Programming Options

Message-Passing Computing

Basics of Message-Passing Programming

Progamming Options

Single Program, Multiple Data (SPMD) model

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

2. A method of sending and receiving messages

1. An efficient execution process for execution on different computers

2. Necessary in any environment; when processes are to be executed, when to pass messages

Here we will concentrate upon the third option.

Procedures for message passing

1. Design a special-purpose programming language

2. Examine the syntax and semantics of the language

3. Designate those constructs that provide a high level of abstraction

4. Examine the syntax and semantics of the language

Different processes are merged into one program.

Within the program, different statements that will customize the code to select different parts for each process.

Different processes are merged into one program.
Multiple Program Multiple Data (MPMD)

Model

Basic Send and Receive Routines

Message-Passing Routines

Figure 2.3

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

\[ \text{send}(&x, \text{destination}_i); \]

\[ \text{recv}(&y, \text{source}_j); \]

An example of a library call for dynamic process creation multiple to the form

\[ \text{spawn(name}_i); \]

which immediately starts another process and both the calling process and the called pro-

cess proceed together:

Then have the form

\[ \text{recv}(\text{destination}_i); \]

\[ \text{send}(\text{source}_j); \]
### Definitions of Blocking and Non-Blocking

**Blocking**
- Routines that do not return until the message transfer has been completed.
- Examples: `send()` and `recv()`.
- The routines are blocked from continuing.

**Non-Blocking**
- Routines that return immediately, regardless of whether the message has been sent or received.
- The routines do not block the continuation of the program.

### Synchronous Message Passing

Synchronous message passing is a type of message passing where the sending and receiving processes synchronize on a message.

#### Code Example

```c
send();
recv();
```

### Blocking and Nonblocking Message Passing

**Blocking Message Passing**
- The routines are blocked from continuing until the message transfer is completed.

**Nonblocking Message Passing**
- The routines return immediately, assuming the message buffer is not modified by the receiving process.

### MPI Definitions of Blocking and Non-Blocking

#### Blocking
- Routines that block until the message transfer is completed.

#### Non-Blocking
- Routines that return immediately, assuming the message buffer is not modified.

### MPI Definitions of Blocking and Non-Blocking

#### Blocking
- Routines that block until the message transfer is completed.

#### Non-Blocking
- Routines that return immediately, assuming the message buffer is not modified.

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

<table>
<thead>
<tr>
<th>Event</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1 send()</td>
<td></td>
</tr>
<tr>
<td>Process 2 receive()</td>
<td></td>
</tr>
<tr>
<td>Suspend</td>
<td></td>
</tr>
<tr>
<td>Process 1 continue</td>
<td></td>
</tr>
<tr>
<td>Process 2 continue</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Event</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1 receive()</td>
<td></td>
</tr>
<tr>
<td>Process 2 send()</td>
<td></td>
</tr>
<tr>
<td>Suspend</td>
<td></td>
</tr>
<tr>
<td>Process 1 continue</td>
<td></td>
</tr>
<tr>
<td>Process 2 continue</td>
<td></td>
</tr>
</tbody>
</table>
Message buffers are used to hold messages being sent prior to being accepted by `recv()`. For a receive routine, the message has to have been received if we want the message. If `recv()` is reached before `send()`, the message buffer will be empty and `recv()` will wait for the message. For a send routine, once the local actions have been completed and the message is safely on its way, the process can continue with subsequent work. In this way, using such send routines can decrease the overall execution time. In practice, 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. It may be necessary to know at some point if the message has actually been received. Generally, a message buffer is needed between the source and destination to hold messages. If special type matching is not required, a wildcard message tag is used, so that the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then the `recv()` will match with any `send()`. If a send routine is matched, it sets the buffer of the message to be returned to the source process, which allows the source process to know to which process the message was sent. The message tag is carried within the message. If special type matching is required, then 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.

Broadcast action does not occur until all the processes have executed their broadcast routine to receive the data passed from the root process. The SPMD model in which each process has the same program is convenient for this type of operation.

Figure 2.6 shows a broadcast operation in action, with each process receiving the same message. Process 0 is identified as the root process within the broadcast parameters. The root process holds the data to be broadcast in buf.

Figure 2.7 shows a scatter operation, with each process receiving a different element of an array of data from the root process.

Scatter

Sending each element of an array of data in the root to a separate process.

The contents of the root can be transmitted to the process.

Broadcast

Sending the same message to all the processes concerned with the problem.
Figure 2.8
Gather operation.

```java
gather();
buf gather();
data gather();
data
```

Process 0
Process 1

Action
Code
Gather

Having one process collect individual values from a set of processes. Gather operation combined with a specific arithmetic or logical operation.

Figure 2.9
Reduce operation (addition).

```java
reduce();
buf reduce();
data reduce();
data
```

Process 0
Process 1

Action
Code
Reduce

Gather operation combined with a specified arithmetic or logical operation. Normally used after some computation has been done by these processes.

Reduce is essentially the opposite of gather. The data from the worker processes is received by the root process and placed in the location of the array set aside to receive the data.
Using Workstation Clusters

### PVM (Parallel Virtual Machine)
- Perhaps the first widely adopted attempt at using a workstation cluster as a multicomputer platform developed by Oak Ridge National Laboratory.
- Provides a message passing environment for message passing between homogeneous and heterogeneous computers.
- Available at no charge.

### MPI (Message Passing Interface)
- Standard developed by a group of academics and industrial partners to foster more widespread use and portability.
- Several free implementations exist.

#### Using PVM and MPI

1. **Problem Decomposition:**
   - 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.

2. **Configuration:**
   - By PVM on the computers, the virtual machine (VM) is defined. PVM processes are started on the computers.

3. **Communication:**
   - Messages are exchanged between computers using PVM.

4. **Implementation:**
   - Provides a message passing environment for message passing between homogeneous computers.

#### PVM Tools

- **PVM Application Program**: Provides the first widely adopted message passing environment.
**Basic Message-Passing Routines**

The key operation of sending and receiving data are done through message buffers.

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

The key operation of sending and receiving data is the same. If you use PVM send routines:

- The message buffer used for sending the message
- Source process
- Message tag
- Length of the message

All PVM receive routines are nonblocking (or asynchronous in PVM terminology) where

- Destination process
- Message tag
- Message buffer
- Length of the message

The message tag is attached to a message to differentiate between types of messages being sent. Both message tag and source wild cards can be used.

### Figure 2.12

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

- Process 1
  - `pvm_psend();`
  - `pvm_precv();`

- Process 2
  - `wait for message`
  - `pack` send buffer
  - `Array Array`

- Application program (executable)

#### Full list of parameters for *pvm_psend() and *pvm_precv() system calls:

- **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)**
Send data composed of various types

Broadcast, Multicast, Scatter, Gather, and Reduce

In PVM, broadcast, scatter, gather, and reduce operations are used with a group of processes as a named group. The operation can be broadcast, scatter, gather, or reduce, depending on the operation.

The PVM multicast operation, `pvm_mcast()`, is not a group operation. It is generally used to send the contents of the send buffer to each of a set of processes that are defined in a task_ID array (but not to itself even if it is named in the array).

A process joins the named group by calling `pvm_joingroup()`. When called, `pvm_joingroup()` returns a group ID that can be used with a group of processes.

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

Similarly, `pvm_scatter()` would scatter values from each member of the named group.

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

In PVM, the data has to be packed into a PVM send buffer prior to sending the data. The receiving process must unpack its receive message buffer according to the format in which it was packed.

Specific packing and unpacking routines for each datatype must be used to send and receive data.

The basic message-passing routines for packed messages are `pvm_send()`, `pvm_recv()`, and `pvm_nrecv()`. These routines allow for nonblocking communication, which is useful in many applications.

In PVM, the named group is formed by calling `pvm_joingroup()`. The group ID returned by the function can be used with a group of processes.

The `pvm_reduce()` function is used to perform operations such as summing the values in the send buffer and sending the result to each member of the named group.

The `pvm_mcast()` function is used to send the contents of the send buffer to each member of the named group.

The `pvm_gather()` function is used to collect values from each member of the named group and send them to the root process.

The `pvm_scatter()` function is used to scatter values from the root process to each member of the named group.

The `pvm_bcast()` function is used to broadcast a message to each member of the named group.

The `pvm_nrecv()` function is used to receive a message from a specified process without blocking the calling process.

The `pvm_recv()` function is used to receive a message from a specified process and block the calling process until the message is received.

The `pvm_send()` function is used to send a message to a specified process.

The `pvm_initsend()` function is used to initialize a send buffer.

The `pvm_pkint()`, `pvm_pkstr()`, and `pvm_pkfloat()` functions are used to pack integers, strings, and floats, respectively, into a send buffer.

The `pvm_upkint()`, `pvm_upkstr()`, and `pvm_upkfloat()` functions are used to unpack integers, strings, and floats, respectively, from a receive buffer.

Figure 2.13: PVM packing messages, sending, and unpacking.
Figure 2.14

Sample PVM program.

```c
#include <stdio.h>
#include <stdlib.h>
#include <pvm3.h>
#define SLAVE "spsum"
#define PROC 10
#define NELEM 1000

int 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;
    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);
}
```

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen
© Prentice Hall, 1999

MPI

A significant difference from PVM is that only static process creation is supported in MPI. A process cannot be created until the MPI run is initialized. MPI defines the scope of a communication operation.

Communicators

Defines the scope of a communication operation.

Process Creation and Execution

MPI is a “standard” that has implementations. MPI uses a large number of routines (over

MP
Using the SPMD Computational Model

Global and Local Variables

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

Procedure 2 does not execute within code only executed by master process and hence the SPMD model does not execute a master-slave approach. Any global declarations of variables will be duplicated in each process.

Example:

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

```c
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.

To facilitate this within a single program, statements need to be inserted to select which portions of the code will be executed by each processor.

Hence, the SPMD model does not preclude a master-slave approach, just that both the master code and the slave code must be in the same program.

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

In this figure, process 0 wishes to send a message to process 1, but there is also message passing between library routines as shown.

Even though each send/recv pair has matching source and destination, incorrect message passing occurs. The use of wild cards makes incorrect operation or deadlock even more likely. Suppose that in one process a non-blocking receive has wild cards in both the tag and source fields. A pair of other processes call library routines that require message passing. The first send in this library routine may match with the non-blocking receive that is using wild cards, causing erroneous actions.

Solution

Communicators

Communicator Types

- Intracommunicator - for communicating within a group
- Intercommunicator - for communication between groups.

A communicator is a communication domain of the library that defines a set of processes that are allowed to communicate between themselves. Each process has a unique rank within the communicator (an integer from 0 to $n-1$, where there are $n$ processes). 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. New communicators are created based upon existing communicators. A set of MPI routines exists for forming communicators from existing communicators, forming communicators for MPI routines, and forming communicators for all of MPI routines.

- Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers

Barry Wilkinson and Michael Allen

Prentice Hall, 1999

Figure 2.15
Unsafe message passing with libraries.

lib() lib() send(…,1,…); recv(…,0,…);

Process 0 Process 1

send(…,1,…); recv(…,0,…);

(a) Intended behavior

(b) Possible behavior

Destination Source
Point-to-Point Communication

Point-to-point communication is the sending of a message to a single process. This is in contrast to collective communication, which involves multiple processes. In point-to-point communication, a single message is sent from one process to another. This can be done using various communication primitives, such as MPI_Send and MPI_Recv.

### Blocking Routines

Blocking routines return when the operation is locally complete. This means that the message has been sent or received, and the process can proceed without being blocked. The general format of parameters for blocking routines is:

- **MPI_Send(data, count, datatype, dest, tag, comm)**
  - `data` is the send buffer.
  - `count` is the number of items to send.
  - `datatype` is the type of data being sent.
  - `dest` is the rank of the destination process.
  - `tag` is a message tag.
  - `comm` is the communicator.

- **MPI_Recv(data, count, datatype, dest, tag, comm, status)**
  - `data` is the receive buffer.
  - `count` is the maximum number of items to receive.
  - `datatype` is the type of data being received.
  - `dest` is the rank of the source process.
  - `tag` is a message tag.
  - `comm` is the communicator.
  - `status` is the status of the operation.

### Example

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

```c
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);
}
```

### Message Tags

Message tags can be used to identify the origin and destination of a message. They are not necessary for point-to-point communication, but they can be used to track the progress of messages. Wild cards can be used in place of the tag (`MPI_ANY_TAG`) and source (`MPI_ANY_SOURCE`).
Nonblocking Routines

A nonblocking routine returns immediately; that is, allows the next statement to execute, whether or not the routine is locally complete.

Nonblocking send - MPI_Isend()

Nonblocking receive - MPI_Irecv()

Completion can be detected by separate routines, MPI_Wait() and MPI_Test().

MPI_Wait() waits until the operation has actually completed and will return then.

MPI_Test() returns immediately with a flag set indicating whether the operation has 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,

Send Communication Modes

Four communication modes that define the send/receive protocol.

Standard Mode Send

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

Synchronous Mode

If buffering is provided, the send and complete before the receive is reached.

Buffered Mode

Buffer space is supplied to the system via the MPI routine MPI_Buffer_attach() and removed with MPI_Buffer_detach().

Il buffering is provided, the send and complete before the receive is reached.

If no buffering is provided, the send and complete before the receive is reached.

It is not assumed that the corresponding receive completes before the send completes. The amount of buffer space is supplied in the application for this mode.

Nonblocking Modes

Four communication modes that define the nonblocking protocol.

Nonblocking send - MPI_Isend()

Nonblocking receive - MPI_Irecv()
Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers

Barry Wilkinson and Michael Allen

Prentice Hall, 1999

Collective Communication

Involves a set of processes. The processes are those defined by an intra-communicator. Message tags are 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_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

```c
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

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_Reduce_scatter()** - Combine values and scatter results

**MPI_Scan()** - Compute prefix reductions of data on processes

Example
Pseudocode Constructs

We shall use a pseudocode for describing algorithms. Our pseudocode will omit the clutter of parameters that are secondary to understanding the operation.

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

\[
\text{send}(\&x, \&y, P\text{slave});
\]

and in the slave process

\[
\text{recv}(\&a, \&b, P\text{master});
\]

where \( x \) and \( a \) are declared as integers and \( y \) and \( b \) are declared as floats. The integer \( x \) will be copied to \( a \), and the float \( y \) copied to \( b \).

Where appropriate, the \( i \)th process will be given the notation \( P_i \), and a tag may be present that would follow the source or destination name; i.e.,

\[
\text{send}(\&x, P_2, \text{data}\_\text{tag});
\]

sends \( x \) to process 2, with the message tag \( \text{data}\_\text{tag} \).

The most common form of basic message-passing routines needed in our pseudocode is the locally blocking \( \text{send()} \) and \( \text{recv()} \), which will be written as given.

In many instances, the locally blocking versions are sufficient. Other forms will be differentiated with prefixes; i.e.,

\[
\text{ssend}(\&\text{data1}, P\text{destination}); \quad /* \text{Synchronous send} */
\]

Evaluating Parallel Programs

The parallel execution time, \( t_p \), is composed of two parts: a communication part, \( t_c \), and a computation part, \( t_r \), i.e.,

\[
t_p = t_c + t_r
\]

The computation time can be estimated in a similar way to that of a sequential algorithm. Communication Time

The communication time can be estimated in a similar way to that of a sequential algorithm.

\[
t_c = n \times t_d + t_s
\]

where \( n \) is the number of data words, \( t_d \) is the transmission time to send one data word, \( t_s \) is the startup time, and \( t_s \) is the message latency. This is essentially the time to send a message with no data. The startup time is assumed to be constant, and the message latency is assumed to be constant.

As a first approximation, we will use

\[
t_c = n \times t_d + t_s
\]

where \( n \) is the number of data words, \( t_d \) is the transmission time to send one data word, \( t_s \) is the startup time, and \( t_s \) is the message latency. This is essentially the time to send a message with no data. The startup time is assumed to be constant, and the message latency is assumed to be constant.

Parallel Execution Time

The parallel execution time is composed of two parts: a computation part, \( t_r \), and a communication part, \( t_c \), i.e.,

\[
t_p = t_c + t_r
\]
Important Note on Interpretation of Equations

Many assumptions in the analysis (see textbook). Only intended to give a starting point to how an algorithm might perform in practice.

The parallel execution time, \( t_p \), will be normalized to be measured in units of an arithmetic operation, which of course will depend upon the computer system.

We might find that the computation requires \( m \) computational steps so that

\[
    \text{t_{comp}} = m
\]

Since we are measuring time in units of computational steps, the communication times, \( \text{t_{comm}} \), have to be measured in the same way.

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

\[
    \text{t_{comm}} = q \left( t_{\text{startup}} + nt_{\text{data}} \right)
\]

Both the startup and data transmission times, \( t_{\text{startup}} \) and \( t_{\text{data}} \), are measured in computational steps, so that we can add \( \text{t_{comp}} \) and \( \text{t_{comm}} \) together to obtain the parallel execution time:

\[
    \text{t_p} = \text{t_{comp}} + \text{t_{comm}}
\]

Latency Hiding

A way to ameliorate the situation of significant message communication times is to overlap communication with subsequent computations. This can be achieved by using nonblocking send routines provided particularly to enable latency hiding.

Threads offer an efficient mechanism for overlapping computations with communication.
Time Complexity

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.

Example

Suppose an algorithm, $A_1$, requires $4x^2 + 2x + 12$ computational steps for $x$ data items.

As we increase the number of data items, the total number of operations will depend more and more upon the term $4x^2$. The first term will “dominate” the other terms, and eventually the other terms will be insignificant. The growth of the function in this example is polynomial.

Another algorithm, $A_2$, for the same problem might require $5\log x + 200$ computational steps. In the function $5\log x + 200$, the first term will eventually dominate the other term, which can be ignored, and we only need to compare the dominating terms. The growth of the function $\log x$ is logarithmic.

For a sufficiently large $x$, logarithmic growth will be less than polynomial growth.

We can capture growth patterns in the $O$ notation (big-oh). Algorithm $A_1$ has a big-oh of $O(x^2)$. Algorithm $A_2$ has a big-oh of $O(\log x)$.

Formal Definition

The $O$ notation $f(x) = O(g(x))$ 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 $f(x) = O(x^2)$, since $0 < 4x^2 + 2x + 12 \leq 6x^2$ for $x \geq 3$.

Unfortunately, the formal definition also leads to alternative functions for $g(x)$ that will also satisfy the definition. Normally, we would use the function that grows the least for $g(x)$.

$\Omega$ notation - upper bound

$f(x) = \Omega(g(x))$ if and only if there exists positive constants $c_1$, $c_2$, and $x_0$ such that $0 \leq c_1g(x) \leq f(x) \leq c_2g(x)$ for all $x \geq x_0$.

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

$\Omega$ notation - lower bound

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

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

We can read $O(f(x))$ as “grows at most as fast as” and $\Omega(f(x))$ as “grows at least as fast as.”

The $\Omega$ notation can be used to indicate the best case situation. For 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 $O(n^2)$ and $\Omega(n\log n)$. The growth of the function $\Omega(n\log n)$ will be less than polynomial growth, and we can capture this in the $O$ notation.
Time Complexity of a Parallel Algorithm

Example

Consider a problem that involves parallel processing with two computers. The problem can be broken down into several steps:

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

As in most parallel algorithms, there is computation and communication, which we will consider separately:

**Computation** (for steps 2 and 4):
- \( t_{comp} = \frac{n}{2} + 1 \)

**Communication** (for steps 1 and 3):
- \( t_{comm} = (t_{startup} + \frac{n}{2} t_{data}) + (t_{startup} + \frac{n}{2} t_{data}) = 2t_{startup} + \frac{n}{2} (2t_{data}) \)

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

\[(\text{parallel time complexity}) \times \text{(number of processors)} = \text{sequential time complexity}\]

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 \(n^2\) processes and has time complexity of \(O(1)\) is not cost optimal.

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

\[
\begin{align*}
000 & \rightarrow 001 \\
000 & \rightarrow 010 \\
000 & \rightarrow 011 \\
000 & \rightarrow 100 \\
000 & \rightarrow 101 \\
000 & \rightarrow 110 \\
000 & \rightarrow 111
\end{align*}
\]

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.
To gather data on a hypercube network, the reverse algorithm can be used. The messages become longer as the data is gathered, and hence the time complexity is increased over time.

**Step 1:**
- Node 000 sends a message to node 100.
- Node 100 sends a message to node 110.
- Node 110 sends a message to node 111.
- Node 101 sends a message to node 011.
- Node 001 sends a message to node 001.

**Step 2:**
- Node 000 sends a message to node 010.
- Node 100 sends a message to node 010.
- Node 110 sends a message to node 011.
- Node 101 sends a message to node 011.

**Step 3:**
- Node 000 sends a message to node 001.
- Node 010 sends a message to node 001.

In the case of gather, the messages become longer as the data is gathered, and hence the time complexity is increased over time.
Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers
Barry Wilkinson and Michael Allen
Prentice Hall, 1999

Figure 2.21
Broadcast in a mesh.

1 2 3
4 5 6

Steps
Broadcast on a Mesh Network
Send message across the top row and down each column as the message reaches the top

Broadcast on a Mesh Network

Broadcast on an Ethernet network.
Broadcast on a Workstation Cluster

Broadcast on a Workstation Cluster

Number of steps because the diameter of a mesh without wrapround is given by 2^n - 1
Requires 2(n - 1) steps or O(n^2) on an n x n mesh, again an optimal algorithm in terms of

Prentice Hall, 1999
Figure 2.23
1-to-N fan-out broadcast.

Figure 2.24
1-to-N fan-out broadcast on a tree structure.
Writing and Evaluating Parallel Programs

Debugging Strategies

Visualization Tools

 Programs can be understood as they are executed in a space-time diagram (or process-time diagram) with a graphical interface. Following a parallel instruction, the program's execution can be tracked with detail.

Debugging and Evaluating Parallel Programs

Visualization Tools

Parallel Programming: Techniques and Applications using Networked Workstations and Parallel Computers

Barry Wilkinson and Michael Allen

Prentice Hall, 1999
Evaluating Programs Empirically

Measuring Execution Time

To measure the execution time between point \( L_1 \) and point \( L_2 \) in the code, we might have a construction such as

\[
\text{L1: time(&t1); /* start timer */}
\]

\[
\text{..}
\]

\[
\text{L2: time(&t2); /* stop timer */}
\]

\[
\text{elapsed_time = difftime(t2, t1); /* elapsed_time = t2 - t1 */}
\]

\[
\text{printf("Elapsed time = \%5.2f seconds\", elapsed_time);}
\]

MPI provides the routine \( \text{MPI_Wtime()} \) 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 \( P_0 \).

\[
\text{P0.}
\]

\[
\text{L1: time(&t1); send(&x, P1);} \]

\[
\text{recv(&x, P1);} \]

\[
\text{L2: time(&t2);}
\]

\[
\text{elapsed_time = 0.5 * difftime(t2, t1);} \]

\[
\text{printf("Elapsed time = \%5.2f seconds\", elapsed_time);}
\]

\[
\text{P1.}
\]

\[
\text{recv(&x, P0);} \]

\[
\text{send(&x, P0);} \]
2-1. Develop an equation for message communication time, \( t_{\text{comm}} \), that incorporates the delay through multiple links as would occur in a static interconnection network. Develop the equation for a mesh and for a hypercube network, assuming that all message destinations are randomly chosen.

2-2. Devise an efficient way that a scatter operation can be done on an 3-dimensional hypercube. What is its time complexity?

2-3. Devise an efficient way that a scatter operation can be done on an \( n \times n \) mesh. What is its time complexity?

2-4. Determine the time complexity of the gather operation on a hypercube using the algorithm described in Section 2.3.4, taking into account the increased message size while gathering data.

2-5. (A suitable first assignment) Compile and run either the PVM program to add numbers or the MPI program to add numbers, as given in Figures 2.14 and 2.16 (or found in http://www.cs.uncc.edu/par_prog as the “sample ... and execute on your system. Modify the program so that the maximum number is found and output as well as the sum.

2-6. Measure the time to send a message in a parallel programming system by using code segments of the form

```c
Master
L1: time(&t1);
send(&x, Pslave);
L2: time(&t2);
tmaster = difftime(t2, t1);
recv(&tslave, Pslave);
print(“Master Time = %d”,tmaster);
print(“Slave Time = %d”,tslave);
```

Slave
```
L1: time(&t1);
recv(&x, Pmaster);
L2: time(&t2);
tslave = difftime(t2, t1);
send(&tslave, Pmaster);
```

Repeat with the ping-pong method described in Section 2.4.4. Experiment with sending groups of multiple messages and messages of different sizes to obtain a good estimate for the time of message transfers. Plot the time of sending a message against the size of the message and fit a line to the results. Estimate the startup time, \( t_{\text{startup}} \) (latency), and the time to send one data item, \( t_{\text{data}} \).

2-7. Repeat Problem 2-6 for broadcast and other collective routines as available on your system.

2-8. Compare the use of broadcast and gather routines using individual send and receive routines.

2-9. Experiment with latency hiding on your system to determine how much computation is possible as messages are being sent. Investigate using both nonblocking and locally blocking send routines.

2-10. If you have both PVM and MPI available (or any two systems), make a comparative study of the communication times on the system by passing messages between processes that have been instrumented to measure the communication times. Write a program to measure the communication times on the system by passing messages between processes that have been instrumented to measure the communication times.

2-11. Compare the performance of using `pvm_psend()` and `pvm_precv()` in PVM with the performance of `pvm_send()` and `pvm_recv()` in MPI. Write a program to measure the communication times on the system by passing messages between processes that have been instrumented to measure the communication times.