para//ab

Parallel Processing: A Self-Study Introduction

A First Course in Programming the Intel Paragon

Jeremy Cook

December 1st, 1993

Report #pl-jc-93.3 – para//ab, Dept. of Informatics, University of Bergen, N-5020 Bergen, Norway

Foreword

This booklet is written in the hope that we can accelerate the learning process of readers using an Intel Paragon computer for the first time. It is not intended as a replacement for the manufacturers programming and reference manuals, but rather to complement them and to guide the new user through the jungle of terminology and programming techniques. We hope that we have written this guide such that the reader can complete this tutorial without requiring such manuals.

Our long experience with advanced computer systems has shown us that these users want to become familiar with a new system as quickly as possible, without having to wade through, say, a metre shelf of manuals to be able to write a simple program. Rather than write *yet another tutorial*, we have tried to give the user guidance in using the Paragon, with concise information, with the hope that they can get up to speed as quickly as possible. Very little instruction on *parallel algorithms* is given in this guide, we have concentrated instead on *use* of the system.

The text and examples in this guide were developed on the 100 node system at para//ab. Almost everything in this guide should be applicable to all Paragon system configurations.

para//ab secretary,
para//ab, Dept. of Informatics,
University of Bergen,
N-5020 Bergen,
Norway

email: `adm@parallab.uib.no`

Further copies may be obtained electronically as file `Paragon_study_guide.ps.Z` (compressed PostScript file) from the anonymous ftp server (`ftp.ii.uib.no`) at para//ab in the directory `/pub/tech_reports`.

Example session:

```
% ftp ftp.ii.uib.no
Connected to gran-fddi.ii.uib.no.
220 gran FTP server (Version 5.60) ready.
Name (ftp.ii.uib.no:jeremy): anonymous
331 Guest login ok, send ident as password.
Password: (YOUR EMAIL ADDRESS)
230 Guest login ok, access restrictions apply.
ftp> cd pub/tech_reports
250 CWD command successful.
ftp> bin
200 Type set to I.
ftp> get Paragon_study_guide.ps.Z
200 PORT command successful.
150 Opening BINARY mode data connection for Paragon_study_guide.ps.Z (164958 bytes).
226 Transfer complete.
local: Paragon_study_guide.ps.Z remote: Paragon_study_guide.ps.Z
164958 bytes received in 0.073 seconds (2.2e+03 Kbytes/s)
ftp> quit
221 Goodbye.
% uncompress Paragon_study_guide.ps.Z
% lpr Paragon_study_guide.ps
```

Copies can also be obtained by sending electronic mail to `adm@parallab.uib.no`.

From time to time, we will update this booklet. Your comments and suggestions for future chapters would be appreciated.

The material in this booklet is based on the original iPSC and iPSC/2 study guides developed at Christian Michelsen Institute by Ron Pickering, Clifford Addison, Jeremy Cook and David Warhurst. Much of the text and examples have been re-worked to fit the Paragon programming model.

Revision History

1989		iPSC version
1990		iPSC/2 version
1st December 1993	v1.0	Initial release for Paragon

Contents

Chapter 1

Introduction

1.1 Aims and Objectives

Through a careful study of these notes, you will be introduced to the basic concepts of parallel processing. This is a way of speeding up the computation by carrying out many of the sub-tasks concurrently. This implies that the hardware must provide independent processors capable of executing their own program instructions. You will find that the Paragon makes use of a set of routines which in conjunction with the usual functions available within either Fortran or C provide the necessary facilities for implementing appropriate algorithms.

The objectives of this booklet are:

- to outline the topology of the Paragon supercomputer,
- to set out the salient features of computing with independent processors, each having its own data,
- to explain the details of the library of routines which provide support for parallel computing,
- to demonstrate some of the aspects of parallel computing by means of simple examples,
- to provide sufficient detail of the operating system to allow readers to create and implement their own program examples.

1.2 Pre-requisites

It will be assumed that readers are familiar with either Fortran 77 or C languages and are experienced in implementing programs on serial machines. A knowledge of UNIX will prove useful but not essential.

1.3 A Parallel Machine

The Intel Paragon is a particular form of parallel machine which makes concurrent computation available at relatively low cost. It consists of a set of independent processors, each with its own memory, capable of operating on its own data. Each processor has its own program to execute and processors are linked by communication channels.

The hardware consists of a number of *nodes*, disk systems, communications networks all mounted together in one or several cabinets with power supply for the whole system. Each node is a separate board, rather like a separate computer. Each node has memory, network interface, expansion port, cache and so on. The nodes are linked together through a back plane which provides high speed communications between them.

Each node has its own operating system which can be considered as permanently resident. It takes care of all the message passing, and also allows more than one executable program, or process as they will be called, to be active on each node at any time. Strictly speaking, it is node processes that communicate with other node processes rather than the nodes themselves.

Figure 1.1: Process communication on the nodes

Remember, nodes use their own copy of the program and have their own memory allocation. No variables are shared between nodes or even between processes on the same node. Data can only be shared by sending them as messages between processes.

The Paragon supercomputer is a distributed-memory multicomputer. The system can accommodate more than a thousand heterogeneous nodes connected in a two-dimensional rectangular mesh. A lightweight *MACH 3.0* based microkernel is resident on each node, which provides core operating system functions. Transparent access to file systems is also provided. Nodes communicate by passing messages over a high-speed internal interconnect network. A general-purpose MIMD (Multiple Instruction, Multiple Data) architecture supports a choice of programming styles and paradigms, including true MIMD and Single Program Multiple Data (SPMD).

In this tutorial we will adopt the SPMD programing paradigm (Single Program Multiple Data) i.e. each process is the same program executing on different processors. Each program executes essentially the same algorithms, but different branches of the code may be active in different processors. This is not by any means the only way to write parallel programs but it is a convenient one for the purpose of this tutorial.

The general architecture of the machine is illustrated in figure 1.2. In the illustration, 98 *compute*[1] nodes are arranged in a 2D mesh. Each compute node consists of two i860XP processors. One of these is an application processor and the other a dedicated communication processor. User applications will normally run using the application processor. The figure illustrates that each compute node may pass messages to neighbouring nodes through a bi-directional communication channel. When messages are to be passed indirectly between non-neighbouring processors, the operating system will handle routing the message between intermediate processors.

File system support and high-speed parallel file access is provided through the nodes labelled *service* and *I/O* in the diagram. Access to the parallel file system is made through standard OSF library routines (*open()*, *close()*, *read()*, *write()*, *etc.*,).

When a user is logged in to the Paragon system, the operating system will allocate the login session to one of the service nodes. Exactly which service node is in use is totally transparent

[1] The system you are using may be configured differently, consult your local system administrator.

to the user. The user will usually edit files, and compile, link and run applications while logged in to one of the service nodes. Note also that most sites will have available a so-called *cross-environment* which allows most of the program development stages – editing, compiling, linking and debugging – to be carried out on a workstation away from the paragon system. Using the cross-environment is <u>highly</u> recommended as the available capacity for such operations is usually greater on a workstation than on the service nodes. Consult your local system administrator to find out how to use this facility.

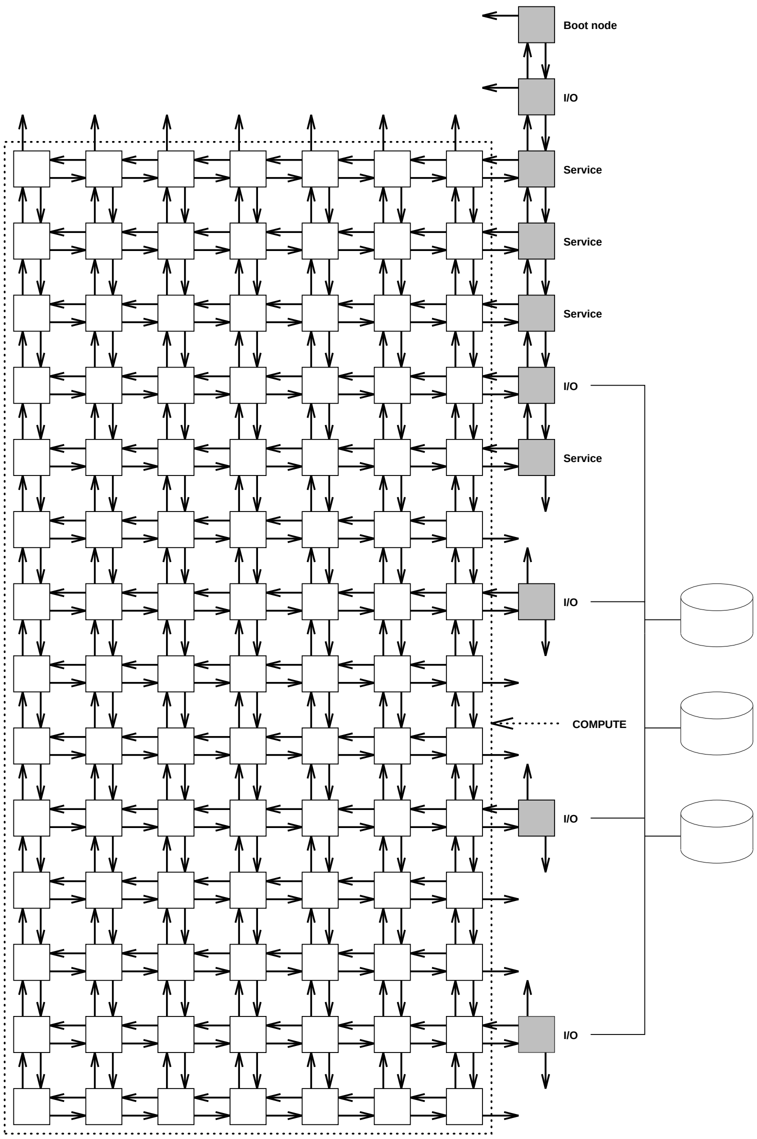

Figure 1.2: Overview of the Paragon system

1.4 Passing Messages on the Paragon

Data needed for processing on the Paragon must be shared between processes by *message passing*. There are a number of libraries available for message passing on the Paragon system. For the purposes of this tutorial, we will concentrate on the proprietary NX message passing library provided by Intel. This offers the best message passing performance for this system. Other message passing libraries such as PVM, P4, or *tcgmsg* may also be available on the system which you are using but these will not be covered in this tutorial.

Later it will become apparent that good parallel algorithms keep message transfers to a minimum, otherwise much of the theoretical advantage of processing concurrently is lost through the communication overheads. First, let us look at the general features of message passing.

1.5 Messages and Their Contents

When messages are sent, this does not imply that they have also been received. The receive procedure must be called in the receiving process in order for the message to pass from source to destination processor, so there can be queues of messages waiting to be transmitted. Generally, when a message is to be physically sent from one processor, the receiving processor must *post* a receive in order for the message to be able to be transmitted, thus avoiding a "deadlock" situation. Messages which are sent without a corresponding receive will eventually cause the process to deadlock. Communication library routines are available, callable from both Fortran and C.

Essentially, when a message is being sent, the following information is required:

- the message type,
- where the message is located (the memory location),
- the length of the message,
- which node the message is going to,
- which process the message is going to. Remember, each node may have more than one active process resident at any time.

There are a number of routines for sending messages. These are covered in detail in chapter 2. The general format of a routine for sending a message is:

```
<m>send( type, buf, count, node, ptype )
```

<m>	The mode of the send. Common use would be csend() for synchronous message passing, and isend() for asynchronous message passing.
type	A user defined code representing the type or class of the message contents (see section 1.5.1).
buf	A user defined buffer containing the data to be sent (see section 1.5.2).
count	The length of the user defined buffer (in bytes).
node	The destination node to send the message to (see section 1.5.3).
ptype	The process on the destination node that will receive the given message.

When a "receive message" is carried out the process only requires the message length and its type. The general format of a routine for receiving a message is:

```
<m>recv( typesel, buf, count )
```

<m>	The mode of the receive. Common use would be crecv() for synchronous message passing, and irecv() for asynchronous message passing.
typesel	A user defined code selecting the type or class of the message to be received (see section 1.5.1).
buf	A user defined buffer in which to receive the incoming message (see section 1.5.2).
count	The length of the user defined receive buffer (in bytes).

Let us consider what is meant by the message type.

1.5.1 Message Type

This term may be confusing because it is probably thought of in terms of a data type (e.g. integer). However, in this context, it is something different. In a message transfer routine, type is an integer value in the range 0 to 1,000,000,000 which is assigned to a message by the user and which serves as a flag to identify a particular kind of message. The nodes will only receive a message which is of the specified "type", i.e. a receive message request for a type 30 message can only be satisfied by a message sent as type 30. It is possible, for example, to distinguish in this way between messages sent by one routine and messages sent by another. A way of doing this might be to adopt a convention such as:

- type 10 an initialisation message,
- type 20 a data exchange message,
- type 30 "send me a new task".

More complicated message type selectors (using indices < 0 or $> 1{,}000{,}000{,}000$) can be used by receiving processes to specify a range of message types but this will not be dealt with in this document. A source node or source process can also be specified to select a message by using the `crecvx()` or `irecvx()` calls. Refer to the C or Fortran reference manuals for advanced usage such as this.

It is now possible to begin looking at the detailed specifications of the message passing routines.

1.5.2 Message Buffer

The NX message passing library is a *low level* library. The programmer is expected to handle the allocation of message buffers. A typical NX send routine requires that the user provides the address of a message buffer and the length (in bytes) of the buffer. Buffers are not required to be just character arrays, they can be integers or floating point numbers, or any programmer defined structured data type. The programmer should however be careful to calculate the correct buffer length in bytes. One simple way to allocate a suitable buffer and pass it to another process might be as in the following C code extract.

For the sending process:

```
int buf[10];
long node, ptype;
:
/* assign buf[0-9], node and ptype */
:
csend(10, buf, sizeof(buf), node, ptype);
```

For the receiving process:

```
int buf[20];

crecv(10, buf, sizeof(buf));
```

Note that in this example, the receiving buffer is actually larger than the buffer that is sent. This is of no consequence as only the first 10 integers will be filled. Information routines can be used to determine how long the incoming message was.

Buffers may also be dynamically allocated and deallocated on C using *malloc()* and *free()*.

1.5.3 Message Destination

The destination of a message being sent is specified as a destination *node* and *ptype*. *node* is usually an integer specifying the number of the destination node. Nodes are numbered from 0 to $n-1$ where n is the number of nodes in use. A destination *node* value of -1 means that the message should be broadcast to all nodes (except for the actual node doing the sending). *ptype* is used to specify the process type of the destination process. Remember that the node operating system is capable of running several processes on each, and it is therefore necessary to specify to which you intend the message to be sent. In most applications the destination *ptype* will be the same as the source *ptype*. It is therefore sufficient to define *ptype* in the following way:

```
#include <nx.h>
long ptype;
:
:
ptype = myptype();
csend(10, buf, sizeof(buf), node, ptype);
```

Further advanced use of *ptype* will not be covered in this tutorial.

1.5.4 Recap

(i) A message type can be either real or integer. True or false?

(ii) By what means can a node process discriminate between different incoming messages?

(iii) Write down four items that must be specified when a message is being sent out.

(Check your answers in the Solutions section in Appendix C)

Chapter 2

Introduction to Fortran / C Routine Libraries

There are nearly 50 routines in the message passing libraries; but there are only 10 or so routines that are necessary for you to know before you can write programs for the Paragon. Only these routines plus one or two other useful ones have been included in the following discussion. If you want to see the complete list of all routines, it is best to consult the Fortran or C Programmer's Reference Manuals.

First, we look at the Fortran routines. C users may prefer to go straight to section 2.2.1.

2.1 The Fortran Routine Library

A summary of these routines can be found in Appendix A.

2.1.1 Subroutines

CRECV receive a message from another process. CRECV blocks the calling process until the message is received.

use: `call crecv(typesel,buf,count)`

There are two input parameters:

- `typesel` integer value referring to the type of the message to be received. This is the only qualifier that can be used to select a particular message,
- `count` the number of bytes in the buffer.

and one output parameter:

- `buf` is the buffer where the received message will be stored. Valid entries include any legal Fortran type.

notes: There are information routines, described below, that can be called to obtain extra information about a received message. There are also ways of using `type` to wait for the first message matching a set of message types.

example:

```
integer msg(4)
call crecv(20,msg,16)
```

will cause the receiving process to wait until it has received a message of type 20, of length 16 bytes which it stores in `msg`.

CSEND

sends a message and waits for completion. CSEND blocks the calling process until the message is sent

use: `call csend(type,buf,count,node,ptype)`

All parameters are input values:

- `type` integer defining message type (`0<=type< 1,000,000,000`),
- `buf` the buffer which contains the message. Valid entries include any legal Fortran data type. It is recommended that data types match in send and receive operations,
- `count` the number of bytes to be transmitted. There is no limit on the message length between nodes,
- `node` the identifier of the destination node,
- `ptype` the identifier of the destination process.

notes: If the node parameter is `-1`, the message will be sent to all processes whose `ptype` matches that given, except for the sending process itself. A successful send operation does not imply that the message was received by a process.

example: `call csend(40,hib,4,mynode()+1,myptype())`
sends a 4 byte message in `hib` to the next node, designating it to be of type 40. Processing is suspended whilst the message is being sent.

CSENDRECV

Sends a message and waits for a reply.

use: `call csendrecv(type,sbuf,scount,node,ptype,typesel,rbuf,rcount)`

`csendrecv()` can be used when a message is to be sent and a reply is expected.

Input parameters:

- `type` integer defining message type (`0<=type< 1,000,000,000`),
- `sbuf` the buffer which contains the message. Valid entries include any legal Fortran data type. It is recommended that data types match in send and receive operations,
- `scount` the number of bytes to be transmitted. There is no limit on the message length between nodes,
- `node` the identifier of the destination node,
- `ptype` the identifier of the destination process.
- `typesel` integer value referring to the type of the message to be received. This is the only qualifier that can be used to select a particular message,
- `rcount` the number of bytes in the buffer.

Output parameters:

- `rbuf` is the buffer where the received message will be stored. Valid entries include any legal Fortran type.

notes: The data in `sbuf` is not guaranteed to be sent before the reply begins to arrive in `rbuf`. It is therefore essential to use two *separate* buffers.

IRECV post a receive for a message from another process and return immediately. IRECV initiates a receive from another process, but does not wait until the message has arrived. Computation can therefore continue whilst a message is arriving.

use: `id = irecv(typesel,buf,count)`

There are two input parameters:

- `typesel` integer value referring to the type of the message to be received. This is the only qualifier that can be used to select a particular message,
- `count` the number of bytes in the buffer.

and one output parameter:

- `buf` is the buffer where the received message will be stored. Valid entries include any legal Fortran type.

Return value:

- `id` integer message ID for use by `msgwait()`.

notes: There are information routines, described below, that can be called to obtain extra information about a received message. There are also ways of using `typesel` to select for the first message matching a set of message types. Before accessing the contents of `buf`, you should ensure that the message has completely arrived by calling `msgwait()`.

example:

```
integer msg(4), rid
rid = irecv(20,msg,16)
   :
   : compute compute compute...
   :
msgwait(rid)
write(*,*) msg(2)
```

will allow the receiving process to compute while a message of type 20 and length 16 bytes is arriving. Before accessing any elements of the receive buffer, we check that the receive is completed with `msgwait()`.

ISEND initiates a message send and returns immediately. ISEND sends a message but returns immediately, allowing computation to continue. A message ID is returned to allow the programmer to determine when the send is complete

use: `id = isend(type,buf,count,node,ptype)`

All parameters are input values:

- `type` integer defining message type (`0<=type< 1,000,000,000`),
- `buf` the buffer which contains the message. Valid entries include any legal Fortran data type. It is recommended that data types match in send and receive operations,
- `count` the number of bytes to be transmitted. There is no limit on the message length between nodes,
- `node` the identifier of the destination node,

`ptype` the identifier of the destination process.

Return value:

`id` integer message ID for use by `msgwait()`.

notes: If the node parameter is `-1`, the message will be sent to all processes whose `ptype` matches that given, except for the sending process itself. You should check that the send has completed before accessing the contents of `buf`, otherwise you could corrupt the buffer being sent. You can do this with the `msgwait()` call.

example:

```
      integer sid, hib
      sid = 0
 100  CONTINUE
      :
      : compute compute compute  ...
      :
c     Wait for previous send to complete before assigning
c     stuff to buffer
      if (sid.ne.0) then
        msgwait(sid)
        sid = 0
      endif
      hib = 42
      sid = isend(40,hib,4,mynode()+1,myptype())
      goto 100
```

sends a 4 byte message in `hib` to the next node, designating it to be of type 40. Processing may continue whilst the message is being sent.

INFOCOUNT an integer function which returns the size (in bytes) of the message received after a receive operation completes.

use: `cnt = infocount()`

notes: All "info" routines must be placed immediately after the related receive call.

INFONODE an integer function which returns the node number of the process that sent the latest message.

use: `node = infonode()`

notes: All "info" routines must be placed immediately after the related receive call.

MCLOCK an integer function which returns the elapsed time in milliseconds since the node was initialised.

use: `result = mclock()`

notes: Do not use the elapsed time given by this function to try and synchronise between different nodes or the host as each will have its own start time.

MSGWAIT causes process to wait until the asynchronous operation identified by `mid` completes.

use: `msgwait( mid )`

notes: other asynchronous utilities such as `msgdone()`, `msgignore()` and `msgmerge()` are available. See the reference manual for further details.

MYNODE returns an integer value which identifies the node on which the process resides.

use: `me = mynode()`

`me` the number of the node executing the process.

notes: When all nodes are loaded with the same program, this is the only way the process can identify its own node.

MYPTYPE an integer function which returns the current processes process type.

use: `ptype = myptype()`

`ptype` contains the current processes type.

NUMNODES an integer function which returns the number of nodes in the application.

use: `nnodes = numnodes();`

`nnodes` gives the number of nodes in the application.

This completes the detailed descriptions of the key routines in Fortran. After a recap, you will be ready to consider some programming examples. C-users will want to work through section 2.2.1.

2.1.2 Recap

1. Explain the action of each of the statements in the following program segments:

   (i)
   ```
   msg(1) = count
   msg(2) = msglen
   call csend(10,msg,8,0,myptype())
   ```
   (ii)
   ```
   me = mynode()
   ptype = myptype()
   count = 16
   ..............
   call csend(20,msg,count,me-1,ptype)
   ...............
   ```

2. Write statements to:

   (i) receive a message of type 30, length 8 bytes. Choose your own buffer.

   (ii) send a message to node 1. The message is of type 10 and is in the integer array `value(1:2)` .

   (iii) the integer array `mesg(1:3)` contains a message. Send this from the resident node to the next node in sequence.

   (iv) receive a message of type 10 into an integer buffer of maximum length 64 words and then find out the number of integers sent as well as the node id of the process that sent the message.

2.2 The Node C Routine Library

A summary of the useful C library routines are given in Appendix A.

2.2.1 Subroutines

CRECV

receive a message from another process. CRECV blocks the calling process until the message is received.

use: `crecv(type,buf,len)`

There are two input parameters:

- `type` integer value referring to the type of the message to be received. This is the only qualifier that can be used to select a particular message,
- `len` the number of bytes in the buffer.

and one output parameter:

- `buf` is the buffer where the received message will be stored. Valid entries include any legal C data type, it should be a pointer.

notes: There are information routines, described below, that can be called to obtain extra information about a received message. There are also ways of using `type` to wait for the first message matching a set of message types.

example: `int msg[4]; crecv(20,msg,4*sizeof(int));`
will cause the receiving process to wait until it has received a message of type 20, which it stores in `msg`. The maximum length is 16 bytes.

CSEND

sends a message and waits for completion. CSEND blocks the calling process until the message is sent

use: `csend(type,count,len,node,ptype);`

All parameters are input values:

- `type` integer defining message type (`0<=type< 1,000,000,000`),
- `buf` the buffer which contains the message. Valid entries include any legal C data type, it should be a pointer. It is recommended that data types match in send and receive operations,
- `count` the number of bytes to be transmitted. There is no limit on the message length between nodes.,
- `node` the identifier of the destination node.
- `ptype` the identifier of the destination process.

notes: If the node parameter is `-1`, all processes currently executing whose `ptype` matches that given will receive the message. A successful send does not imply that a message has been received by a process.

example: `csend(40,count,4,mynode()-1,myptype());`

sends a 4 byte message in count to processor with number 1 less than my own, designating it to be of type 40. Processing is suspended whilst the message is being sent.

CSENDRECV Sends a message and waits for a reply.

use: `csendrecv(type,sbuf,scount,node,ptype,typesel,rbuf,rcount);`

`csendrecv()` can be used when a message is to be sent and a reply is expected.

Input parameters:

- `type` integer defining message type (`0<=type< 1,000,000,000`),
- `sbuf` the buffer which contains the message. Valid entries include any legal C data type, it should be a pointer. It is recommended that data types match in send and receive operations,
- `scount` integer defining the number of bytes to be transmitted. There is no limit on the message length between nodes,
- `node` the identifier of the destination node,
- `ptype` the identifier of the destination process.
- `typesel` integer value referring to the type of the message to be received. This is the only qualifier that can be used to select a particular message,
- `rcount` integer defining the number of bytes in the buffer.

Output parameters:

- `rbuf` is the buffer where the received message will be stored. Valid entries include any legal C type, it should be a pointer.

notes: The data in `sbuf` is not guaranteed to be sent before the reply begins to arrive in `rbuf`. It is therefore essential to use two *separate* buffers.

IRECV post a receive for a message from another process and return immediately. IRECV initiates a receive from another process, but does not wait until the message has arrived. Computation can therefore continue whilst a message is arriving.

use: `id = irecv(typesel,buf,count);`

There are two input parameters:

- `typesel` integer value referring to the type of the message to be received. This is the only qualifier that can be used to select a particular message,
- `count` the number of bytes in the buffer.

and one output parameter:

- `buf` is the buffer where the received message will be stored. Valid entries include any legal C type, it should be a pointer.

Return value:

- `id` integer message ID for use by `msgwait()`.

notes: There are information routines, described below, that can be called to obtain extra information about a received message. There are also ways of using `typesel` to select for the first message matching a set of message types. Before accessing the contents of `buf`, you should ensure that the message has completely arrived by calling `msgwait()`.

example:

```
{
  int msg[4], rid;
  rid = irecv(20,msg,16);
            :
            : compute compute compute...
            :
  msgwait(rid);
  printf("%8d\n", msg[1]);
}
```

will allow the receiving process to compute while a message of type 20 and length 16 bytes is arriving. Before accessing any elements of the receive buffer, we check that the receive is completed with `msgwait()`.

ISEND initiates a message send and returns immediately. ISEND sends a message but returns immediately, allowing computation to continue. A message ID is returned to allow the programmer to determine when the send is complete

use: `id = isend(type,buf,count,node,ptype);`

All parameters are input values:

- `type` integer defining message type (`0<=type< 1,000,000,000`),
- `buf` the buffer which contains the message. Valid entries include any legal C data type, it should be a pointer. It is recommended that data types match in send and receive operations,
- `count` the number of bytes to be transmitted. There is no limit on the message length between nodes,
- `node` the identifier of the destination node,
- `ptype` the identifier of the destination process.

Return value:

- `id` integer message ID for use by `msgwait()`.

notes: If the node parameter is **-1**, the message will be sent to all processes whose `ptype` matches that given, except for the sending process itself. You should check that the send has completed before accessing the contents of `buf`, otherwise you could corrupt the buffer being sent. You can do this with the `msgwait()` call.

example:

```
int sid=0, hib;
for (;;)
  {
    :
    : compute compute compute  ...
    :
    /* Wait for previous send to complete before assigning
       stuff to buffer
    */
    if (sid) {
      msgwait(sid);
      sid=0;
    }
    hib = 42;
    sid = isend(40,&hib,sizeof(hib),mynode()+1,myptype());
```

```
}
```

sends a 4 byte message in `hib` to the next node, designating it to be of type 40. Processing may continue whilst the message is being sent.

INFOCOUNT an integer function which returns the size (in bytes) of the message received after a receive operation completes.

use: `cnt = infocount()`

notes: All "info" routines must be placed immediately after the related receive call.

INFONODE an integer function which returns the node id the process that sent the latest message.

use: `node = infonode()`

notes: All "info" routines must be placed immediately after the related receive call.

INFOTYPE an integer function which returns the message type of the latest message.

use: `type = infotype()`

notes: All "info" routines must be placed immediately after the related receive call.

MCLOCK an integer function which returns the elapsed time in milliseconds to obtain extra information

use: `result = mclock()`

notes: Do not use the elapsed time given by this function to try and synchronise between different nodes or the host as each will have its own start time.

MSGWAIT causes process to wait until the asynchronous operation identified by `mid` completes.

use: `msgwait( mid );`

notes: other asynchronous utilities such as `msgdone()`, `msgignore()` and `msgmerge()` are available. See the reference manual for further details.

MYNODE returns an integer value which identifies the node on which the current process is executing.

use: `me = mynode();`

`me` the number of the node executing the process.

MYPTYPE an integer function which returns the current processes type.

use: `ptype = myptype();`

`ptype` contains the current process type.

NUMNODES an integer function which returns the number of nodes in the application.

use: `nnodes = numnodes();`

`nnodes` gives the number of nodes in the application.

2.2.2 Recap

1. Explain the action of each of the statements in the following program segments:

   (i)
   ```
   msg[0] = num;
   msg[1] = msglen;
   csend(10,msg,8,0,myptype());
   ```

   (ii)
   ```
   me = mynode();
   ptype = myptype();
   count = sizeof(msg);
   ............
   csend(20,msg,count,me+1,ptype);
   ............
   ```

2. Write statements to

   (i) receive a message of type 30, length 8 bytes. Choose your own buffer.

   (ii) send a message to node 1. The message is of type 10 and is in the array `int value[2]`.

   (iii) the array `int mesg[3]` contains a message. Send this from the resident node to the next node in sequence.

   (iv) receive a message of type 10 into an integer buffer of maximum length 64 words and then find out the number of integers sent as well as the node and process id of the process that sent the message.

Chapter 3

Program Examples and Exercises

Firstly, an example is presented for you to work through. Secondly, there are some exercises for you to attempt for yourself. Lastly, a rather longer problem is outlined with some hints, the coding is left as an exercise for the reader.

Refer to chapter 4 for instructions for editing, compiling and running programs on the Paragon.

3.1 A Programming Example

The problem: To find a root of the equation $f(x) = 0$ when the solution cannot be found directly. A well-known method of doing this is to find some interval $[a, b]$ in which $f(x)$ changes sign and to successively keep halving this interval (the method of bisection).

Start with $[a, b]$
Repeat:
    Set $m = (a + b)/2$
    if $f(m) \times f(b) < 0$ then
        Repeat with $[m, b] \ldots, a = m$
    else
        Repeat with $[a, m] \ldots, b = m$
    endif
until $|b - a| < \varepsilon$

If evaluating $f(x)$ is time-consuming, distributing this task over several nodes will produce a significant increase in speed. When we have several processors available, we can modify this method as we can determine where the sign changes in n sections, where n is the number of processors. For this problem, we shall use the algorithm as follows: Divide $[a, b]$ into $n - 1$ intervals and evaluate $f(x)$ at each boundary point. Except for node 0, send the value of $f(x)$ one processor to the "left". When a node discovers a sign change, that node then takes charge and broadcasts a new interval to all other nodes. The operation is repeated until the interval is small enough.

(Whilst not a perfect example of an efficient parallel program on the Paragon due to the large amount of message passing in relation to processing, it illustrates well most aspects of communication and processing on the machine.)

e.g. If $[a, b]$ is replaced with $[x_{n-1}, x_n]$, with nine nodes, the interval is reduced to $(b - a)/8$. We can now divide the tasks between the nodes. The process must:

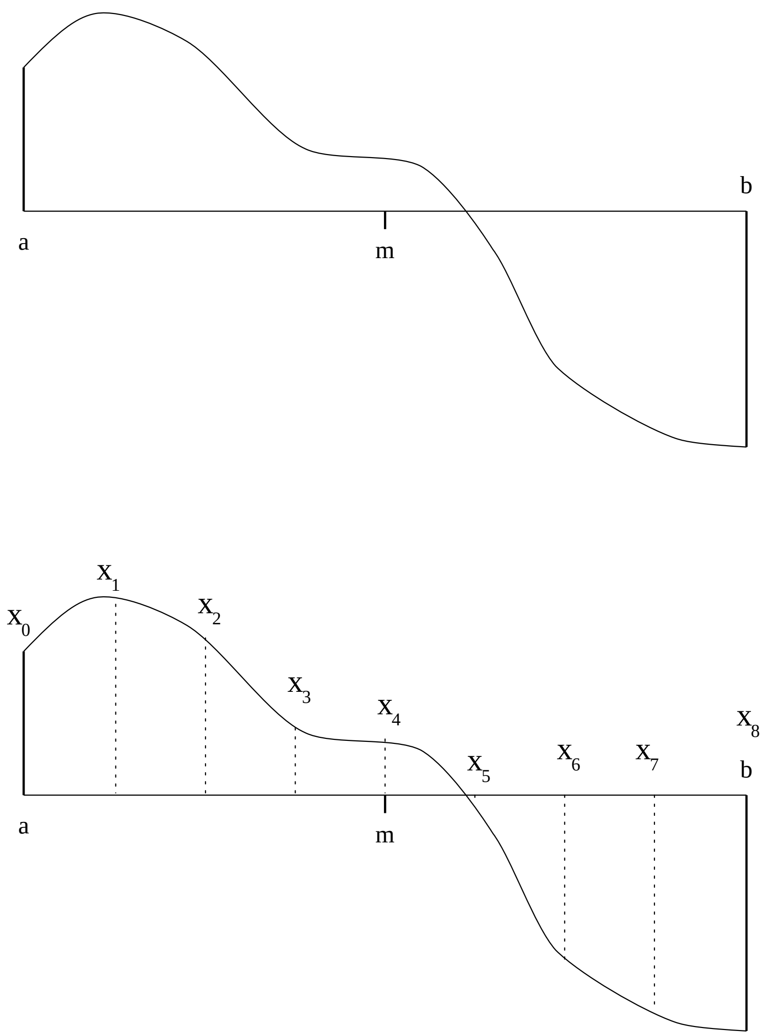

Figure 3.1: Method of bisection in parallel

(i) receive a value of x from the *master*,

(ii) calculate $f(x)$,

(iii) if not the last node, pass on the value of $f(x)$ to the node to my "left",

(iv) check for a sign change between the two values of $f(x)$,

(v) if a sign change exists, become the *master* process with the value of x,

(vi) go back to (i) and repeat as required.

From these observations we can construct the following program algorithm:

begin
  elect one processor to input the
    initial limits a,b and the desired accuracy eps;
  repeat (on all nodes)
    set step h between each node = (b-a)/(nnodes-1);
    broadcast a and h to all nodes;
    determine x := a + my_node_number*h
    calculate the value of $f(x)$;
    if (nodeno $\neq$ last node) send value of $f(x)$ to previous node;
    if (nodeno $\neq$ 0) receive mesg from next node;
    set $f(xn)$:=msg;
    if ($f(xn) \times f(x) < 0$) then (this node holds root)
      a = x
  until ($| b - a | < \varepsilon$) or (count>= 5);
  winner output root
end.

N.B. For the sake of simplicity, in this algorithm, it has been assumed that one and only one root will lie in the starting interval, hence exactly one sign change will be found. Dealing with multiple roots or no roots at all is left as an exercise for the reader. Here are the programs. There are two versions, a C version followed by a Fortran version. In this example, the program will find the root of the equation $x^3 - 0.7948x^2 - 9.3844x - 7.5896 = 0$ in the interval, say, $[1, 10]$ by using the initial values of 1, 10 and 0.00005.The answer is then 3.7948

Study these short programs in detail and determine how they work. It is a very useful exercise to type in one of these programs and to get it working on your local Paragon system. Chapter 4 explains how to compile and run programs.

The C version is:

```
/* Find the root of the equation f(x) = x^3 -0.7948x^2 -9.3844x-7.5896 */
#include <stdio.h>
#include <nx.h>

#define TASK_MSG 10
#define RESULT_MSG 20

typedef struct { float a; float h; } packet;

void main() {
    long me, ptype, nnodes;
    long idecide, ndecide, iwork;
    long sid, it;

    float b, x, eps;
    float fx, fxh;
    packet msg;

    me = mynode();
    ptype = myptype();
    nnodes = numnodes();

    if (me==0) { /* Initial starting values */
        printf(" Enter a, b and eps: \n");
        scanf("%f %f %f", &msg.a, &b, &eps);
        printf("Input values: %f %f %f   \n", msg.a, b, eps);

        msg.h = b-msg.a;
```

```
        csend( TASK_MSG, &eps, sizeof(eps), -1, ptype);
        idecide = 1;
    } else {
        crecv( TASK_MSG, &eps, sizeof(eps));
        idecide = 0;
    }

    for (it=0 ;it<100 ;it++ ) {
        if (idecide) { /* Send new ranges to all nodes */
            if ( msg.h < eps) msg.h = -1; /* Result has required accuracy */
            else msg.h = msg.h/(nnodes - 1);
            csend( TASK_MSG, &msg, sizeof(msg), -1, ptype);
            idecide = 0;
        }
        else  crecv(TASK_MSG, &msg, sizeof(msg));

        if (msg.h<0.0) break;

        x = msg.a + me*msg.h;
        fx = ((x - 0.7948)*x - 9.3844)*x - 7.5896;

        sid=0;
        if ( me != 0 ) sid = isend(RESULT_MSG, &fx, sizeof(fx), me-1, ptype);

        if ( me != nnodes-1 ) {
            crecv(RESULT_MSG, &fxh, sizeof(fxh));
            if ( idecide = (fx*fxh < 0.0) ) msg.a = x;
        }
        if (sid) msgwait(sid);

        /* Check for only one winner ! */
        ndecide = idecide;
        gisum(&ndecide, 1, &iwork);
        if (ndecide!=1) {
            if (me==0) fprintf(stderr, "ERROR: %d roots found\n",ndecide);
            if (idecide) fprintf(stderr, "%2d : x = %7.4f h = %g\n", me, x, msg.h);
            break;
        }
    }
  if (me==0) {
    if (msg.h<0.0) printf("Result is %7.4f\n",msg.a);
    else fprintf(stderr, "ERROR: no convergence!\n");
  }
}
```

And in FORTRAN

```
program root
include 'fnx.h'
integer tskmsg, rslmsg
integer me, ptype, nnodes
integer idecide, ndecide, iwork
integer sid, it
integer sz

real b, x, eps
real fx, fxh
real msg(0:1)

tskmsg = 10
rslmsg = 20

me = mynode()
ptype = myptype()
nnodes = numnodes()

sz = 4
if (me.eq.0) then
  write(*,700)
  read(*,*) a, b, eps
  write(*,*)a, b, eps
  call csend(tskmsg, eps, sz, -1, ptype)
  idecide = 1
  h = b-a
else
  call crecv(tskmsg, msg, sz)
  eps = msg(0)
  idecide = 0
endif

sz = 8
do 20 it=1,100
  if (idecide.ne.0) then
    if (h.lt.eps) then
      h = -1.0
    else
      h = h/(nnodes-1)
    endif

    msg(0) = a
    msg(1) = h
    call csend(tskmsg, msg, sz, -1, ptype)
    idecide = 0
  else
    call crecv(tskmsg, msg, sz)
    a = msg(0)
    h = msg(1)
  endif

  if (h.lt.0.0) goto 30

  x = a + me * h
  fx = ((x - 0.7948)*x - 9.3844)*x - 7.5896

  sid=0
  if (me.ne.0) sid = isend(rslmsg, fx, 4, me-1, ptype)

  if (me.ne.(nnodes-1)) then
    call crecv(rslmsg, fxh, 4)
    if ((fx*fxh).lt.0) then
      idecide = 1
      a = x
```

```
          endif
        endif

        if (sid.ne.0) call msgwait(sid)

c       check for one winner!
        ndecide = idecide
        call gisum( ndecide, 1, iwork)
        if (ndecide.ne.1) then
          if (me.eq.0) write(*, 800) ndecide
          goto 30
        endif
 20   continue
 30   continue
      if (me.eq.0) then
        if (h.lt.0.0) then
          write(*,900) a
        else
          write(*,1000)
        endif
      endif

 700  format('    Type in the values of a, b, and eps')
 800  format('    ERROR ',I8,' roots found')
 900  format('    Result is ',f7.4)
 1000 format('    ERROR: no convergence!')
      end
```

The implementation presented here solves the problem by allowing the "winner" at each iteration, *ie.,* the process with the root change, to broadcast the range values for the next iteration. Note that when each node must communicate its value **fx** to its neighbour, we have used **isend()** to *asynchronously* send the message to the neighbouring node.

The use of **isend()** means that the program will continue to the next line which will carry out a receive. *In other words*, we should always assume that we *must* post a receive before a message can be transmitted from the sending node. Deadlock can occur if a synchronous send is attempted without a corresponding receive being posted. The final call to **msgwait()** ensures that the **isend()** operation is completely finished before we continue to the next iteration.

In this example we have also put in an extra check to ensure that only one root was found. Finding multiple roots is left as an exercise.

3.2 Programming Exercises

Try the following three exercises for yourself. Your solutions should function correctly for any number of nodes.

3.2.1 Exercise

One node (for example node 0) elects to start. This node then sends a variable with the value *1* to the next node, which increments the value by *1* and passes it on. When each node has received the value and passed it on, the final value should arrive at the starting node and be printed out.

3.2.2 Exercise

Each node in turn is delegated to be a master process. When a node becomes a master, it sends a message to all other nodes and receives a reply from each. The master should display the message that it receives from each "slave". You should be able to observe that the incoming messages do not necessarily arrive in numerical order.

Specimen answers can be found in Appendix C.

3.2.3 Exercise

Each node generates a random double precision number. Write a program to find the maximum number from all nodes in $\log n$ steps where n is the number of nodes. Your program should also keep track of the node which has the original maximum value. (hint: this program will require that the number of nodes is a power of two, you should put an extra check for this at the beginning.)

3.3 An Extended Exercise

This is based on the Trapezium rule in numerical integration. The area beneath the curve $y = f(x)$ is divided up into strips, each of which is approximated by a trapezium. The sum of the areas of these trapezia gives the approximate value of the integral. e.g.

$$\int f(x)\, dx = h \left\{ \frac{f_0}{2} + f_1 + \ldots + f_{n-1} + \frac{f_n}{2} \right\}$$

If we have p processing nodes then we can split the integration into p separate parts:

$$\int_a^b f(x)\, dx = \sum_{j=0}^{p-1} \int_{a_j}^{b_j} f(x)\, dx$$

where the interval r, $[a_j, b_j]$ for each node is:

$$r = \frac{b-a}{p}.$$

It is left as an excercise to show that if we use this rule in each sub-integral with the same strip size h that the formula for the overall integral has the same form.

The accuracy of the result increases as h gets smaller, but this requires more and more function evaluations. Since function evaluation is time-consuming, distributing this task amongst the nodes can speed up the process considerably.

In this algorithm, the complete interval is divided equally between n nodes and each node does one section of the integral, thus giving a speed-up of roughly n times. Alternatively, we could use the extra computational power to allow a smaller value of h, thus increasing accuracy. For this example, assume the integrand is of the form $f(x) = c_0\sqrt{c_1 + c_2x^2}$ where the coefficients c_0, c_1 and c_2 are supplied by the user. Each node will be given the values of the coefficients together with its own local limits for the integration a, b. Each node will divide its own integral into 8 strips and apply the trapezium rule. The algorithm may be expressed as follows:

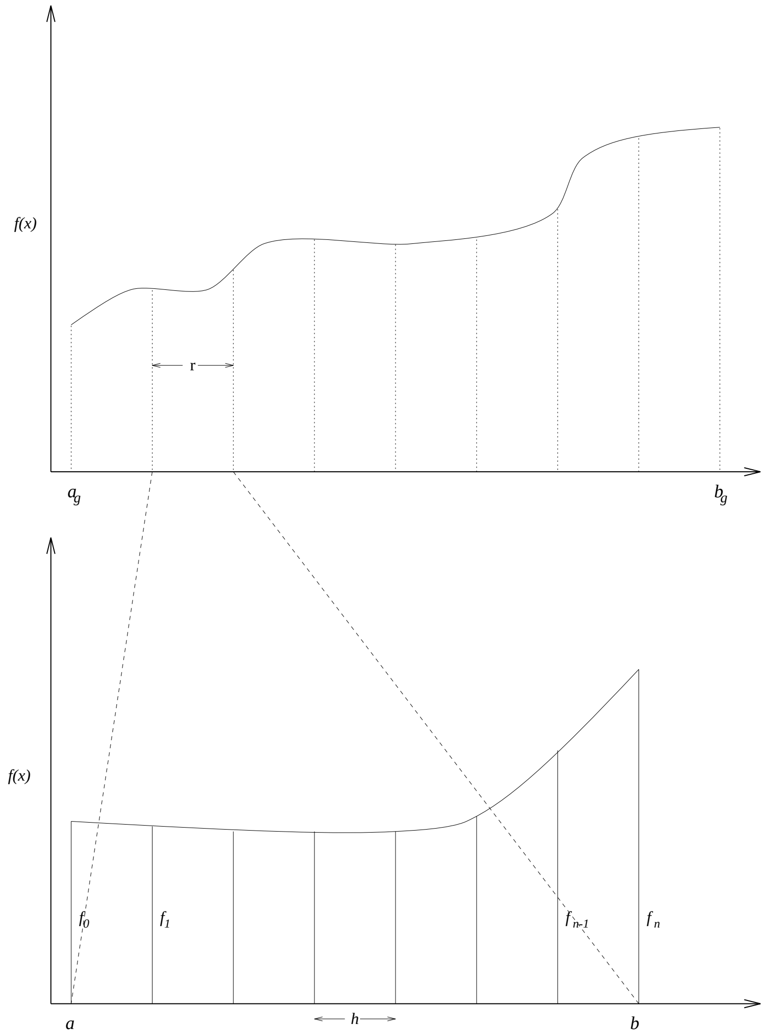

Figure 3.2: Applying the trapezium rule to each sub-interval

begin
  elect one processor to input the
    initial (global) limits a_g, b_g and the function coefficients;
  compute the domain r of the integral for each node $r = (b_g - a_g)/itnnodes$;
  broadcast global lower limit a_g, r and function coefficients;
  compute $a = a_g + my\_node\_num \times r$
  compute $b = a_g + (my\_node\_num + 1) \times r$
  compute $h = (b - a)/num\_strips$
  compute the integral
    sum $= \frac{f_0}{2} + f_1 + f_2 + \ldots + f_{n-1} + \frac{f_n}{2}$;
  $area$ = sum×h;
  compute global sum of $area$
end

Code this exercise and test your result on the example

$$f(x) = 2\sqrt{1-x^2}, \quad \int_{-1}^{1} f(x) = \pi,$$

Modify your program so that the node starts with 2 strips and computes the area, doubling the number of strips each time until two successive answers agree to within a preset value, *ie.,*

$$\mathrm{abs}\left(\frac{\mathrm{area} - \mathrm{areaprev}}{\mathrm{area}}\right) < \varepsilon,$$

Modify this example to compute π to as many decimal places as you can, use double precision arithmetic.

Now that you have written some programs, you will be eager to try them out on the Paragon. The following section explains how to enter the program files on your workstation and to edit them. Finally there are a subset of operating system commands which will allow you to run your programs and see how well you have got on.

Chapter 4

Creating, Compiling and Running Programs

There are four tasks to consider in this section:

- you must create source files of your programs,
- you must compile your source files,
- the executable versions must be linked to the appropriate subroutine libraries,
- you will need to know a few commands in order to run your programs.

4.1 Creating and Editing Source Code

The Unix operating system supports several standard editors. In this section we present the fundamentals of one of these – vi. There are also several editors associated with workstations running under window management software, but we shall not cover these here. Use your favorite editor if you have one.

Assuming that you have logged in successfully and received the "following is a small subset of editing commands but it should be sufficient for you to create and edit the programs which you write without spending a great amount of time learning to use the editor. N.B. In the text, <cr> represents the return key and <esc> the escape key.

To create a file

type vi<cr> to invoke the editor
I to insert text.
. . . .
. . . . lines of text.
<esc>

To write the display to a file

type **:w filename<cr>**

To edit a file

type **vi filename<cr>**

Then use the following commands to edit the displayed file:
To move the cursor around the screen

use the arrow keys or

<cr>	to start of next line,
-	to start of previous line.
Ctrl-F	moves the display forward one page,
Ctrl-W	moves the display back one page.
h	moves the cursor one character to the left,
j	moves the cursor one row down,
k	moves the cursor one row up,
l	moves the cursor one character to the right,

To delete text:

x	deletes character at the cursor position,
dw	deletes the word at the cursor position,
dd	deletes the current line,
D	deletes from the cursor to the end of the line.

To insert text:

i	insert text starting before the cursor position
I	insert text starting on the next line

To exit from vi:

ZZ	saves and exits from the editor.
:q!	exits without saving.

NB. It is strongly recommended that the *shift-lock* is not used. Vi is very case sensitive.

4.2 Compiling and Linking Program Files

This section covers the command syntax for compiling and linking on the Paragon. FORTRAN and C are the two languages installed on the Paragon. The following sections give the details for both languages.

4.2.1 Fortran Source Files

To compile and link a Paragon program source file `yyyy.f`, type:

```
if77 -o yyyy yyyy.f -nx
```

which results in a *node* executable file called `yyyy`.

4.2.2 C Source Files

To compile and link a Paragon C program, type:

```
icc -o yyyy yyyy.c -nx
```

The result is a *node* executable file called `yyyy`.

4.2.3 Executing programs

To execute a program on the Paragon you should normally only need to specify the number of nodes which your application requires. The user should type:

```
prog -sz <nnodes>
```

prog is the name of the program, and the `-sz` flag is used to specify the number of nodes required. For example:

```
a.out -sz 8
```

will run the program `a.out` on 8 processors. The Paragon operating system will choose the nodes for you so that they do not conflict with other users on the system.

Many other run-time options are available for advanced use. See the manual page for *application(1)* for more details.

This concludes the introduction to parallel processing on the Paragon. We hope that you have enjoyed working through this booklet and, more importantly, have become familiar with the basic ideas that have been introduced. Clearly there are aspects of using the Paragon which in an introduction cannot find a place. However, if this little booklet has been successful, you will be in a position to learn more about parallel computing. We hope that this is the case.

Appendix A

Summary of Key Fortran Routines

In the following , all variables are of type `integer*4`, except `buf`, which can be **any** valid Fortran type.

Each fortran module which uses these functions should include the NX header file as follows:

```
include 'fnx.h'
```

Routine	Call
CRECV	`call crecv(typesel,buf,count)`
CSEND	`call csend(type,buf,count,node,ptype)`
CSENDRECV	`call csendrecv(type,sbuf,scount,node,` `ptype,typesel,rbuf,rcount)`
IRECV	`mid = irecv(typesel,buf,count)`
ISEND	`mid = isend(type,buf,count,node,ptype)`
INFOCOUNT	`cnt = infocount()`
INFONODE	`node = infonode()`
INFOTYPE	`type = infotype()`
MCLOCK	`tick = mclock()`
MSGWAIT	`call msgwait(mid)`
MYNODE	`node = mynode()`
MYPTYPE	`ptype = myptype()`
NUMNODES	`nproc = numnodes()`

Appendix B

A Summary of the C Routines

The following is a synopsis of the prototype definitions for a selection of the most used NX routines. In order to use these routines you should have the line:

```
#include <nx.h>
```

at the beginning of each C module which uses them.

Routine	Call

CRECV	`void crecv(`
	`     long typesel,`
	`     char *buf,`
	`     long count );`

CSEND	`void csend(`
	`     long type,`
	`     char *buf,`
	`     long count,`
	`     long node,`
	`     long ptype );`

CSENDRECV	`long csendrecv(`
	`     long type,`
	`     char *sbuf,`
	`     long scount,`
	`     long node,`
	`     long ptype,`
	`     long typesel,`
	`     char *rbuf,`
	`     long rcount );`

IRECV	`irecv(`
	`     long typesel,`
	`     char *buf,`
	`     long count );`

ISEND	`long isend(`
	`     long type,`
	`     char *buf,`
	`     long count,`
	`     long node,`
	`     long ptype );`

INFOCOUNT	`long infocount(void);`
INFONODE	`long infonode(void);`
INFOTYPE	`long infotype(void);`
MSGWAIT	`void msgwait( long mid );`
MYNODE	`long mynode(void);`
MYPTYPE	`long myptype(void);`
NUMNODES	`long numnodes(void);`

Appendix C

Solutions to the Exercises

1.5.4

(i) False; in this context, type is a user-defined integer.

(ii) It can use the message type.

(iii) (a) the message type.
(b) where the message is located.
(c) the length of the message.
(d) which node the message is going to.
(e) which process on the node the message is going to.

2.1.2

1 (i) The first item in `msg` is assigned the value in count, the second item is assigned the value in `msglen`. A message is sent of type 10 consisting of the 8 byte contents of `msg` to node 0.

(ii) The node id on which the process is running is assigned to `me`, the process type is assigned to `ptype`. A message is sent of type 20, from buffer `msg` of size `count` bytes to the node specified by `me - 1` and process type `ptype`. The calling process waits until the message has been sent.

2 (i) `call crecv(30,buffer,8)`

(ii) `call csend(10,value,8,1,myptype())`

(iii) `call csend(10,mesg,12,mynode()+1,myptype())`

(iv)
```
      crecv(10,buffer,4*64)
      count = infocount()
      fnode = infonode()
      fptype = infoptype()
```
where `count`, `fnode`, `fptype` are declared as integers.

2.2.2

1 The results will be as in 2.1.2 question 1 above.

2 (i) `crecv(30,buffer,8);`

(ii) `csend(10,value,8,1,ptype);`

(iii) `csend(10,mesg,12,mynode()+1,ptype);`

(iv)

```
crecv(10,buffer,sizeof(int)*64);
count = infocount();
fnode = infonode();
fptype = infoptype();
```

where `count`, `fnode`, `fptype` are declared as integers.

3.2.1 – C solution

```
/* Send a message in a ring */
#include <stdio.h>
#include <nx.h>

void main() {
  long me = mynode(),
       ptype = myptype(),
       nnodes = numnodes();

  long msg_out = 1, msg_in = 0, sid = 0;

  if (me==0) sid = isend(10, &msg_out, sizeof(long), (me+1)%nnodes, ptype);

  crecv(10, &msg_in, sizeof(long));
  printf("%2d : received %2d\n", me, msg_in);
  msg_out = msg_in + msg_out;

  if (sid) { msgwait(sid); sid=0; }
  if (me!=0) csend(10, &msg_out, sizeof(long), (me+1)%nnodes, ptype);

  gsync();
  if (me==0) printf("Node 0 received ring message = %d\n", msg_in);
}
```

3.2.1 – Fortran solution

```
c     Send a message in a ring
      program ring

      include 'fnx.h'

      integer me, ptype, nnodes
      integer msg_in, msg_out, sid

      integer szint
      parameter (szint=4)

      me = mynode()
      ptype = myptype()
      nnodes = numnodes()

      msg_in = 0
      msg_out = 1
      sid = 0

      if (me.eq.0) then
        sid = isend(10, msg_out, szint, mod(me+1, nnodes), ptype)
      endif

      call crecv(10, msg_in, szint)
      msg_out = msg_in + msg_out
      write (*,800) me, msg_in

      if (sid.ne.0) then
        call msgwait(sid)
        sid = 0
      endif

      if (me.ne.0) then
        call csend(10, msg_out, szint, mod(me+1, nnodes), ptype)
      endif

      call gsync()

      if (me.eq.0) then
        write (*,700) msg_in
      endif

 700  format('    Node 0 received ring message = ',I)
 800  format('    ',I2,' : received ',I2)
      end
```

3.2.2 – C solution

```
/* Send broadcast from master node, receive back replies */
#include <stdio.h>
#include <nx.h>

void main() {
  long me = mynode(),
       ptype = myptype(),
       nnodes = numnodes();

  long msg_out = 1,
       msg_in = 0,
       i, master;

  for (master = 0; master < nnodes; master++) {
    if (me==master) {
      csend(10+master, &msg_out, sizeof(long), -1, ptype);
      for (i = 0; i<nnodes-1; i++) { /* receive nnodes-1 messages */
        crecv(10+master, &msg_in, sizeof(long));
        printf( "node %2d received %2d from node %2d\n",
                me, msg_in, infonode());
      }
    } else {
      crecv(10+master, &msg_in, sizeof(long));
      msg_out += msg_in;
      csend(10+master, &msg_out, sizeof(long), infonode(), ptype);
    }
  }
}
```

3.2.2 – FORTRAN solution

```
c     Send broadcast from master node, receive back replies
      program master_slave
      include 'fnx.h'

      integer me, ptype, nnodes
      integer szint
      parameter (szint=4)

      integer msg_out , msg_in , i, master

      me = mynode()
      ptype = myptype()
      nnodes = numnodes()

      msg_out = 1
      msg_in = 0

      do master = 0, nnodes-1
        if (me.eq.master) then
          call csend(10+master, msg_out, szint, -1, ptype)
c         receive nnodes-1 messages
          do i=0, nnodes-2
            call crecv(10+master, msg_in, szint)
            write (*,700) me, msg_in, infonode()
          enddo
        else
          call crecv(10+master, msg_in, szint)
          msg_out = msg_out + msg_in
          call csend(10+master, msg_out, szint, infonode(), ptype)
        endif
      enddo
 700  format( 'node ',I2,' received ',I2,' from node ',I2)
      end
```

3.2.3 – C solution

```
/* Find global maximum in log(n) steps, together with the owner */
#include <stdio.h>
#include <stdlib.h>
#include <nx.h>

typedef struct { double val; long node; } packet;

void main() {
  long me = mynode(),
       ptype = myptype(),
       nnodes = numnodes();
  unsigned long mask, out, p2=0;

  packet msg_in, msg_out;
  double rnum;

  /* First check that we're running on a square machine */
  for (mask=1; mask && p2==0; mask<<=1) {
    if ((mask&nnodes) && !(~mask&nnodes)) p2=1;
  }
  if (p2==0) {
    if (me==0)
      fprintf(stderr,"ERROR: number of nodes must be a power of 2!\n");
    exit(1);
  }

  /* Seed random generator and get a random number */
  srand48(me+(long)dclock());
  rnum = drand48();

  msg_out.val = rnum;
  msg_out.node = me;

  printf("%d : has %7.4f\n",me, rnum);
  gsync(); /* This isn't strictly necessary, but helps make the printout nice*/

  /* In log2(nnodes) steps, we find the maximum */
  out = 0;
  for (mask=1; mask<nnodes && !out; mask<<=1) {
    if ((me & mask)==mask) {
      /* printf("%2d : sending to %2d [0x%x 0x%x]\n", me, me^mask, me, mask);*/
      csend(10, &msg_out, sizeof(msg_out), me^mask, ptype);
      out = 1; /* Drop out if we've taken part already */
    } else {
      crecv(10, &msg_in, sizeof(msg_in));
      if (msg_in.val > msg_out.val) msg_out = msg_in;
    }
  }

  if (me==0)
    printf("...and the winner is... node %2d with %7.4f\n",
           msg_out.node, msg_out.val);

  gsync(); /* we get stuck here if someone is stuck in a crecv */
}
```