JOINT INSTITUTE FOR COMPUTATIONAL SCIENCE

IPSC

A Beginner's Guide to the Intel iPSC/860

Siddharthan Ramachandramurthi
104 South College
University of Tennessee
Knoxville, TN 37996–1301, USA

Last Revision: February 22, 1995
Author's Email Address: `jics@cs.utk.edu`

Copyright (C) 1991, 1992, 1993, 1994, 1995 by the Computational Science Education Project

This electronic book is copyrighted, and protected by the copyright laws of the United States. This (and all associated documents in the system) must contain the above copyright notice. If this electronic book is used anywhere other than the project's original system, CSEP must be notified in writing (email is acceptable) and the copyright notice must remain intact.

1 Introduction

This guide is intended for those who are learning to use the Intel iPSC/860 parallel supercomputer (such as the one at the Oak Ridge National Laboratory). Our goal is to present some of the salient features and capabilities of the iPSC/860 in a concise manner and not to overwhelm the reader with details. However, we have included some of the details that are essential for beginners to get their first program to compile successfully. Programming tools and sources for additional information are also discussed.

The hardware and/or software configuration tend to vary with site. Please consult your local system administrator for site-specific information.

It is assumed that most users will program either in C or in FORTRAN. All illustrative examples are presented in C. Everything we say about C programs usually applies to FORTRAN programs as well.

The rest of this guide is organized as follows. Section 2 gives an overview of the iPSC/860. Section 3 is about accessing the iPSC/860. Issues specific to programming on the iPSC/860 are discussed in Section 4. Section 5 surveys system commands and Section 6 describes some useful system calls. Sections 7 and 8 discuss interprocessor communication by message passing. Files are discussed in Section 9. Section 10 surveys some of the software tools and libraries available. Sources for more information are listed in Section 11.

2 iPSC/860 System Overview

We start with a brief overview of the iPSC/860 system hardware and software.

2.1 Processors

The iPSC/860 is a high performance parallel computer system. The processing power of the iPSC/860 comes from its processing nodes. Each node in the iPSC/860 is either a CX or an RX processor. Each CX processor may also be accompanied by an SX scalar processor and/or a VX vector processor. Every iPSC/860 system contains at least one RX node. The CX node is based on the Intel386 microprocessor. An RX node consists of an Intel i860 microprocessor capable of a peak performance of 80 MFLOPS. The i860 has multiple arithmetic units: an integer unit, a floating point adder and a floating point multiplier.

The iPSC/860 has a System Resource Manager (SRM) known as the host which serves as a front-end to the nodes in the hypercube. Often, one or more workstations are also configured to serve as hosts to the iPSC/860. These workstations are connected to the iPSC/860 by Ethernet.

2.2 The Interconnection Network

The processing nodes of the iPSC/860 are interconnected in a hypercube architecture. In a hypercube of dimension n, each node has n neighbors and the total number of nodes in the hypercube is 2^n. For example, the machine at Oak Ridge National Laboratory (ORNL) has $2^7 = 128$ RX nodes. The nodes of a hypercube are assigned unique addresses so that the address of any two neighbors differs in only one binary digit. The maximum data transfer rate between adjacent nodes on the hypercube is 20Mbps.

2.3 Memory

The iPSC/860 has a distributed memory architecture. This means that each processor has its own primary memory. The size of memory varies with location. For instance, the machine at ORNL has 8 megabytes of memory at each node. Some of this memory is used by the operating system for internal purposes. Secondary storage is usually available in the form of disks in the Concurrent File System (CFS).

2.4 Operating System

The iPSC/860 is a multi-user, multi-programming system. Each node on the iPSC/860 runs the NX/860 operating system. The host runs the UNIX operating system.

The iPSC runtime environment consists of a set of system commands that you can use at the shell prompt from the host and a set of system calls that you can use in node programs. It is important to note the difference between system commands and system calls. The system commands are part of the iPSC system UNIX extension. The system calls on the node are part of the NX/860 operating system.

Each user program runs on all the nodes of a hypercube of specified dimension. The user can specify a sub-hypercube of dimension ranging from 0 to the maximum available. A user can also run several different programs, each on a distinct hypercube.

3 Accessing the iPSC/860

The iPSC/860 can be accessed using `rlogin` or `telnet` to reach the host. If a remote host is available, then it is better to use it for most editing and compiling.

4 Programming on the iPSC/860

Programs written for the iPSC/860 are based on the control parallel model. In control parallel programs, each processor independently executes its version of the program on its own data and may communicate with other processors from time to time. The programmer controls the distribution of the data to the nodes and also specifies the task to be performed by each node.

There are two different methods for control parallel programming on the iPSC/860, namely, host/node programs and hostless programs. Hostless programming is the recommended method because it is sufficient to write a single program that will execute on all the processing nodes. In the host/node programming model, the host program is distinct from the node program.

4.1 Important Header File

As a general rule, it is a good practice to include the file `cube.h` in all C programs as follows:

```
#include <cube.h>
```

and the file `fcube.h` in FORTRAN programs as follows:

```
include 'fcube.h'
```

These files contain important definitions and declarations needed by iPSC/860 system calls.

4.2 Compiling

The compiler used varies according to the type of the node processors and whether you are on the SRM or on a remote host. The host program is compiled using `cc` or `f77`. All host programs must be compiled with the `-host` switch on a local host or the `-lhost` switch on a remote host.

RX node programs are compiled using the `icc` or `if77` cross-compilers. CX node programs are compiled on the local host using the native compilers `cc` and `f77`. On remote hosts, CX node programs are compiled using the cross compilers `rcc` and `rf77`. All node programs should be compiled with the `-node` switch.

Example
Suppose we have an iPSC/860 system consisting only of RX nodes. Let `myhost.c` and `mynode.c` be the host and node programs respectively. The host program can be compiled on the local host (SRM) as follows:

```
cc -host myhost.c -o myhost
```

The node program can be compiled on the local host or a workstation as follows:

```
icc -node myprog.c -o myprog
```

Consult your system administrator for details of your local configuration. For more information on compiling, see the section on System Commands in the iPSC/860 User's Guide.

4.3 Input and Output

Here we discuss how a program can communicate with the external world. On sequential computers, input can be read from the keyboard and output written to the screen. However, on the iPSC/860, since there are several processes all executing in parallel, we must do things differently.

The first difference is that the nodes cannot read input from the keyboard. In particular, C programs cannot use the `scanf` function to read input. We describe two different ways of providing input to the program.

Passing the Input as Arguments to `Main`

In this case, all the nodes would automatically receive the command line arguments to the program.

Example
The program `myprog.c` requires two integers as input and looks as follows:

```
#include <cube.h>
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[];
{
   int x, y;
   sscanf(argv[1], "%d", &x);
   sscanf(argv[2], "%d", &y);
   ...
   ...
}
```

Suppose the object program `myprog` is loaded on the nodes of the hypercube along with two input parameters with the following command:

```
load myprog 7 12
```

`myprog` assigns the value 7 to `x` and 12 to `y`.

Reading the Input from a File

In this case, the programmer has the option of making either one node read the input from the file and broadcast it to all the others or have each node read its input from a file. Files are discussed in Section 8.

Writing to the Standard Output

`printf` statements can be used in C programs to write to the standard output. However, when several nodes are executing processes simultaneously, the sequence of execution of print statements is not easily predictable. Thus, using `printf` may be unsuitable for some applications. In such cases, files could be used for output.

Similar restrictions apply to FORTRAN programs too.

5 iPSC/860 System Commands

The first step before loading programs onto nodes is to allocate a hypercube.

5.1 Allocating and Releasing Hypercubes

The command `getcube` at the shell prompt will allocate a hypercube of the specified dimension. On the iPSC/860, each user can allocate more than one hypercube. Several users can use the iPSC/860 at the same time, each on their own hypercubes. Once allocated, each user gets exclusive use of his/her hypercubes and they are not available to other users until released. The `relcube` command releases a hypercube. It is a good practice to release all your cubes before exiting the system.

Each hypercube can be assigned a name or the default name is assumed. If you allocate more than one cube, then each cube should be assigned a distinct name. The last cube allocated is the current cube. The current cube can be changed using `attachcube`. `cubeinfo` returns information on allocated cubes.

5.2 Managing Processes

After allocating a hypercube, the next step is usually to load one or more processes onto the nodes and run them.

Every process executing on a node of a hypercube has a *pid*. Each RX node can execute only one process at a time, and its NX *pid* is 0.

The `load` command loads a program on the specified nodes of a hypercube. The `killcube` command kills one or more processes on a cube and also cleans up after a process (for example, by getting rid of pending messages). Therefore, issuing a `killcube` before each `relcube` is a good practice. `load -H` will load the program but not start it until the `startcube` command is issued. `waitcube` waits until one or more processes have completed.

The following example shows how to use some of the system commands mentioned above.

Example

The following program is called `hello.c`:

```
#include <cube.h>
#include <stdio.h>
main()
{
    printf("Hello World!  --from Node-%d\n", mynode());
}
```

The program is then compiled and the executable is called `hello`. We allocate a hypercube of four nodes and load `hello` on all four nodes. When all the processes terminate, we kill the cube and then release it. The commands and their output look as follows. Note the sequence of execution of the `printf` statement by the nodes.

```
> getcube -t4
getcube successful:  ...
> load hello
Hello World!  --from Node-0
Hello World!  --from Node-1
Hello World!  --from Node-3
Hello World!  --from Node-2
> killcube
> relcube
relcube released 1 cube
```

5.3 Using the Node Shell

The `nsh -s` command loads the C shell onto node 0 of the current hypercube. In order to use `nsh`, the hypercube must have at least four nodes. The shell process on node 0 then manages the distribution of processes created under the C shell to the other nodes of the cube. The `exit` command returns you to the UNIX shell from the node's C shell.

nsh uses three special files:

```
.login.ipsc
.cshrc.ipsc
.logout.ipsc
```

You should create these files in your home directory. These files are used in the same way that UNIX uses the `.login`, `.cshrc`, and `.logout` files.

The node shell is typically used only when using the Concurrent File System (CFS) of the iPSC/860 (see Section 9).

6 The iPSC/860 System Calls

iPSC system calls are available that allow a node process to obtain information, control processes, send and receive messages, redirect input and output, handle errors and exceptions and many other functions. Node programs can also issue UNIX system calls.

Here we briefly survey some of the more frequently used calls. System calls for message passing are discussed in Section 7.

6.1 System Calls for Information

There are several system calls that provide information about a hypercube or a node.

Associated with each node of a hypercube is a logical node number which ranges from 0 to n-1 where n is the number of nodes in the hypercube. `mynode()` returns the number of the node on which the calling process is executing. `numnodes()` returns the number of nodes in the current hypercube and `nodedim()` returns the dimension of the current hypercube.

In many of the system calls, a node number, a pid or a message type have to be specified. `-1` is the wild card for system calls. It can be used to specify all the nodes, all the processes, or all message types.

6.2 Controlling Processes

`load()` loads a program on the specified nodes of a hypercube. This call has the same function as the `load` system command. Note that on the iPSC/860, each node can only execute one process at a time.

The two calls `waitall()` and `waitone()` can be used to synchronize node processes. `waitall()` blocks further execution of the process until the processes on all the specified nodes have completed. On the contrary, `waitone()` returns after one process in the specified set completes. Beware that a node process waiting for its own completion will hang indefinitely.

When a node program executes a `fork()`, a new process is created on an available node of the hypercube. An available node is one that does not have any process running on it. If a process has been loaded with `load()` and has terminated, then that node is unavailable until it has been waited on with the `waitcube` command, or the `waitall()` or `waitone()` calls.

A process on a specified node can be killed with the `killproc()` call. `killproc()` does not flush any messages pending for the killed process. The `killcube()` call kills all processes on all the nodes of the current hypercube and flushes pending messages.

6.3 Timing an Execution

The `hwclock()`, `dclock()` and `mclock()` system calls can be used to obtain timing information. `dclock()` is the most useful one. It reads the hardware counter and returns a double precision time in seconds. The actual time returned by these calls is usually unimportant. To time an interval in a program, the difference in the time between an initial call and final call to `dclock()` is used.

Example
Here we measure the time taken by a loop. Node 0 prints the time to the screen.

```
double start_time, end_time, elapsed_time;
int i, imax;
start_time = dclock();
for (i=0; i<imax; i++) {
    ...
}
end_time = dclock();
elapsed_time = end_time - start_time;
if (mynode() == 0)
printf("Time taken = %8.6lf\n", elapsed_time);
```

---

7 Interprocessor Communication

In order for multiple processors to cooperate on a single problem, it is necessary for them to communicate with each other. On the iPSC/860, the nodes do not share physical memory. Communication among processors is accomplished by passing messages. System calls are available both in C and in FORTRAN for a node to send messages to or receive messages from other nodes in the hypercube. Node processes send and receive messages through local buffers.

7.1 Message Characteristics

Messages are characterized by a length, a type and an id. The message length is measured in bytes. There is no limit on the length of messages between nodes. Zero-length messages are allowed.

The type of a message is an integer determined by the programmer, before it is sent. Normally, only message types in the range from `0` to `999999999` should be used. A process can use the type selection parameter to receive only messages of specific types.

The id is an identifier returned by the system. It can be used to check for the completion of asynchronous messages.

There are two different types of message passing: synchronous and asynchronous.

7.2 Synchronous Message Passing

The program that initiates the send (receive) waits until the send (receive) is complete. This waiting is called *blocking*. Completion of a send only means that the message buffer can be reused and does not imply that the message has been received. `csend()` and `crecv()` are the synchronous send and receive system calls.

Example
The following code synchronously sends the value of `pi` in a message of type 1 to process 0 on node 2.

```
double pi;
pi = 3.14159;
csend(1, &pi, sizeof(double), 2, 0);
```

7.3 Asynchronous Message Passing

`isend()` and `irecv()` are the asynchronous send and receive system calls. An asynchronous send or receive does not block. It returns a unique message id, which can be used later to check for completion of the message. To check on the completion of an asynchronous call, use the asynchronous `msgdone()` call, or the synchronous `msgwait()` call. Because the iPSC system has a limited number of message ids, it is a good practice to release ids that are no longer needed. `msgwait()`, `msgdone()` or `msgcancel()` can be used for this.

Example
The following code asynchronously sends the value of `pi` in a message of type 1 to process 0 on node 2.

```
double pi;
long msgid;
pi = 3.14159;
msgid = isend(1, &pi, sizeof(double), 2, 0);
...
...
msgwait(msgid);
```

7.4 Process Buffers vs. System Buffers

When a message is sent for which a receive has not yet been issued, the message goes into a system buffer if its length is 100 bytes or less. It is then referred to as a pending message. The message will be received only after a receive has been issued for it. If a receive had already been issued, then the message bypasses the system buffer and goes directly to the buffer of the receiving process. The `cprobe()` and `iprobe()` calls determine whether there is a message pending for a process. If the length of the message is more than 100 bytes, then it will remain in the sender's buffer and not go into a system buffer.

8 Efficient Communication

In order to obtain high performance on the iPSC/860, it is important to maximize the ratio of time spent on computation to the time spent on communication. Next we discuss some of the efficient ways of passing messages.

8.1 Size and Alignment of Buffers

Although the message passing calls will work with any size or alignment of buffers, the hardware works only with well-aligned buffers. Therefore, the software must copy misaligned messages into aligned buffers before sending, thereby degrading performance.

On the iPSC/860, it is a good practice to ensure that the size of each buffer is a multiple of four bytes. For this, in C, buffers should be declared with `int` or `long`, rather than `char` or `short`. Buffers allocated with `malloc` or `calloc` will be correctly aligned.

8.2 Short vs. Long Messages

Messages of up to 100 bytes in length are sent in a single step. Longer messages are sent using a slower, three-step method. This difference in time can be significant for some applications. Therefore, it is often advantageous to send several short messages instead of one long message.

8.3 Broadcasting

In some applications, it might be necessary for a node to broadcast some information to all the other nodes in the hypercube. The most efficient way of accomplishing this is by specifying the wild card `-1` for the receiving node in the send call.

8.4 Global Communication Calls

Several global communication calls are available on the iPSC/860. These global operations are high-level constructs for communication among node processes and for some applications they are more efficient than the primitive message passing calls.

For instance, `gsync()` is the global synchronization operation (also known as a *barrier*). When a node process calls `gsync()`, it waits until all other nodes in the hypercube have also executed `gsync()` before continuing on with the process.

Global calls are available for several arithmetic and logic functions as well.

Example
`gdsum()` is used to sum a double precision number across all nodes.
Suppose `x` is an array that is evenly divided into blocks of m elements each and each node of a hypercube is assigned one block. The dot product `x.x` can be computed efficiently using `gdsum`. Here, `dummy` is a temporary variable used by each node in order to accumulate the contributions from the other nodes. When `gdsum` completes, `dot` contains the correct dot product. The following program is executed by each node of the hypercube.

```
double x[m], dot, dummy;
int i;

dot = 0;
for (i=0; i<m; i++)
dot = dot + x[i]*x[i];
gdsum(&dot, 1, &dummy);
```

8.5 Force Types

Force types are a fast way to send long messages. Message types in the range from `1073741824` to `1999999999` are special force types that bypass the normal flow control mechanisms and do not match the `-1` wildcard.

Messages of force types are discarded if no receive is posted at the time of their arrival at the destination. Therefore, a force type message should be sent only if it is certain that a receive has been issued for it. Force type messages are sent faster because the message avoids the system buffer and goes directly to the buffer of the receiving node.

9 File Systems on the iPSC/860

Usually, two different file systems are accessible to a node program.

9.1 The UNIX File System

This is the file system mounted on the SRM or the remote workstations and is accessible to all node programs. Accessing this file system is straightforward.

9.2 The Concurrent File System (CFS)

This is the secondary storage of the iPSC/860. The nodes use the CFS for high-speed simultaneous access to secondary storage. `/cfs` is the default directory for CFS. You should create a directory `/cfs/yourname` before using the CFS. This will be your CFS directory.

Files are accessed using the standard UNIX system calls in either C or FORTRAN. A node program can access a concurrent file in one of four I/O modes.

Mode 0: Individual file pointer(IFP). This is the default mode.
Mode 1: Unsynchronized, Common file pointer(CFP).
Mode 2: Synchronized, CFP, variable length.
Mode 3: Synchronized, CFP, fixed length.

The system call `setiomode()` is used to set a file I/O mode, and `iomode()` to determine a file's current mode.

10 Software Tools and Libraries

Several software tools and libraries are available to help develop efficient software for the iPSC/860. Here, we take a quick look at some of them.

10.1 The Interactive Parallel Debugger

The iPSC Interactive Parallel Debugger (IPD) is a symbolic debugger available for parallel C and FORTRAN programs on the iPSC/860. IPD has many of the features of sequential-program debuggers and also includes special functions for debugging parallel programs. For example, it is possible to set breakpoints for individual node processes, monitor message queues and display the process status table. More information on the IPD may be found in the iPSC manuals. On-line manual pages are also available for `ipd`.

10.2 PICL

A Portable Instrumented Communication Library (PICL) is fully implemented on the iPSC/860. PICL is a subroutine library developed at ORNL, that implements a generic message-passing interface on a variety of multiprocessors. Programs written using PICL routines instead of the native commands for interprocessor communication are portable in the sense that the source can be compiled on any machine on which the library has been implemented. Programs written using PICL routines will also produce timestamped trace data on interprocessor communication, processor busy/idle times and simple user-defined events if a few additional statements are added to the source code. A separate facility called ParaGraph can be used to graphically view the trace data.

You must link to the PICL libraries when compiling C or FORTRAN programs. For C, use the `-lcpicln` switch. For FORTRAN, use the `-lfpicln` switch.

Send the following message to `netlib@ornl.gov` for information on how you can obtain manuals for PICL as well as download and install PICL if necessary.

```
send index from picl
```

10.3 ParaGraph

ParaGraph is a graphical display system for visualizing the behavior and performance of parallel programs on message-passing multiprocessor architectures. It takes as input execution profile data provided by PICL.

Information on ParaGraph can be obtained by sending the following message to `netlib@ornl.gov`

```
send index from paragraph
```

10.4 BLAS

The Basic Linear Algebra Subroutines (BLAS) can be used both in C and in FORTRAN programs by compiling and linking the appropriate routines during compilation.

BLAS is also available from `netlib@ornl.gov`. For more information, send the message

```
send index from blas
```

11 Sources for Further Information

There is an abundance of information available on the iPSC/860 in the form of manuals and research reports. However, for the programmer, the most useful and easily accessible source of help is likely to be the on-line manual pages.

11.1 On-Line Manual Pages

The on-line manual pages for iPSC system commands and calls are usually located in the directory `/usr/local/man` on the SRM. You must add this directory to the MANPATH in your `.cshrc` file in order for the `man` command to find them.

The iPSC System Commands are located in the `man1` subdirectory. The iPSC Library Calls for C and FORTRAN77 are in the `man3` subdirectory. The `man` command will look in the `man1` directory first. Therefore, if you are only interested in the Library Calls, then you must specify that.

For example, `getcube` is available both as a system command and as a library call. If you just type `man getcube`, then only the system command manual will be displayed. In order to find the C and FORTRAN library call version of `getcube`, you should type `man 3 getcube`.

11.2 iPSC/860 System Manuals

The following two manuals contain most of the information that you may need:

1. iPSC/2 and iPSC/860 User's Guide, Intel Corporation, April 1991.
2. iPSC/2 and iPSC/860 Programmer's Reference Manual, Intel Corporation, April 1991.

11.3 Performance Reports on the iPSC/860

The performance of the `iPSC/860` has been studied by many researchers. Both message passing performance and the computational performance have been studied and some of the results are available in the following two reports:

1. Jack J. Dongarra, *"Performance of Various Computers Using Standard Linear Equations Software,"* Technical Report CS-89-85, Department of Computer Science, University of Tennessee, April 1994.
2. Tom H. Dunigan, *"Performance of the Intel iPSC/860 and NCube 6400 Hypercubes,"* Technical Report ORNL/TM-11790, Oak Ridge National Laboratory, March, 1991.

11.4 Manuals for Software Tools and Libraries

1. iPSC/2 and iPSC/860 Interactive Parallel Debugger Manual, Intel Corporation.
2. G. A. Geist, M. T. Heath, B. W. Peyton and P. H. Worley, *"A User's Guide to PICL - A Portable Instrumented Communication Library,"* Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, June, 1991.

11.5 Programming Examples

The Technical Publications Department of Intel's Supercomputing Systems Division has provided on-line versions of all the C examples in the iPSC/860 Programmer's Reference Manual. These examples are complete running programs and are available by `anonymous ftp` from `export.ssd.intel.com`. They are short and simple programs, seldom more than a page in length. FORTRAN examples will be available shortly. Beware that there are numerous examples and they might occupy a lot of space.

To obtain the examples, you must have Internet access. First, make an `ftp` connection, `login` as `anonymous` (please use your e-mail address as your password), go into binary mode, enter the directory `pub/R3.3-examples`, and transfer the file `Cexamples.tar.Z`. Here is a sample session:

```
> ftp export.ssd.intel.com
Connected to export.ssd.intel.com
```

```
220 export FTP server (SunOS 4.0) ready.
Name (export.ssd.intel.com:siddhart):  anonymous
331 Guest login ok, send ident as password.
Password:siddhart@cs.utk.edu
230 Guest login ok, access restrictions apply.
ftp> binary
200 Type set to I.
ftp> cd pub
250 CWD command successful.
ftp> cd R3.3-examples
250 CWD command successful.
ftp> get Cexamples.tar.Z
200 PORT command successful.
150 Binary data connection for Cexamples.tar.Z
(137.46.201.3,1303) (169190 bytes).
226 Binary Transfer complete.
local:  Cexamples.tar.Z remote:  Cexamples.tar.Z
32538 bytes received in .14 seconds (2.3e+02 Kbytes/s)
ftp> quit
221 Goodbye.
```

When the file is transferred, move it to the directory in which you want your examples to reside, `uncompress` the file, and then `tar` it as follows:

```
uncompress Cexamples.tar.Z
tar -xvf Cexamples.tar
```

A number of individual C programs will appear. Instructions are in the README files.

Acknowledgements

I thank Ruth Ann Manning and Steve Moulton for providing valuable information about the iPSC/860. Thanks also to Clay Breshears and Christian Halloy for their valuable comments and criticisms which greatly contributed to the improvement of the presentation of this guide.

References

[1] "iPSC/2 and iPSC/86 User's Guide," Intel Corporation, April 1991.

[2] "iPSC/2 and iPSC/86 Programmer's Reference Manual," Intel Corporation, April 1991.