Porting and Optimizing a Routing Library for C* on the iPSC/860

Chetan Uberoy
Department of Computer Science
Oregon State University
Corvallis, OR 97331

uberoyc@research.cs.orst.edu

A project report
submitted in partial fulfillment of
the requirements for the degree of
Master of Science

Major Professor : Dr. Michael J. Quinn
Minor Professor : Dr. Timothy Budd
Other Commitee Member : Dr. Prasad Tadepalli

April 2, 1993

Abstract

High level data-parallel languages are easy to use and shield the programmer from machine specific details. A simple and efficient way of providing an interface to such languages is to develop a machine-independent compiler and a routing library, which isolates the low-level machine dependent communication functions. The compiler translates the high-level language source code into C (or some other high-level sequential language) code. It also includes calls to the routing library routines whenever it comes across a statement in the source code, which requires a communication.

C* is a data-parallel language designed for Connection Machine computers by Thinking Machines corporation. A routing library for C* on the Intel Touchstone Delta was developed at the University of New Hampshire. This project dealt with porting this library to the Intel iPSC/860 system by making alterations to the library routines and the makefile. The original mesh-based communication routines were also converted to hypercube style communications. The assembly code generated for these hypercube routines was optimized. Various benchmark C* programs were written and the timings for these were recorded, and the speedup curves plotted.

Acknowledgments

I would like to thank my advisor, Dr. Mike Quinn, who guided me throughout the course of this project and patiently listened to the questions I asked him. His support, help and encouragement went a long way to make this project a success.

I would also like to thank Dr. Phil Hatcher of the University of New Hampshire who helped me in the initial and most difficult stage of this project.

Contents

Chapter 1
Introduction

1.1 Data-parallel programming and C*

In the data-parallel computing model, the programmer writes a program which is run in a lock-step synchronized fashion on different processors, each processor having some associated memory. By lock-step we mean that all processors must be executing the same program statement at any time during the program execution. This is also known as the SIMD (Single Instruction Multiple Data) style of parallel programming.

C* (pronounced 'sea-star'), is one such data-parallel language designed for Connection Machine computers by Thinking Machines Corporation. The major goal of developing a language like C* is to provide the programmer with a high -level, machine independent parallel language that is insulated from the details of the underlying hardware. C* can be efficiently implemented on a variety of multicomputer architectures and has the major advantage that it is an extension of ANSI C. The parallel extensions provided by C* are few in number, easy to learn, and do not significantly alter the underlying structure of the code.

1.2 What is a routing library ?

The task of providing a high-level parallel language interface for C* has been divided into two phases for reasons of portability and simplicity. The first phase is developing a compiler for the language which will translate the source code into C code, and the second phase is implementing a **routing library** which will take care of the communication between the processors. The compiler is, therefore, machine and architecture independent–it just generates calls to the routing library whenever it comes across a place in the C* program where the processors need to communicate. The routing library then implements these communications by making calls to the low-level machine dependent C functions that exist on the parallel machine.

1.3 The aim of this project

A compiler and routing library for C* were developed at the University of New Hampshire. The target architecture was the Intel Touchstone Delta system which is

a mesh topology. Therefore, the routing library routines were optimal for a mesh configuration of processors.

This project dealt with porting the routing library routines to the Intel iPSC/860, a hypercube topology, and changing some of the communication routines to hypercube style. It also involved optimizing the assembly code generated for these new routines. Various benchmark C* programs were written and the improvement in timings, achieved as a result of the new hypercube style routines and the assembly optimizations, was recorded.

Chapter 2
The iPSC/860 System

2.1 Architecture

In the iPSC/860 system, up to 128 processors (called nodes) can work concurrently on parts of a single problem. The user communicates with these nodes through a front end processor called the host. The host can be local, in which case it is also known as the System Resource Manager (SRM), or it can be a remote workstation [1]. See Figure 2.1(a).

The nodes on the iPSC/860 are based on the Intel i860™ microprocessor and are organized in a hypercube topology, as shown in Figure 2.1(b). These nodes are also called RX nodes. Each node has its own **Direct Connect Module (DCM)** which allows a message to be passed from one node to another without interrupting the intermediate CPUs. As a consequence, nearest neighbors in a cube (nodes directly linked to one another) have only slightly lower message latency than those that are not.

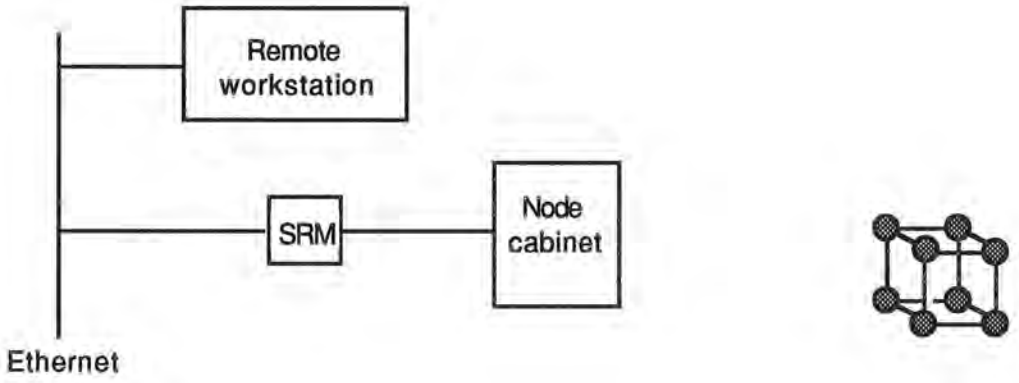


Figure 2.1 : The iPSC/860 system. (a) The SRM and a remote workstation connected to the node cabinet. (b) An 8-node hypercube.

The DCM has eight channels numbered 0 through 7. Channels 0-6 of each DCM are connected to the nearest neighbors. Channel 7 on node 0 of the cube is connected to the SRM, giving the SRM the same kind of access to the cube that the nodes have.

2.2 Allocating, releasing and loading a cube

Running a concurrent application involves allocating a cube of processors, loading the application on to the cube and finally releasing the cube. This section describes some system commands which facilitate these operations. For a complete reference, see [1].

getcube : This command "gets" or allocates a cube. The `-t` switch specifies a particular number or type of nodes. Some examples will illustrate this system command :

`getcube -t8rx`	–allocates a cube with 8 RX nodes.
`getcube -t4m8rx`	–4-node RX cube, each node having 8M memory.

Multiple cubes can also be allocated at the same time. Names must be assigned to the cubes in order to do this. The `-c` switch allows you to name a cube. A cube allocated without this switch is given the name `defaultname`. Multiple cubes cannot be allocated with the same name. Some examples :

`getcube -t2rx`	–this cube is named `defaultname`.
`getcube -c mycube -t4rx`	–this cube is called `mycube`.

Redirecting the output of `getcube` into a file as in

```
getcube > myfile
```

redirects the standard output and standard error of the node processes, too.

cubeinfo : Use this to get information about the *current* cube or all the allocated cubes. The *current* cube is the last one allocated. The `-a` switch gives information about all your cubes while the `-s` switch tells you about all the cubes allocated on the iPSC system belonging to that SRM. This command, when used without any switches, gives the details of the *current* cube.

relcube : Releases one or more cubes allocated by the user. Without any switches, it releases the current cube. The `-a` switch releases all your cubes. Used with the `-c` option, this command releases the named cube. For example,

```
relcube -c mycube      –will release the cube with the name mycube.
```

attachcube : Changes the *current* cube. Used without a switch, it makes the cube with name `defaultname` the *current* cube. The `-c` switch can be used to make a particular cube the *current* one.

load : Loads the program on a cube. Some examples will illustrate its use :

```
load myprogram          –loads myprogram on each node of the current
                         cube.
load -c mycube node     –loads executable node on mycube.
```

Command line arguments may also be passed to the node program by specifying them after the name of the executable. A host program may be used alternatively to load a node program on a cube.

killcube : The `killcube` command kills the processes running on a cube. The `-c` switch works in its usual way for this command, too.

rebootcube : Performs a limited system reboot. Should only be used when there is a problem with the host daemons.

2.3 Using the `icc` and `cc` compilers

The `icc` system command is used to compile a node program on the iPSC/860 system. Besides the usual `cc` options and switches, the switch `-node` must be used with `icc` to compile a node C program. The `cc` compiler when used with a `-host` switch compiles a host C program.

The `icc` command produces optimized code when the `-O` switch is used. The levels of optimization can be selected by specifying a number from 1 to 4 (4 being the

maximum) after the switch. The `-Mvect` switch can also be used for code optimization [1].

2.3.1 Example

An example of a script for compiling a host program called `host_prog.c` and a node program called `node_prog.c` is illustrated below :

```
cc -host -O4 -Mvect -o host_prog host_prog.c
icc -node -O4 -Mvect -o node_prog node_prog.c
```

The executables are called `host_prog` and `node_prog` respectively.

2.4 iPSC system calls

iPSC system calls provide the basic functions for cube management, message passing and performing I/O from the node processes. These calls can be used in either host, or node programs, or both. The file `cube.h` must be included in any program making a system call. A brief description of the important system calls follows. See [1] for a complete reference.

`mynode()` : Returns the number of the node on which the program is executing.

`nodedim()` : Gives the dimension of the cube.

`numnodes()` : Returns the number of nodes in the cube.

`crecv( int MSG_TYPE, char *rec_buf, int sizeof(rec_buf) )` : Receives a message synchronously (the program waits till the message is received) into `rec_buf`.

`csend( int MSG_TYPE, char *send_buf, int sizeof(send_buf), DEST_NODE, 0 )` : Synchronous (waits till the message has left the sending process) send to processor `DEST_NODE`. The last parameter, which represents the process ID, is always 0 on an iPSC/860 because all RX nodes are single process nodes.

`gcol(), gcolx(), gdhigh(), gdlow(), gdprod(), gdsum(), giand(), gihigh(), gilow(), gior(), giprod(), gisum(), gixor(), gland(), glxor(), glor(), gopf(), gsendx(), gshigh(), gslow(), gsprod(), gssum(), gsync()` : These are global

operations which can be used for communication among node processes. In some cases, they are more efficient than the primitive message passing calls. For example,

```
gdsum(double *partial_int, 1, double *work);
```

sums up the `partial_ints` on all processors of the cube and puts the result in each `partial_int`. This is a O(log N) operation where "N" is the number of processors in the cube. Similarly, `gihigh()` finds the global maximum of a given integer variable, and so on. `gsync()` performs a global synchronization.

`hwclock()` : This call is used for execution timing. It returns the value of the node's hardware counter as a 64-bit unsigned integer. The value returned by `hwclock` is a structure of type `esize_t` defined in `/usr/include/estat.h`. An example will illustrate how `hwclock` can be used to measure times.

```
esize_t    start_time, end_time, diff_time;
char       str_time[21];

hwclock(&start_time);
     •
     •
hwclock(&end_time);
diff_time = esub(end_time, start_time)/1000;
etos(diff_time, str_time);
printf("Time in seconds is %s\n", str_time);
```

In this example, `esub` is used to subtract one `esize_t` from another and `etos` is used to convert an `esize_t` to a string.

Chapter 3
An overview of C*

C* is a superset of ANSI C. In addition to all the features of C, C* provides new features that make data-parallel computing possible. Some of these features are :

- A method for describing size and shape of parallel data.
- New operators and expressions for parallel variables along with new meanings of standard operators.
- Methods for choosing parallel variables and specific points within the parallel variables, upon which operations are to performed.
- A parallel variable pointer.
- Changes to the way functions work to allow parallel variables to be passed as arguments.
- Library functions that allow communication among parallel variables.

The C* compiler developed at the University of New Hampshire supports all except the last feature. For a complete description of all features of C*, see [4].

3.1 Shapes

C* provides a new type specifier, **shape**, which is used as a template for distributing parallel data [4]. The **shape** keyword is used for declaration of a shape.

```
shape [4][8] shapeA;
```

declares a **shape** `shapeA` of two dimensions. The *rank* of a shape or a parallel variable is the number of dimensions in the **shape** declaration. A dimension is also referred to as an *axis*. In the above example, `shapeA` has 4 elements in axis 0 and 8 elements (or *positions*) in axis 1. A 4-by-8 **shape** is shown in Figure 3.1.

3.2 Parallel variables

A parallel variable, along with having a type and storage class, has a **shape**. The **shape** defines how many elements of the variable exist, and how they are organized.

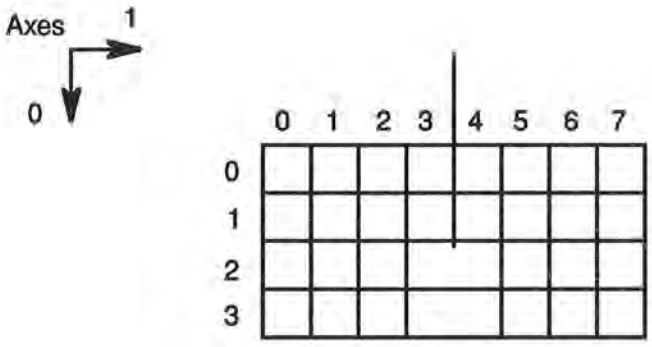


Figure 3.1 : A 4-by-8 shape (rank 2).

Each element in a parallel variable corresponds to a position in the **shape** and contains a single value for the parallel variable. For example,

```
shape[4][8] shapeA;
int: shapeA foo;
```

declares a parallel variable `foo` of type integer with 32 elements (as shown in Figure 3.1). The advantage of a parallel variable is that C* allows a program to carry out operations on all elements (or any subset of elements) of a parallel variable at the same time.

C* allows the use of parallel structures and arrays, too. An example of a parallel array declaration is :

```
shape [128] foo;
int: foo my_array[3];
```

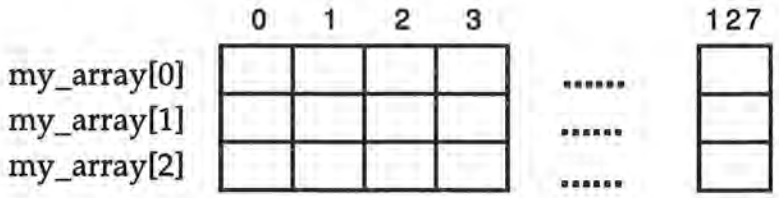


Figure 3.2 : A parallel array (3 parallel variables, each with 128 positions).

Parallel variables can be initialized by assigning a single value to all elements at the time of declaration. For example, in the following statement, the value 5 is assigned to each element of the parallel variable `psi` which belongs to shape `foo`:

```
int:foo psi = 5;
```

The intrinsic functions **positionsof**, **rankof** and **dimof** can be used to obtain information about parallel variables as well as shapes. They return the total number of elements, the rank, and the number of elements in a particular axis respectively. For example,

`positionsof(foo)`	returns the number of elements of `foo`.
`dimof(foo, 0)`	returns the number of elements in axis 0 of `foo`.

An individual element within a parallel variable can be described by its coordinates along the axes of the shape. The coordinates appear in brackets to the left of the variable name, starting with the coordinate for axis 0. These coordinates are also referred to as a *left index*. For example,

`[5][8]foo`	specifies the element in row 5, column 8.

Using this form, individual elements can be referred to in sequential code as well [4].

3.3 The "with" statement

A C* program can operate on parallel data from only one **shape** at a time. This **shape** is known as the *current* shape and is selected using the **with** statement as in :

```
shape[4][16]foo;
.
.
with(foo) {
```

Any statement in the **with** can operate on parallel variables of the *current* **shape** (the **shape** selected with the **with**). Nested **with** statements are also allowed (see [4]).

3.4 Operations on parallel variables

3.4.1 A scalar and a parallel operand

Consider the assignment :

```
p1 = s1;
```

where `p1` is a parallel variable and `s1` is a scalar. This is quite similar to initializing a parallel variable to a constant value. The value of `s1` is copied into all the elements of `p1`.

An assignment statement with a scalar on the left hand side and a parallel variable on the right hand side is not allowed. But, an explicit cast can be used to achieve this –the value assigned to the scalar can be any one of the values in the positions of the parallel variable.

`s1 = p1;`	is not allowed.
`s1 = (int)p1;`	results in an arbitrary value to be assigned to `s1`.

3.4.2 Two parallel operands

A statement like

```
p1 = p2;
```

where `p1` and `p2` both are parallel variables (they have to be of the same **shape** because a **with** selects only one **shape** to be the current one), assigns the value in each element of `p2` to the element of `p1` in the corresponding position (Figure 3.3).

Similarly, a conditional expression like

```
p1 >= p2
```

returns, for each element , 1 if it is greater than or equal to the corresponding element of `p2`, and 0 if it is not.

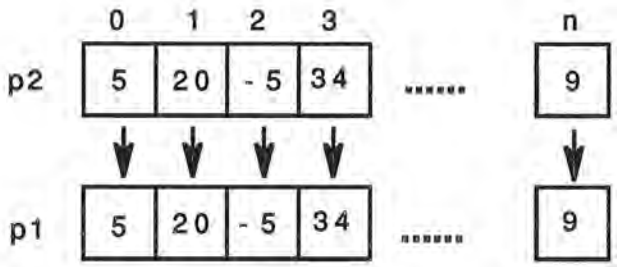


Figure 3.3 : Copying a parallel variable to another.

3.5 New C* operators

3.5.1 Reduction operators

Reduction operators *reduce* the values of all elements of a parallel variable to a single scalar value.

The += operator sums the value in each element of a parallel variable and adds the sum to the scalar variable on the left hand side. For example,

```
s1 += p1;
```

The += operator is also a unary operator. Operators like **-=, and=,** ^= and |= work in a similar way as the += operator.

The **<?=** and **>?=** operators find the minimum and maximum respectively, of all elements of a parallel variable. These operators can also be used with a parallel variable on the left hand side as in :

```
p1 <?= p2;
```

This assigns the lesser of `p1` and `p2` to `p1`, for every pair of corresponding elements of these parallel variables.

3.5.2 The %% operator

This operator is quite similar to the % operator in C and provides the modulus of its operands. The only difference is when one or both of the operand is negative, in which case the sign of the result depends upon the implementation of the operator [4].

3.5.3 The "bool" data type

This is very similar to the **Boolean** data type in Pascal. Usually, a variable of type **bool** occupies one bit of memory and therefore, stores a 1 (True) or 0 (False). Casting a larger data type variable to **bool** gives it a *logical* behavior. The operator **boolsizeof** introduced in C*, is used to obtain the exact size of a variable in units of **bools**.

3.6 Selecting positions in a parallel variable

3.6.1 The "where" statement

When a **with** statement is first entered, all positions of the shape are active; i.e., the statements operate on all elements of a parallel variable. The **where** statement is used to select a subset of these positions to remain active. For example,

```
with(shapeA)
    where(foo > 25) {
    .
    }
```

In this section of code, all statements inside the **where** block operate on only those positions of **shape** `shapeA` in which the value of `foo` is greater than 25 (Figure 3.4). In the figure, the active positions of `foo` are shown shaded. `p1` is another parallel variable of **shape** `shapeA` and all references to `p1` in the **where** body, operate on only the elements of `p1` *corresponding* to the active positions of `foo`, as shown in the figure.

The context set by the **where** remains in effect for any procedures called within its body. An exit from the body of the **where** resets the context to back to what it was before entering it (see [4]).

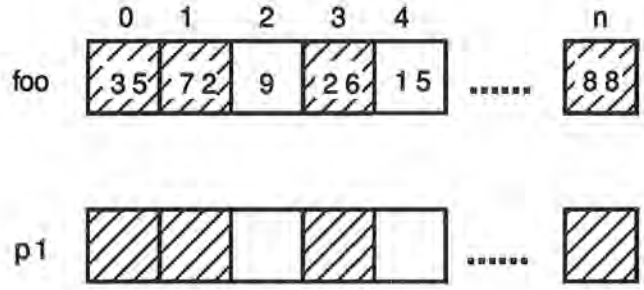


Figure 3.4 : Selecting positions with a **where**. All positions in which the values of elements of `foo` is greater than 25 are selected.

3.6.2 The "else" clause

The **else** following a **where** inverts the active/inactive status of all the elements. That is, all positions which were inactive in the **where** now become active, and vice-versa.

3.7 An example C* program

A C* program for finding the sums of the *odd* and the *even* positions of a shape is illustrated below :

```
#define ODD 0
#define EVEN 1

shape [64]a;
int:a node_no;

main()
{
     int odd_sum = 0, even_sum = 0;
     with(a) {
          node_no = pcoord(0);
          where(node_no % 2)
               odd_sum += node_no;
          else
               even_sum += node_no;
     }
}
```

In this program, a is a template or **shape** for parallel data distribution with *rank* 1 and 64 *positions* in the 0th *axis*. node_no is a parallel variable of **shape** a (and type **int**), and thus has 64 *elements*. The **with** statement makes a, the *current* **shape**. Function call **pcoord(0)** returns the number of the position in the 0th axis and thus, the parallel variable node_no now contains the position numbers (the 1st position of node_no contains 1, the 63rd position contains 63, and so on). The **where** statement makes active all those positions for which node_no is odd. A *reduce* inside the **where** adds up all the odd position numbers and places the result in odd_sum. The **else** selects all those positions which were not selected in the **where**; i.e., all the even numbered positions. Again, the sum of all these active position numbers is placed in even_sum by virtue of the reduction operator +=.

Chapter 4
The routing library routines

This chapter briefly describes some of the routing library routines developed at the University of New Hampshire. These routines were written for a mesh architecture (Intel Touchstone Delta) originally, but included flags which allowed them to be used for a hypercube, too [5].

4.1 Mapping routines

4.1.1 Mesh to hypercube

A shape in C* has the topology of a mesh. Thus, a mapping from a mesh to a hypercube needs to be done when the communication routines are implemented on an iPSC/860. The library takes care of this by using Gray codes . A Gray code has the following property :

For any number *i*, *G(i)*, the gray code of *i*, differs from *G(i+1)* by exactly one bit.

As an example of this, consider the case of mapping a 2X4 mesh to an 8-node hypercube. Two bit positions are used for the column number and 1 bit for the row number of the mesh. The Gray codes for the column are {00, 01, 11, 10} and that for the row are {0, 1}. The 3 bit code for the mesh is formed by concatenating, for each processor, the Gray code for the column and the Gray code for the row. So, the processor in say, the 1st row and the 2nd column (assuming rows and columns are numbered starting from 0) will have the code 111. This means that the processor numbered 111 in the hypercube is responsible for row 1, column 2 in the mesh (Figure 4.1).

The file `set-up.c` contains the various routines for implementing the mapping. The function `CS__Setup`, which is one of the initialization functions, calculates the *row* and the *column* of the node (on which the program is executing) at runtime by using the array `CS__yarg` (in file `gray.h`), which contains the gray codes.

4.1.2 Shape to physical mesh

The next step involves mapping the **shape** positions to the physical mesh [5].

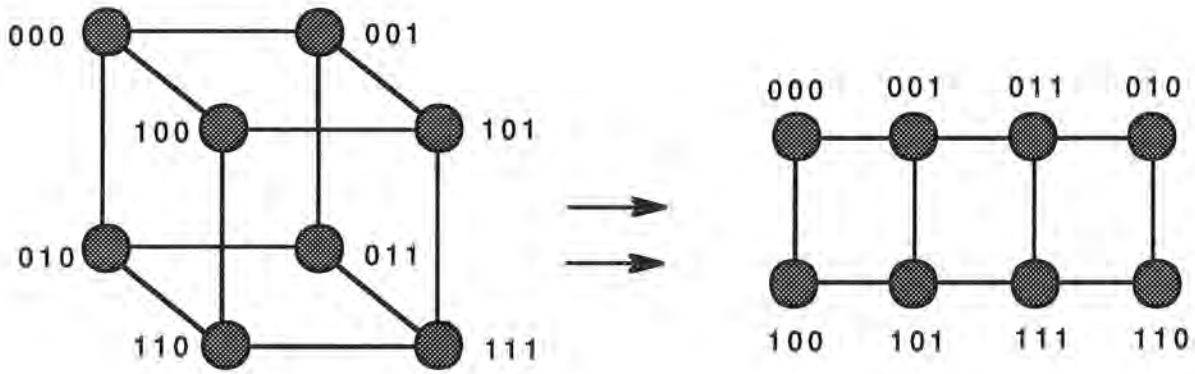


Figure 4.1 : Mapping a 2 X 4 mesh to an 8-node hypercube.

For a shape of rank one mapped to a two-dimensional mesh, the mesh is viewed as a one-dimensional array of processors and the data (parallel variables) are mapped to the array, as shown in Figure 4.2 [7].

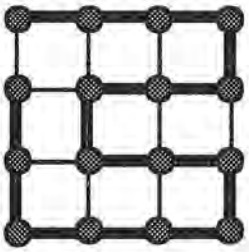


Figure 4.2 : Viewing a 2D mesh as a one dimensional array of processors.

Data are distributed evenly to the processors, and in case the number of positions in the **shape** is not a multiple of the number of processors, the *extra* positions are assigned to the lower numbered processors (Figure 4.3).

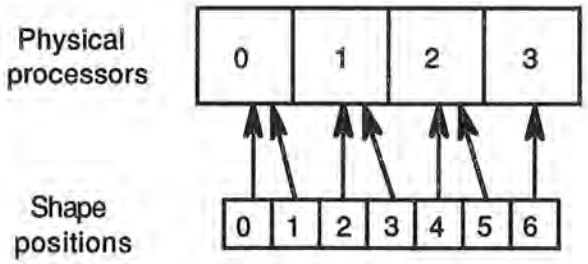


Figure 4.3 : Mapping *extra* positions of a **shape** to the physical processors.

Mapping a shape of rank two to a 2-D mesh is pretty straightforward with the rank 0 positions mapped to the rows, and the rank 1 positions mapped to the columns of the mesh. In case the positions of the shape do not divide evenly into the mesh processors, a procedure similar to the one illustrated in Figure 4.3 is used.

For any shape of rank greater than two, the first two axes are distributed as usual and all positions contained in the additional axes, are held locally.

This mapping is implemented in `mapping.c` in a function called `CS__MapAxes`. The function is invoked when a shape declaration is seen in the C* program.

4.2 Allocating shapes and parallel variables

Shapes and parallel variables are allocated at run-time in C*, which gives it greater flexibility because size and dimension of shapes can be based on the characteristics of the input data set.

4.2.1 Shape allocation

The compiler calls the function `CS__allocate_shape_array_1` in file `CS__alloc.c` when it sees a **shape** declaration in the source code. This function stores the rank, the length (number of positions) of each axis and the total number of positions in the shape, in a C **struct** called `CS__shape_struct` [5]. The shape positions are then mapped to the physical processor mesh by calling function `CS__MapAxes`, as mentioned earlier. For example, when the compiler sees the statement

```
shape  [8192]row;
```

it emits the following C code :

```
struct CS__shape_struct CS__temp_0;
CS__Shape row = &CS__temp_0;

/* The following function is executed before main () */

void _GLOBAL_I_SHAPE_warshall_cs_0(void)
{
     CS__SetUp();
     CS__allocate_shape_array_1 ((CS__Shape *) &row,
                                 (CS__Shape) &CS__temp_0,
                                 sizeof(row)/sizeof(CS__Shape), 1,
                                 (int) (8192));
}
```

Here, `CS__Shape` is a pointer to `CS__shape_struct`. All shape allocations are done by calling up a startup function (in this case `_GLOBAL_I_SHAPE_warshall_cs_0`) which is executed before function `main()`.

4.2.2 Parallel variable allocation

The function `CS__PvarAlloc` in `CS__alloc.c` takes care of parallel variable allocation. For parallel variables declared globally, this function is called from one of the startup functions (called before `main()`). The shape to which the parallel variable belongs, is passed as an argument to the `CS__PvarAlloc` function. The function stores this shape pointer, the `stride` and the `start` (to be used in VP emulation loops) in a **struct** called `CS__Pvar` (see [5] and [6]). It returns the pointer to the struct. For example, a declaration like

```
double: chunk  x;
```

causes

```
CS_Pvar x;
x = CS__PvarAlloc ((chunk), sizeof(double), CS__USERMEM_POOL);
```

to be emitted. For a parallel array declaration like

```
char: row  a[64];
```

the following code is emitted :

```
CS__Pvar a;
a = CS__PvarAlloc ((row), sizeof(char [64]), CS__USER_MEM_POOL);
```

4.3 Communication routines

The communication routines implemented in this routing library are *broadcast, reduce, scatter, gather, point to point, grid write, grid read, owner assign* and *sync*. This section briefly describes these routines along with the algorithms used to implement the communication.

4.3.1 Sync

Called with the name `CS__Sync`, this is just a barrier synchronization routine that returns only when all the processors have made the call to this routine.

4.3.2 Owner assign

Function `CS__OwnerAssign` is called when the compiler sees a statement like

```
[5]p9.mem2 = scalar;
```

No communication is required; only the scalar value needs to be assigned to the parallel variable on the processor holding the parallel variable. The compiler has the capability of combining any number of owner-assign operations into one library call by passing the number of assignments in a parameter to `CS__OwnerAssign`.

4.3.3 Broadcast

A *broadcast* is used to distribute a value residing on one processor to all the other processors. The compiler calls function `CS__Broadcast` in `broadcast.c` on seeing a statement like

```
scalar = [2]p7;
```

This is implemented by first assigning the value of the parallel variable to the local copy of `scalar`, and then broadcasting the value to all other processors so that they can update their values of `scalar`.

The algorithm used (see [6]) for performing the broadcast is pictorially represented in figures 4.2 (a) and 4.2 (b).

The algorithm first assigns each node with a new *pseudo-number*, obtained by rotating the source node's ID to zero and adjusting the others accordingly . A jump or span is calculated with which the physical dimension is to be divided. This is equal to the ceiling of log(N) where "N" is the number of nodes. The source node sends the value to its partner, located in the middle of the renumbered array. The span is divided in half, and each new subdivision then participates in the communication. This step is repeated till the span equals one. At each step, twice as many nodes as in the previous step, are involved in the communication.

In the first step, the value is broadcast to all the processors in the same row in the mesh as the source processor. In the next step, each of these processors (which now have a copy of the value to be broadcasted) broadcast along their respective columns.

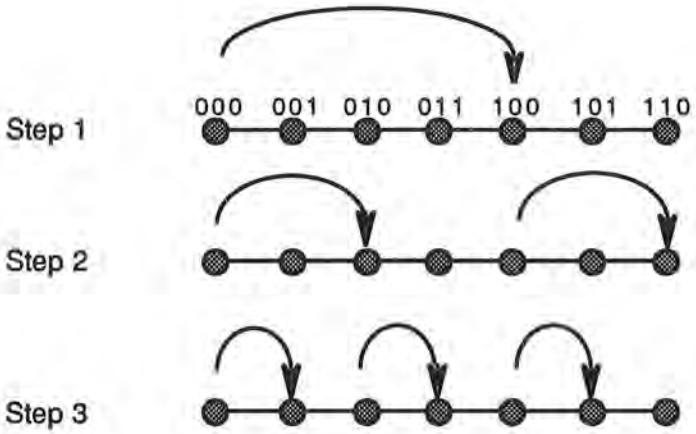


Figure 4.2 (a) : Broadcast algorithm on a non power-of-two number of processors with the source being node 0.

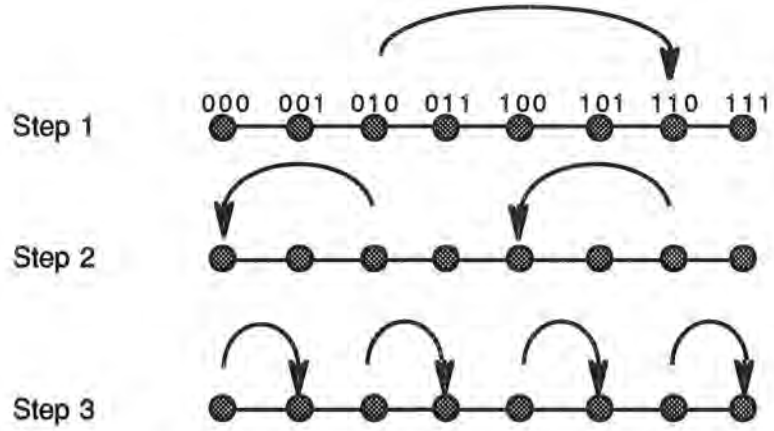


Figure 4.2 (b) : Broadcast algorithm with source node not being the 0th node (node 2 is the source here).

This algorithm requires a total of (ceiling(log R) + ceiling(log C)) communication steps where R is the number of rows, and C is the number of columns in the physical mesh.

4.3.4 Point to point

Direct assignment of one position of a parallel variable to another position of some parallel variable (both *positions* must be known at compile time), results in a call to function `CS__PointToPoint`. An example of such a statement is (see Figure 4.3) :

```
[15]p4 = [8]p6;
```

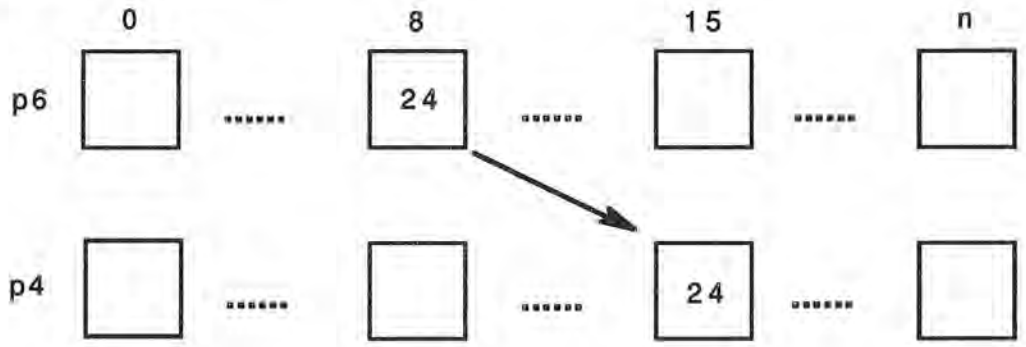


Figure 4.3 : Point to point communication.

The library determines the source and destination nodes of the call by calling function `CS__GetRowAndCol` in file `mapping.c`. The left indices of the source and destination positions of the parallel variables, are extracted from the parameters `dlindex[]` and `slindex[]`. The library then buffers the data at the source and sends it to the destination, while the destination receives the packet and moves it to the appropriate address in its local memory [5].

4.3.5 Grid write

When all outgoing values from a single processor are routed to a single destination processor known at compile time, and all incoming values to a processor are coming from a single processor, also known at compile time, the compiler emits a call to function `CS__GridWrite`. An example of such a statement is (Figure 4.4) :

```
[(pcoord(0) - 1) %% 8]p7 = p7;
```

In this statement, each element of the parallel variable `p7` writes its value to its immediate left neighbor. The `%%` operator is used so that element 0 writes to the rightmost element (wraparound). The algorithm used for the *grid write* operation

is summarized below. See [5] for more details.

(1) Separate the local data to be sent to other nodes from the data that will remain local.
(2) Determine the list of destinations and calculate how many incoming messages to expect.
(3) For each destination, collect the outgoing data destined for that node and send it.
(4) Alternate the reading of incoming messages with performing local work from the buffer receiving the data.

The *grid read* operation involves reading values from neighboring elements of the same parallel variable and storing it as your own value. The context of the source element (whether it is active or not) is ignored. The *grid read* operation is, therefore, very similar to the *grid write*, and is performed by making a call to function `CS__GridWrite` with a `ignore_context` flag. The appropriate values are copied from a temporary parallel variable (which is used to avoid overwriting positions' values before they have been read) into the destination parallel variable, after the read operation is completed. This avoids the step in which processors request data from the sources (see [5] for more details).

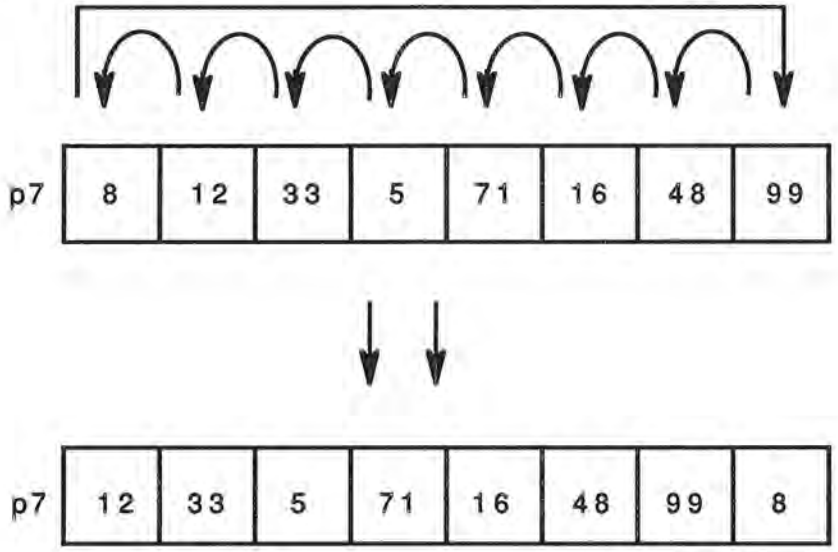


Figure 4.4 : Grid write - each position writes its value to its neighbor (left neighbor in this case).

4.3.5 Reduce

A *reduce* is performed on all or some elements of a parallel variable, the result of the operation (such as sum, subtraction, product, maximum or minimum) being stored in a scalar variable. For example, the statement

```
scalar += pl;
```

calculates the sum of all the active elements of `pl` and stores it in `scalar`.

The C* compiler performs the *local* reduction (on all elements of the parallel variable held on the node) before invoking function `CS__Reduce`, and provides the library with the scalar source of distribution rather than a parallel variable [6].

The algorithm for the reduction (see [6]) first calculates the largest power-of-two that fits within the actual processor array size. The processors which fall in this power-of-two are marked as *filled* and the others are marked *partially-filled*. In the first iteration, all *partially-filled* processors send their values to the *filled* ones. The *filled* processors are then divided in half, and they swap their values with their partners across the half. This division and swap is repeated, and in log(N) steps, where "N" is the greatest power-of-two, all the *filled* processors hold the desired result. They now relay this result to the *partially-filled* processors. An illustration of this algorithm for an array of 7 processors is shown in Figure 4.5.

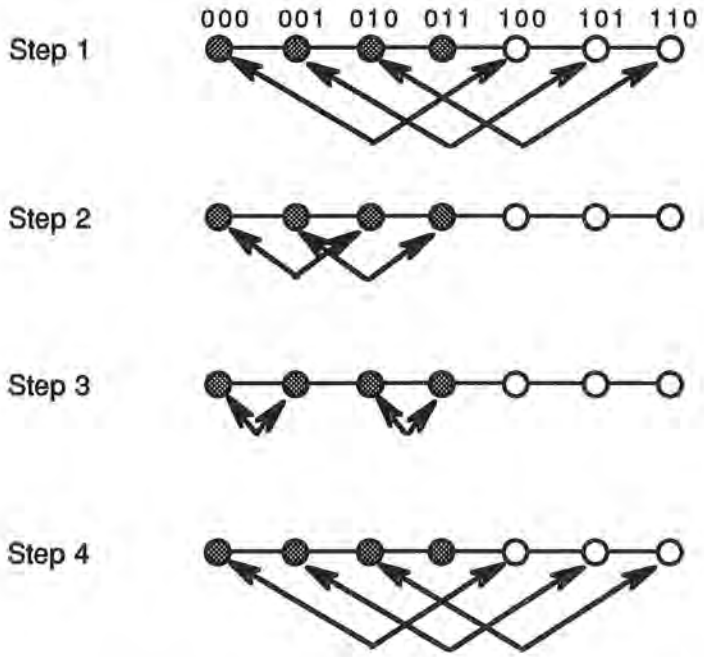


Figure 4.5 : Reduce algorithm–in the first step, the processors on the *filled* (with IDs less than the greatest power-of-two that fits in the processor array size) side exchange their values with those on the *partially-filled* side. Then, the *filled* processors swap their values with each other (steps 2 and 3). In the final step, the result of the *reduce* is sent to the *partially-filled* side. In this figure, the *filled* processors are the ones shown shaded.

4.3.6 Scatter and gather

Function `CS__Scatter` is called when values are to be distributed in an unpredictable pattern across the nodes. An example of C* code requiring a *scatter* is

```
[p8]p7 += p8;
```

The communication which takes place as a result of this statement is shown in Figure 4.6. 4.6(a) shows the values before, and 4.6(b) shows the values after the communication.

For a *scatter* operation, the context of the parallel index is examined to determine which positions are active and what the values are. These values are then used to calculate the destination nodes and offsets for the corresponding source values. Finally, the synchronized algorithm is executed and the values sent at the appropriate steps [6].

An example of a *gather* is :

```
p4 = [p2]p3;
```

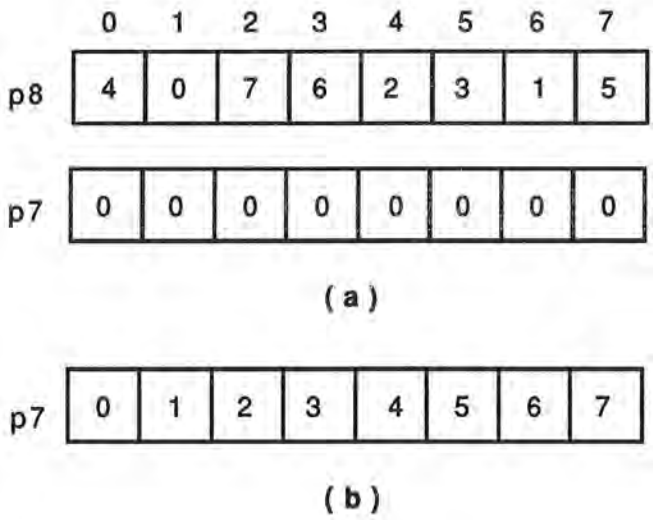


Figure 4.6 : Scatter–values of `p8` are added to values of `p7` indexed by `p8`. (a) shows the values in the parallel variables before the *scatter* and (b) shows the values in `p7` after the *scatter*.

The *gather* algorithm requires two *scatters*. First the requests are scattered to the nodes where the source values reside, and then these nodes scatter back the values.

The algorithm used for *scatter* and *gather* is identical to the *reduce* algorithm, except that in this case, packets are sent instead of a single value. At each step, the processors check the buffer and keep the values meant for them, and send the other values.

Chapter 5
Porting the library to the iPSC/860

Some changes had to be made to the existing library routines and the `makefile` to port them to the iPSC system. This chapter describes those changes.

5.1 Installing the library

The C* routing library and the associated files are installed in the directory `/usr/uberoyc/cstar` on the iPSC system. A layout of the files is illustrated in Figure 5.1.

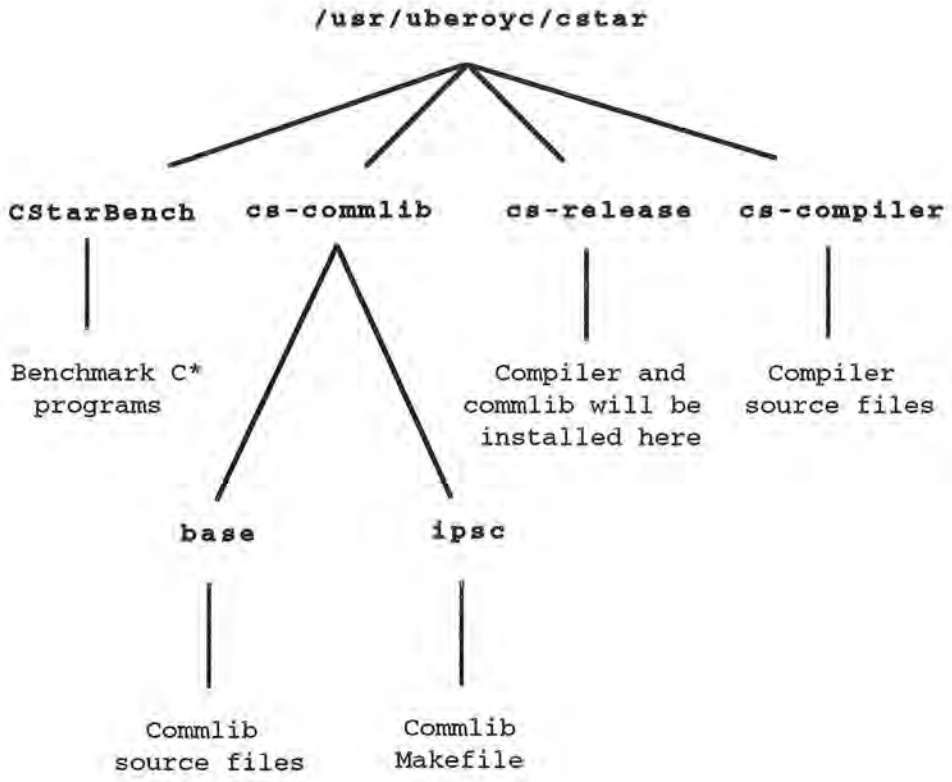


Figure 5.1 : Layout of the communication library files on the iPSC.

5.2 The `makefile`

The communication library `makefile` compiles the source files to produce object files. An executable for a C* program is created by first using the C* compiler to obtain C code. This is then compiled to produce an object file using the `icc` compiler, which is then linked with the communication library object files to produce the executable. This section gives a brief description of the communication library makefile.

The `icc` compiler is used to compile the library source files for the iPSC nodes. The `-node` switch is used for this purpose. The `-O4` and `-Mvect` switches, as discussed in Chapter 2, are used for optimizing the code. The `-DCS__IPSC` and `-DCS__RUNTIME_LIB` switches are required so that the code which is compiled, is suited to the iPSC system (The communication library routines are portable to other systems. For this purpose, the library source files contain many **#ifdef** pre-processor directives which filter out the code meant for other systems). After compiling the source files, the `makefile` links them to produce a communication test program called `test`.

The original `makefile` contained a `ranlib` command which created a library archive called `cs-commlib.a`. The UNIX system V release 3.2, however, does not support this command. Therefore, the original command for creating a library archive

```
make cs-commlib.a
```

issued from the command line, does not create a symbol table for the archive.

The `makefile` also provides the options of compressing and uncompressing the object files (`make small` and `make big`) and linting the C files. A table describing all the `makefile` options is shown in Figure 5.2.

5.3 Changing the library source files

The communication library makes use of system independent macros to implement the communication routines. The macros are implemented using the system calls supported by the particular system. Therefore, a few changes had to be made to these macro implementations to port the library to the iPSC/860. Some of the important changes made are described in this section.

Command	Description
make	Compile library source files into object files and produce a program "test".
make cs-commlib.a	Create library archive. The "ranlib" command is not available for this option.
make from_compiler_build	Creates library archive and copies .h files from source dir to "release" dir.
make cs-commlib.a lib	Removes old archive and creates it again.
make release install	Installs the archive in the "release" dir.
make small	Compresses all files in the directory.
make big	Uncompresses all files in the directory.

Figure 5.2 : The communication library `makefile` commands.

5.3.1 Mapping

Some iPSC system calls were added to the function `CS__SetUp` in `set-up.c` which, as mentioned earlier, maps the physical mesh to a hypercube. A section of the added code is shown below :

```
#elif defined(CS__IPSC) /* Preprocessor directive to filter the
                           iPSC code */

CS__nodenum = mynode();    /* Get the node number */
CS__mesh_size = numnodes();/* The total number of nodes */
CS__cube_dim = nodedim(); /* The cube dimension */
```

Using this information about the allocated cube, a mapping is carried out from the mesh to a hypercube as described in chapter 4.

5.3.2 Macro implementations

The file `macros.h` contains the implementation of message passing and other macros using system calls. The Intel Touchstone Delta, for which this routing library was originally written, has system calls which are exactly similar to the iPSC/860 calls. So all that needed to be changed to the message read, message write and most of the timing macros, was the addition of a logical OR preprocessor directive as :

```
#if defined (CS__DELTA) || defined(CS__IPSC)
```

One of the timing routines in the same file, which returns the time in milliseconds when passed the number of clock ticks elapsed, had to be changed for the iPSC/860 in the following manner :

```
#if defined(CS__IPSC)
#define CS__MSECS(units) \
     (double) (ediv(units, (long)10000)
#endif
```

In this routine, `units` is divided by 10000 to obtain the time in milliseconds. This is because the hardware count rate, defined in constant `HWHZ`, (returned by system call `hwclock()` used in the iPSC/860 timing routines) is 10000000 for RX nodes. To obtaine the time in milliseconds, you divide `units` by `HWHZ/1000` which equals 10000. See [1] for details.

5.3.3 Synchronization

The iPSC system call for global synchronization (`gsync`) is the same as that for the Intel Delta so a logical OR preprocessor directive, as shown previously, was all that was needed in the routine `CS__Sync` in file `sync.c`.

5.3.4 Timer and profiling routines

Again, these have the same implementation as for the Delta. The files `sys/types.h` and `estat.h`, needed to perform extended arithmetic with the timer values, have to be included in `timer.c`, `profile.c` and `profile1.c`, just like in the case of the Delta.

Chapter 6

Optimizing the library routines

The communication library routines, as mentioned earlier, were originally written for the Intel Touchstone Delta which is a mesh architecture. To achieve better timings, the communication routines were changed to hypercube style communications. This chapter provides a brief description of the algorithms used, and the changes made to the code.

All these new functions are called from within a preprocessor `#if defined(CS__IPSC)`, thus mantaining the portability of the library.

6.1 Reduce

The routine `HypercubeReduce` in file `reduce.c` implements a hypercube style *reduce* algorithm.

As described in chapter 4, a *reduce* reduces a set of parallel values to a *scalar*. The new hypercube style routine takes into account only the *active* elements of a parallel variable, and reduces them. The algorithm proceeds by repeatedly dividing the cube into subcubes. At each step, every node swaps its value with the corresponding node in the newly created subcube. At the end of log(N) steps, all nodes have the final *scalar* value. The implementation is described in Figure 6.1 (**D** denotes the cube dimension). See [3].

For each **i**, from **D** down to 1

Shift left 1 **i** times.

Calculate your **partner** by XORing your node number with the result of the shift left.

Send your value to your **partner**.

Receive **partner**'s value.

Perform the required operation on both (your and **partner**'s) values and store it as "your" value.

End For.

Figure 6.1 : Implementation of the hypercube *reduce* algorithm on a hypercube of dimension **D**.

The original *reduce* algorithm required (log(R') + 2) + (log(C') + 2) steps where R' is the greatest power-of-two which fits in R, the number of rows (when the mesh is mapped to the hypercube), and C' is the greatest power of two which fits in C, the number of columns. For a hypercube, R=R', C=C' and R*C = N, the total number of nodes. This means that a total of (log(N) + 4) steps are required to perform a *reduce* using the original *reduce* algorithm. The hypercube *reduce* algorithm, on the other hand, requires log(N) steps. Appendix A.1 shows the improvement in timings achieved for some C* programs by using the hypercube style *reduce.*

6.2 Broadcast

A *broadcast* is required when a single processor needs to communicate a set of values to all other processors.

The function `HypercubeBroadcast` in file `broadcast.c` implements a hypercube style broadcast. The algorithm for this implementation (see [3]) has the form of a binomial tree with the source node being the root of this tree. In the first step, only two processors–the source and one of its neighbors participate, with the source sending its value to the neighbor. In the next step, each of these two nodes send their values to two other nodes, thus involving a total of 4 nodes in this step. This process is repeated with the number of nodes participating being multiplied by 2 in each step. Hence, for a cube of dimension D, it takes D steps for the value to reach every node. This algorithm is illustrated in Figure 6.2.

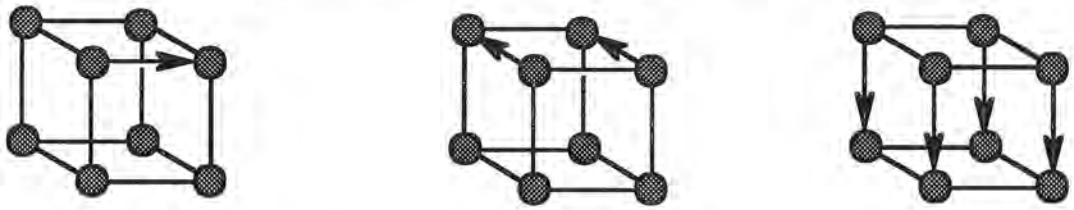


Figure 6.2 : Hypercube broadcast.

An implementation of this algorithm is described in Figure 6.3.

The mesh *broadcast* algorithm requires (ceiling(log R) + ceiling(log C)) steps where R*C = N, the number of nodes in the hypercube. This is equal to log(N) steps for a hypercube because R and C are both powers-of-two. The hypercube algorithm also requires log(N) steps. A considerable improvement in timings was still achieved for one of the benchmark C* programs because of the fact that the hypercube algorithm involves less traffic on the node links than the mesh algorithm. The mesh algorithm also involves a lot of communication between processors which do not

have direct links, resulting in slower timings. Appendix A.2 shows the figures for the hypercube *broadcast* performance.

(1) Find the physical node number of the source node.
(2) Shift left 1 (**D** - 1) times (this is equal to half of the number of nodes in the cube).
(3) Xor the source node number with your own node number (this assigns each node a value relative to the source node, with the source node being node 0). This is called **xored_no**.
(4) For each **i**, from 1 to **half**, multiply **i** by 2 in each iteration,

Determine whether this node will be sending a value in this iteration.Only the nodes whose relative node number (**xored_no**) is less than **i**, send in each step. Thus, **xored_no** is compared with **i**.

If **i** is greater,

Calculate **partner** by xoring your node number with **i**. This is the corresponding processor across the **i**th dimension, which will receive your value in this step.

Else

Determine whether this node will be receiving a value in this iteration. Only those nodes whose **xored_no** is less than **2i** but greater than or equal to **i**, receive. Thus, **xored_no** is compared to **2i**.

If **xored_no** is lesser,

Calculate the node number of the source by xoring your node number with **i** and receive the value from it.

EndFor.

Figure 6.3 : Implementation of the hypercube *broadcast* algorithm on a hypercube of dimension **D**.

6.3 Scatter and gather

A *scatter*, as discussed earlier, is required when values are to be distributed in an unpredictable pattern across the nodes. A *gather* is 2 *scatters*.

The hypercube algorithm for a *scatter* is quite similar to that of a *reduce*. Each node maintains a read and a write buffer and in each iteration, it checks every data item in the read buffer. If the data item belongs to the same node, it keeps the item. If it doesn't, another check is performed to determine whether the data item belongs to a

node in the same partition (or subcube) of the current iteration. If it doesn't, then the data is transferred into the write buffer. Otherwise, nothing is done. Then the nodes swap their read buffers with their partners. The cube is subdivided in each iteration, with the above sequence of steps performed every time. After log(N) steps, the values are scattered to the desired destinations.

Hypercube *scatter* is implemented by the function `HypercubeScatter` in the file `scatter.c`. The `CS__Gather` function, which is emitted by the compiler when it sees a statement requiring a *gather* makes a call to `HypercubeScatter` two times– once for scattering the requests and then for scattering the values.

The hypercube *scatter* algorithm requires log(N) steps and a *gather* requires 2log(N) communication steps, "N" being the number of processors in the hypercube. The mesh scatter algorithm required (log(R) +2) + (log(C) + 2) steps where R*C = N. This is equal to (log(N) +4) steps for a *scatter* and (2log(N) + 8) steps for a *gather*. Appendix A.3 shows the improvement in timings achieved for a benchmark C* program by using the hypercube style *scatter* and *gather*.

Chapter 7
Assembly Optimization

The `-S` switch, when used with the `icc` compiler, produces the assembly code for the file which is being compiled. This chapter describes optimizations made to the assembly code of some of the communication routines of the routing library. The functions which were optimized were `HypercubeScatter`, `HypercubeBroadcast` and `HypercubeReduce`.

7.1 The i860™ instruction set

The iPSC/860 RX nodes use the Intel i860™ microprocessor, which has a 64-bit design [2]. This section gives a brief explanation of some of the instructions which were involved in the optimizations.

7.1.1 The conditional branch statements

`bc` and `bnc` : These instructions check the contents of the *Condition Code (CC)* field of the processor status register of the i860™ microprocessor. This field is set or cleared by a previous add or subtract instruction. Depending on the CC field, the instructions branch or do not branch to a specified branch target. The `bc.t` and `bnc.t` instructions are similar to `bc` and `bnc` except for the fact that the instruction following `bc.t` and `bnc.t` is executed even if a branch is taken.

7.1.2 Register loading instructions

`ld.l`, `mov` and `fxfr` : The `ld.l` instruction loads the value stored in a memory location into a register. The `fxfr` instruction transfers a value from a floating point register to an integer register. `mov` moves the contents of one register to another.

7.1.3 Arithmetic operations

`adds` and `subs` : These instructions add or subtract the values stored in two registers. The CC field is set depending on the relative magnitudes of the two values.

7.2 Optimizations

7.2.1 Removing redundant statements

Although the `-O4` switch passed to the `icc` compiler removed almost all the unnecessary statements, some further redundancies in function `HypercubeReduce` were removed. These included some `mov` and `adds` statements whose operation did not have any effect on the execution.

7.2.2 Moving around instructions

A reference to the *destination* of `ld.l` and `fxfr` in the next instruction causes a delay of one clock. Thus, one optimization dealt with looking for such cases and inserting statements between the two instructions to get rid of the delay. The inserted instruction was usually plucked out from the code preceding the two instructions, taking care that no data dependencies were disturbed. An optimization which caused a considerable improvement in timing in function `HypercubeScatter` is illustrated in the following code :

```
ld.l 76(sp), r22
fxfr f8, r25
adds r25, r17, r16
```

The reference to `r25` in the `adds` statement following the `fxfr` whose *destination* is `r25`, causes a delay of one clock. This is because the instructions are pipelined and the `adds` instruction must wait for the `fxfr` to complete before it can be executed. Inserting the first of those instructions between the last two as shown in the following, gets rid of the delay.

```
fxfr f8, r25
ld.l 76(sp), r22
adds r25, r17, r16
```

A `bc`, `bnc`, `bc.t` or `bnc.t` following an `adds` or `subs` causes a delay of one clock which is also optimized by inserting an instruction between the two. The delay is again a result of the instruction pipeline–the `subs` and `adds` instruction set the *CC* flag and the `bc.t` decides whether to branch or not based on the contents of this flag. So, the `bc.t` instruction must wait for the completion of execution of the `subs` or `adds` instruction. An example illustrates this :

```
st.l r16, 128(sp)
.
.
subs r4, r18, r0
bc.t .B838
```

(a) Before

```
subs r4, r18, r0
st.l r16, 128(sp)
bc.t .B838
```

(b) Optimized

7.2.3 Loop invariant code

Invariant code inside a loop (which does not depend on the iteration being executed) was moved to before the loop. Most of these optimizations were taken care of by using the `-O4` switch.

Optimizing `HypercubeReduce` and `HypercubeBroadcast` using the above described optimizations did not result in any improvements in timings. `HypercubeScatter` gave a considerable improvement, the results of which are summarized in Appendix B.1.

Chapter 8
Compiler Bugs

A bug in the C* compiler was discovered while benchmarking one of the C* programs. A description of the bug is given in this chapter.

The C* benchmark program `shallow-2d.cs` has a procedure called `init_p_u_v` whose compilation caused the bug to appear. The C* code for the part of this procedure is shown :

```
void init_p_u_v()
{
   int i, j;

   with( physics) {
     float:physics   new_psi;
     float:physics   new_east_psi;
     float:physics   new_southeast_psi;
     float:physics   new_south_psi;


     new_psi = a*sin((pcoord(0)+.5)*di)*sin((pcoord(1)+.5)*dj);
     new_east_psi = [.][(.+ 1) %% dimof(physics,1)]new_psi;
     .
     .
}
```

An error message from the routing library :

Unsupported distribution type in CS__GetRowAndCol

was displayed when the above piece of code was executed. It was traced to the statement in which `new_east_psi` is being assigned its value. On examining the translated C code for the above C* code, it was found that the shape map of `new_psi` changed when a value was assigned to it (in the previous statement). The change in the map was traced to a VP emulation loop in which `new_psi` was assigned the values.

When the declarations for `new_psi`, `new_east_psi`, `new_southeast_psi` and `new_south_psi` were moved to before the **with**, the program compiled and executed correctly.

Chapter 9
Summary

- The routing library was ported successfully to the iPSC/860 system.
- The communication routines for *reduce, broadcast* and *scatter* were changed to hypercube style. Benchmark C* programs were written and the improvement in timings recorded for the new routines.
- The assembly code for the new routines was optimized which resulted in a further improvement in timings.
- The timing data was compiled for all the benchmark C* programs. Appendix C.1 summarizes this data.
- C* programs can now be written on the iPSC/860 making use of the ported routing library.

Appendix A.1
Hypercube Reduce performance

relprime.cs (128)

Number of Processors	Original Timings (in secs)	Timings after Hypercube reduce (in secs)
1	14.295	6.243
2	7.453	3.345
4	3.854	1.762
8	1.993	0.941

warshall.cs (64X64)

Number of Processors	Original Timings (in secs)	Timings after Hypercube reduce (in secs)
1	21.087	10.941
2	11.068	6.005
4	6.174	3.613
8	3.716	2.434

Appendix A.2
Hypercube Broadcast performance

warshall.cs (64X64)

Number of Processors	Original Timings (in secs)	Timings after Hypercube broadcast (in secs)
1	10.941	6.565
2	6.005	3.816
4	3.613	2.476
8	2.434	2.052

Appendix A.3
Hypercube Scatter-Gather performance

red_black.cs

Number of Processors	Original Timings (in secs)	Timings after Hypercube scatter (in secs)
1	96.031	82.887
2	54.173	48.408
4	28.261	25.861
8	15.865	14.597

Appendix B.1
Scatter assembly optimization performance

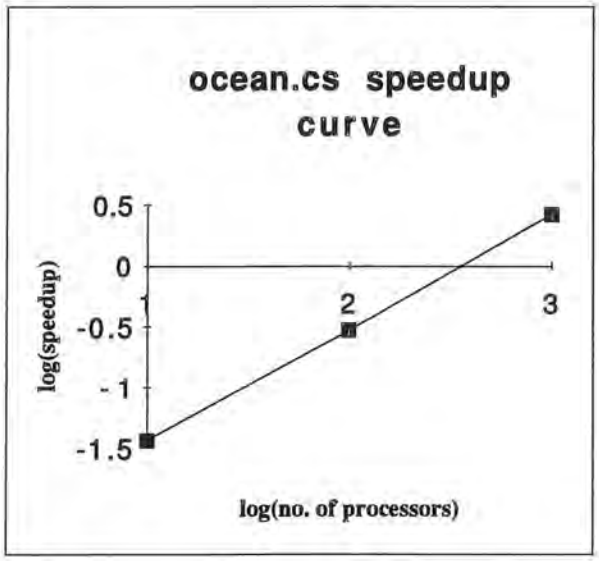
red_black.cs

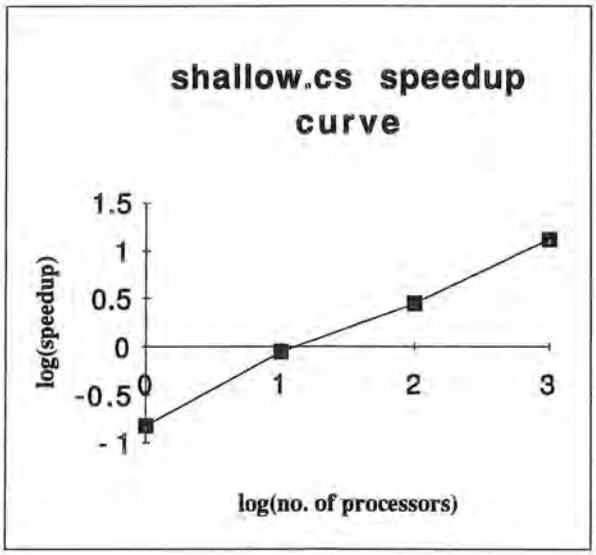
Number of Processors	Original Timings (in secs)	Timings after assembly opt. (in secs)
1	82.887	69.062
2	48.408	40.316
4	25.861	21.667
8	14.597	12.436

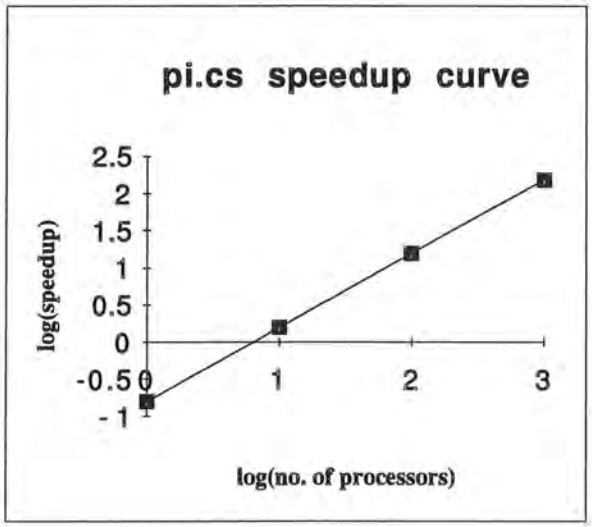
Appendix C.1
Intel iPSC/860 Performance Data - Feb 1993

Performance of C* programs on IPSC/860 (all times in secs)					
Program	Size	Number of Processors			
		1	2	4	8
pi	100,000	1.061	0.532	0.266	0.134
pi	200,000	2.119	1.063	0.532	0.267
pi	400,000	4.238	2.126	1.064	0.532
relprime	128	6.243	3.345	1.762	0.941
relprime	256	49.395	25.836	13.347	6.861
fastmat1	64	0.043	0.036	0.031	0.029
fastmat1	256	2.931	1.833	1.248	0.805
warshall	64	6.565	3.816	2.476	2.052
warshall	128	26.468	15.301	9.733	7.301
warshall	256	109.697	65.636	43.504	33.411
gauss	64	No memory	3.639	2.165	1.479
gauss	128	No memory	No memory	7.348	5.059
gauss	256	No memory	No memory	No memory	22.156
sieve	64,000	0.331	0.638	0.969	1.418
sieve	512,000	30.115	51.773	74.673	100.981
shallow-1d	64	303.797	150.796	72.571	38.665
shallow-2d	64	130.028	76.391	53.982	33.806
ocean		No memory	604.573	323.524	167.295
jacobi		26.995	13.795	7.516	4.231
red-black		69.062	40.316	21.667	12.436

Appendix C.2
Speedup Curves






pi.cs speedup curve
2.5
2
1.5
1
0.5
0
-0.5
-1
0
1
2
3
log(speedup)
log(no. of processors)

Bibliography

[1] Intel Corporation. 1991. *iPSC/2 and iPSC/860 user's guide.*

[2] Intel Corporation. 1990. *i860^{TM} 64-bit Microprocessor Programmer's reference manual.*

[3] Philip J. Hatcher and Michael J. Quinn. 1991. *Data-Parallel Programming on MIMD Computers.* MIT Press, Cambridge, MA.

[4] Thinking Machines Corporation. 1990. *C* Programming Guide, Version 6.0 Beta.*

[5] Kathleen P. Herold. 1992. *A Retargetable C* Communication and Run-time library for Mesh-Connected MIMD Multicomputers.* Masters' thesis, Department of Computer Science, University of New Hampshire, Durham, NH.

[6] Anthony J. Lapadula and Kathleen P. Herold. 1992. *A Retargetable C* Communication library for Mesh-Connected MIMD Multicomputers.*

[7] Michael J. Quinn. 1991. *Designing Efficient Algorithms for Parallel Computers.* MCGraw-Hill Book Company.