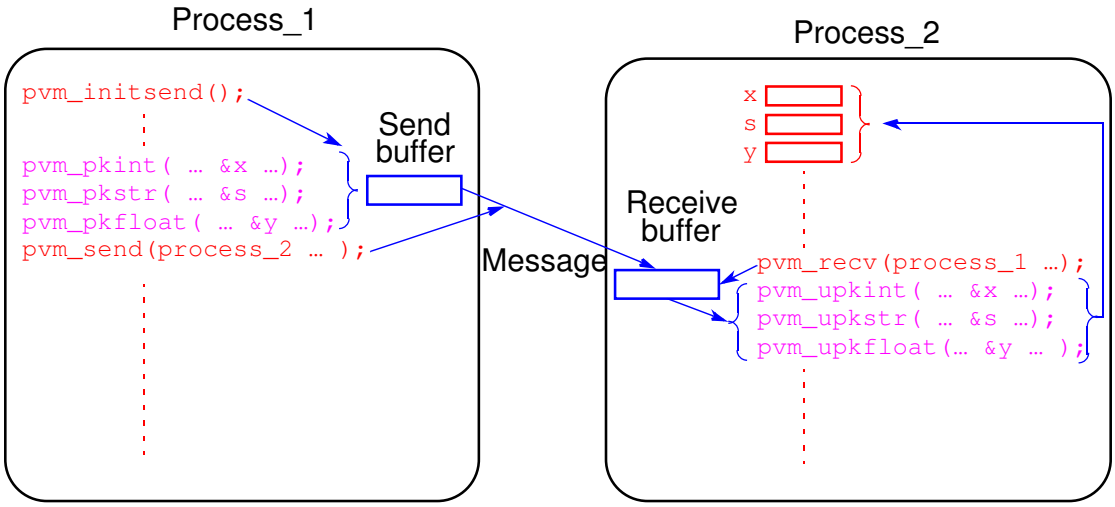
Sending Data Composed of Various Types

Data packed into send buffer prior to sending data.

Receiving process must unpack its receive buffer according to format in which it was packed.

Specific packing/unpacking routines for each datatype.

Sending Data Composed of Various Types Example



Broadcast, Multicast, Scatter, Gather, and Reduce

pvm_bcast()
pvm_scatter()
pvm_gather()
pvm_reduce()

operate with defined group of processes.

Process joins named group by calling **pvm_joingroup()**

Multicast operation, **pvm_mcast()**, is not a group operation.

Sample PVM program.

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]);
      printf("%d from %d\n",result[who],who);
```

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;
```

Slide 81

```
/* 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);
}
```

Broadcast data

Receive results

```
   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);
}
```

Slide 82

MPI (Message Passing Interface)

Standard developed by group of academics and industrial partners to foster more widespread use and portability.

Defines routines, not implementation.

Several free implementations exist.

Slide 83

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 processes must be defined prior to execution and started together.

Orginally SPMD model of computation.

MPMD also possible with static creation - each program to be started together specified.

Slide 84

Communicators

Defines *scope* of a communication operation.

Processes have ranks associated with 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();
}
```

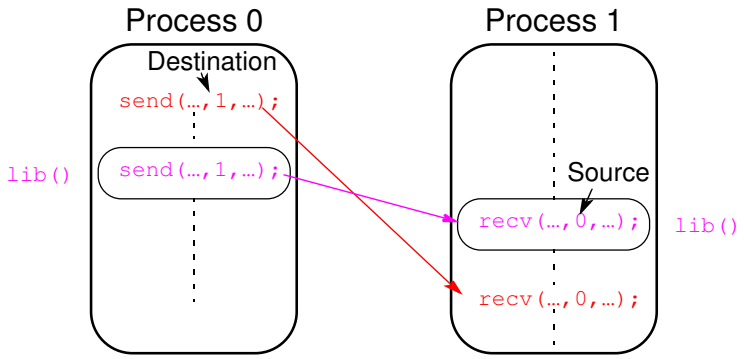
where **`master()`** and **`slave()`** are procedures to be executed by master process and slave process, respectively.

"Unsafe" Message Passing

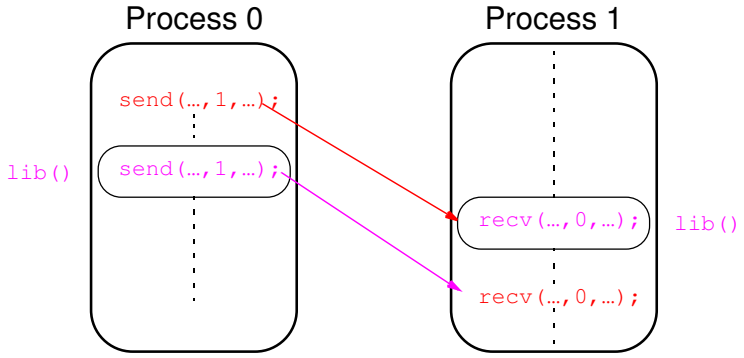
MPI specifically addresses unsafe message passing.

Unsafe message passing with libraries

(a) Intended behavior



(b) Possible behavior



Slide 87

Slide 88

MPI Solution

"Communicators"

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

The communication domain of the library can be separated from that of a user program.

Used in all point-to-point and collective MPI message-passing communications.

Default Communicator

`MPI_COMM_WORLD` exists as the first communicator for all the processes existing in the application.

A set of MPI routines exists for forming communicators.

Processes have a “rank” in a communicator.

Point-to-Point Communication

PVM style packing and unpacking data is generally avoided by the use of an MPI datatype being defined.

Slide 91

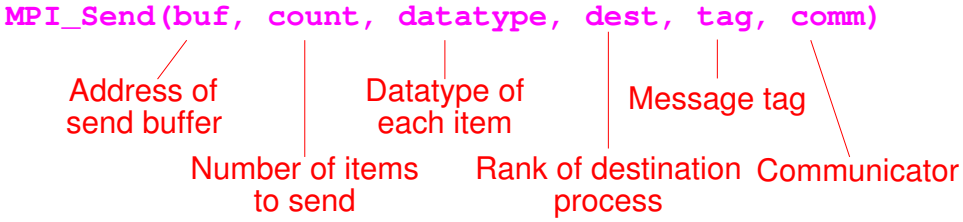
Blocking Routines

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

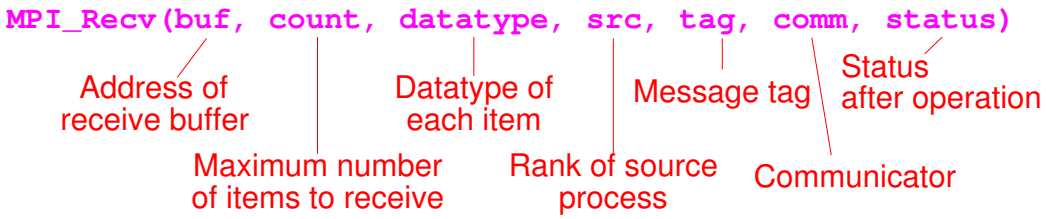
A blocking send will send the message and return. This does not mean that the message has been received, just that the process is free to move on without adversely affecting the message.

Slide 92

Parameters of the blocking send



Parameters of the blocking receive



Example

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

```
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);    /* find 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.

Nonblocking Routine 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 operation completed and returns then.

`MPI_Test()` returns with flag set indicating whether operation completed at that time.

Need to know whether particular operation completed.

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 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,1,MPI_INT,0,msgtag, MPI_COMM_WORLD, status);
}
```


Four Send Communication Modes

Standard Mode Send

Not assumed that corresponding receive routine has started. Amount of buffering not defined by MPI. If buffering 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()`**

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 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.

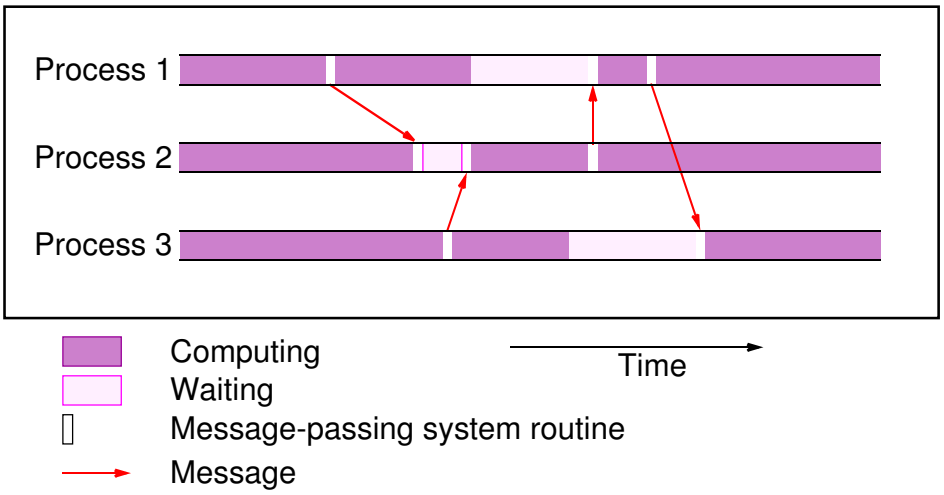
Sample MPI program.

```
#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();
}
```


Debugging and Evaluating Parallel Programs

Visualization Tools

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



PVM has a visualization tool called XPVM.

Implementations of visualization tools are available for MPI. An example is the Upshot program visualization system.

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).