The
Connection Machine
System

Introduction to Programming in C/Paris

Version 5
June 1989

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, June 1989

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142-1214
(617) 876-1111

Contents

Part V Example Programs

Figures

Examples

Customer Support

Thinking Machines Customer Support encourages customers to report errors in Connection Machine operation and to suggest improvements in our products.

When reporting an error, please provide as much information as possible to help us identify and correct the problem. A code example that failed to execute, a session transcript, the record of a backtrace, or other such information can greatly reduce the time it takes Thinking Machines to respond to the report.

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142-1214
Internet Electronic Mail:	customer-support@think.com
Usenet Electronic Mail:	ames!think!customer-support
Telephone:	(617) 876-1111

About This Manual

Objectives

This manual is intended to help new users of C/Paris get started quickly and be able to write simple but complete programs for the CM system.

Intended Audience

Readers are assumed to have a working knowledge of C programming and a very general understanding of the components of the CM system. Prior knowledge of assembly-level programming is helpful but not required.

Revision Information

This is a new manual. It is a companion volume to the *Paris Reference Manual*, Version 5.0, and its updates.

Organization

Part I Getting Started
These two chapters introduce C/Paris as an assembly-level subroutine library and provide a step-by-step explanation of a simple program.

Part II Basic Concepts and Techniques
These three chapters introduce the basics of C/Paris and data parallel programming. They describe simultaneous computations within many processors at once, Paris-level memory management, the data parallel techniques of program control, and the means of configuring sets of virtual processors to express large data sets.

Part III Interprocessor Communications
These two chapters introduce communications among CM virtual processors. Once a program has configured virtual processors to express both the size and the shape of various data sets, the processors can send intermediate results to each other in a variety of patterns, and the program can perform cumulative operations along any of the dimensions of a data set.

Part IV Commands and Utilities
These two chapters outline the procedures for compiling and executing a C/Paris program and introduce some utilities for debugging, run-time safety checking, and monitoring program execution time.

Part V Examples
The appendixes present six complete example programs.

Associated Documents

- *CM User's Guide: UNIX System Front End,* Connection Machine documentation set
- *Paris Reference Manual,* Version 5.0, Connection Machine documentation set

Associated On-Line Directory

- **/cm/src/cparis-examples**

All the example programs shown in this manual appear in the above directory under the file name given in the example's caption. A **Makefile** is included. (Check with the site system manager if the example directory has been placed in another location.)

Notation Conventions

Convention	Meaning
boldface	UNIX and CM System Software commands, command options, and file names.
boldface	C/Paris and C language elements, such as keywords, operators, and function names, when they appear embedded in text.
italics	Parameter names and placeholders in function and command formats.
`typewriter`	Code examples and code fragments.
% **boldface** `typewriter`	In interactive examples, user input is shown in **boldface** and system output is shown in `typewriter` font.

Part I
Getting Started

Chapter 1

What Is Paris?

The data parallel programming model assumes an array of small processors, each with some associated memory, all acting under the direction of a distinguished processor called the front end. In the Connection Machine system, the front end is a standard serial computer—a Sun-4 or certain models of VAX—with a bus interface to the processor array within the CM itself.

The essence of the programming model is that each processor stores the information for one data point in its local memory, and then all processors perform the same operation on all the data points at the same time. For instance, a text retrieval program might store articles one-per-processor and then have each processor search its article for a key word. Similarly, a graphics program might store pixels one-per-processor and then have each processor compute the color value for its pixel.

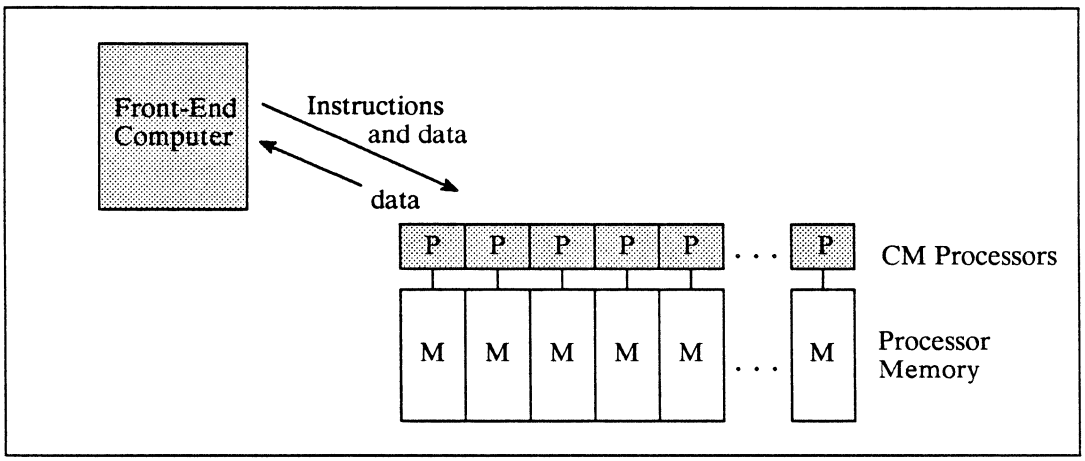

Figure 1. Interactions between front end and CM

Refinements of the programming model allow the user to direct that only a selected set of processors perform a given operation. In the text retrieval program, for instance, the processors that find the key word might be instructed to search further for another key word, while those that did not find the key word remain idle. Another refinement enables processors to pass messages to each other. For instance, color shading in a graphic image requires that each processor obtain surface information from surrounding pixels (processors) in order to calculate its result.

The front-end computer runs a serial program that explicitly "parallelizes" certain operations by transferring data and instructions over the bus to the CM processors. The instructions might include:

- The data parallel equivalent of serial looping operations, as in the text retrieval example just mentioned. These kinds of instructions are similar to the arithmetic and relational operations of a serial computer, except that they are executed by many processors simultaneously.
- Directions to the CM processors to compute some value that determines whether they are to participate in the next instruction.
- Directions to the CM processors to communicate with each other, as in the color-shading example. These kinds of instructions cause each processor to send (or get) some intermediate result from another processor before proceeding to the next instruction.
- Directions to the CM processors to return their results (or some aggregate result) to the front end.

This set of instructions by which the front end directs the actions of the processor array is Paris, the Connection Machine **PAR**allel Instruction Set.

NOTE on I/O

Paris instructions transfer data into and out of CM memory only by way of the front end. Specialized CM systems exist for transferring data directly between CM memory and peripheral storage or graphic display devices. See the volumes *Connection Machine I/O Programming* and *Connection Machine Graphics Programming.*

1.1 What Is C/Paris?

Paris is a low-level protocol in which the user can write data parallel programs for the Connection Machine system. It exists as three interfaces—C/Paris, Lisp/Paris, and Fortran/Paris—which are essentially subroutine libraries in the languages.

A Subroutine Library

C/Paris should be seen as a library rather than as a full language—it cannot be used independently of C (or some other language with a suitable interface). A C/Paris program is written in C, with Paris calls as needed to express parallel operations.

It is important to recognize that a CM program is directing two different computers: the serial front end and the parallel CM. The higher-level CM languages mask this duality somewhat, but at the Paris level it is always explicit:

- C code directs front-end (serial) operations, including the manipulation of serial data on the front end and all control flow in the program.
- Paris calls direct *only* the handling of data by CM processors and the transfer of data between the front end and the CM.

An Assembler-Like Language

Paris is intended primarily as a base upon which to build the higher-level CM languages—C*, CM Fortran, and *Lisp. For instance, the C* compiler generates serial C code with calls to C/Paris routines. This output is passed to the front end's C compiler, which proceeds in the normal way to produce an executable load module.

Users can, however, gain finer control over program behavior by calling Paris routines directly, either from within C* or from within an ordinary C program. As with any low-level language, the user is trading off the convenience of high-level language abstractions for program efficiency.

1.2 A Program Template

Every C/Paris program must contain two particular lines of code, as shown in this program template:

Example 1. C/Paris program template: **template.c**

```
#include <cm/paris.h>

main()
{
     CM_init();

     /* [C code] */
     /* [Paris function calls occurring within C code] */
     /* [C code] */
}
```

The **#include** directive must occur at the beginning of any C/Paris program. The header file **paris.h** sets up the C/Paris programming environment by declaring the function names and defining data types and certain CM global configuration variables used by Paris.

The C/Paris routine **CM_init**, which takes no arguments, must appear in the program before any other calls to Paris instructions. It has two effects:

- **CM_init** initializes the values of the global configuration variables, such as:
  - **CM_physical_processors_limit**, determined by the size of the CM that is executing the program
  - **CM_maximum_integer_length**, determined by implementation restrictions, if any, in the version of Paris being used

  Programs can use these variables (rather than constant values) to ensure portability across CM software versions and hardware configurations. Programs should *never* set the values of CM configuration variables.

- **CM_init** also *warm boots* the CM. Warm booting prepares the system for the upcoming program by clearing the queue for the instruction bus between the front end and the CM, clearing error status indicators, and initializing certain system memory areas in the CM. (The user memory in the CM processors is not affected.)

  Warm booting should not be confused with *cold booting*, a more thorough initialization that completely resets the state of the CM hardware and clears user memory. Warm booting is always done from within a C/Paris program; cold booting is done only from the UNIX shell, using procedures described in the volume *CM Front-End Subsystems*.

The template shown is in fact a working C/Paris program—the simplest program possible. Its effects, however, are limited to changes in CM state that would not be visible to the programmer.

The kinds of C/Paris calls that do produce useful results are the subject of the remainder of this manual.

Chapter 2

A Simple Program

This chapter examines a very simple C/Paris program: one that simply adds two integer constants within each processor. The purpose of the exercise is to illustrate the basic features of parallel instructions and parallel data, ignoring for the moment the details and refinements.

The program is the parallel analogue of this trivial C program:

Example 2. A simple C program: **add-scalar-constants.c**

```
#include <stdio.h>

main()
{
     int a, b, sum;

     a = 2;
     b = 3;

     sum = a+b;
     printf( "\nThe sum of a and b is %d.\n", sum );
}
```

In this program, **a**, **b**, and **sum** are *scalar* (single) values that reside on the front end. Now, consider the same operations in **add-parallel-constants.c**, where the three variables indicate multiple values in CM processors' memories.

After showing the C/Paris program, the remainder of this chapter walks through the program step by step, pointing out some basic features of C/Paris programming and the ways in which it differs from serial C programming.

NOTE

This program, and all complete programs shown in this manual, can be found on line in **/cm/src/cparis-examples** (or another location determined by the site system manager).

Example 3. A simple C/Paris program: **add-parallel-constants.c**

```
#include <cm/paris.h>
#include <stdio.h>

#define LEN 32

main ()
{

/* ========================================================*/
/* 1. Declare variables and warm boot CM */

     CM_field_id_t field_a, field_b, field_sum;
     int single_sum, agg_sum;

     CM_init();

/* ========================================================*/
/* 2. Allocate CM memory fields */

     field_a = CM_allocate_heap_field( LEN );
     field_b = CM_allocate_heap_field( LEN );
     field_sum = CM_allocate_heap_field( LEN );
```

```
/* ========================================================*/
/* 3. Specify that all processors participate */

    CM_set_context();

/* ========================================================*/
/* 4. Put values into the two "source" fields */

    CM_s_move_constant_1L( field_a, 2, LEN );
    CM_s_move_constant_1L( field_b, 3, LEN );

/* ========================================================*/
/* 5. Add the two source fields in each processor and
   place the result in the "destination" field */

    CM_s_add_3_1L( field_sum, field_a, field_b, LEN );

/* ========================================================*/
/* 6. Read the value from the destination field in a single
   processor and print it. */

    single_sum =
        CM_s_read_from_processor_1L( 0, field_sum, LEN );
    printf( "\nThe sum in processor 0 is %d.\n", single_sum );

/* ========================================================*/
/* 7. Add the destination fields in all processors, return
   the aggregate value to the front end, and print it */

    agg_sum = CM_global_s_add_1L( field_sum, LEN );
    printf( "\nThe sum of all the sums is %d.\n", agg_sum );

/* ========================================================*/
/* 8. Deallocate CM memory fields */

    CM_deallocate_heap_field( field_a );
    CM_deallocate_heap_field( field_b );
    CM_deallocate_heap_field( field_sum );
}
```

The most immediately obvious difference between this program and its serial C analogue is the sheer length of the C/Paris program. Its length is, however, largely a result of its being low-level rather than of being parallel. The higher-level CM languages, such as C*, can express parallel operations nearly as economically as C expresses serial operations. This program is an example of the trade-off between convenience and fine-tuned control that we see in comparing any high-level and low-level languages.

2.1 Allocating and Referencing Parallel Data

User data is always stored in *fields* in CM memory. A field is simply one or more contiguous bits that start at the same bit location *in every processor.* A CM field is roughly analogous to a C variable in that it is associated with an address in memory. However, there are two significant differences:

- Unlike a C variable, a CM field must be allocated explicitly. The allocation instructions return a field-id, which contains the address of the allocated field along with other information. The program can (optionally) assign the field-id to a front-end variable of type **CM_field_id_t**.
- Fields are not associated with types. The length of a field (in bits) is specified when the field is allocated. Later, data of any supported CM type—signed or unsigned integer or floating-point number—can be placed in any field.

In the example program, Step 2 allocates fields and assigns them to front-end variables that have been declared in Step 1.

```
CM_field_id_t field_a;                          /* Step 1 */
field_a = CM_allocate_heap_field( LEN );        /* Step 2 */
```

These two steps allocate memory to hold parallel data and provide a front-end variable with which to reference that portion of memory. The "stripe" of memory allocated consists of 32 contiguous bits that begin at the same bit location in every processor. All Paris instructions that operate on user data, including those that move data into fields, take one or more field-id's as operands.

Figure 2 shows the state of CM system memory at this point in the program. The program calls the allocation instruction three times to allocate three fields, and it performs three assignments to previously declared front-end variables.

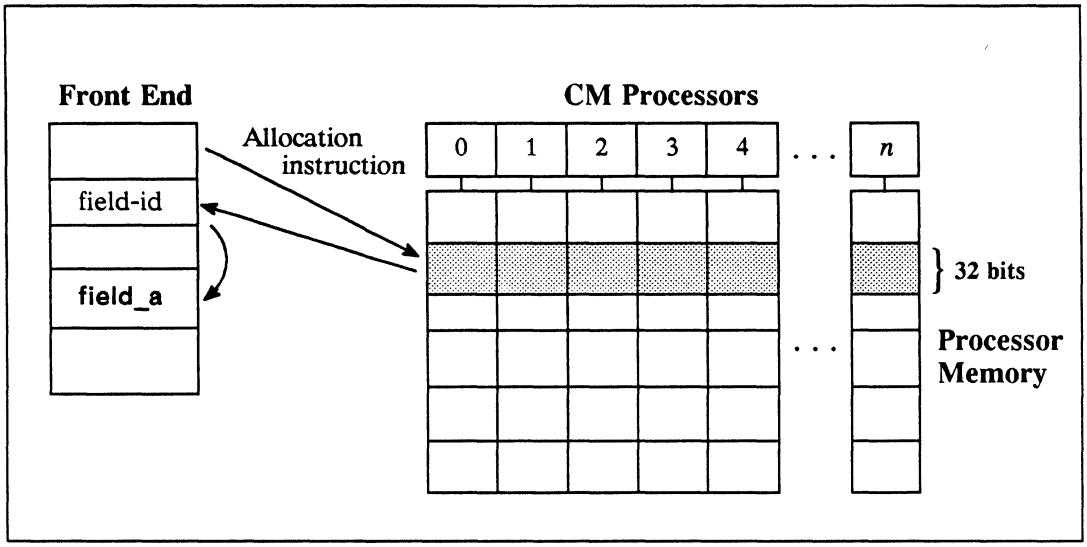

Figure 2. Allocating a field and returning a field-id

Note that the last step of the program explicitly deallocates the fields.

```
CM_deallocate_heap_field( field_a );                    /* Step 8 */
```

This step is not strictly necessary in the example program, since the fields would be deallocated at program termination in any case. However, as in C, it is generally good practice to free up memory by deallocating items that are no longer needed.

2.2 Setting CM Context

Most CM programs use all the processors for some instructions and a selected subset of them for other instructions. For instance, in the text retrieval example given in the previous chapter, all processors execute the first instruction to search for key-word-1, but only those that met success execute the next instruction to search for key-word-2.

The set of processors that are to execute the next instruction is called the *selected set* or *active set*. At any given time, the program's *context* may change to a different selected set. At the Paris level, context is always set and reset explicitly. Step 3 of the example program is a Paris call that selects all processors; this context is retained throughout the program.

```
CM_set_context();                                       /* Step 3 */
```

2.3 Moving Data into Fields

When fields are allocated, they contain no useful data. This program illustrates one of several ways to initialize fields, in this case with signed integer constants. The **CM_move** family of instructions copy data into a field, either from another field or from the front end. If the front-end data to be moved is a constant or zero, the front end "broadcasts" a copy of the quantity to the specified field in each processor.

```
CM_s_move_constant_1L( field_a, 2, LEN );          /* Step 4 */
CM_s_move_constant_1L( field_b, 3, LEN );
```

Like all Paris instructions that operate on fields, this instruction takes a field-id as an operand to indicate the bit address at which to begin storing the constant (**2** or **3**). In addition, when calling any Paris instruction that operates on fields, the program must specify the number of bits (**LEN = 32**) on which to operate. In this program, the number corresponds to the allocated length of the field. (Chapter 3 shows cases where the number might be smaller than the field.)

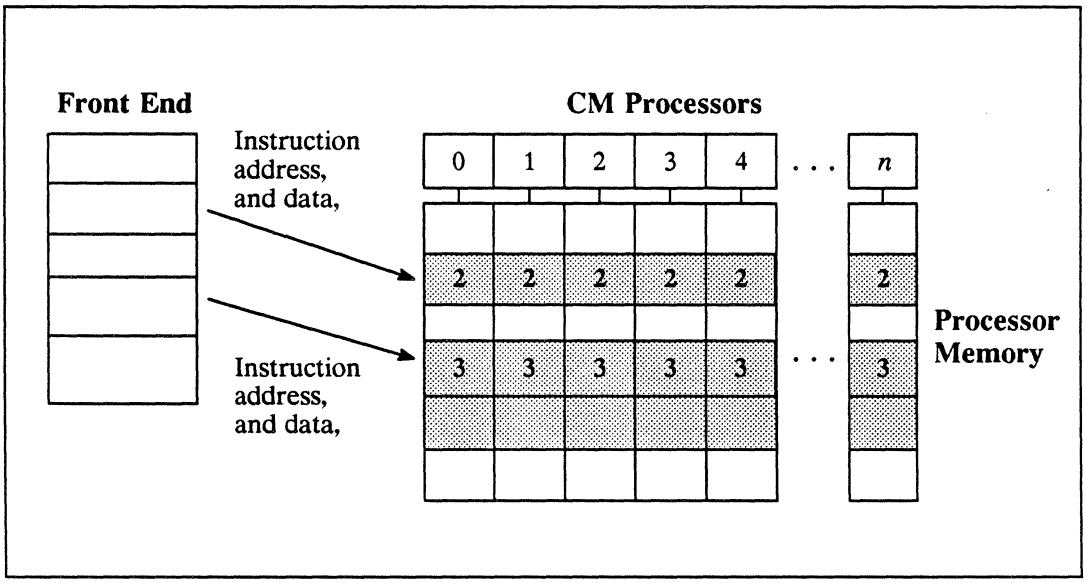

Figure 3. Moving constant values into CM memory fields

Notice that it is the instruction, not the field, that expects a certain type of data. The element **_s** in the instruction name indicates that the instruction treats its data operand as a signed integer. (Alternatively, the element could be **_u** or **_f**, for unsigned integer or floating-point number.) However, there is nothing analogous to type checking in

C/Paris. If the constant provided were **2.0**, the instruction would simply take it to be a signed integer and blithely put the wrong value in the specified field.

2.4 Computation on Fields

Given the development of the program so far, the instruction that actually performs the computation is straightforward:

```
CM_s_add_3_1L( field_sum, field_a, field_b, LEN );    /* Step 5 */
```

The *destination* field is always specified first in a Paris call, followed by one or more *source* fields as appropriate to the instruction. In this case, the instruction adds the values in **field_a** and **field_b**, treating them as signed integers, and places the result in **field_sum**.

Note the element **_3** in the instruction name. This element indicates that the instruction takes three field operands: two source fields and a different destination field. An alternative that Paris provides is **CM_s_add_2_1L**, which places the result back into one of the source fields. For instance, the following call increments the value in **field_a** by the value in **field_b**:

```
CM_s_add_2_1L( field_a, field_b, LEN );
```

The last element of the instruction name, **_1L**, indicates the number of different lengths of the operands. This example used the **_1L** version because the three fields are all of the same length (32 bits) and the program uses the full length of the fields. An alternative is the **_3L** version, used when the operands are of different lengths. In the **_3L** version, all the lengths are specified and in the same order as the field operands. For instance:

```
CM_s_add_3_3L( field_sum, field_a, field_b, 32, 16, 16 );
```

These variations on the theme of **add** illustrate the major reason for the large size of the Paris instruction set. The **CM_add** family of instructions includes variants for **_u**, **_s**, and **_f** numbers; within each of these, there are variants for adding constants or adding variables; and within each of these, there are variants for **_2** and **_3** field operands and for **_1L** and **_3L** different lengths. The number of variations is a reflection of the low level of the Paris instructions: each of these subtly different system operations is expressed by a different Paris instruction.

2.5 Moving Data to the Front End

The results computed by the CM processors are of no use to the **main** program until some result is returned to the front end.

The example program shows two of the several ways in which the front end retrieves results from the CM processors. In both these cases, the data retrieved is a scalar, not parallel, quantity.

From a Single Processor

The first method of moving data from CM memory to the front end involves reading a field's contents from one specified processor (see Figure 4). The value returned is then assigned to a front-end variable.

```
single_sum =                                              /*Step 6*/
    CM_s_read_from_processor_1L( 0, field_sum, LEN );
```

The first argument is a unique processor-id, or *send address*. Send addresses are more commonly used when processors send messages to each other. However, the front end can use a send address to access a value from a single CM processor.

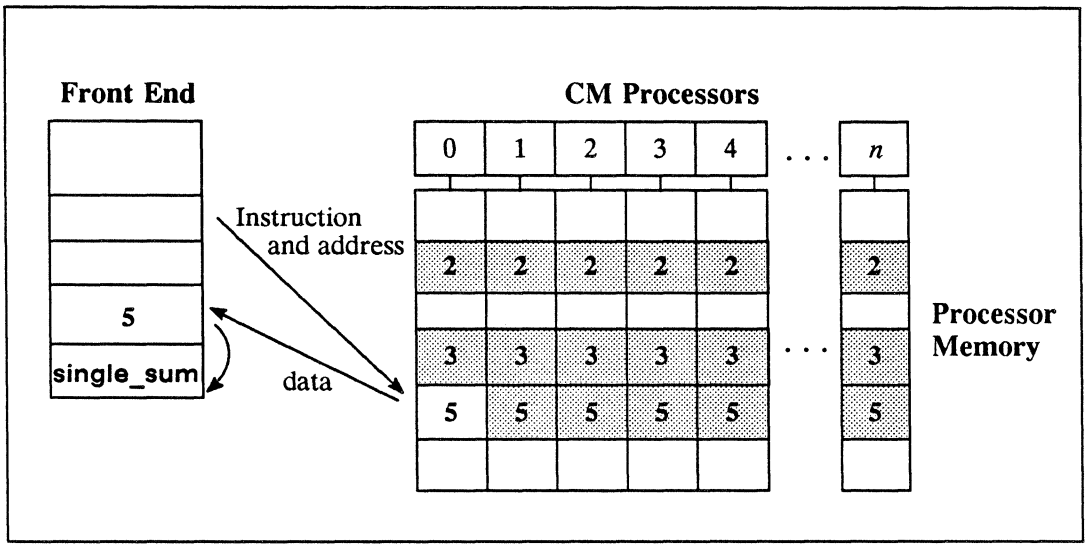

Figure 4. Returning one processor's result to the front end

Send addresses, and the means of computing them, are introduced in Chapter 7. Paris does not guarantee that send addresses are consecutive integers, but we can be confident that there will always be a processor numbered **0**. We can thus check one result of a parallel computation by using **CM_*type*_read_from_processor_1L**.

From a Computation across Processors

A second method of retrieving scalar data from the CM processors is to perform some combining operation across all the values in the specified field and return the single result to the front end (see Figure 5). Paris provides several such instructions, all with the element **_global** in their names.

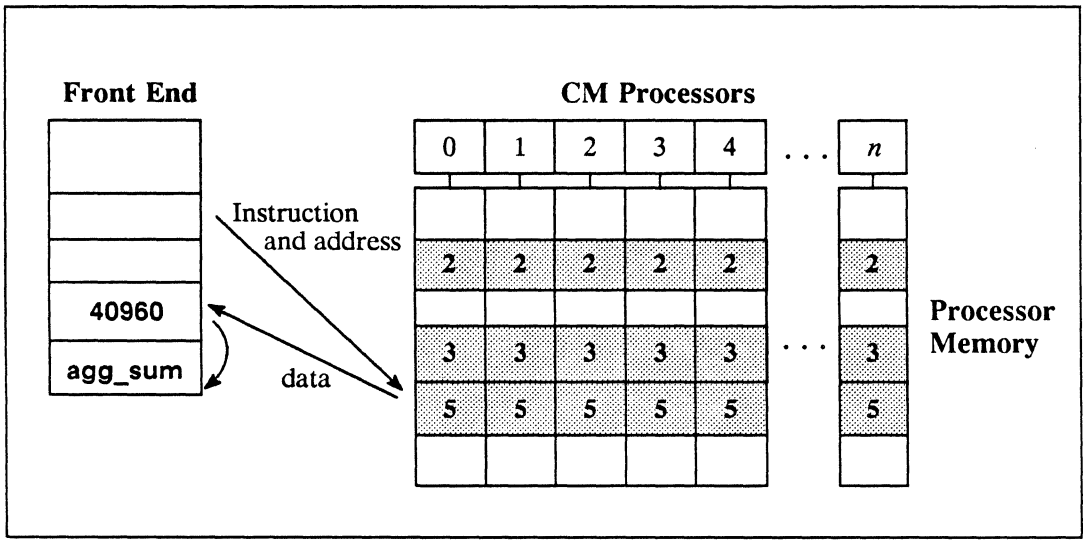

Figure 5. Returning a global-reduction result to the front end

In the example shown, Step 7 adds the values across **field_sum** and returns their aggregate sum to the front end. The result is then assigned to the front-end variable **agg_sum**.

```
agg_sum = CM_global_s_add_1L( field_sum, LEN )        /* Step 7 */
```

The value returned in this example is $n \times 5$, where n is the number of processors participating. For instance, imagine that the program is executing on a 16K CM system. The 16,384 instances of **5** are added and the total **81920** is returned. If the program executes on an 8K CM system, the aggregate sum is **40960**.

2.6 Compiling and Executing the Program

A C/Paris program is compiled with the front end's C compiler **cc** and executed with the CM System Software command **cmattach**. The detailed procedures for compiling and executing a C/Paris program are described in Chapter 8.

Those who have faith can simply type the following command lines:

```
% cc add-parallel-constants.c  -lparis -lm

% cmattach -w a.out
Attaching the Connection Machine system [ name ]...
coldbooting... done.
Attached to 8192 physical processors

The sum in processor 0 is 5.

The sum of all the sums is 40960.

Detaching... done.
%
```

NOTE

The command lines just shown apply to Paris Version 5.0. In later versions, please consult the current CM documentation to determine whether these command options have changed. In particular, the means of specifying the Paris library on the **cc** command line may change.

Part II
Basic Concepts and Techniques

Chapter 3

Computing within Processors

The vast majority of Paris instructions cause some computation to occur within each selected processor, independently of the others. Most of these instructions are arithmetic and relational operations that will be familiar to assembly-language programmers and not surprising to C programmers.

The major differences between intraprocessor Paris instructions and the analogous C operations arise from the low level of Paris instructions, rather than from parallelism:

- The data on which Paris operates, the CM *data formats*, are unlike C types.
- Working with CM *memory fields* requires more explicit storage management than working with C variables.

This chapter illustrates intraprocessor Paris instructions, although it does not attempt to give an exhaustive listing of them. Rather, the focus is on using CM data formats and managing CM memory fields in the course of executing intraprocessor instructions.

NOTE

The emphasis in this chapter is on differences from C programming. Users who are experienced in assembly-language programming can skim quickly over this chapter.

3.1 What Is a "Processor"?

This chapter adopts an artificially constrained programming model: the number of processors is assumed to be equal to the size of the physical CM on which a program executes, and the processors do not exchange data with each other.

In fact, the "processors" referred to in this chapter are not physical CM processors. Paris supports a *virtual processing mechanism* whereby each physical processor simulates some number of virtual processors. Further, the virtual processors can be logically configured into a variety of shapes of up to 31 dimensions, which facilitates the exchange of data between logical "neighbors." Chapter 5 introduces the virtual processing mechanism, along with the methods for configuring the virtual machine.

The programs shown in the present chapter make use of the default virtual processor configuration, which is a 2-dimensional array of processors the same size as the physical machine. The procedures introduced here for intraprocessor computations and storage management are the same in any configuration of virtual processors.

3.2 Arithmetic and Relational Instructions

Paris provides a large selection of arithmetic and relational instructions, analogous to those found in the instruction sets of many serial computers. These include:

- Binary arithmetic, such as **add**, **subtract**, **multiply**, **divide**, **max**, **min**, **truncate**, **round**, **rem**, **mod**, **power**, and others.
- Unary arithmetic, such as **negate**, **sqrt**, **abs**, **signum**, **integer-length**, **logcount**, **floor**, **ceiling**, **truncate**, as well as transcendental and trigonometric functions **sin**, **cos**, **tan**, **sinh**, **cosh**, **tanh**, **exp**, and **ln**.
- Bitwise booleans on two operands, **logand**, **logior**, **logeqv**, **lognand**, **lognor**, **logandc1**, **logandc2**, **logorc1**, **logorc2**, as well as **lognot** for one operand.

Instruction Variants and Names

Each of the intraprocessor instructions takes as operands:

- The field-id's of one or more source fields and a destination field (which may be the same field as one of the source fields)
- One or more length specifiers

Also, most of the instructions have variants for signed and unsigned integers and for floating-point numbers. A few variants are not provided either because they are nonsensical (there is no unsigned **abs**, for example) or because they are not generally useful (for instance, the trigonometric functions are provided for floating-point data only).

The user can recognize the action of a Paris instruction—and often predict the existence of related instructions—by parsing its name:

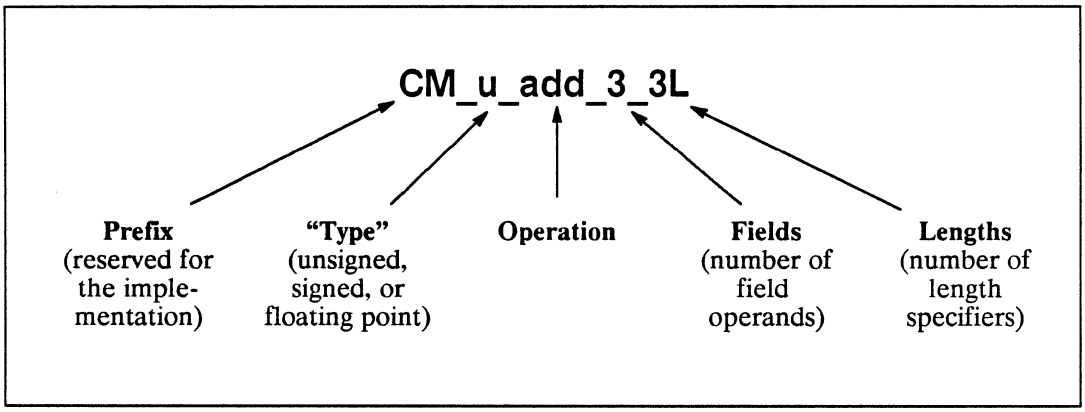

Figure 6. Naming convention for C/Paris instructions

At this point, it would be helpful for the new Paris user to scan the early sections of Chapter 5, "Instruction Set Overview," in the *Paris Reference Manual*, Version 5.0. This chapter lists all Paris instructions by name under various functional categories and identifies the variants of each instruction.

Example of Intraprocessor Computations

The following example, **unsigned-arithmetic.c**, illustrates some of the intraprocessor instructions in action. The instruction variants shown all place their results in a separate destination field, rather than back into a source field. In a real program this would be a profligate use of memory, but it does facilitate showing the results of all the operations in a single snapshot (Figure 7).

This example also shows some additional methods for getting data into and out of CM memory, beyond those shown in earlier examples:

- For input, the example uses a **move** instruction (Step 1), an implicit **move** instruction (Step 4), and the unsigned variant of **random**, which places random integers up to a specified limit into the destination field (Step 1).
- For output, the example uses two **_global** reduction instructions (Steps 7 and 8). These instructions are available as **add**, **multiply**, **max**, and **min**, each for **_u**, **_s**, and **_f** data, as well as **logand**, **logior**, and **logxor**.

Example 4. Computing within processors: **unsigned-arithmetic.c**

```
#include <cm/paris.h>
#include <stdio.h>

#define LEN 8


main ()
{
     CM_field_id_t a, b, c, d, e, f, g;
     unsigned int max_value, min_value;
     char *string;

     CM_init();

     a = CM_allocate_heap_field( LEN );
     b = CM_allocate_heap_field( LEN );
     c = CM_allocate_heap_field( LEN );
     d = CM_allocate_heap_field( LEN );
     e = CM_allocate_heap_field( LEN );
     f = CM_allocate_heap_field( LEN );
     g = CM_allocate_heap_field( LEN );


     CM_set_context();
```

```
/* ========================================================*/
/* 1. Initialize fields a and b */

    CM_u_random_1L( a, 100, LEN );
    CM_u_move_constant_1L( b, 2, LEN );

/* ========================================================*/
/* 2. Compute the max of a and b */

    CM_u_max_3_1L( c, a, b, LEN );

/* ========================================================*/
/* 3. Multiply a by b */

    CM_u_multiply_3_1L( d, a, b, LEN );

/* ========================================================*/
/* 4. Multiply a by the constant 2 */

    CM_u_multiply_constant_3_1L( e, a, 2, LEN );

/* ========================================================*/
/* 5. Divide a by b and round the result toward zero  */

    CM_u_truncate_3_1L( f, a, b, LEN );

/* ========================================================*/
/* 6. Take the integer square root of f */

    CM_u_isqrt_2_1L( g, f, LEN );

/* ========================================================*/
/* 7. Find the maximum value in g, return it to the
   front end, and print it */

    max_value = CM_global_u_max_1L( g, LEN );
    printf("The largest value in field g is %d.\n", max_value);
```

```
/* ========================================================*/
/* 8. Determine whether any of the values in g is zero
   and print that information */

    min_value = CM_global_u_min_1L( g, LEN );
    string = min_value ? " not" : "";
    printf( "Field g does%s contain a zero.\n", string );
}
```

Note in Step 8 that a CM integer of length 8 is assigned to a front-end **unsigned int** (which is of length 32 on VAX and Sun). The two numbers need not be of the same length. The only constraint is that the front-end type should be large enough to hold the value retrieved from the CM. In this example, the returned value would fit comfortably into a **char.**

The changes that occur in CM memory from this set of computations are shown in Figure 7, which depicts an arbitrary set of five processors.

CM Processors

	P	P	P	P	P	
a	34	98	0	77	12	1. Initialize **a** and **b**
b	2	2	2	2	2	
c	34	98	2	77	12	2. Max of **a** and **b**
d	68	196	0	154	24	3. Multiply **a** by **b**
e	68	196	0	154	24	4. Multiply **a** by **2**
f	17	49	0	38	6	5. Divide **a** by **b** and truncate
g	4	7	0	6	2	6. Square root of **f** and truncate

Figure 7. Changes in CM state from **unsigned-arithmetic.c**

3.3 Data Formats

User data on the CM is always stored in bit fields, that is, in sets of contiguous bits that begin at the same memory location in each processor and extend for some arbitrary length. However, many Paris instructions interpret bit fields as being of certain data "types" or *storage formats*. The currently supported formats are:

- Unsigned integer, represented in straight binary form
- Signed integer, represented in two's-complement form
- Floating-point number, represented in three subfields (for significand, exponent, and sign) in a format like the IEEE standard

The format of the data in any given field is determined by the instruction that moves data into that field, rather than by any feature of the field itself.

Operand Lengths

Each CM processor is a one-bit serial processor. Since the basic granule of operation is a bit—rather than a byte or a word, as in more complex processors—Paris does not enforce any length or alignment requirements on data formats. In this respect, CM data formats are completely unlike C types.

Any of the CM "types" or formats can be of almost any length:

- Signed integers can be any length from 2 bits up to the value of the CM configuration variable **CM_maximum_integer_length**, which is version-dependent but never less than 128 bits.
- Unsigned integers can be any length up to the value of **CM_maximum_integer_length**.
- The subfields for floating-point numbers can be:
  - Significand: from 1 bit up to the value of the configuration variable **CM_maximum_significand_length**, which is never less than 96 bits.
  - Exponent: from 2 bits up to the value of the configuration variable **CM_maximum_exponent_length**, which is never less than 32 bits.
  - Sign: always 1 bit.

To correspond with IEEE single- and double-precision floating-point formats, the allocated field should be 32 bits or 64 bits, and the length arguments in procedure calls should be specified as **23, 8** or **52, 11**, respectively. (The last bit is the sign bit.) Thus:

```
field_name = CM_allocate_heap_field( 32 );
CM_f_random_1L( field_name, 23, 8 );
```

or,

```
field_name = CM_allocate_stack_field( 64 );
CM_f_random_1L( field_name, 52, 11 );
```

The choice of length for a field depends primarily on the size of the numbers it is to contain and the degree of precision desired. Specifically, dynamic range increases with field size for integers and floating-point exponents; precision increases with field size for floating-point significands. Other considerations in the choice of field length are:

- *Memory usage.* Like a C type, a CM field need be no longer than the number of bits required to represent its largest value. For instance, the program shown in Example 4, **unsigned-arithmetic.c**, uses 8-bit fields because the largest value to be represented is 198. (The exclusive upper bound of the random number initialization is 100, and the only increase in value is multiplication by 2.)

- *Overflow.* Shorter lengths run the risk of overflow, where the value moved into a field is too large to be represented in that field. Most Paris instructions that operate on fields set a state bit called the *overflow flag* when overflow has occurred. (The means of checking for overflow is described in Chapter 4.)

  For instance, the following call results in an overflow condition, producing an undefined result and setting the overflow flag as a side effect in each active processor:

  ```
  CM_s_move_constant_1L( dest_field, 1000000, 8 );
  ```

- *Execution time.* The extra bits needed to guarantee against overflow are not "free." Since the CM processors act upon one bit at a time in a serial fashion, there is an at least linear increase in processing time as field length increases.

- *Use of floating-point hardware.* For most data formats, length does not affect portability across CM system components, such as the various supported front ends (barring overflow, of course). However, on systems equipped with

an optional 32-bit floating-point accelerator, floating-point operands must be of length **23, 8** to be executed on hardware.

Format Conversions

A very important difference between C and C/Paris is that there is *no* type checking and *no* coercion in C/Paris. Paris instructions simply take the values that are passed to them and treat them as the format they are designed to operate upon. It is entirely the responsibility of the program to ensure the compatibility of operands.

Because Paris has no strong typing, programs are vulnerable to two kinds of errors that do not ordinarily occur in C:

- *No type checking.* If the operand format is not what an instruction expects, the instruction simply produces incorrect results. For instance:

```
CM_s_move_constant_1L( field_a, 5, 32 );
CM_f_negate_1_1L( field_a, 23, 8 );     /* wrong results */
```

  After broadcasting an integer constant into **field_a**, this code negates what it takes to be a floating-point number, obediently breaking it into subfields and so on. The result is neither the integer value **-5** nor the floating-point value **-5.0**.

- *No coercion.* If operands are incompatible with each other, the instruction simply produces incorrect results. For instance:

```
CM_s_move_constant_1L( field_f, 5, 32 );
CM_f_move_constant_1L( field_g, 5.0, 23, 8 );
CM_s_add_2_1L( field_f, field_g, 32 );  /* wrong results */
```

Paris programs must convert operands explicitly to the desired format, using the unary conversion instructions provided. These instructions have two "type" elements in their names, signifying the result format and the operand format, respectively.

CM_f_s_float_2_2L *dest source source-len dest-sig-len dest-exp-len*
CM_f_u_float_2_2L *dest source source-len dest-sig-len dest-exp-len*
CM_s_f_floor_2_2L *dest source dest-len source-sig-len source-exp-len*
CM_s_f_truncate_2_2L *dest source dest-len source-sig-len source-exp-len*

Correct versions of the above code fragments might be the following (assuming that all the needed fields have been previously allocated and assigned):

```
CM_s_move_constant_1L( field_a, 5, 32 );
CM_f_s_float_2_2L( field_b, field_a, 32, 23, 8 );
CM_f_negate_1_1L( field_b, 23, 8 );          /* correct */
```

and,

```
CM_s_move_constant_1L( field_f, 5, 32 );
CM_f_move_constant_1L( field_g, 5.0, 23, 8 );
CM_s_f_floor_2_2L( field_h, field_g, 32, 23, 8 );
CM_s_add_2_1L( field_f, field_h, 32 );       /* correct */
```

3.4 More on Fields

Paris programs can use two kinds of data storage, *heap fields* and *stack fields*. The distinction between the two is analogous to the distinction between static and automatic variables in C.

In addition, programs can divide a field into subfields by means of an offset instruction, and then treat the subdivided field somewhat like a structured data type.

Using Heap Fields

Heap fields are the "general-purpose" form of CM storage. These fields are intended to be used like C global variables: that is, they have indefinite extent, and they can be used within any lexical scope. Heap fields must be explicitly allocated, and they can be deallocated at any time and in any order.

The commonly used instructions that pertain to heap fields are:

CM_allocate_heap_field *length*
CM_deallocate_heap_field *field-id*
CM_is_field_in_heap *field-id*

The first instruction takes a *length* in bits, and returns a field-id to the front end. The second two take a *field-id* argument and perform the action indicated. Notice that the space allocator makes no reference to the format of the data the field will contain. Any field can be used for any CM data format.

Using Stack Fields

Stack fields, like C local variables, are intended for use within some bounded lexical scope such as the body of a procedure. However, stack fields do not simply disappear when the procedure is exited. Stack fields must be explicitly deallocated and in the reverse order from which they were allocated. Deallocating a stack field causes all stack fields that were allocated later to be deallocated.

The commonly used instructions that pertain to stack fields are:

CM_allocate_stack_field *length*
CM_deallocate_stack_through *field-id*
CM_is_field_in_stack *field-id*
CM_is_stack_field_newer *stack_query_field stack_base_field*

An example of stack fields in use is the following:

Example 5. Using a stack field: **swap-signed-integers.c**

```
#include <cm/paris.h>

swap_signed_integers( a, b, len )
     CM_field_id_t a, b;
     unsigned int len;
{
     CM_field_id_t temp;

     temp = CM_allocate_stack_field( len );

     CM_s_move_1L( temp, a, len );
     CM_s_move_1L( a, b, len );
     CM_s_move_1L( b, temp, len );

     CM_deallocate_stack_through( temp );
}
```

Notice that the stack field is deallocated before the procedure exits. This is the way that C compilers make local variables "automatically" disappear. Notice also that the

same front-end type, **CM_field_id_t**, is used to store the ID's of both stack fields and heap fields.

Storage Management

The difference in the recommended use between heap fields and stack fields does not arise from differences in extent: both heap fields and stack fields persist until they are deallocated (or until the program terminates). The difference in recommended use arises from the way the system manages the two kinds of memory:

- Heap fields, if explicitly deallocated, can be deallocated in any order without side effects on other heap fields.
- Stack fields should be explicitly deallocated, and they must be deallocated in the reverse order in which they were allocated. If a stack field is not deallocated explicitly, it may be deallocated as a side effect of deallocating an earlier stack field.

When fields are allocated, storage is reserved for heap fields and stack fields at opposite ends of CM memory (see Figure 8). The stack is managed with the standard Last In First Out (LIFO) stack protocol: new stack fields are always allocated—and always deallocated—at the top of the stack. As a result, the stack remains packed.

Heap fields, in contrast, are allocated in the first available space. Early in the program, the space is probably the top of the heap. However, since heap fields can be deallocated in any order, gaps tend to form over time and later fields may be placed in these gaps. See Figure 8 for the result: the stack is packed and the fields appear in the order allocated, but the heap is fragmented and the fields are not in the order allocated.

The consequences of these features of storage management for system efficiency are:

- Stack storage is always space-efficient, whereas the space efficiency of heap storage tends to deterioriate over time.
- Allocating a stack field is somewhat faster than allocating a heap field, depending on the degree of fragmentation of the heap. However, the difference is not as great as between static and automatic variables on a serial machine.

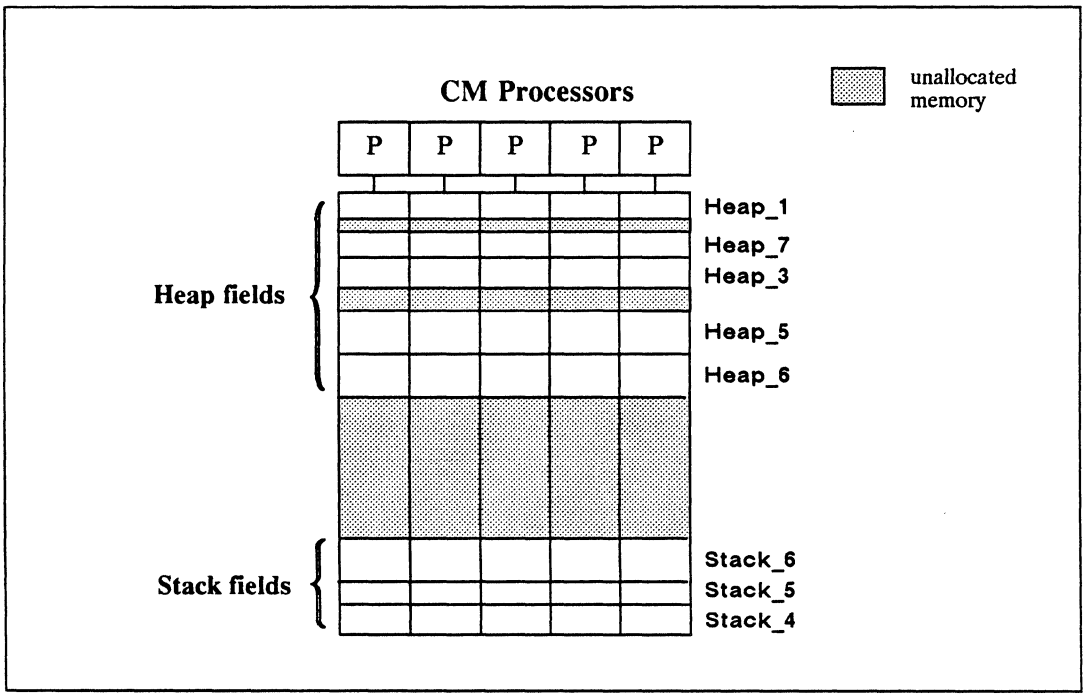

Figure 8. State of CM memory at a midpoint in program execution

3.5 Bit-Field Operands

Paris distinguishes between a data operand and the field that contains it. A field is simply some number of bits that have been allocated and which can therefore be accessed. A data operand, however, is the set of bits specified in a call to a Paris routine. The bit-field operand should fall entirely within an allocated field, but it need not be coterminous with the field.

Field-id's and Lengths

A bit-field operand is specified by:

- A "pointer" to the bit address at which to begin operating in each processor. The field-id indicates the first bit of an allocated field.
- The number of bits on which to operate. This length specifier can be any value between the minimum length of the data format and the length of the field.

NOTE

Although field-id's are associated with CM memory addresses, they are *not* C pointers and programs should never attempt to use them for indirect addressing. To be precise, in the present implementation of Paris, field-id's are indices into a front-end array of structures, where each structure describes a CM field.

A bit-field operand need not begin at the beginning of an allocated field. The following instruction takes a field-id and an unsigned integer offset and returns a new field-id.

CM_add_offset_to_field_id *field-id offset*

As example layouts, consider a 32-bit field that may contain either an integer or a floating-point number. The address indicated by the field's ID is the first bit of the field (bit 0), as shown in Figure 9. This is the least significant bit of the integer or of the significand. The number's sign, if any, is stored in the last bit (bit 31). If the length specifiers of the floating-point number were **23** and **8**, the exponent begins at bit 23.

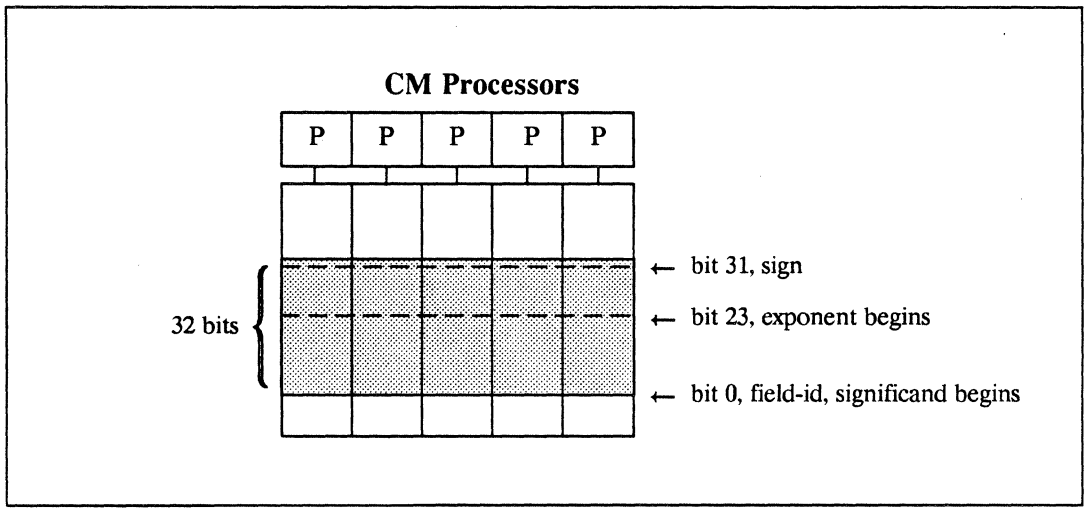

Figure 9. Field-id and data layout in a 32-bit field

By adjusting the starting point and the length, a program can extract the constituent subfields of this 32-bit floating-point number. The following fragment copies the significand into field **b**, the exponent into field **c**, and the sign into field **d**.

```
a = CM_allocate_heap_field( 32 );
b = CM_allocate_stack_field( 32 );
c = CM_allocate_stack_field( 8 );
d = CM_allocate_stack_field( 1 );

CM_f_move_constant_1L( a, 5.86, 23, 8 );

CM_u_move_1L( b, a, 23 );
CM_u_move_1L( c, CM_add_offset_to_field_id( a, 23 ), 8 );
CM_u_move_1L( d, CM_add_offset_to_field_id( a, 31 ), 1 );
```

Similarly, by adjusting starting points and lengths, programs can use **CM_move** instructions to perform bit shifts, as shown in Example 6. These shift routines move specified bits from a source field into a position in the destination field that is offset to the right or left, and then move zeros in to fill the offsets.

For instance, Figure 10 shows the change that results in each processor from a left ("upward") shift by one bit position. The most significant bit is lost, and a separate operation moves a zero into the least significant bit position. Of course, the same operation could be performed in place by specifying the source field as the destination field.

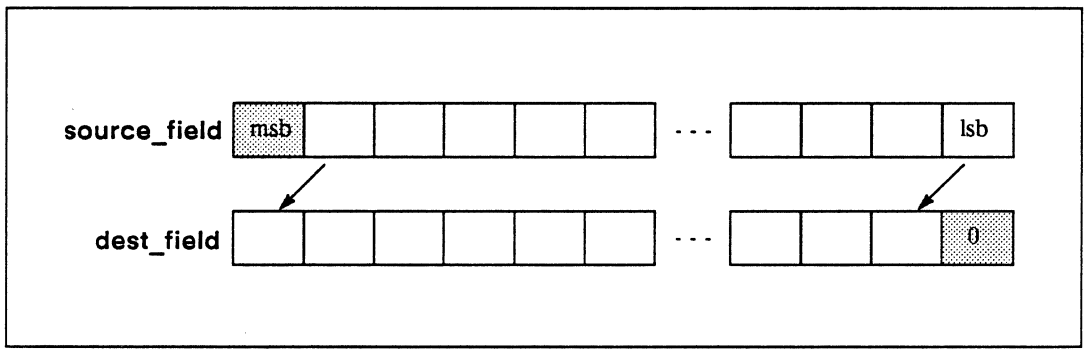

Figure 10. A left shift by one bit position

Routines that perform right and left shifts by any number of bit positions might be implemented as follows:

Example 6. Performing unsigned shifts: **unsigned-shift.c**

```
#include <cm/paris.h>

shift_left( dest, source, shift_amt, len )
     CM_field_id_t dest, source;
     int shift_amt, len;
{
     CM_u_move_1L( CM_add_offset_to_field_id( dest, shift_amt ),
                   source,
                   len - shift_amt );
     CM_u_move_zero_1L( dest, shift_amt );
}


shift_right( dest, source, shift_amt, len )
     CM_field_id_t dest, source;
     int shift_amt, len;
{
     CM_u_move_1L( dest,
                   CM_add_offset_to_field_id( source, shift_amt ),
                   len - shift_amt );
     CM_u_move_zero_1L( CM_add_offset_to_field_id(dest,
                                                  len-shift_amt),
                        shift_amt );
}
```

Using Subfields

Although Paris does not provide structured data types, programs can treat a subdivided field somewhat like a per-processor array or structure. Instructions can operate either on the entire field or on some or all of its subfields.

The advantages of using subfields are:

- *Reduced allocation time.* Be aware that allocating a CM field, even a stack field, requires much more overhead than allocating a C variable.

- *More efficient data movement.* The time required by the operations that move data between processors (including to and from the front end) is reduced by "packetizing" data items in a single field.

The major disadvantage is:

- *Disables safety checking.* The run-time safety utility (Chapter 10) checks, among other things, that data operands do not exceed the allocated length of the field that contains them. However, the safety utility has no information on the boundaries of subfields.

A program that treats data both as a whole field and as subfields is shown in Example 7. A complex number is stored in a 64-bit field as two 32-bit subfields, one for the real part and one for the imaginary part:

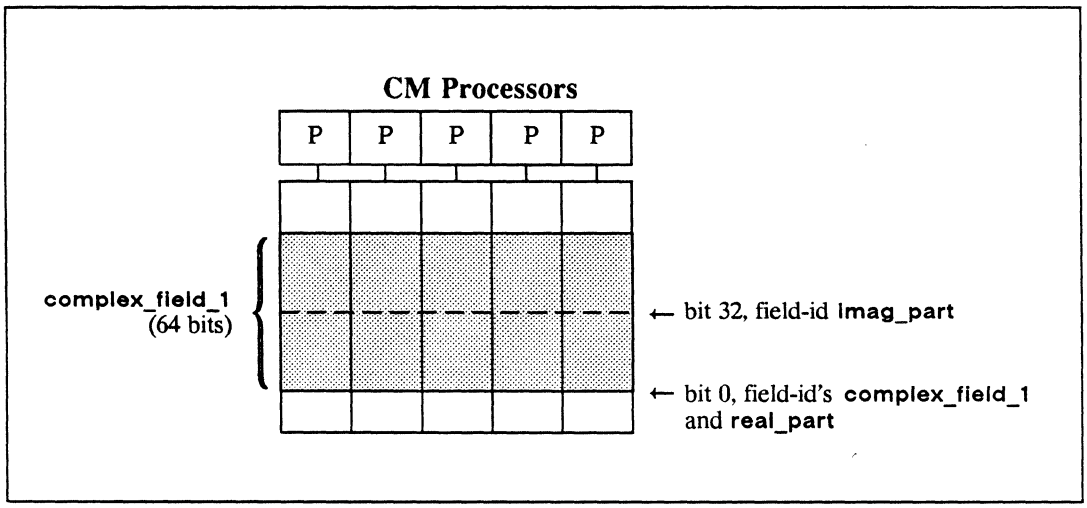

Figure 11. Field-id's in a subdivided field that represents a complex number

The following program operates separately on the subfields to multiply the complex number by 10.0. It then operates on the whole field to copy the complex number to another field.

Example 7. Using subfields: **simulate-complex-number.c**

```
#include <cm/paris.h>
#include <stdio.h>

#define LEN 32

main ()
{
     CM_field_id_t complex_field_1, real_part, imag_part,
                   complex_field_2;

     CM_init();

     complex_field_1 = CM_allocate_stack_field( 2*LEN );
     real_part = complex_field_1;
     imag_part = CM_add_offset_to_field_id( complex_field_1,LEN );

     CM_set_context();

/* Initialize the subfields with 32-bit floating-point numbers */

     CM_f_random_1L( real_part, 23, 8 );
     CM_f_random_1L( imag_part, 23, 8 );

/* Multiply the complex number by the real value 10.0 */

     CM_f_multiply_constant_2_1L( real_part, 10.0, 23, 8 );
     CM_f_multiply_constant_2_1L( imag_part, 10.0, 23, 8 );

/* Copy the complex number to another field */

     complex_field_2 = CM_allocate_stack_field( 2*LEN );
     CM_u_move_1L( complex_field_2, complex_field_1, 2*LEN );

/* Other code */

/* Signal program completion from the front end */

     printf( "\nProgram execution completed.\n" );

}
```

Notice that the last Paris call in this program uses the **_u** variant of **CM_move** to copy the field's contents. This variant treats the contents as a straight binary number (the format of an unsigned integer). The **_f** variant should not be used, since it would take two length specifiers and treat the field's contents as a single floating-point number.

Creating a Parallel Structure

A program can define a front-end structure and then mimic its memory layout within a CM field. The correspondence between the two layouts permits convenient data movement between the front end and the CM by means of the instructions that read and write to specified CM processors (see Chapter 7).

This section shows a simple case of mimicking a structure in a field, followed by some example macros that users can define to implement the general case.

First, consider an arbitrary front-end structure:

```
typedef struct {
    int a;    /* 32 bits */
    char b;   /*  8 bits */
} new_type;

new_type my_fe_structure;
```

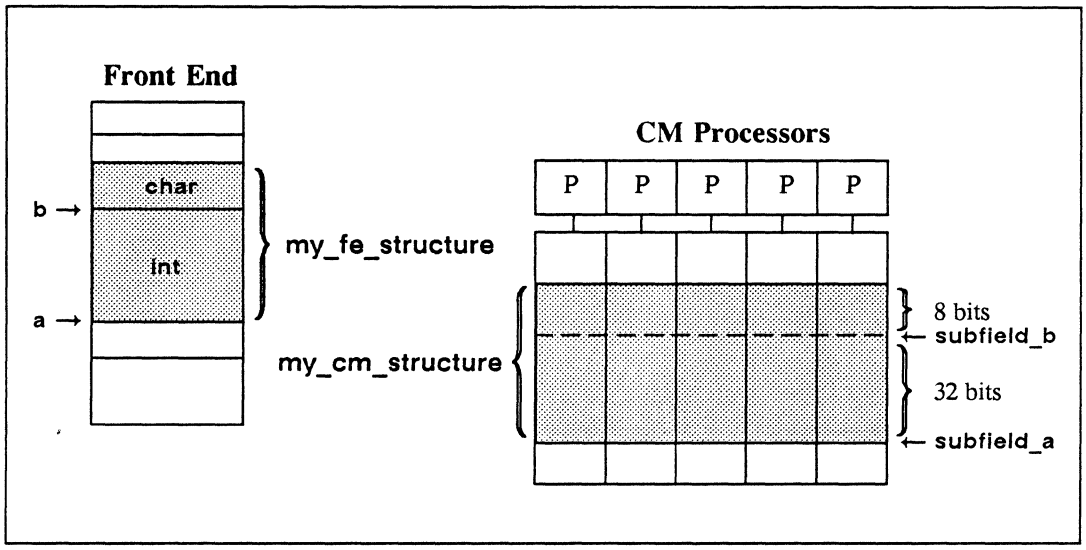

Figure 12. A CM field subdivided to match the layout of a front-end structure

Creating the parallel analogue of this particular layout is straightforward:

```
CM_field_id_t my_cm_structure, subfield_a, subfield_b;

my_cm_structure = CM_allocate_stack_field( 40 );
subfield_a = my_cm_structure;      /* specify len=32 when using */
subfield_b = CM_add_offset_to_field_id( my_cm_structure, 32 );
                                   /* specify len=8 when using */
```

More generally, a program can use macros to match a CM field to the layout of any specified front-end structure. Example 8 shows a possible implementation of such macros. These macros use **sizeof** to find the length in bytes of the front-end structure and then calculate the offsets of the structure's members. These values are multiplied by **8** to convert the lengths to bits for the CM calculations. The macro parameters are:

type	the type of the front-end structure
slotname	the name of a front-end structure member
obj	the field-id of the CM field that mimics the structure

Example 8. Creating a parallel structure: **create-cm-struct.h**

```
#define TYPELEN(type) ( 8 * sizeof( type ))

#define ALLOCATE_TYPE(type)    \
     CM_allocate_stack_field( TYPELEN( type ))

#define SLOT_OFFSET(type,slotname)  \
     (8 * (int) &(( ( type * )0 )->slotname ))

#define CMREF(obj,type,slotname)    \
     CM_add_offset_to_field_id( (obj), SLOT_OFFSET(type,slotname))
```

Thus, to mimic the layout of **my_fe_structure** in a CM field called **my_cm_structure**:

```
my_cm_structure = ALLOCATE_TYPE( new_type );
```

To reference the subfield that corresponds to member **b**:

```
CMREF( my_cm_structure, new_type, b )
```

Chapter 4

Context and Control

Data parallel programming is distinguished from both serial programming and control parallel programming by its single in-line thread of control. Each processor in the CM processor array contains a different data point, but all execute exactly the same instruction on their data points at the same time. Any processors for which the instruction is not relevant are instructed to sit idle for that time. We say that the *context* has narrowed to a smaller *active set* of (virtual) processors.

NOTE

Context relates to subselecting elements of a data set; it does not relate to selecting among data sets. For example, given a program that operates on points and lattices, context manipulation serves to subselect certain points *or* to subselect certain lattices. Context in the sense of choice among data sets—the points versus the lattices—is expressed in Paris through the concept of *vp-sets* and the *current vp-set* (see Chapter 5).

In Paris programming, the flow of control is handled entirely on the front end under the direction of the C code in the program. That is, C statements direct all branching, looping, and recursive operations and all subroutine calls. These actions often determine which Paris instructions are sent to the CM. Once the instructions are sent, however, there is no further variation in the thread of control. The only control-related determination made on the CM is which processors are to participate in the next in-line instruction.

4.1 The Context Flag

Each CM (virtual) processor contains a predefined one-bit entity called the *context flag*. This flag serves as a processor mask: the processors whose context flag is set to **1** participate in the next instruction; those whose context flag is set to **0** do not. Context is always determined explicitly by the user program. There are no Paris instructions that set the context flag as a side effect.

Although the context flag is like a field, Paris supports no way to reference it. Instead, Paris provides specialized instructions that operate on the context flag; all such instructions have the element **_context** in their names. The commonly used instructions for manipulating context are:

CM_set_context
Makes all processors active (sets context flags to **1**)

CM_clear_context
Makes all processors inactive (sets context flags to **0**)

CM_load_context *source*
Moves the value from a specified one-bit source field into the context flag in all processors

CM_store_context *dest*
Moves the value of the context flag into a one-bit destination field in all processors

CM_logand_context *source*
Clears the context flag in all processors where the value in the one-bit source field is **0**

CM_logior_context *source*
Sets the context flag in all processors where the value in the one-bit source field is **1**

CM_global_count_context
Returns the number of active processors

CM_global_logior_context
Returns **0** if no processors are active or **1** if any processor is active

For example, the following fragment adds **1** to all the odd values in **field_a**, leaving the even values unchanged. To narrow the context to only those processors whose **field_a** value is odd, the code calls **CM_logand_context** with **field_a** as *source*. Since this in-

struction operates on only one bit, the effect is to check the least significant bit in **field_a** and clear the context flag if that bit is **0** (that is, if the value is even).

```
CM_set_context();
CM_u_random_1L( field_a, LEN, 255 );

CM_logand_context( field_a );
CM_u_add_constant_2_1L( field_a, 1, LEN );
```

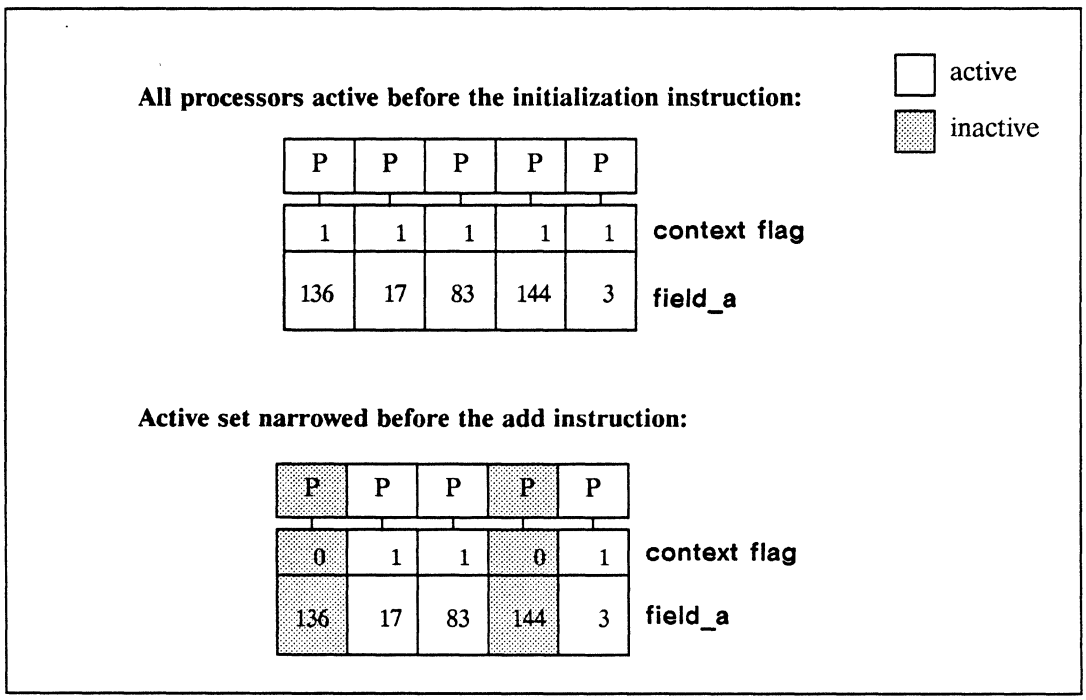

Figure 13. Narrowing the active set of processors

Unconditional Instructions

Practically all Paris instructions are *conditional*: they execute only in those processors whose context flag is set to **1**. A few instructions are *unconditional:* they execute in all processors regardless of the state of the context flag. The major categories of unconditional instructions are:

- Instructions whose name contains the element **_context**. These instructions operate on the context flag itself.

- Instructions whose name contains the element **_always**. These instructions ignore the context flag.

  For example, **CM_s_move_always_1L** is the unconditional variant of **CM_s_move_1L**. The **_always** instructions are faster than the conditional variants; they are useful when the program is unconcerned with preserving the values in inactive processors.

- Certain specialized communications instructions, identified in the *Paris Reference Manual.*

Another category of instructions that ignore the context flag are allocation intructions, such as **CM_allocate_heap_field**. Allocation is primarily a front-end operation; it simply earmarks certain CM memory addresses as allocated and associates them with a front-end field-id. No action happens on the CM processors from an allocation instruction, the context flags are not checked, and the field in question is always allocated across all processors.

The Test Flag

A common source of the values used to determine context is another predefined state bit, the *test flag.*

The test flag is the destination for the Paris comparison instructions: **eq, ne, gt, ge, lt,** and **le**. These operations are implemented for signed and unsigned integers and floating-point numbers. Each is provided in three forms: compare two fields, compare a field with a constant, and compare a field with zero. Within each processor, the instruction compares the two values and returns true or false (**1** or **0**) to the test flag.

For example:

```
CM_s_eq_2L( field_a, field_b, 32, 16 );

CM_f_gt_zero_1L( field_c, 23, 8 );

CM_u_le_constant_1L( field_d, 100, 32 );
```

As with the context flag, Paris provides no way to reference the test flag directly. Instead, Paris provides specialized instructions with the element **_test** in their names. These instructions are similar to the list of **_context** instructions shown above.

Since the test flag is most often used as a source for loading the context flag, Paris provides a specialized (unconditional) instruction for making this transfer:

CM_logand_context_with_test

For example:

```
CM_set_context();
CM_f_gt_zero_1L( field_c, 23, 8 );
CM_logand_context_with_test();
```

In processors where the value in **field_c** is greater than zero, the comparison instruction sets the test flag; in other processors, it clears the test flag. The **logand** instruction then narrows the context by setting the context flag to **0** in all processors where the test flag contains **0**. The next in-line instructions will execute only in those processors where the value in **field_c** is greater than zero.

4.2 Conditional Constructs

C provides one form of conditional operation, the **if** statement (with or without an appended **else** clause). In C/Paris, we can distinguish three kinds of conditional operations, which differ according to the nature of the control expression. These are:

- Branching on a front-end condition
- Branching on a front-end condition that is a reflection of CM state
- Choice of actions within the CM according to each processor's value in a specified field.

The last operation is more properly called *contextualization*. This construct uses a field as the analogue of a control expression and manipulates CM context—activates and deactivates processors—according to the values in that field.

With a Front-End Condition

C/Paris does not extend the C **if** statement. The control expression provided must be scalar, and the action is what we would expect in C:

```
if (condition) {
     /* block 1: Paris calls, with or without serial C code */
}
else {
     /* block 2: Paris calls, with or without serial C code */
}
```

In this paradigm, *condition* is a regular scalar expression on the front end. If it is non-zero (true), then block 1 executes; if it is zero (false), then block 2 executes. In no case do both blocks execute.

With a Scalar CM Condition

A useful variant in C/Paris is to use a scalar CM value as the control expression. As shown in Chapter 2, scalar CM values are returned as the result of either a global reduction operation or of a call to **CM_*type*_read_from_processor_1L**.

For example, the overflow flag, mentioned in Chapter 3, is set as a side effect in any processor where an arithmetic result overflows the destination field. A program can check whether overflow has occurred and conditionally take action:

```
CM_s_s_power_2_1L( dest, source, LEN );

if ( CM_global_logior_overflow() ) {
   fprintf( stderr,"Overflow from exponentiation.\n" );
   exit();
}
else
   printf( "Exponentiation succeeded.\n" );
```

This fragment raises the value in **dest** to the **source** power, an operation that invites overflow in small fields. The instruction **CM_global_logior_overflow** returns **1** if any overflow flag is set. If overflow has not occurred, the global reduction value is **0** and only the **else** statement executes.

With a Parallel CM Condition

Paris does not support branching on the CM. In the parallel analogue of a conditional construct, all blocks are executed on the front end *and* on the CM. The program determines which CM processors participate in any given block by manipulating context to reflect the local (per-processor) value in a field.

For example, compare the serial and parallel versions of a construct that increments **a** if **a** is less than **100**, and decrements it otherwise.

```
if ( a<100 )
   a++ ;
else
   a-- ;
```

Where **a** is a field, processors need to take different actions according to the local value of **a**. All the actions are specified as Paris calls, and all are executed one at a time in exactly the order called. The context flag is set and cleared to indicate which processors perform which action:

```
CM_set_context();

CM_u_lt_constant_1L( a, 100, LEN );          /* "condition" */
CM_logand_context_with_test();
CM_u_add_constant_2_1L( a, 1, LEN );         /* "then" */

CM_invert_context();
CM_u_subtract_constant_2_1L( a, 1, LEN );    /* "else" */
```

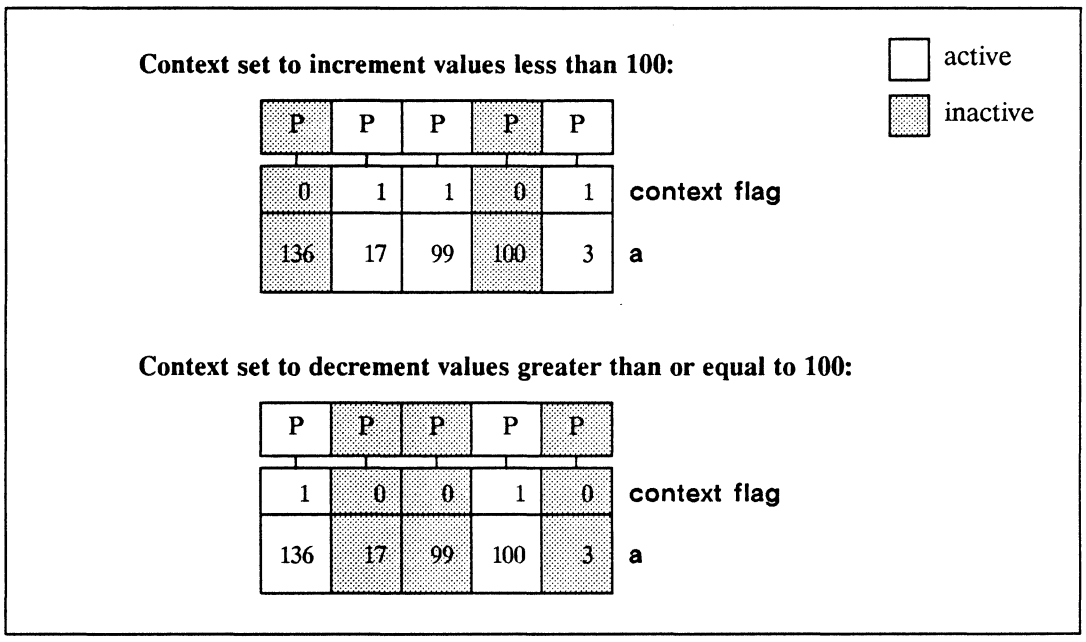

Figure 14. Manipulating context to express a parallel condition

Notice that there is no branching in this fragment: all lines of code are executed. The processors where **a** is less than **100** execute the **add** instruction while the others sit idle; then, the processors where **a** is greater than or equal to **100** execute the **subtract** instruction while the others sit idle.

Because contextualization does not involve branching, there is no reason why the respective sets of active processors have to be mutually exclusive. The active sets in the above fragment are mutually exclusive only because we used **CM_invert_context**, which reflects the initial state of the field.

In contrast, consider a fragment where the second contextualization reflects the state of the field after the first operation, rather than the initial state.

```
CM_set_context();

CM_u_lt_constant_1L( a, 100, LEN);          /* "condition" */
CM_logand_context_with_test();
CM_u_add_constant_2_1L( a, 1, LEN);         /* "then" */

CM_set_context();

CM_u_ge_constant_1L( a, 100, LEN);          /* "condition" */
CM_logand_context_with_test();
CM_u_subtract_constant_2_1L( a, 1, LEN );   /* "then" */
```

Values in field **a** that were originally less than **100** might become equal to **100** after the **add** operation. Therefore, it is quite possible for a processor to participate in both "branches" of the operation, as the middle processor in Figure 15 does.

Notice that the above fragment calls **CM_set_context** before the second comparison instruction. The comparison instructions are conditional; if the context were not reset, the second comparison would execute only within the narrowed context, and only the middle processor in Figure 15 would execute the **subtract** instruction.

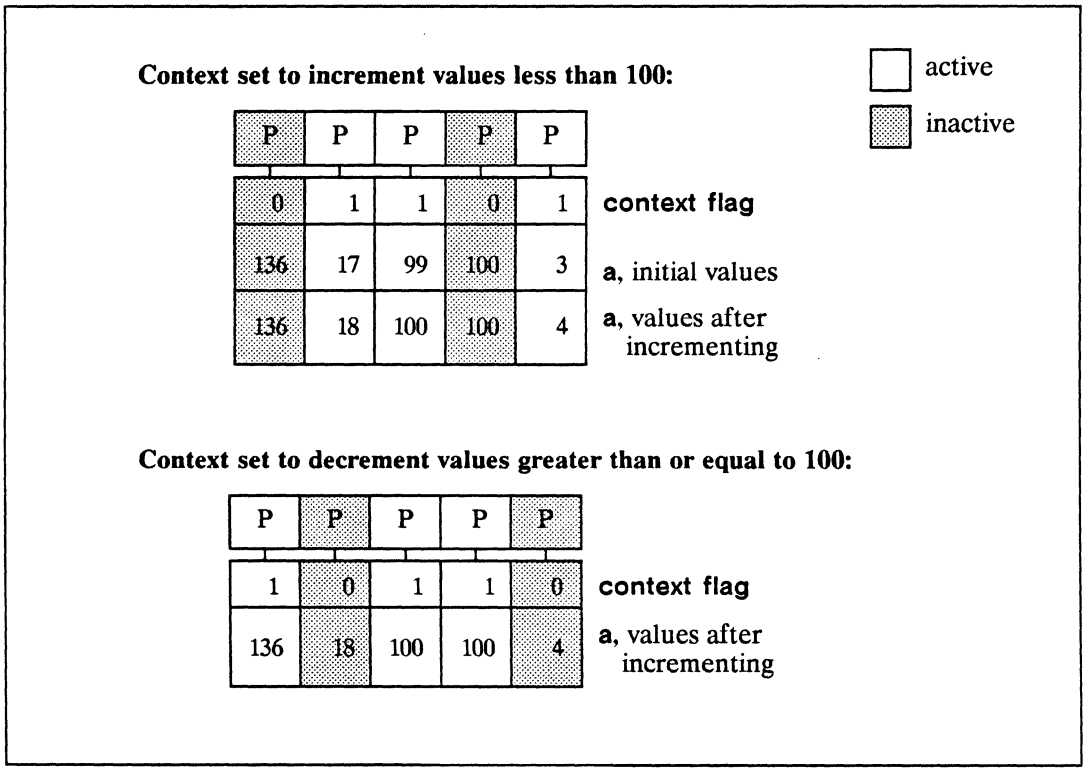

Figure 15. Manipulating context into non-mutually-exclusive active sets

With Nesting and Returns

Finally, the explicit manipulation of context permits constructs that resemble nested **if** statements and returns to previous "levels" or contexts:

- Nested conditionals are expressed by progressively narrowing the context.
- Returns are expressed by saving the context at a given point and later restoring the saved context.

For example, the following fragment begins by narrowing the context. This code shows the methods for further narrowing and for inversion within the narrowed context (the "nested" conditional). It also illustrates returning to a previous context after the last operation.

Example 9. Simulating nested conditionals and returns: **nesting-and-returns.c.fragment**

```
/* Narrow the initial context in some way */
     CM_load_context( source );

/* Save the current context */
     CM_store_context( saved_context );

/* The condition */
     /* any one-bit field or flag, such as the test flag
        after a comparison instruction */

/* Deactivate processors where the condition is false */
     CM_logand_context_with_test();

/* Block 1: the "then" block */
     /* various Paris and front-end operations */

/* Invert the context but do not activate any processors that
   were not active at the time of the condition */

     CM_invert_context();
     CM_logand_context( saved_context );

/* Block 2: the "else" block */
     /* various Paris and front-end operations */

/* Restore context as it was at the time of the condition */
     CM_load_context( saved_context );
```

4.3 Iterative Constructs

The data parallel model provides several constructs that are analogous to serial looping operations. These are:

- "Iteration" over data points
- Iteration with a front-end termination condition
- Iteration with a CM termination condition

The behavior of the C **while**, **do**, and **for** statements is not extended in any way, but the embedded statements can be Paris calls and the control expression can be a reflection of CM state.

Iteration and Parallelism

"Iteration" over data points, when performed on CM data, requires no special control constructs: it is the essence of data parallel programming. For example, consider a trivial C program that iterates several times over array elements:

Example 10. An iterative C program: **add-array-elements.c**

```
#include <stdio.h>
#define  ARRAY_SIZE  16384

main()
{
     int a[ ARRAY_SIZE ], b[ ARRAY_SIZE ], sum[ ARRAY_SIZE ];
     int i, agg_sum;

     for( agg_sum = 0, i=0; i<ARRAY_SIZE; i++ ){
            a[i] = 2;
            b[i] = 3;
            sum[i] = a[i] + b[i];
            agg_sum += sum[i];
     }

     printf( "\nThe sum of all the sums is %d.\n", agg_sum );
}
```

This program is the functional equivalent of the simple C/Paris program **add-parallel-constants.c** shown in Chapter 2. The operations that are iterative in C—initializing the arrays, summing the elements, and computing the aggregate sum—are expressed in Paris as parallel operations:

```
CM_s_move_constant_1L( field_a, 2, LEN );
CM_s_move_constant_1L( field_b, 3, LEN );
```

```
CM_s_add_3_1L( field_sum, field_a, field_b, LEN );
agg_sum = CM_global_s_add_1L( field_sum, LEN );
```

Iteration with Scalar Termination

Iteration can also be performed within CM processors. In one construct, the loop continues in every processor until a front-end control expression is false. The paradigm is straightforward:

```
while (front-end expression) {
    /* Various Paris calls with or without serial C code */
}
```

For example, the following program loops over subfields in a field as if they were a per-processor array. It creates a subfield and initializes it, and then repeats this action in every processor until the front-end condition **i<SUBFIELD_COUNT** is false. The value moved in is incremented with each iteration: the first subfield gets **0**, the second gets **1**, and so on. In this example, five subfields are created in each processor, as shown in Figure 16.

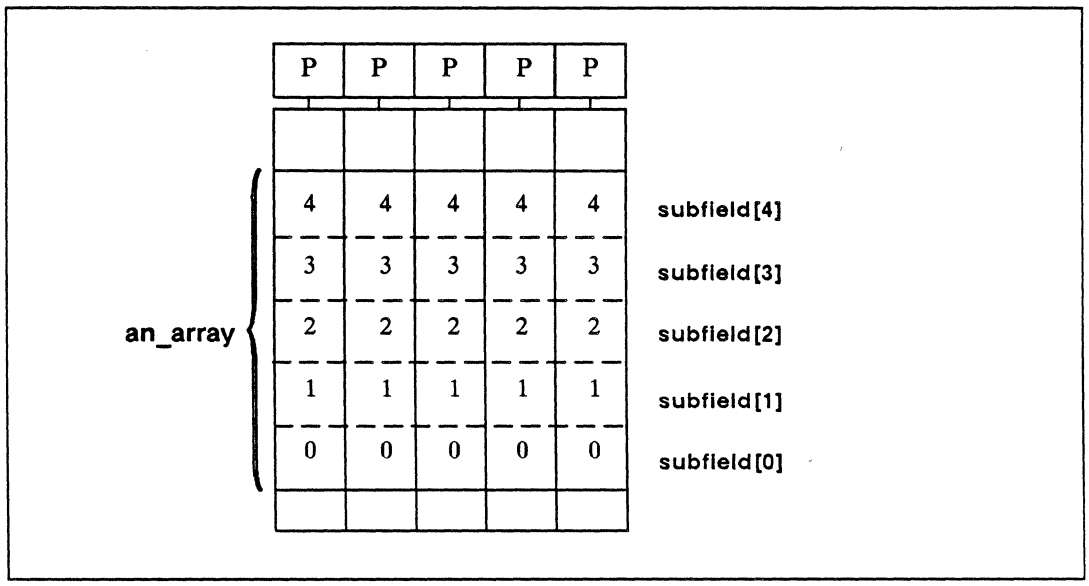

Figure 16. Results of an iterative subdivide-and-initialize operation on a field

Example 11. Per-processor iteration with front-end termination: **seed-local-arrays.c**

```
#include <cm/paris.h>
#include <stdio.h>

#define SUBFIELD_COUNT 5
#define SIZE 16

main()
{
     CM_field_id_t an_array, subfield[ SUBFIELD_COUNT ];
     int i;

     CM_init();
     CM_set_context();

     an_array = CM_allocate_heap_field( SIZE * SUBFIELD_COUNT );

     for ( i=0; i<SUBFIELD_COUNT; i++ ) {
          subfield[i] =
             CM_add_offset_to_field_id( an_array, i * SIZE );
          CM_s_move_1L( subfield[i], i, SIZE );
     }

     /* other code */

     printf( "\nProgram execution completed.\n" );
}
```

Notice that because termination is determined by a front-end constant and the CM context does not change, all the CM processors perform the body of the **for** construct the same number of times.

Iteration with Parallel Termination

When the termination of an iterative construct depends on a CM value in each processor, the processors do not necessarily perform the action the same number of times. The control expression must be a scalar value, but it can be a reflection of CM state. That is, the control expression can be either the result of either a global reduction operation or of a call to **CM_*type*_read_from_processor_1L**. A paradigm is:

```
while (global-reduction-result) {
     /* Various Paris calls with or without serial C code */
}
```

With this construct, it is possible to have CM processors repeat some action until a local condition is met. As each processor reaches the termination condition, that processor is deactivated but the others continue. The front end repeatedly checks CM context and continues the front-end loop until no CM processors are left active.

For example, the following procedure takes a field of integers and its length and calculates the base-2 logarithm of each value in the field. The action is to divide by **2** repeatedly until the quotient is less than **1**; the number of iterations required to meet this condition is the the floor of $\log_2$ of the original value. When the value in a processor becomes less than **1**—or, since we are working here with integers, equal to **0**—that processor becomes inactive. The front end calls **CM_global_logior_context** before each iteration, and terminates the loop when this instruction returns **0**.

Example 12. Per-processor iteration with CM termination: **log2-of-int.c**

```
#include <cm/paris.h>

log2_of_int( source, result, s_len, r_len )
     CM_field_id_t source, result;
     unsigned int s_len, r_len;
{
     CM_field_id_t temp_source, saved_context;

     temp_source = CM_allocate_stack_field( s_len );
     saved_context = CM_allocate_stack_field( 1 );

     CM_store_context( saved_context );
     CM_s_move_1L( temp_source, source, s_len );
     CM_s_move_zero_1L( result, r_len );

     CM_s_ne_zero_1L( temp_source, s_len );
     CM_logand_context_with_test();

     while( CM_global_logior_context() ){
          CM_s_truncate_constant_2_1L( temp_source, 2, s_len );
          CM_s_add_constant_2_1l ( result, 1, r_len );
```

```
            CM_s_ne_zero_1L( temp_source, s_len );
            CM_logand_context_with_test();
        }

        CM_load_context( saved_context );
        CM_deallocate_stack_through( temp_source );
    }
```

Chapter 5

Configuring Virtual Processors

Paris presents to the user an abstract machine architecture that is, not surprisingly, very much like the physical Connection Machine architecture. The one major extension is virtual processing, which permits a program to specify nearly any number of processors when it allocates data on the CM.

The virtual processing mechanism enables each CM physical processor to simulate some specified number of virtual processors (VPs). The processor's memory is shared among several VPs, and the physical processor is automatically time-sliced among the data that pertains to all the virtual processors that it is simulating. For instance, with a *virtual processor ratio* of 4, four instances of each memory field are allocated in each physical processor's memory and the physical processor performs each instruction four times in sequence.

The mapping of virtual processors onto physical processors is transparent to the user. Paris encourages programmers to think entirely in terms of virtual processors, both for memory allocation and for computation. In this view, the virtual machine appears as an *n*-dimensional array of processors whose size and shape is under program control.

5.1 Why Virtual Processors?

Given that the data parallel programming model associates a processor with each data point, virtual processing greatly increases the expressive power of Paris:

- *Large data sets.* Although the CM system provides up to 65,536 processors, it is not unusual for data sets to have hundreds of thousands or even millions of elements. The virtual processor abstraction permits the number of CM processors to be logically increased to that needed for any size data set.

- *Scalability.* Since the processors devoted to a problem are virtual rather than physical, a data parallel program is not tied to any given machine size. The virtual processor mechanism automatically provides the number of "processors" called for, which allows a program to run unchanged on any size CM system.

- *Natural data layout.* The virtual processor mechanism simulates the shape, as well as the size, of a data set. For instance, graphics applications are usually laid out as a 2-dimensional grid of pixels, while modeling the diffusion of heat through a metal block requires a 3-dimensional layout of data points. Programs can configure virtual processors into *n*-dimensional grids and specify the length of each dimension, thus reflecting the natural shape of *n*-dimensional data points.

- *Optimized communications.* Each virtual processor has a unique address, which allows any other processor (including the front end) to transfer data to it. In addition, specialized Paris instructions rely on the logical shape of a virtual processor grid to perform high-speed nearest-neighbor communications and cumulative computations along any of the axes of the grid.

- *Multiple data sets.* Many problems, of course, have several data sets of different sizes and shapes. The virtual processor mechanism is dynamic: it allows different virtual configurations to coexist, and it permits a program to create and destroy virtual configurations as needed at run time.

5.2 Overview of Configuration Procedure

A particular configuration of virtual processors is called a *vp-set*. Each vp-set has a *geometry*, which specifies the number of virtual processors in the vp-set and their logical organization in *n*-space. When CM memory is allocated, the field is associated with exactly one vp-set, and the field shares the geometry—the size and shape—of its vp-set.

For example, consider an arbitrary CM field, **field_a**. The many flavors of **field_a** shown previously in this manual have all been allocated within Paris's default vp-set. When a program does not explicitly create a vp-set, the system creates one with a geometry that is 2-dimensional (as nearly square as possible) and the same size as the physical machine. Thus, **field_a** in all previous illustrations is 2-dimensional, and its size is determined at run time. The virtual processor ratio (VPR) of the default vp-set is of course 1.

Alternatively, **field_a** could be allocated in an explicitly defined vp-set of nearly any size and shape. If the size is twice that of the physical machine, then the VPR of **field_a**'s vp-set is 2. If allocated in a vp-set with a multidimensional geometry, **field_a** is multidimensional and the VPR is the product of the dimension sizes divided by the size of the physical machine.

The following fragment shows the essential procedure for creating a vp-set and associating memory with it. The remainder of this chapter elaborates on these four steps:

1. Create a geometry, using **CM_create_geometry**
2. Create a vp-set associated with that geometry, using **CM_allocate_vp_set**
3. Make the vp-set the *current vp-set*, using **CM_set_vp_set**
4. Allocate CM memory in the current vp-set, using any of the field allocation instructions

Example 13. Creating a vp-set: **create-vp-1dim.c.fragment**

```
int                dimensions[1];
CM_geometry_id_t   geometry;
CM_vp_set_id_t     vp_set;
CM_field_id_t      field;

CM_init();

dimensions[0]   =  16384;
geometry        =  CM_create_geometry( dimensions, 1 );
vp_set          =  CM_allocate_vp_set( geometry );

CM_set_vp_set( vp_set );
field           =  CM_allocate_heap_field( 32 );

CM_set_context();

/* various operations on the field */
```

5.3 Creating a Geometry

A *geometry* is a front-end object that describes an *n*-dimensional grid of elements. When later associated with a vp-set, the geometry defines the configuration of the processors in that vp-set.

Procedure

The following instruction creates a geometry and returns on the front end a geometry-id, which the program can (optionally) assign to a variable of type **CM_geometry_id_t**:

CM_create_geometry *dimension-array rank*

The *dimension-array* operand is a C array whose element values are the lengths of the axes of the geometry. In the example above, this argument is an array of one element, initialized as 16384 (16K):

```
int dimensions[1];
dimensions[0] = 16384;
```

The second argument to **CM_create_geometry** is a *rank*, an unsigned integer that specifies the number of dimensions of the geometry. This value can be any integer from 1 to 31, inclusive. In this example, the rank is 1 and the call that creates the geometry is:

```
CM_geometry_id_t geometry;
geometry = CM_create_geometry( dimensions, 1 );
```

Restrictions

The current restrictions on defining geometries are:

- The length of each axis must be a power of 2. Their product—the total number of virtual processors—is therefore a power of 2.
- The product of the axis lengths must be an integer multiple of the physical size of the CM system or section that executes the program. This integer is the virtual processor ratio, which varies according to physical machine size.
- It follows that the VPR must also be a power of 2.

Future versions of Paris may remove the restriction that VPRs must be a power of 2, but the restriction that they must be integer multiples of physical machine size is likely to remain.

Examples

For example, the following geometry definitions are all legal.

- A 2-dimensional geometry of total size 32,768:

```
CM_geometry_id_t geometry_2D;
int dim[2]  = { 8192, 4 };

geometry_2D = CM_create_geometry( dim, 2 );
```

  If a program containing this geometry executes on a 32K CM, the VPR is 1; on a 16K system, the VPR is 2. This program cannot execute on 64K physical processors because the VPR would be less than 1, violating the second restriction.

- A 3-dimensional geometry of total size 16,384:

```
CM_geometry_id_t geometry_3D;
int dim[3];

dim[0] = 16;
dim[1] = 512;
dim[2] = 2;

geometry_3D = CM_create_geometry( dim, 3 );
```

  This program can execute only on a 16K or 8K set of CM processors.

- A 1-dimensional geometry that is set to current machine size:

```
CM_geometry_id_t geometry_1D;
int dim[1];

dim[0] = CM_physical_processors_limit;
geometry_1D = CM_create_geometry( dim, 1 );
```

  This program can execute on any size CM system or section.

Retrieving Attributes

Paris provides a number of instructions that inquire about the size and shape of a geometry and the attributes of its axes. All such instructions take a geometry-id, and some take an integer that identifies the axis of interest. The integer is the appropriate subscript of the dimension-array that was used to define the geometry.

Examples of the inquiry instructions are:

CM_geometry_total_processors *geometry–id*
CM_geometry_total_vp_ratio *geometry–id*
CM_geometry_rank *geometry–id*
CM_geometry_axis_length *geometry–id axis*
CM_geometry_axis_vp_ratio *geometry–id axis*

The complete list of inquiry instructions appears in the *Paris Reference Manual.* Since these instructions pertain to a front-end object (the geometry), they are all unconditional.

Optimization

The rank and dimension sizes define a geometry sufficiently for the system to provide a correct mapping of virtual to physical processors. Many such mappings are possible, and all are equally efficient for programs that involve little or no communication between logical neighbors on a grid axis.

In programs that do perform communications between nearest neighbors or cumulative computations along grid axes, the programmer might wish to specify further properties of a geometry. These properties include:

- The *ordering* of the axes, which influences the particular embedding of the logical grid into the physical grid
- The *weight* of the axes, which influences whether the virtual processors on an axis are laid out within, rather than across, physical processors or laid out across processors that are all located on the same CM chip

The system optimizes interprocessor communication along certain axes of the geometry according to their weight and ordering properties. To specify these properties, create the geometry by calling **CM_create_detailed_geometry**, which is described in the *Paris Reference Manual* and illustrated in Appendix G, "Drawing Lines."

5.4 Creating a Vp-Set

Procedure

Once a geometry is defined, the program can use that geometry to create one or more vp-sets.

CM_allocate_vp_set *geometry-id*

This instruction takes a geometry-id and creates a vp-set of the size and shape described by the geometry. The instruction returns a vp-set-id of type **CM_vp_set_id_t**. As shown in Example 13 above:

```
CM_vp_set_id_t vp_set;

    /* ... */

geometry = CM_create_geometry( dimensions, 1 );
vp_set   = CM_allocate_vp_set( geometry );
```

Changing Shape

The size of a vp-set is fixed at the time of the vp-set's creation. It shape, however, can be changed at any time by associating it with a different geometry:

CM_set_vp_set_geometry *vp-set-id geometry-id*

For example, a program that operates on fields in a 3-dimensional configuration might need to change the shape temporarily to 1-dimensional, perhaps to permit a cumulative operation across all the processors. (Cumulative computations are performed along a grid axis, as shown in Chapter 6.) The virtual processors are therefore reconfigured into a different logical organization, although their total number does not change:

```
int first_dim_array[3]  = { 512, 16, 2 };
int second_dim_array[1] = 16384;

    /* ... */
```

```
geometry_3D = CM_create_geometry( first_dim_array, 3 );
my_vp_set   = CM_allocate_vp_set( geometry_3D );

    /* various operations in 3 dimensions */

geometry_1D = CM_create_geometry( second_dim_array, 1 );
CM_set_vp_set_geometry( my_vp_set, geometry_1D );

    /* operations on the same data points in 1 dimension */
```

This fragment reconfigures a 3-dimensional grid of processors into a 1-dimensional grid of the same total size. No data actually moves, but the logical layout of data points is changed. Be aware that the mapping of processors between the two grids is not what one might expect from similar operations on serial computers: specifically, the layout of processors is neither row-major nor column-major. Paris programs should not depend on any particular mapping of processors from one grid to another.

The geometry-id currently associated with any vp-set can be retrieved by executing:

CM_vp_set_geometry *vp-set-id*

Deallocating Geometries

Associating a vp-set with a new geometry implicitly destroys its association with its previous geometry. If the previous geometry will not be used again, the program can free up system resources by deallocating it:

CM_deallocate_geometry *geometry-id*

It is an error to deallocate a geometry that is still associated with some vp-set.

Deallocating Vp-Sets

A program can also deallocate vp-sets that are no longer needed, provided that they no longer have storage allocated within them. Deallocating a vp-set does not affect its associated geometry.

CM_deallocate_vp_set *vp-set-id*

It is an error to deallocate a vp-set that still has memory fields associated with it.

5.5 Setting the Current Vp-Set

A program can create any arbitrary number of vp-sets, but only one vp-set is active at a time. This vp-set, known as the *current vp-set,* is the only vp-set in which Paris instructions can execute. All other vp-sets are latent: they cannot execute instructions.

Certain interprocessor communication instructions can operate across vp-sets, as described in Part III of this manual. However, only the VPs in the current vp-set perform the action of the instruction (sending or getting messages). The VPs in the other, non-current, vp-set are simply the passive destination or source of the messages.

Procedure

To make a vp-set current, use:

CM_set_vp_set *vp–set–id*

As shown in Example 13:

```
CM_set_vp_set( vp_set );
```

The default vp-set coexists with any vp-sets that the program creates. Until this instruction is executed, the default vp-set is the current vp-set. After this instruction is executed, all Paris instructions operate *only* within the argument vp-set until such time as the current vp-set is changed again.

Retrieval

The ID of the current vp-set is always available as the value of the Paris variable **CM_current_vp_set**. For example, to determine the number of processors in the current vp-set, a program could call:

```
int n;
n = CM_geometry_total_processors
        ( CM_vp_set_geometry( CM_current_vp_set ));
```

If the program has performed any operations within the default vp-set, it is wise to give this vp-set an identifier before making a user-defined vp-set current. Without such an identifier, the default vp-set cannot be referenced or made current again.

For example:

```
CM_vp_set_id_t apples, oranges;

     /* various operations within the default vp-set */

apples = CM_current_vp_set;
CM_set_vp_set( oranges );

     /* various operations within the user-defined vp-set */

CM_set_vp_set( apples );

     /* various operations within the original vp-set */
```

5.6 Allocating Memory

At the time of its creation, a vp-set has the full complement of flags (context, test, overflow, and carry) but no associated memory. The program uses the storage allocation instructions to allocate memory fields in vp-sets. Each field is allocated in all virtual processors in the vp-set.

Procedure

The instructions introduced earlier allocate fields in the current vp-set:

CM_allocate_heap_field *length*
CM_allocate_stack_field *length*

Programs can also allocate fields in a vp-set that is not necessarily current:

CM_allocate_heap_field_vp_set *vp-set-id length*
CM_allocate_stack_field_vp_set *vp-set-id length*

To determine which vp-set a field is associated with, use the following instruction (which returns a vp-set-id):

CM_field_vp_set *field-id*

Memory Layout

Normally, it is useful to think of each virtual processor as having its own allocated memory and to picture virtual processors' memories as separate, per-processor heaps and stacks. The diagrams shown in Chapter 3 in the discussion of CM storage management are intended to be abstractions of virtual, not physical, processors and their memories.

However, a brief digression into physical layout is useful for clarifying the restrictions on field deallocation and for predicting the efficiency of Paris programs.

The determining factor in physical layout is the VPR, which is the number of virtual processors that a physical processor is simulating for each vp-set. In the mapping of a set of virtual processors onto physical processors, *all* physical processors are used and the virtual processors are "spread out" as much as possible across the physical processors. This amounts to saying that the VPR is never less than 1 and that it is kept as low as possible.

The VPR of each vp-set is determined at run time by the number of physical processors available to the program in relation to the size of the vp-set. For example, imagine that the ubiquitous **field_a** is allocated in a vp-set of size 32,768. Figure 17 shows the physical layout of this field when the program executes on 32K processors (at left) and on 16K processors (at right).

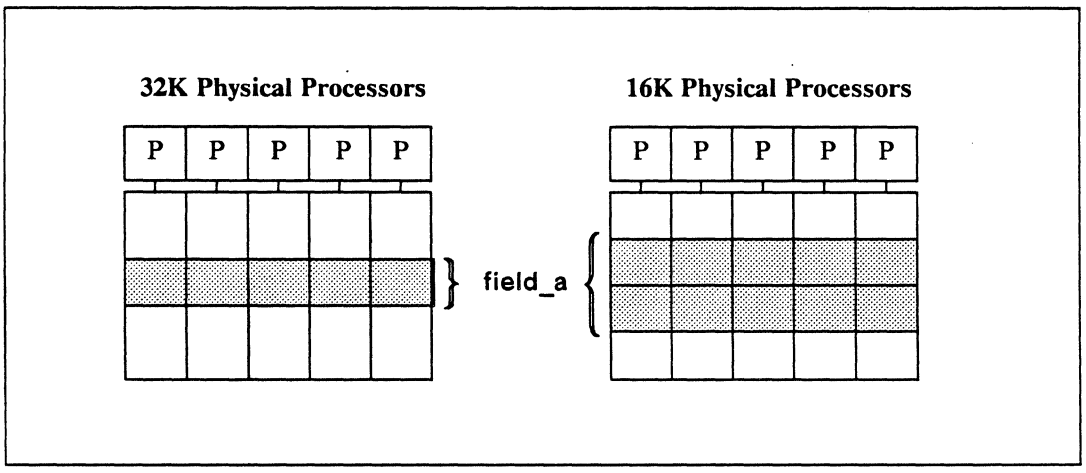

Figure 17. Physical layout of a field in a 32K vp-set

Notice that the shape of the vp-set is irrelevant to its physical layout in memory. Shape determines which VPs are considered nearest neighbors along a logical axis, but VPR

alone determines the number of "banks" required in each physical processor to accommodate the fields of a vp-set. The field shown in Figure 17 could be of any logical dimensionality, regardless of how many memory banks its VPR requires.

When multiple vp-sets exist, each physical processor's memory contains fields from *every* vp-set, and each field is replicated the number of times that its vp-set's VPR requires.

The VPs that are co-resident on a physical processor share a single heap and a single stack, and their fields are intermingled in more-or-less the order allocated. That is, stack fields are stored in exactly the order allocated, regardless of vp-set, whereas heap fields may depart from the allocated order to the extent that field deallocation has freed up space that was previously used, again regardless of vp-set.

For example, consider a program that has three vp-sets: **apples**, **oranges**, and **pears** (shown below as Example 14). For each of the three vp-sets, the total number of virtual processors is the product of its dimension sizes (axis lengths), and its VPR is the number of virtual processors divided by the physical machine size. Thus, when executing on 16,384 (16K) CM processors:

apples:	size = 128×128	=	16,384 (16K)
	VPR = 16K/16K	=	1
oranges:	size =		65,536 (64K)
	VPR = 64K/16K	=	4
pears:	size = 64×16×4×8	=	32,768 (32K)
	VPR = 32K/16K	=	2

The physical memory layout of the fields in the three vp-sets is shown in Figure 18. In this figure, each physical processor is simulating 7 virtual processors: 1 processor in vp-set **apples** (4 fields allocated), 4 processors in vp-set **oranges** (2 fields allocated), and 2 processors in vp-set **pears** (3 fields allocated). The order of the fields reflects the order of their allocation, certainly within the physical stack and probably within the physical heap.

The ranks and dimension sizes shown above and in Example 14 are arbitrary—what determines physical layout is the total number of processors and thus the VPR. For example, vp-set **apples** could as well have 1 dimension of size 16K or 3 dimensions of sizes 2, 8, and 1024. As long as the product of the dimension sizes is 16K, the virtual processors in **apples** will be laid out one-per-physical-processor on a 16K CM system or section.

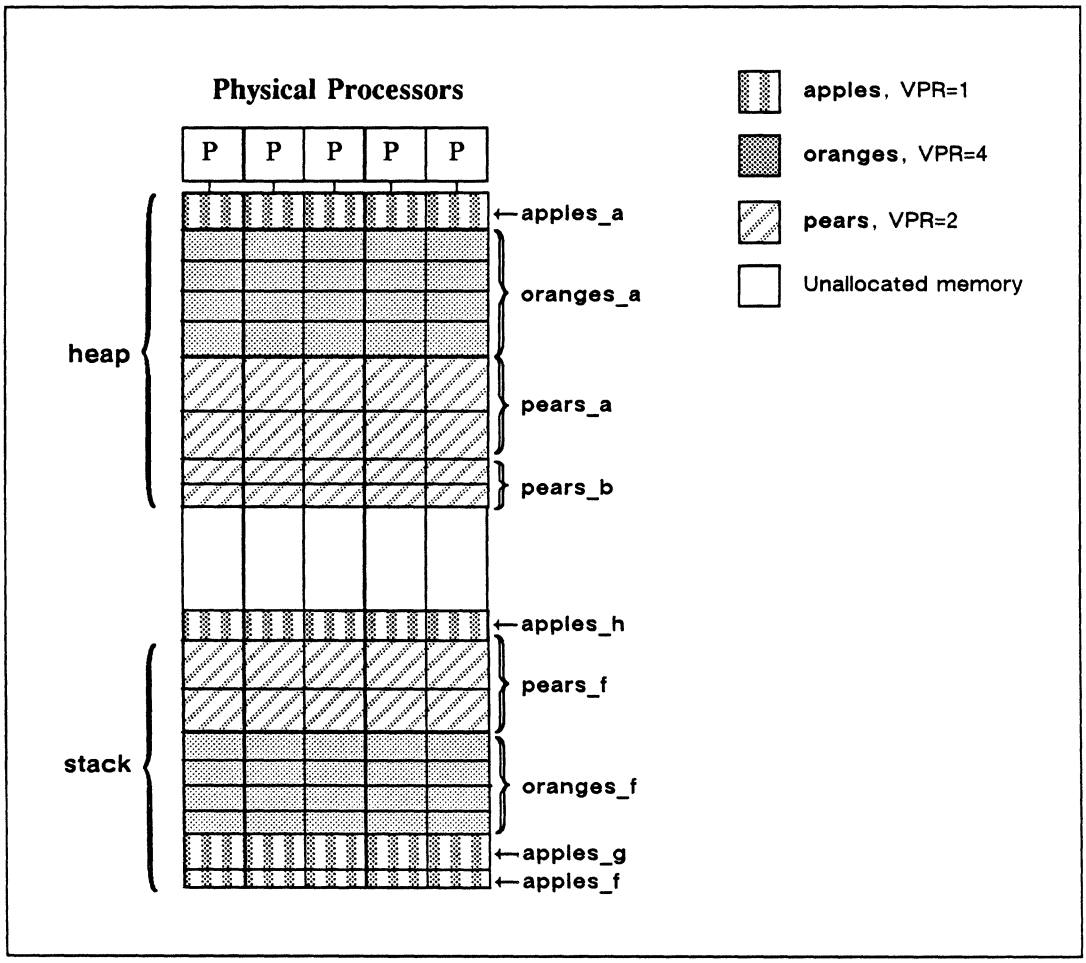

Figure 18. Physical layout of 9 fields in 3 vp-sets

Recall from Chapter 3 that the heap fields can be deallocated in any order. However, *the LIFO stack protocol applies to the physical stack*, not to the stack fields associated with a given VP. That is, deallocating field **apples_f** causes the deallocation not only of **apples_g** and **apples_h** but also of **oranges_f** and **pears_f**.

NOTE

It is an error to access any stack field after an earlier stack field has been deallocated, regardless of vp-set.

Example 14. Code underlying Figure 18: **vp-sets.c.fragment**

```
int                     dim_apples_geom[2],
                        dim_oranges_geom[1],
                        dim_pears_geom[4];

CM_geometry_id_t        apples_geom, oranges_geom, pears_geom;
CM_vp_set_id_t          apples, oranges, pears;
CM_field_id_t           apples_a, apples_f, apples_g, apples_h,
                        oranges_a, oranges_f,
                        pears_a, pears_b, pears_f;

CM_init();

/* ============================================================ */
/* Create vp-set apples, make it current, and allocate memory */

dim_apples_geom[0] =  128;
dim_apples_geom[1] =  128;
apples_geom        =  CM_create_geometry( dim_apples_geom, 2 );
apples             =  CM_allocate_vp_set( apples_geom );

CM_set_vp_set( apples );
apples_a           =  CM_allocate_heap_field( 32 );
apples_f           =  CM_allocate_stack_field( 8 );
apples_g           =  CM_allocate_stack_field( 32 );

CM_set_context();

     /* various operations on fields in vp-set apples */

/* ============================================================ */
/* Create vp-set oranges from within vp-set apples */

dim_oranges_geom[0]=  65536;
oranges_geom       =  CM_create_geometry( dim_oranges_geom, 1 );
oranges            =  CM_allocate_vp_set( oranges_geom );

/* Make vp-set oranges current and allocate memory */

CM_set_vp_set( oranges );
oranges_a          =  CM_allocate_heap_field( 16 );
orange_f           =  CM_allocate_stack_field( 12 );
```

```
    /* various operations on fields in vp-set oranges */

/* ============================================================ */
/* Create vp-set pears and allocate memory in pears while still
   within vp-set oranges */

dim_pears_geom[0]  =  64;
dim_pears_geom[1]  =  16;
dim_pears_geom[2]  =  4;
dim_pears_geom[3]  =  8;
pears_geom         =  CM_create_geometry( dim_pears_geom, 4 );
pears              =  CM_allocate_vp_set( pears_geom );

pears_a            =  CM_allocate_heap_field_vp_set( pears, 32 );
pears_b            =  CM_allocate_heap_field_vp_set( pears,  8 );
pears_g            =  CM_allocate_stack_field_vp_set(pears, 32 );

    /* more operations on fields in vp-set oranges */

/* ============================================================ */
/* Make vp-set pears current and operate on its fields */

CM_set_vp_set( pears );

    /* various operations on fields in vp-set pears */

/* ============================================================ */
/* Make vp-set apples current and operate on its fields */

CM_set_vp_set( apples );

    /* various operations on fields in vp-set apples */

/* Deallocate the earliest stack field; all stack fields in all
   vp-sets are deallocated. */

CM_deallocate_stack_through( apples_f );
```

Aside from the constraint on deallocating stack fields, the physical layout of CM memory never affects program behavior (although it may affect program performance). An understanding of physical layout does, however, help to clarify two points made earlier in this chapter:

- The size of a vp-set is fixed, but its shape can change. The total size of a vp-set is reflected in its physical layout, which cannot change; its rank and individual dimension sizes, however, are unrelated to layout and thus can change.
- The weight of an axis of virtual processors influences its mapping onto physical processors. In the more fine-tuned geometries created by **CM_create_detailed_geometry**, the user can specify the axis on which the most interprocessor communication will occur. If this axis fits within the number of memory banks required by that vp-set (that is, if VPR $\geq$ **dim**[x], where x is the heavily used axis), the system can lay out the VPs for that axis entirely within a single physical processor, thus enhancing the speed of inter-*virtual*-processor communications.

This discussion also makes it obvious that VPRs greater than 1 involve two penalties:

- Memory usage increases with VPR in a linear fashion. The virtual processor mechanism sets no limit on the size of data sets that can be handled on a one-element-per-processor basis. However, physical memory can become a constraint at very high VPRs.
- Execution time for most operations increases with VPR in a linear fashion. With a VPR of 4, for instance, each physical processor loops serially over 4 banks of memory and thus performs 4 times as many instructions as if virtual processors were not in use. The MIPS and FLOPS rates, however, are about the same as they would be for a data set the same size as the physical machine.

  An important exception is interprocessor operations that rely on local (same-physical-processor or same-chip) communications. The speed of these operations at high VPRs is often sublinear—that is, faster than a linear extrapolation would predict.

5.7 Processor Addresses

In all the operations shown thus far, each virtual processor performs computations independently of other processors. However, very few useful applications decompose into such totally independent subproblems. Instead, processors often need to transfer

data among themselves; and the front end, of course, often needs to access specified processors for purposes of I/O or manipulating context.

To facilitate interprocessor communications, each virtual processor in a vp-set has two addresses, each of which uniquely identifies that processor within that vp-set. The two addresses correspond to the two general models of interprocessor communication in the CM system:

- A *send address*: a single integer that remains constant for each virtual processor for the life of its vp-set. Any processor (including the front end) can access any other processor in the CM processor array by specifying the destination processor's send address.

  Every processor can send a message to (or get a message from) another specified processor, all at the same time. The procedures for communicating in arbitrary patterns to specified processor addresses are described in Chapter 7 of this manual.

- A *NEWS address*: a set of coordinates that reflects a virtual processor's grid position in the current geometry of its vp-set. Unlike send addresses, NEWS addresses are dependent on the current geometry of a vp-set and will change if the geometry changes.

  Nearest-neighbor communications and cumulative computations along grid axes are dependent on grid positions, although the program need specify only the pattern of communication (not actual coordinates) to perform these operations on all processors in parallel. The procedures for communicating in regular patterns are described in Chapter 6 of this manual.

Part III
Interprocessor Communications

Chapter 6

Communicating in Regular Patterns

The virtue of organizing virtual processors as a logical grid is that each processor can access the memory of another processor without computing the other's address. Instead, processors can communicate in regular patterns by specifying only a grid axis and a pattern.

The possible patterns for grid communication in Paris are:

- Nearest-neighbor communication, where each processor gets a message from the processor that is next to it on a grid axis.
- Remote-neighbor communication, where each processor gets a message from the processor that lies at some specified distance and in some specified direction in the grid.
- Block transfers of data between a front-end array and the CM processors that make up a grid of comparable size and shape.
- Cumulative, or *parallel prefix,* computation, where some combining operation (such as addition) is performed cumulatively across all processors on a grid axis in a specified direction.

Grid communications are sometimes called NEWS operations. The term *NEWS* has historical significance in designating the four nearest neighbors to any processor on a 2-dimensional grid: North, East, West, and South. Paris now supports grids from 1 dimension to 31 dimensions; the number of nearest neighbors to any given processor is $2n$, where n is the rank (number of dimensions) of the current geometry.

Although grid communication is only a subset of the communications that are possible on the CM system, it is an important subset that is optimized for speed. Specifically:

- A processor need not calculate or check the address of the processor whose memory it is to access.

- There is no possibility of *collisions*, where more than one message arrives at a processor from a single operation, and thus no need to combine in-coming messages.
- Paris guarantees that virtual processors that are nearest neighbors on a logical grid are also nearest neighbors on the physical grid.

  Paris performs virtual-to-physical mapping such that any two nearest-neighbor virtual processors are located either within the memory of a single physical processor or in physical processors on the same chip, or they are linked directly by a single wire. Thus, the CM hardware directly supports grid communications, no matter what the shape of the logical grid.

All communication along grid axes—including nearest-neighbor, remote-neighbor, and cumulative communications—necessarily occur within the current vp-set. The only grid operation that involves more than one vp-set is **CM_cross_vp_move_1L**, which copies data from a grid in the current vp-set to a grid in another vp-set. See the *Paris Dictionary Supplement,* Version 5.1, for information.

6.1 Grid Coordinates

Each virtual processor has a set of coordinates that define its position in the grid described by the current geometry of its vp-set. The *n*-tuple of the grid coordinates for each processor is its grid address or *NEWS address*, by which another processor (including the front end) can identify that processor.

The coordinates are specific to the geometry; changing the geometry of the vp-set changes the grid coordinates of the individual virtual processors. (See Chapter 7 for information on determining the new NEWS address of a given processor after a change of geometry.)

Numbering Axes and Processors

The axes of a grid and the virtual processors on each axis are numbered as one would expect in a Cartesian coordinate system: both the axes and the processors on each axis are numbered with sequential (contiguous) unsigned integers beginning with zero. For example, the numbering of axes and processors that results from the following vp-set definition is shown in Figure 19.

```
unsigned int dim[3];
dim[0]    = n;
dim[1]    = m;
dim[2]    = 2;

3D_geom      =  CM_create_geometry( dim, 3 );
my_vp_set    =  CM_allocate_vp_set( 3D_geom );
```

Recall that the product of *n*, *m*, and 2 must be a power-of-2 multiple of the physical machine size.

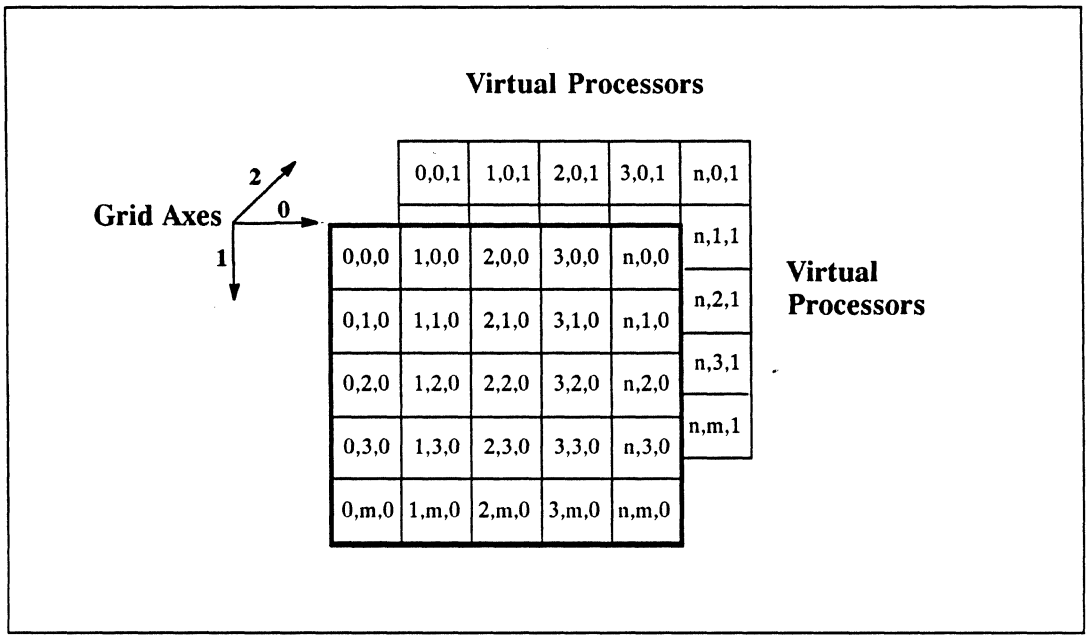

Figure 19. Grid coordinates and axis numbers in a 3-D geometry

Retrieving Grid Coordinates

Paris provides an instruction by which each virtual processor can determine its own coordinate on a specified axis of the current geometry. The instruction places the coordinate value for each processor in a destination field in the same processor.

CM_my_news_coordinate_1L *dest axis dest-length*

This instruction is conditional and operates only within the current vp-set.

In most cases, it is safe to specify *dest-length* as, say, 16 or 32 bits. To determine the *minimum* length needed for the destination field, the program needs to compute the number of bits required to represent the highest coordinate value on a specified grid axis. For this purpose, we use:

CM_geometry_coordinate_length *geometry-id axis*

Example 15 uses these instructions to compute a 2-dimensional identity matrix. Each processor computes its own coordinates on the two axes, placing the values in fields x and y, respectively, as shown in Figure 20.

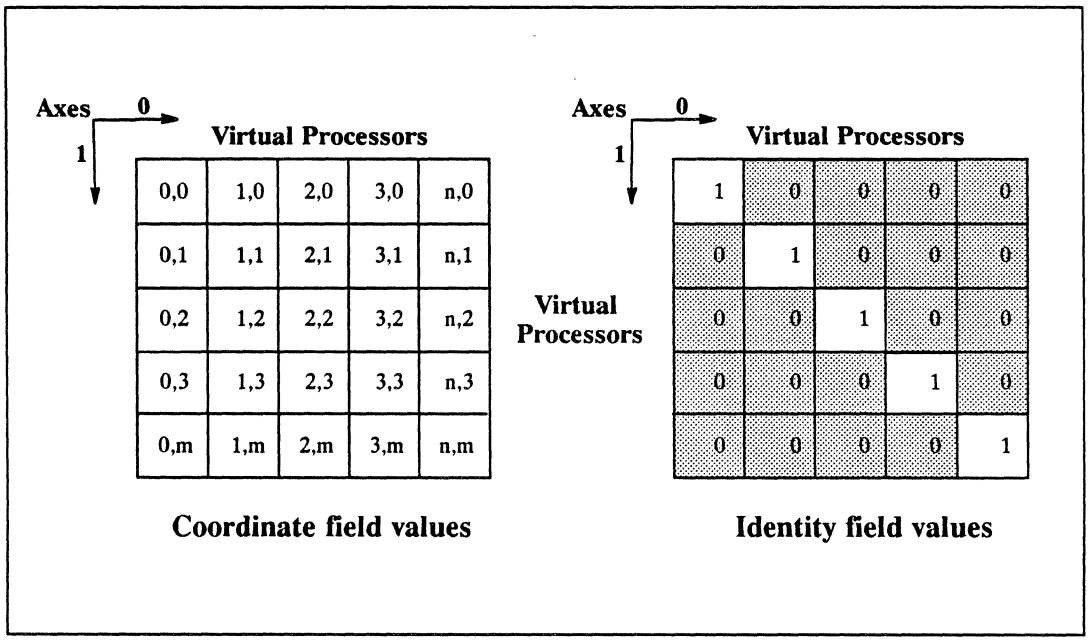

Figure 20. Grid coordinates and identity field values in a 2-dimensional geometry

The code also creates a field **identity** and initializes it to **0** in all processors. The example then compares the values of **x** and **y** in each processor; in processors where the two are equal, the comparison instruction sets the test flag to **1**. Loading the test flag into the context flag then deactivates all processors where the two coordinates are *not* equal. Finally, the example moves the value **1** into the **identity** field in all active processors, which are those for which the two coordinate values are equal.

Example 15. Retrieving grid coordinates: **identity-matrix.c.fragment**

```
current_geom = CM_vp_set_geometry( CM_current_vp_set );
x_len        = CM_geometry_coordinate_length( current_geom, 0 );
y_len        = CM_geometry_coordinate_length( current_geom, 1 );

x        = CM_allocate_stack_field( x_len );
y        = CM_allocate_stack_field( y_len );
identity = CM_allocate_stack_field( 1 );

CM_my_news_coordinate_1L( x, 0, x_len );   /* 0 specifies axis */
CM_my_news_coordinate_1L( y, 1, y_len );   /* 1 specifies axis */

CM_u_move_zero_1L( identity, 1 ); /* initialize identity field*/

CM_u_eq_2L( x, y, x_len, y_len ); /*  set test flag if x = y  */
CM_logand_context_with_test();    /*  deactivate if x not = y */

CM_u_move_constant_1L( identity, 1, 1 ); /* set identity to 1  */
```

6.2 Nearest-Neighbor Communication

Nearest-neighbor, or NEWS, communication is the simplest and one of the most efficient means of interprocessor communication on the CM. Every virtual processor accesses the memory of an immediate neighbor on the logical grid, all at the same time and in the same direction.

Basic NEWS Instructions

The basic NEWS communication instructions direct each active processor to send or get a message from the memory of its nearest neighbor in a specified direction. Since the concept of "neighbor" has meaning only on a grid axis, the instructions also take an axis operand:

CM_get_from_news_1L *dest source axis direction length*
CM_send_to_news_1L *dest source axis direction length*

The *direction* operand is specified in C/Paris as either **CM_upward** or **CM_downward**, both of type **CM_communication_direction_t**. The upward pattern indicates the neighbor with the next-higher grid coordinate on the axis; the downward pattern indicates the neighbor with the next-lower coordinate.

Notice that these instructions take only one length specifier: the *dest* and *source* field operands are taken to be of the same length. Also, the two fields must be either disjoint (no shared bits) or identical (all bits shared); they may not overlap partially.

NEWS Accesses and Context

If all processors are active, then **CM_get_from_news_1L** with direction **CM_upward** is exactly equivalent to **CM_send_to_news_1L** with direction **CM_downward**. The difference between the two instructions concerns context. The processor that performs the action of the instruction (getting or sending) must be active; the neighbor processor need not be active. (See Figure 21, which shows NEWS transfers between memory fields in a 1-dimensional grid of processors.)

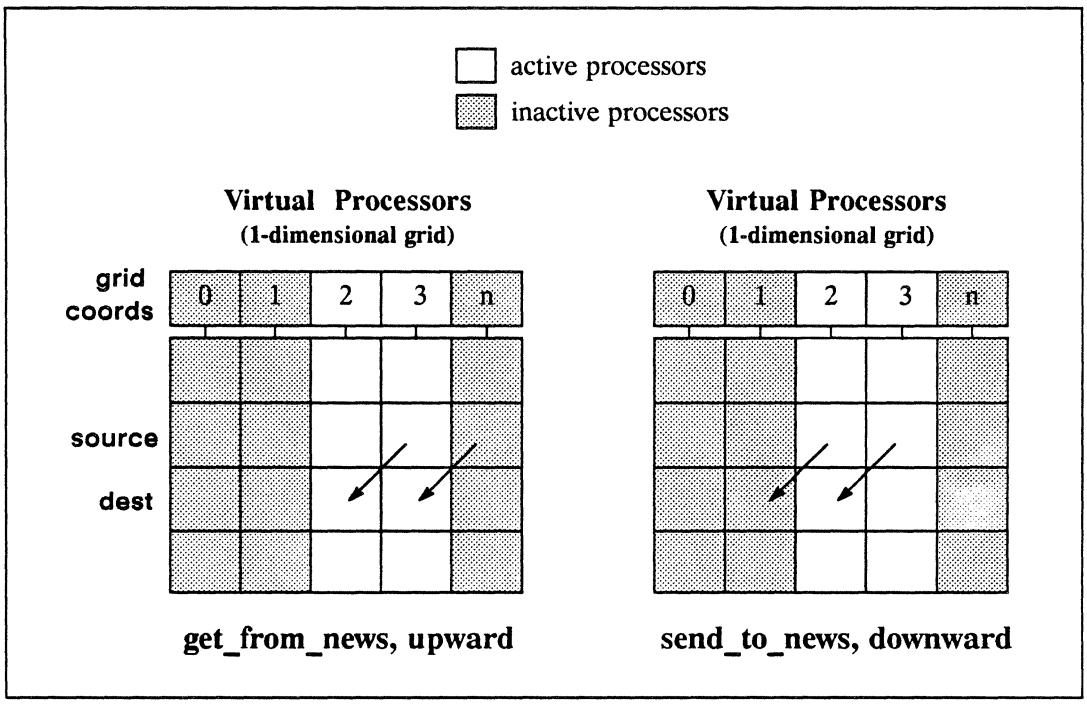

Figure 21. Effect of context on NEWS communication

Paris also provides unconditional variants of the basic NEWS communication instructions. The action of the two unconditional instructions is exactly equivalent if the *direction* operands are opposite.

CM_get_from_news_always_1L *dest source axis direction length*
CM_send_to_news_always_1L *dest source axis direction length*

Example of NEWS Communication

To illustrate data transfers between nearest neighbors, Example 16 shows a simple procedure whereby each processor averages its value in a source field with the values of its two nearest neighbors in the same field.

Example 16. Accessing two nearest neighbors: **neighbor-average.c**

```
#include <cm/paris.h>

neighbor_average( source, axis, source_len )
     CM_field_id_t source;
     unsigned int  axis, source_len;
{
     CM_field_id_t  my_value, neighbor_up, neighbor_down;

     /* allocate temporary storage as subfields in each VP */
     my_value = CM_allocate_stack_field ( source_len * 3 );
     neighbor_up =
          CM_add_offset_to_field_id( my_value, source_len );
     neighbor_down =
          CM_add_offset_to_field_id( my_value, source_len * 2 );

    /*initialize subfields*/
     CM_u_move_1L( my_value, source, source_len );
     CM_u_move_zero_1L ( neighbor_up, source_len * 2 );

    /* get value from source field in two neighbors */
     CM_get_from_news_1L( neighbor_up, source,
                          axis, CM_upward, source_len );
     CM_get_from_news_1L( neighbor_down, source,
                          axis, CM_downward, source_len );
```

```
    /*add the three values, divide by three, and truncate */
    CM_u_add_3_1L( my_value, neighbor_up, neighbor_down,
                   source_len );
    CM_u_truncate_constant_2_1L( my_value, 3, source_len );

    /*move result into source field and deallocate temporaries*/
    CM_u_move_1L( source, my_value, source_len);
    CM_deallocate_stack_through( my_value );
}
```

Notice that this procedure does not need to identify any NEWS coordinates or even the current geometry (although it is an error if the *axis* argument exceeds the rank of the current geometry when the procedure is called). Because the only communication that occurs is between neighbors in a regular pattern, no processor needs to be identified by its address.

Border Behavior

The Paris NEWS instructions wrap when a processor is on the border of a grid. That is, the processor with coordinate 0, when accessing downward, accesses the highest-numbered processor on the axis, and the highest-numbered processor accesses processor 0 when accessing upward. Thus, the grid is by default a toroidal mesh.

The program can change the default border behavior by identifying the processors on the border of the grid and deactivating them for the NEWS operation. It can then activate only the border processors and specify some other operation. These actions require the program to retrieve the NEWS coordinates of the processors on the axis of interest and compare them with the result returned by **CM_geometry_axis_length** and with 0.

For example, the following program (Example 17) performs the same NEWS operations as the procedure shown above: it averages the values of each set of three neighbors on a grid axis. However, the processors on the ends of the axis do not participate in the operations. (This program differs from the example above in various arbitrary ways; for instance, it manages storage somewhat differently, and it initializes the source field with random numbers.)

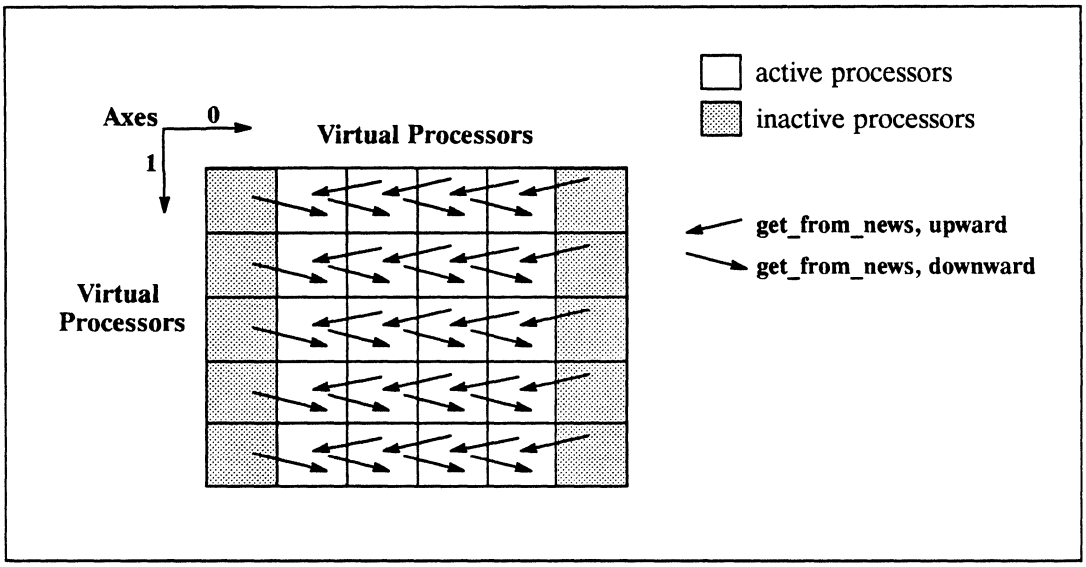

Figure 22. Deactivating border processors before NEWS communication

The program in Example 17 uses the default (2-dimensional) vp-set, and all NEWS operations occur on axis 0. The actions can be depicted as in the diagram above.

Example 17. Controlling grid border behavior: **neighbor-average-no-wrap.c**

```
#include <stdio.h>
#include <cm/paris.h>

#define FIELD_LENGTH 16

/* ============================================================ */

main()
{
    CM_field_id_t     my_value, neighbor_up, neighbor_down,
                      news_coordinate;
    unsigned int      coord_length, end_coordinate;
    CM_geometry_id_t current_geometry;

    CM_init();
```

```
/* ========================================================== */

    /* identify the geometry of the current (default) vp-set */
    current_geometry = CM_vp_set_geometry( CM_current_vp_set );

    /* determine how many bits are needed for the coordinate on
    axis 0 of the current geometry */
    coord_length =
        CM_geometry_coordinate_length( current_geometry, 0 );

    /* allocate storage as subfields in each virtual processor */
    news_coordinate =
        CM_allocate_heap_field(coord_length + FIELD_LENGTH * 3);
    my_value =
        CM_add_offset_to_field_id( news_coordinate,
                                   coord_length );
    neighbor_up =
        CM_add_offset_to_field_id( my_value, FIELD_LENGTH );
    neighbor_down =
        CM_add_offset_to_field_id( my_value, FIELD_LENGTH * 2 );

/* ========================================================== */

    /* set context and initialize fields */

    CM_set_context();

    CM_u_move_zero_1L( my_value, FIELD_LENGTH * 3 );
    CM_u_random_1L( my_value, FIELD_LENGTH,
        1<<( FIELD_LENGTH - 2 ) );  /* the limit operand */
    CM_my_news_coordinate_1L( news_coordinate, 0, coord_length );


/* ========================================================== */

    /* Deactive the end processors on the first axis */

    end_coordinate =
        CM_geometry_axis_length( current_geometry, 0 ) - 1;
    CM_u_ne_constant_1L( news_coordinate, end_coordinate,
                         coord_length );
    CM_logand_context_with_test();
    CM_u_ne_constant_1L( news_coordinate, 0, coord_length );
    CM_logand_context_with_test();
```

```
/* ========================================================== */

    /* get values from neighbors and average them */

    CM_get_from_news_1L( neighbor_up, my_value,
                         0, CM_upward, FIELD_LENGTH );
    CM_get_from_news_1L( neighbor_down, my_value,
                         0, CM_downward, FIELD_LENGTH);

    CM_u_add_3_1L( my_value, neighbor_up, neighbor_down,
                   FIELD_LENGTH);
    CM_u_truncate_constant_2_1L( my_value, 3, FIELD_LENGTH );

/* ========================================================== */

    /* signal program completion from the front end */
    printf( "Program execution completed." );
}
```

6.3 Remote-Neighbor Communication

Remote-neighbor communication refers to the parallel transfer of data in regular patterns between processors that are not nearest neighbors on a grid axis.

This form of communication differs from general communication (covered in Chapter 7) in that each pair of communicating processors is in exactly the same spacial relationship as all other pairs. Remote-neighbor communication thus shares the performance optimizations of other grid communications (no address needed, no collisions possible), as well as hardware support for what is in effect a series of nearest-neighbor transfers.

The most straightforward method of communicating with remote neighbors is by simply making repeated calls to a NEWS instruction, perhaps with changes to the *axis* and *direction* operands. For example, in the Game of Life problem shown in Appendix A, each processor needs to check a value in each of *eight* neighbors on a 2-dimensional grid (the four NEWS neighbors and the four diagonal neighbors). The diagonal neighbors cannot be accessed directly, since the underlying hardware does not support them as nearest neighbors. However, two calls to **CM_get_from_news_1L** suffice to access a diagonal neighbor:

```
CM_get_from_news_1L( neighbor_N, source, 1, CM_upward, LEN );
CM_get_from_news_1L( neighbor_NE, neighbor_N, 0,
                     CM_downward, LEN );
```

In the first line, every active processor gets a value from its upward neighbor on axis 1 and stores the value in its own **neighbor_N** field. In the second line, every active processor gets the value that its downward neighbor on axis 0 has just acquired and stores that value in its own **neighbor_NE** field. (See Appendix A for the complete program.)

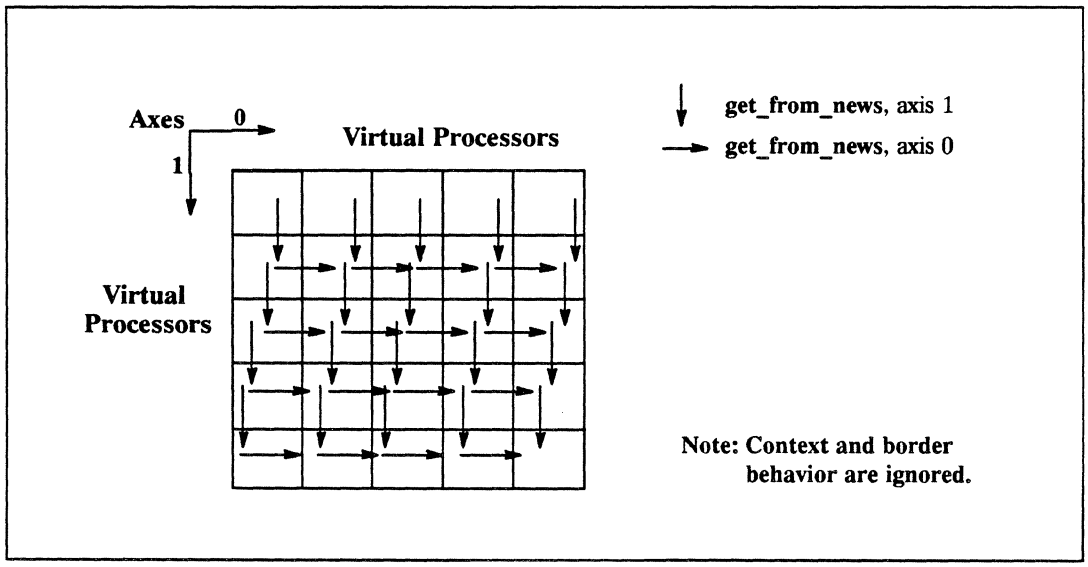

Figure 23. Accessing diagonal neighbors in two NEWS operations

For accessing remote neighbors that lie a power-of-2 distance on the same grid axis, Paris provides instructions that perform the operation in one step:

CM_get_from_power_two_1L *dest source axis distance direction length*
CM_get_from_power_two_always_1L *dest source axis distance direction length*

These instructions take the same operands as **CM_get_from_news_1L**, plus a *distance* operand. The *distance* operand is the base-2 log of the number of grid positions to be traversed.

6.4 Front-End Array Transfers

The most convenient and efficient means of transferring large amounts of data between CM memory and the front end is the Paris instructions that read and write NEWS arrays. These instructions transfer data between a virtual processor grid and a front-end array of comparable size and shape. Their implementation is optimized for comparatively high throughput.

Separate array-transfer instructions are implemented for signed and unsigned integers and for floating-point numbers. All the variants are unconditional.

The instructions take a large number of operands that describe the front-end array and the NEWS grid, as well as the CM memory field to be accessed. The array or grid can be a subarray or a portion of a grid. See the *Paris Reference Manual* for a complete description of the operands to the array-transfer instructions.

Briefly, the operands are:

CM_*type*_read_from_news_array_1L

front-end-array	front-end (dest) array
offset-vector	offsets for dest array
start-vector	lowest NEWS coords
end-vector	highest NEWS coords
axis-vector	axes of NEWS grid
source	CM source (src) field
source-len	length (bits) of src
rank	rank of dest array
dimension-vector	axes of dest array
element-length	length (bytes) of dest array elements

CM_*type*_write_to_news_array_1L

front-end-array	front-end (src) array
offset-vector	offsets for src array
start-vector	lowest NEWS coords
end-vector	highest NEWS coords
axis-vector	dims of NEWS grid
dest	CM dest field
dest-len	length (bits) of dest
rank	rank of src array
dimension-vector	dims of src array
element-length	length (bytes) of src array elements

For example, the following procedure transfers data from a 2-dimensional grid on the CM to a specified array on the front end. The procedure is simplified by having the grid be square; that is, the parameter *array_edge_size* specifies the highest NEWS coordinate on both dimensions of the grid. However, the procedure could easily be extended to higher ranks and unequal axis lengths. Also, we would need to change only the Paris call to have this procedure write from the front end to the CM.

Example 18. A block data transfer: **read-news-array.c**

```
#include <cm/paris.h>

#define RANK 2

get_square_array_from_cm( front_end_array, source_field,
                          source_length, array_edge_size )
    unsigned int    *front_end_array;
    CM_field_id_t    source_field;
    unsigned int     source_length, array_edge_size;
{
    /* offsets into front_end_array */
    int fe_offset_vector[RANK];      /* note signed integers */

    /* the "start" NEWS coordinate of the CM grid */
    unsigned cm_start_vector[RANK];

    /* the "end" NEWS coordinate of the CM grid */
    unsigned cm_end_vector[RANK];

    /* the NEWS axes to transfer */
    unsigned cm_axis_vector[RANK];

    /* the dimensions of the front-end array */
    unsigned fe_dim_vector[RANK];


    /* Initialize parameters for the array transfer */
    fe_offset_vector[0]   = 0;
    fe_offset_vector[1]   = 0;
    cm_start_vector[0]    = 0;
    cm_start_vector[1]    = 0;
    cm_end_vector[0]      = array_edge_size;
    cm_end_vector[1]      = array_edge_size;
```

```
    cm_axis_vector[0]      = 0;
    cm_axis_vector[1]      = 1;
    fe_dim_vector[0]       = array_edge_size;
    fe_dim_vector[1]       = array_edge_size;

    /* Perform the array transfer */
    CM_u_read_from_news_array_1L
        ( front_end_array, fe_offset_vector,
          cm_start_vector, cm_end_vector, cm_axis_vector,
          source_field, source_length, RANK,
          fe_dim_vector, sizeof( unsigned int ));
}
```

6.5 Cumulative Communications

Paris provides a number of extremely powerful instructions that combine computation and communication along an axis of a logical grid. Such operations are called *parallel prefix* operations, on the analogy of prefix operations in array processing (where the operation applies some combiner over all prefixes of an array).

For instance, it is frequently useful to compute partial sums of the values along a grid axis, where each processor computes the total of itself and all processors before it in a specified direction. The last processor computes the grand total. This seemingly serial operation is in fact one of the most efficient *parallel* operations on the CM system.

This section introduces the most basic of the parallel prefix operations, **CM_scan**. The extensions of this operation, **CM_reduce**, **CM_spread**, and **CM_multispread**, are described in the *Paris Reference Manual.*

Scan Operations

The **CM_scan** instructions take a binary associative operator @ and an ordered set of elements [a_0, a_1, a_2, . . .], and compute the ordered set

[a_0, (a_0 @ a_1), (a_0 @ a_1 @ a_2), . . .]

The action is intuitively obvious as the procedure for balancing a checkbook. Consider a set of checkbook transactions (credit and debit) stored one-per-processor along a NEWS grid in chronological order. The following instruction computes a running balance:

CM_scan_with_s_add_1L *dest source axis length direction inclusion smode sbit*

Ignoring the last two operands for the moment, we see that most of the others are familiar from the grid communications discussed previously. The *inclusion* operand determines whether a processor's initial value is included in the computation; it can be either **CM_inclusive** or **CM_exclusive**.

If the check transactions are stored in field **transactions** and the running balance is to be placed in field **year_balance**, the call looks like:

```
CM_scan_with_s_add_1L( year_balance, transactions, /* axis= */ 0,
                       LEN, CM_upward, CM_inclusive,
                       CM_none, CM_no_field );
```

The last two operands are discussed in the next section.

The **CM_scan** operations are provided with the arithmetic combiners **add**, **min**, and **max**, as well as with **logand**, **logior**, **logxor**, and **copy**. The arithmetic combiners are provided in **s**, **u**, and **f** variants for signed and unsigned integers and floating-point numbers. Some examples:

CM_scan_with_f_max_1L *dest source axis s-len e-len direction inclusion smode sbit*

CM_scan_with_copy_1L *dest source axis length direction inclusion smode sbit*

In a **scan_with_***type***_max** operation, for instance, each processor receives the largest of the values from the processors that precede it on the axis in the specified direction. In a **scan_with_copy** operation, each processor receives the value from the *first* processor on the axis in the specified direction.

All the **CM_scan** instructions are conditional. Inactive processors are treated as if they did not exist: their *dest* field values are not changed, and their *source* field values are not considered in the running computation.

Segmented Scan Operations

The last two operands of the **CM_scan** instructions support *segmented scans,* where the running computation "restarts" at specified points on the grid axis.

For example, a simple **scan_with_add** of a year's checkbook transactions yields partial totals (subtotals) for each of the transactions, ending with the net total for the year. To find the net total for each month, however, we need to divide the grid axis into monthly segments and restart the **scan** operation with each month.

For this purpose, we use a one-bit field where the value **1** indicates the beginning of a new segment, or *scan set.* Assume for the moment that the field **month_segment_marker** is set in this way. That is, assume that each processor whose **transactions** value is the first transaction in a month has the value **1** in the field **month_segment_marker** and all other processors have the value **0** in that field.

The call that computes net monthly cash balance is:

```
CM_scan_with_s_add_1L( month_balance, transactions, /* axis=*/ 0,
                       LEN, CM_upward, CM_inclusive,
                       CM_start_bit, month_segment_marker );
```

This call restarts the running total at the beginning of each month. The **month_balance** value associated with the last transaction of each month is the net balance for that month only, not for the year to date.

The *smode* operand **CM_start_bit** indicates that the instruction is a segmented **scan**. The alternative is **CM_none**, which was used in the non-segmented **scan** operation shown above. The last operand, *sbit,* is the field-id of the segment field, in this case **month_segment_marker**. If *smode* is **CM_none**, then *sbit* can be the dummy field-id **CM_no_field.**

How do we compute the segment bit? Assume that each transaction is associated with a date and thus has a field **my_month**. Since the transactions are sorted chronologically along a grid axis, we can use NEWS instructions to find the spots where adjoining processors have different **my_month** values. Each processor gets its downward neighbor's month and compares it with its own month. If the two values are not equal, the processor sets its own **month_segment_marker** to **1**.

The procedure is:

```
CM_get_from_news_1L( neighbor_month, my_month, 0,
                     CM_downward, LEN);
```

```
/* clear the segment bit */
CM_clear_bit_always( month_segment_marker );

/* compare month with downward neighbor's month in each proc */
CM_u_ne_1L( my_month, neighbor_month, LEN );

/* if the two months are not equal, set the segment bit to 1 */
CM_store_test( month_segment_marker );
```

Other examples that use the **CM_scan** instructions can be found in Appendix F, "Lines of Sight," and Appendix G, "Drawing Lines."

Chapter 7

Communicating in Arbitrary Patterns

Perhaps the most distinctive feature of the CM system is *general communication*, where each virtual processor transfers a message to *any* other processor by specifying the destination processor's address.

As with grid communication, every active processor communicates with some other processor, all at the same time. Unlike grid communication, the pairs of communicating processors need not be in any particular spatial relationship to each other. In fact, the source and destination processors can be in different vp-sets.

General communication is sometimes called *router communication*, a reference to the underlying packet-switching mechanism by which each message is routed along one of the many possible paths to its destination. The path chosen may vary according to the distance to be traversed and the amount of message "traffic" on a particular wire.

This chapter presents the basic information needed to perform general communication in a C/Paris program:

- Computing processor addresses
- Using the general communication instructions
- Transferring data between designated CM processors and the front end

7.1 Processor Addresses

Every processor within a vp-set is uniquely identified by an unsigned integer called its *send address*. Like NEWS coordinates, send addresses are unique only within a vp-set; it is possible for virtual processors in different vp-sets to have the same send address. Unlike NEWS coordinates, however, a processor's send address remains constant for the life of its vp-set, regardless of changes in the vp-set's geometry.

In the present version of Paris, the send addresses in each vp-set are consecutive unsigned integers beginning with 0 and extending to the total size of the vp-set minus 1. However, this feature is an artifact of the present restriction of total vp-set size to powers of 2. In future versions, the send addresses for a vp-set may *not* be consecutive.

NOTE

Paris programs should not assume that send addresses occupy a contiguous range. In particular, we discourage arithmetic on send addresses. For a contiguous ordering of all processors, please use a 1-dimensional NEWS grid.

Computing Self-Addresses

Within the current vp-set, each active processor can compute its own send address and store that value in a destination field in the same processor:

CM_my send_address *dest*

The field size needed to store the send address can be determined from the vp-set's geometry:

CM_geometry_send_address_length *geometry-id*

For example:

```
unsigned int     dest_len;
CM_field_id_t    dest_field;
CM_geometry_id_t current_geom;

current_geom  =  CM_vp_set_geometry( CM_current_vp_set );
dest_len      =  CM_geometry_send_address_length( current_geom );
dest_field    =  CM_allocate_stack_field( dest_len );

CM_my_send_address( dest_field );
```

The send address reflects only the size, not the shape, of a vp-set. If the vp-set is later associated with a different geometry, the send address of each processor remains the same because the second geometry must be of the same total size as the first (see the discussion of changing geometries in Chapter 5).

The send address of a virtual processor is composed of two parts, the physical part and the virtual part. The physical part indicates which physical processor supports the VP; the virtual part indicates a particular VP on that physical processor. The address is thus a reflection of the mapping of virtual to physical processors, which does not vary during program execution. The address may change between program runs, however, since the physical part varies with physical machine size and the virtual part varies with the vp-set's virtual processor ratio (also a function of physical machine size).

Computing Send Addresses on the Front End

Any processor, including the front end, can compute the send address of any arbitrary CM processor from that processor's NEWS coordinates in a geometry. Aside from **CM_my_send_address**, the instructions that convert NEWS coordinates to send addresses are the only supported way to obtain a send address in Paris.

Paris provides two instructions that compute, entirely on the front end, the send address of any single CM processor:

CM_sendaddr_t
CM_fe_make_news_coordinate *geometry axis news-coord*

CM_sendaddr_t
CM_fe_deposit_news_coordinate *geometry send-address axis news-coord*

The first instruction takes a geometry-id, an axis within that geometry, and the unsigned integer coordinate of a processor on that axis. It converts the NEWS coordinate into a send address on the assumption that all coordinates other than the one specified are **0**. If *geometry* is in fact 1-dimensional, the send address returned is complete. When a send address is stored on the front end, it is of type **CM_sendaddr_t**.

If the geometry is multidimensional, we need to build on the partial send address returned by **CM_fe_make_news_coordinate** by adding information about the other coordinates. The procedure is to pass that partial address as an argument to **CM_fe_deposit_news_coordinate**, along with the processor's NEWS coordinate on another axis of the geometry. By successive calls to **CM_fe_deposit_news_coordinate**, the front end can build the complete send address of a CM processor from all its NEWS coordinates.

For example, suppose we want to compute the send address of the processor whose NEWS coordinates on axes 0, 1, and 2 of **geometry_3D** are 10, 50, and 200:

```
CM_sendaddr_t      send_addr;
unsigned int       axis;
CM_geometry_id_t   geometry_3D;

send_addr = CM_fe_make_news_coordinate( geometry_3D, 0, 10 );
send_addr = CM_fe_deposit_news_coordinate( geometry_3D,
                                           send_addr, 1, 50 );
send_addr = CM_fe_deposit_news_coordinate( geometry_3D,
                                           send_addr, 2, 200 );
```

Computing Send Addresses on the CM

Analogous instructions direct each active CM processor to construct a send address from a set of NEWS coordinates. The parallel instructions are similar to the front-end instructions except that the *news-coord* operand and the "result" are fields. Also, these instructions take a length specifier for the field *news-coord*:

CM_make_news_coordinate_1L *geometry dest axis news-coord coord-len*
CM_deposit_news_coordinate_1L *geometry dest axis news-coord coord-len*

For example, suppose we want each processor in **geometry_3D** to compute its own send address from its NEWS coordinates. (This trivial exercise is an inefficient way to simulate **CM_my_send_address**, but it suffices to illustrate the generic procedure. More-useful examples are shown later in this section.) The procedure is:

1. Compute the NEWS coordinates for all the processors.
2. Create a field in the current vp-set to store the send addresses.
3. Call **CM_make_news_coordinate_1L** on the coordinates of one axis.
4. Call **CM_deposit_news_coordinate_1L** successively on the coordinates of each remaining axis.

Step 1 NEWS Coordinates

Compute the processors' NEWS coordinates (as explained in Chapter 6) and place them in, say, fields **x**, **y**, and **z**.

```
x_len = CM_geometry_coordinate_length( geometry_3D, 0 );
y_len = CM_geometry_coordinate_length( geometry_3D, 1 );
z_len = CM_geometry_coordinate_length( geometry_3D, 2 );

x     = CM_allocate_stack_field( x_len );
y     = CM_allocate_stack_field( y_len );
z     = CM_allocate_stack_field( z_len );

CM_my_news_coordinate_1L( x, 0, x_len );
CM_my_news_coordinate_1L( y, 1, y_len );
CM_my_news_coordinate_1L( z, 2, z_len );
```

Step 2 Destination Field

Next, create a field long enough to contain the send address. This field will be the destination field for all the calls to the address-building instructions.

```
dest_len = CM_geometry_send_address_length( geometry_3D );
dest     = CM_allocate_stack_field( dest_len );
```

Step 3 First NEWS Coordinate

Then, beginning with any axis of the geometry, call **CM_make_news_coordinate_1L**, specifying the coordinate field on that axis and the length of the coordinate field:

```
CM_make_news_coordinate_1L( geometry_3D, dest, 0, x, x_len );
```

This instruction converts a news coordinate into a send address on the assumption that all coordinates other than the one specified are **0**. If *geometry* were in fact 1-dimensional, the send address would now be complete.

Step 4 Other NEWS Coordinates

Since the geometry is multidimensional, build the send address with successive calls to **CM_deposit_news_coordinate_1L**, once for each remaining axis of the geometry. Specify the destination field used in Step 3 in all these calls.

```
CM_deposit_news_coordinate_1L(geometry_3D, dest, 1, y, y_len);
CM_deposit_news_coordinate_1L(geometry_3D, dest, 2, z, z_len);
```

Converting Send Addresses to NEWS Coordinates

Paris also provides instructions that extract NEWS coordinates back out from send addresses, either on the front end or in each active CM processor:

unsigned int
CM_fe_extract_news_coordinate *geometry axis send-address*

CM_extract_news_coordinate_1L *geometry dest axis send-address dest-len*

The front-end instruction returns an unsigned integer that is the NEWS coordinate of the specified processor along the specified geometry axis. The *send-address* operand of the front-end instruction is of type **CM_sendaddr_t**.

The CM instruction directs each active processor to perform the parallel analogue of that action: derive the appropriate NEWS coordinate of the processor specified in its *send-address* field and place the result in the *dest* field (which is of length *dest-len*). If *geometry* is multidimensional, we can compute the full NEWS address by making successive calls to **CM_extract_news_coordinate_1L** and storing the results in separate fields or subfields.

Example of Address Conversions

The instructions that convert between send addresses and NEWS coordinates are particularly useful when a vp-set's geometry changes. As mentioned in Chapter 5, the grid ordering of the processors in the new geometry is not what one might expect from serial computers. In fact, it is not possible to predict the new NEWS coordinates of particular processors from their coordinates in the previous geometry.

The program can determine which processors are which by having them compute their send addresses before changing the geometry. Afterward, each processor can derive its new NEWS coordinates from the send address, which has remained unchanged. For example, consider a change in geometry from 2-dimensional to 1-dimensional:

Example 19. Addresses and changing geometries: **send-addr-to-news.c.fragment**

```
CM_geometry_id_t geom_2D, geom_1D;
CM_vp_set_id_t   my_vp_set
CM_field_id_t    send_addr, news_x;
unsigned int     send_addr_len, x_len;
```

```
    /* create geometries, etc. */

my_vp_set = CM_allocate_vp_set( geom_2D );
CM_set_vp_set ( my_vp_set );

    /* various operations on the 2-D grid */

/* compute self-addresses */
send_addr_len = CM_geometry_send_address_length( geom_2D );
send_addr     = CM_allocate_heap_field( send_addr_len );
CM_my_send_address( send_addr );

/* change geometry to 1-D */
CM_set_vp_set_geometry( my_vp_set, geom_1D );

/* prepare a field for the new grid coordinates */
x_len  = CM_geometry_coordinate_length( geom_1D, /*axis =*/ 0 );
news_x = CM_allocate_heap_field( x_len );

/* Extract new grid coordinates from send addresses */
CM_extract_news_coordinate_1L( geom_1D, news_x, 0,
                               send_addr, x_len );

    /* various operations on the 1-D grid */
```

7.2 The Basic send Instruction

General interprocessor communications are performed by the **CM_send** family of instructions. These instructions move the contents of a source field in each active processor to a destination field in any specified processor. (The opposite operation, **CM_get**, is described briefly in Section 7.4 below.)

The simplest of the **send** instructions is:

CM_send_1L *dest send-address source len notify*

This instruction—and all the variants of **CM_send**—takes field-id's for the message (*source*), the send address of the destination processor, and the field (*dest*) in the destination processor where the message is to be deposited.

The **send** instructions also take a length specifier that applies to both *source* and *dest*. The message can be any length that is legal for its CM data format, as detailed in Chapter 3. In addition, all the **send** instructions take a one-bit field, *notify*, that is set in the destination processor when the message arrives.

The action of **CM_send_1L** is illustrated in Figure 24 for an arbitrary set of five virtual processors. (In this example, all the processors send to different processors; Section 7.3 discusses cases where multiple processors send to the same processor.)

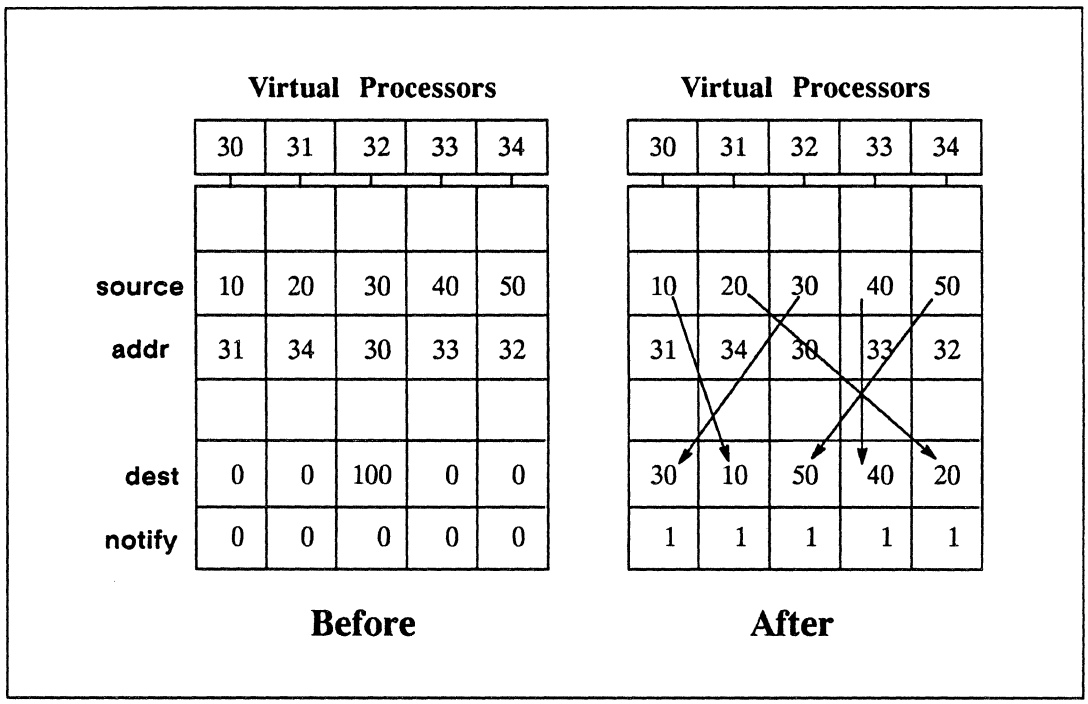

Figure 24. Change in CM state from executing **CM_send_1L**

Notice that the destination field need not be cleared before a call to **CM_send_1L**. In Figure 24, processor 32 starts with the value 100 in its *dest* field, but it is overwritten by the message from processor 34. It is wise, though, to clear the *notify* field, since a preexisting 1 in a processor that receives no message would defeat the purpose of notification.

Notice also that the four fields are disjoint in this example, which is generally advisable. Some field overlap is permitted, however, as detailed in the descriptions of the individual **CM_send** instructions in the *Paris Reference Manual.*

Effect of Context

All the **CM_send** instructions are conditional: only active processors can *send* a message. A processor does not need to be active to *receive* a message. (See Figure 25.)

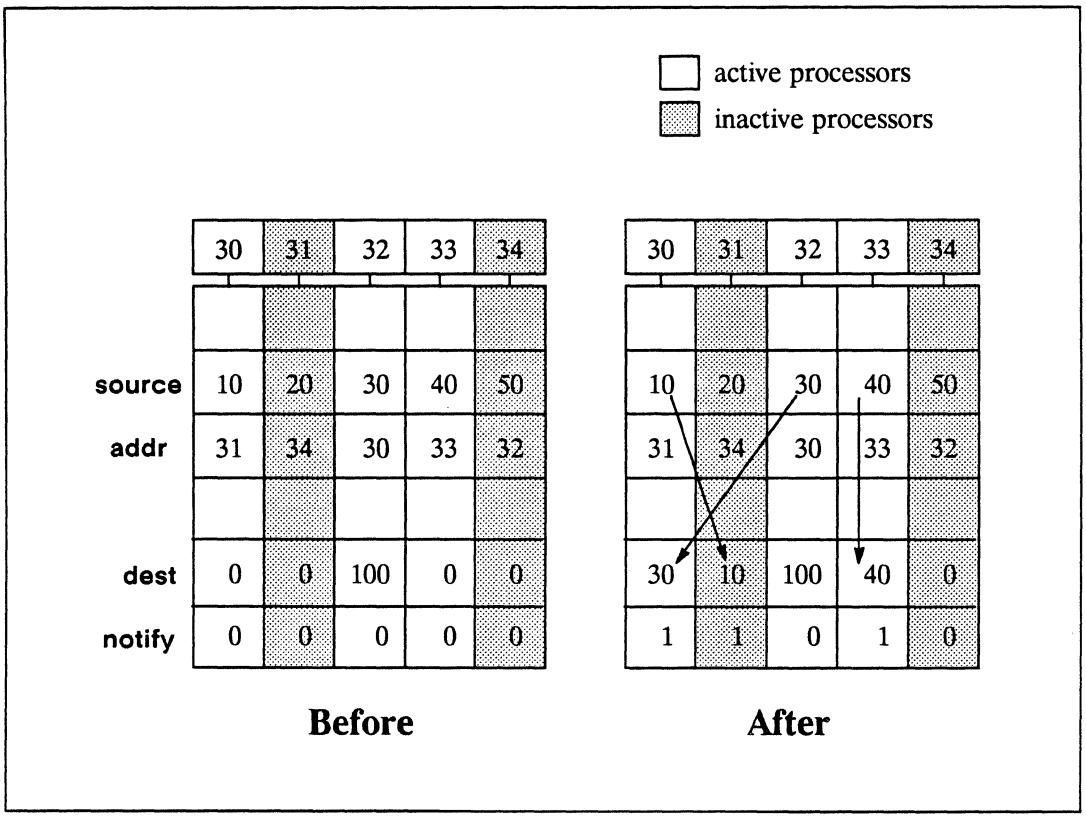

Figure 25. Effect of context on **CM_send_1L**

The *notify* bit is particularly useful when the program has manipulated context and needs to check which messages were sent. There is, of course, a performance penalty associated with notification. If no notification is desired, the program can supply the dummy field-id **CM_no_field** as the *notify* operand to the **send** instructions.

Example of a Simple Send

A simple example of general communication on the CM is a program that transposes a square array (flips it over its diagonal). What follows are the essential actions of the program that appears in Appendix C, "Transposing an Array."

Assume that the program has created a vp-set **matrix_vp_set** with a 2-dimensional geometry **matrix_geometry**. The objective is for each processor to transfer the value in its own field **matrix_value** to the same field in the processor that is its "opposite number" in the array.

Determining the NEWS coordinates of the destination processors is straightforward, since each processor needs only to reverse its own NEWS coordinates in building the send address (that is, processor *x,y* sends to processor *y,x*).

```
/* compute news coordinates */

    CM_my_news_coordinate_1L( x, 0, news_coord_length );
    CM_my_news_coordinate_1L( y, 1, news_coord_length );

/* build send address by reversing coordinate values */

    CM_make_news_coordinate_1L( matrix_geometry, send_addr,
                                0, y, news_coord_length );
    CM_deposit_news_coordinate_1L( matrix_geometry, send_addr,
                                1, x, news_coord_length );
```

The processors right on the matrix diagonal, where $x = y$, in effect compute their own send addresses. Therefore, the program should send to a temporary field rather than to the source field, since a processor cannot send to itself if the source and destination fields overlap. (See the discussion of field overlap under each of the **send** instructions in the *Paris Reference Manual.*)

Assuming that the fields **temp** and **notify** have been allocated, the call that transfers the value in field **matrix_value** from the source processors to the destination processors is:

```
CM_send_1L( temp, send_addr, matrix_value, value_length, notify);
```

The program then moves the value from the **temp** field into the original source field:

```
CM_u_move_1L( matrix_value, temp, value_length );
```

Example of a Simple Send across Vp-Sets

The procedure for sending messages across vp-sets is similar to sending within a vp-set. The source, or "sending," processors must be in the current vp-set. The destination processors can be in any vp-set. Although send addresses are unique only within a vp-set, the system can determine from the *dest* operand which vp-set is meant, since a field is associated with exactly one vp-set.

The only significant difference in the **send** procedure when working with multiple vp-sets is in determining the length of the send-address field. The send addresses are computed in the current vp-set, but their minimum length is determined by the geometry of the destination vp-set. The *notify* field, if any, should be in the destination vp-set.

For example, consider the simple point-drawing procedure shown in Example 20. This procedure sends color values from one vp-set to specified processors in another vp-set. The destination **image** is a field in a 2-dimensional vp-set where each processor represents a pixel in a graphic image. The operation is performed in the vp-set where the color values are stored. (The change in CM state from this procedure is shown afterward in Figure 26.)

Example 20. Sending across vp-sets: **draw-points.c**

```
#include <cm/paris.h>

draw_points( image, x, y, color, coord_length, color_length )

     CM_field_id_t     image, x, y, color;
     unsigned int      coord_length, color_length;
{
     CM_field_id_t     send_addr;
     CM_geometry_id_t geometry;
     unsigned int      send_addr_length;

/* determine length of send address in image field's vp-set */

     geometry = CM_vp_set_geometry( CM_field_vp_set( image ));
     send_addr_length =
          CM_geometry_send_address_length( geometry );
```

```
/* allocate and initialize field for send address */
     send_addr = CM_allocate_stack_field( send_addr_length );
     CM_u_move_zero_1L ( send_addr, send_addr_length );

/* build send address from one news axis coordinate at a time */
     CM_make_news_coordinate_1L( geometry, send_addr, 0, x,
                                 coord_length);
     CM_deposit_news_coordinate_1L( geometry, send_addr, 1, y,
                                    coord_length );

/* send color values into the image field (no notification) */
     CM_send_1L( image, send_addr, color, color_length,
                 CM_no_field );

/* deallocate stack field */
     CM_deallocate_stack_through( send_addr );
}
```

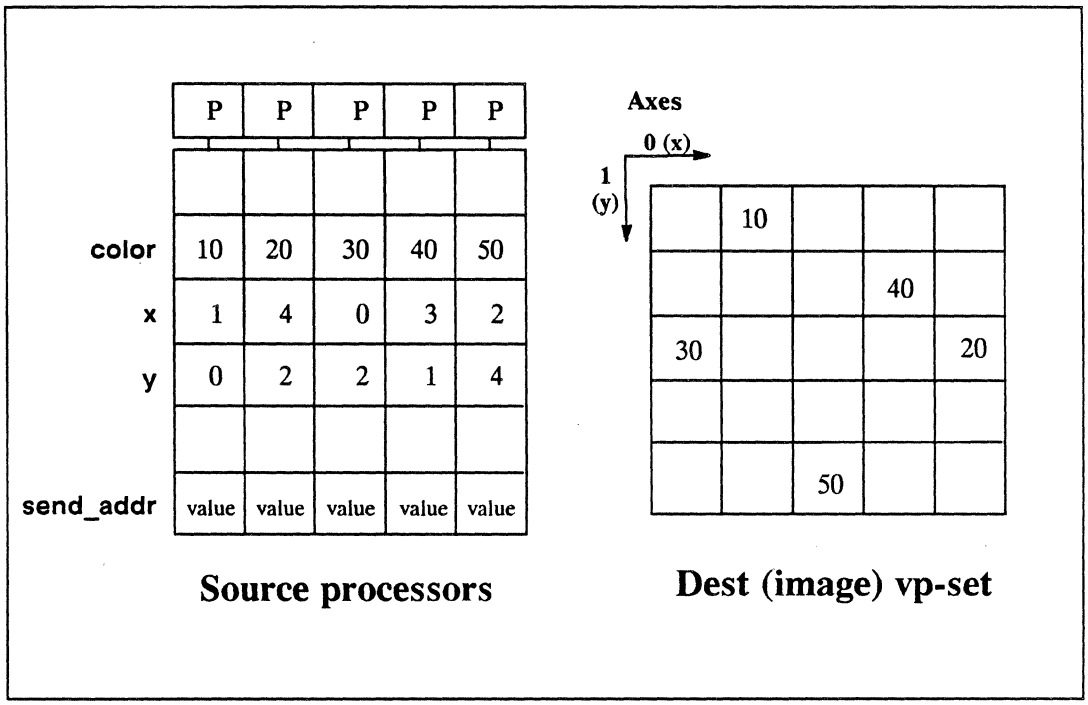

Figure 26. Change in CM state from executing **draw-points.c**

7.3 Handling Message Collisions

The examples just shown of **send** instructions are special cases in that no processor received messages from more than one other processor, that is, there were no *collisions*. Real programs frequently need to select among or combine multiple messages that arrive at the same processor.

The variants on **CM_send_1L** are all means of specifying how to handle collisions. The combiners provided are **overwrite**, **add**, **max**, **min**, **logand**, **logior**, and **logxor**. The arithmetic combiners are each provided in **s**, **u**, and **f** variants for signed and unsigned integers and floating-point numbers. A few examples:

CM_send_with_overwrite_1L	*dest send-address source len notify*
CM_send_with_s_add_1L	*dest send-address source len notify*
CM_send_with_f_max_1L	*dest send-address source sig-len exp-len notify*

Unlike **CM_send_1L**, the **CM_send_with_***combiner***_1L** instructions do include the original contents of the destination field when selecting among or combining messages. For instance, **CM_send_with_overwrite_1L** chooses any one of the messages *or* the original value (the one chosen is unpredictable). The instruction **CM_send_with_s_add_1L** adds the original value and all incoming messages. To exclude the original value from the operation, the program should prepare the destination field in the way specified for the particular instruction in the *Paris Reference Manual.*

An example of handling collisions in general communication is seen in computing a histogram. This program, shown in its entirety in Appendix D, sends a **1** for each instance of a particular value in a source field to the appropriate "bin" or accumulator cell in another vp-set. Since the purpose of the exercise is to determine how many instances there are of each value, the **send** instruction used is **CM_send_with_u_add_1L**. Example 21 shows the accumulator procedure from this program.

Example 21. Sending with a combiner: **accumulate-votes.c**

```
#include <cm/paris.h>

accumulate_votes(  accumulator_field, src_field,
                   accumulator_length, src_length )

    CM_field_id_t accumulator_field, src_field;
    unsigned int  accumulator_length, src_length;
```

```
{
    CM_geometry_id_t  dest_geometry;
    CM_field_id_t     send_address, vote;
    unsigned int      send_address_length;

/* Get information about actual arguments. */
    dest_geometry =
       CM_vp_set_geometry( CM_field_vp_set( accumulator_field ));
    send_address_length =
       CM_geometry_send_address_length( dest_geometry );

/* Allocate temporary storage. */
    CM_set_vp_set( CM_field_vp_set( src_field ));
    vote = CM_allocate_stack_field( accumulator_length +
                                    send_address_length );
    send_address =
       CM_add_offset_to_field_id( vote, accumulator_length );

/* Initialize temporary fields; setting vote to 1 with the full
   field length specified sets subfield send_address to 0. */
    CM_u_move_constant_1L( vote, 1, accumulator_length +
                                    send_address_length);

/*  To construct send addresses for the accumulator vp-set, use
    the src_field value as the destination processor's news
    coord; then send a "vote" from each source processor. */

    CM_make_news_coordinate_1L( dest_geometry,
                                send_address,
                                /* axis = */ 0,
                                /* coord = */ src_field,
                                src_length);

    CM_send_with_u_add_1L ( accumulator_field,
                            send_address,
                            vote,
                            accumulator_length,
                            CM_no_field );

    CM_deallocate_stack_through( vote );
}
```

7.4 A Word on CM_get_1L

The opposite instruction to **CM_send_1L** is

CM_get_1L *dest send-address source len*

This instruction directs each active processor in the current vp-set to get a message from a specified field (*source*) in a specified virtual processor (*send-address*). The *source* processors need not be active, and they can be in any vp-set. The length specifier describes both the *source* and *dest* operands; it can be any unsigned integer that is a legal length for the CM data format of the message.

Multiple processors can read (get) from a single processor without contention; thus, **CM_get_1L** has no variants analogous to the combiner variants on **CM_send_1L**.

Be aware that **CM_get_1L** uses comparatively large amounts of temporary storage, since it computes and stores information about the path each message is to traverse between the source and destination processors. This instruction should be used judiciously in programs where CM memory is at a premium. Also, because of the need to store the path and then perform the data transfer, a **get** operation takes about twice the time of a comparable **send** operation.

7.5 Front-End Communications

The front-end computer can use a send address to read from or write to any single CM processor. The instructions that transfer data between the CM and the front end are provided in three variants for signed and unsigned integers and floating-point numbers, respectively. These instructions are all unconditional.

CM_s_write_to_processor_1L *send-address dest source-value len*
CM_u_write_to_processor_1L *send-address dest source-value len*
CM_f_write_to_processor_1L *send-address dest source-value s-len e-len*

int **CM_s_read_from_processor_1L** *send-address source len*
unsigned **CM_u_read_from_processor_1L** *send-address source len*
float **CM_f_read_from_processor_1L** *send-address source s-len e-len*

For all these instructions, the *send-address* operand is a front-end variable of type **CM_sendaddr_t**, computed according to the procedures shown above in Section 7.1.

The *source* or *dest* operand is a CM field (**CM_field_id_t**) of length *len*. The *source–value* (for **write**) and the result (for **read**) are the front-end types implied by the instruction names.

The following examples illustrate the transfer of data between the CM and the front end. These simple procedures compute the send addresses of some specified number of processors (*num_procs*) on a specified axis of the current CM vp-set. They then read the values from a source field in those processors and print the values on the front end.

Example 22. Printing CM values on the front end: **cm-print.c**

```
#include <cm/paris.h>

u_display_cm_memory( source, len, axis, num_procs, header_string)

     CM_field_id_t   source;
     unsigned int    length, num_procs, axis;
     char            *header_string;
{
     unsigned int    i;
     CM_sendaddr_t   send_addr;

     printf( "%s\n", header_string );

     for ( i=0; i<num_procs; i++ ) {
          send_addr =
             CM_fe_make_news_coordinate
                ( CM_vp_set_geometry( CM_current_vp_set ),
                  axis,
                  i );

          printf( "  %d\n",
             CM_u_read_from_processor( send_addr, source, len ));
     }
}

/* ======================================================== */

f_display_cm_memory( source, s_len, e_len, axis, num_procs,
                     header_string )

     CM_field_id_t   source;
```

```
        unsigned int    s_len, e_len, num_procs, axis;
        char            *header_string;
{
        unsigned int    i;
        CM_sendaddr_t   send_addr;


        printf( "%s\n", header_string );

        for ( i=0; i<num_procs; i++ ) {
             send_addr =
                CM_fe_make_news_coordinate
                   ( CM_vp_set_geometry( CM_current_vp_set ),
                     axis,
                     i );

             printf("  %f\n",
                    CM_f_read_from_processor( send_addr,
                                              source,
                                              s_len, e_len ));
        }
}
```

Part IV
Commands and Utilities

Chapter 8

Compiling and Executing Programs

To experiment with the procedures in this chapter, use any of the C/Paris example programs shown in the previous chapters. Alternatively, a simple program that produces some visible output is the following:

Example 23. A program with visible output: **count-active-set.c**

```
#include <cm/paris.h>
#include <stdio.h>

main()
{
     CM_init();

     CM_set_context();
     printf( "\nThe number of processors participating is %d.\n",
            CM_global_count_context() );
}
```

8.1 To Compile

A C/Paris program is compiled with the front end's C compiler in the same way as any C program that is to be linked with a specialized library (in this case, the Paris library). The program can be compiled on any VAX or Sun-4 that has CM System Software installed.

In Version 5.0, the Paris library is specified on the **cc** command line as **-lparis**. (Please consult the documentation for later versions to determine whether the means of specifying the Paris library has changed.)

```
% cc count-active-set.c -lparis
```

Many Paris routines rely on definitions in the UNIX (serial) math library, but this library is not prelinked with the Paris library. Therefore, C/Paris programmers should specify the **-lm** option on the **cc** command line, placing it *after* **-lparis**.

```
% cc filename.c -lparis -lm
```

8.2 To Attach

Before a C/Paris program can be executed, a front-end bus interface (or *FEBI*) must be *attached* to one or more sequencers within the CM. The CM System Software command **cmattach** establishes this logical connection. It reserves for the program's use the processors that the sequencer controls and initializes (cold boots) the sequencer and its processors.

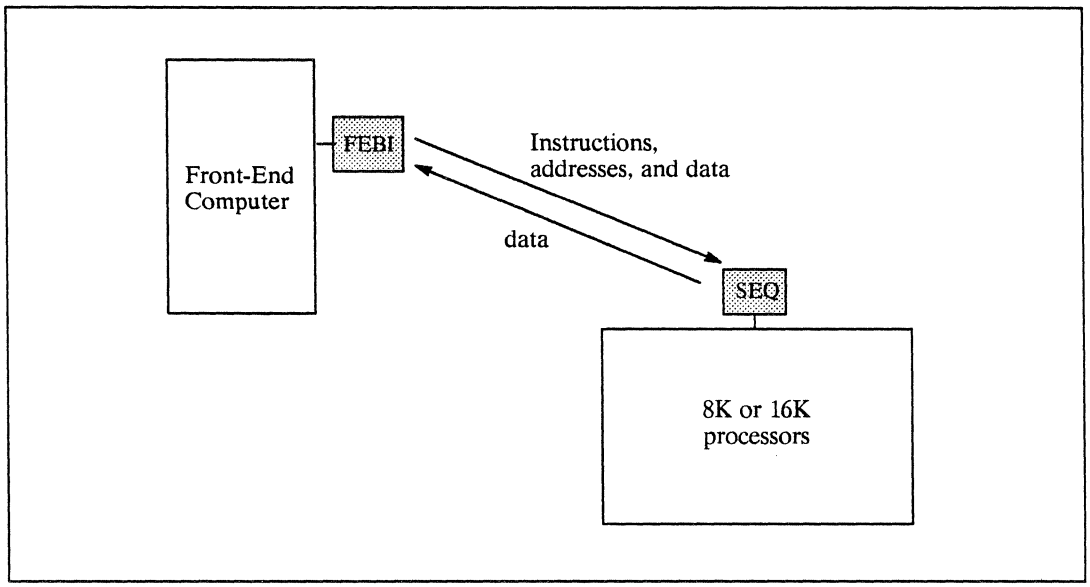

Figure 27. A front-end FEBI attached to a CM sequencer

Any number of users can develop and compile C/Paris programs at the same time on the front end. However, the number of users that can *execute* programs at once is limited by the number of FEBIs on the front end and the number of sequencers on the CM. For example, Figure 28 shows a CM system with two FEBIs and four sequencers. One sequencer is free, but it cannot be accessed until one of the FEBIs becomes detached.

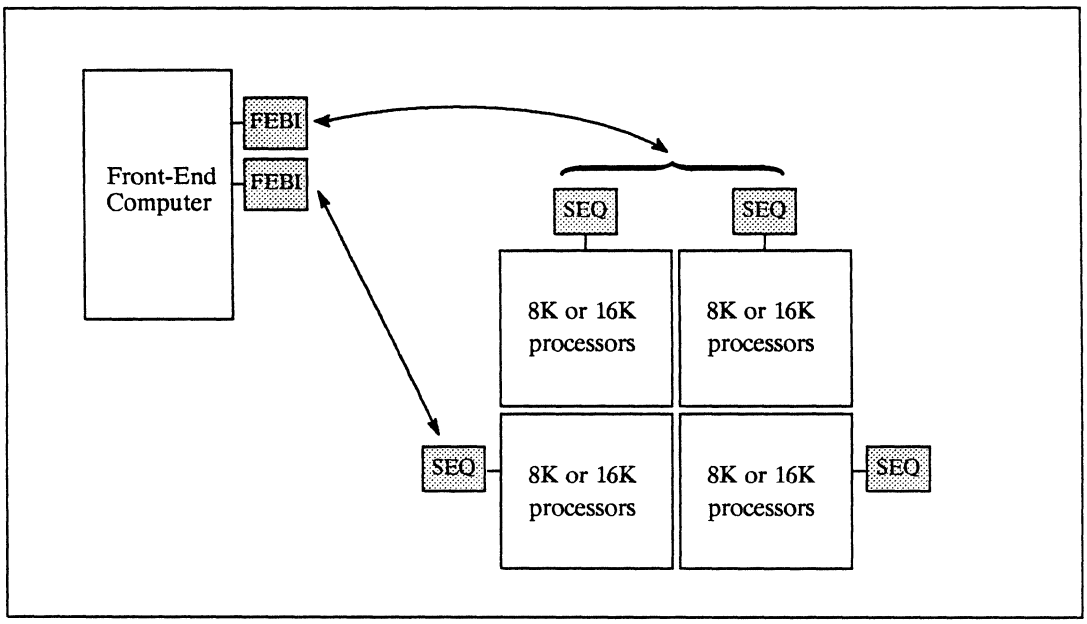

Figure 28. Front-end FEBIs attached to CM sequencers

The further action of **cmattach**, beyond attaching and initializing the CM processors, depends on whether the command is invoked in batch or interactive mode. If an executable filename is supplied on the command line, **cmattach** executes the program in batch (see next section); with no filename, **cmattach** prepares for interactive execution.

When executed without options, **cmattach** attaches the first available FEBI to the first available sequencer. The on-line manual page for **cmattach** provides a full list of **cmattach** options and their defaults and legal values. Some commonly used options are:

-w "Wait for resources." The front end keeps trying to attach if no sequencer or no FEBI is available at the first try. This option is recommended when multiple users are competing for a share of CM hardware. For example:

```
% cmattach -w my-program
```

-S "Sequencer." The front end attaches to a particular sequencer or to more than one sequencer. A particular sequencer might be requested because it is associated with an optional floating-point accelerator or with a system I/O device. Multiple sequencers might be requested for programs with very large data sets. Some example command lines are:

```
% cmattach -w -S1 my-program
% cmattach -w -S0-3  my-program
```

-i "Interface." The front end attaches a particular FEBI to the specified (or default) sequencer. This option is useful when the interfaces on a front end are physically connected to different CMs and a particular CM is desired. Some example command lines are:

```
% cmattach -i0 my-program
% cmattach -w -i1 -S0-1 my-program
```

8.3 To Execute in Batch

Paris programs can be executed in batch mode in the UNIX foreground or background or on a remote machine that is also a CM front end.

In the UNIX Foreground

To invoke **cmattach** in batch mode in the foreground, supply the executable filename on the command line:

% cmattach [*options*] *executable-filename* [*arguments*]

When invoked in this way, **cmattach** performs the following actions:

1. Attaches a FEBI to a CM sequencer
2. Cold boots the sequencer and its processors
3. Executes the specified program
4. Detaches the FEBI from the sequencer

If no options beyond **-w** are specified, the screen display might look like the following example. Note the **printf** output from the sample program **count-active-set.c**.

```
% cmattach -w count-active-set
Attaching the Connection Machine system...
coldbooting... done.
Attached to 8192 physical processors

The number of processors participating is 8192.

Detaching... done.
%
```

In the UNIX Background

A program can also be executed in the UNIX background:

% cmattach [*options*] *executable-filename* [*arguments*] **>&** *output-filename* **&**

For example:

```
% cmattach -w -q my-program >& output &
%
```

In this example, program output and any error messages are redirected to the file **output**. It is important to redirect *both* standard output and standard error, using **>&**; if both streams are not redirected, the program could be suspended waiting to write to the terminal. Note also the use of the option **-q** to suppress screen display of informational messages arising from program execution.

On a Remote Machine

Finally, a program can be executed in batch on a remote VAX or Sun-4 that is also a CM front end. This is done in the normal UNIX manner with the command **rsh** and the name of the remote machine. In this case, it is especially important to specify the full pathname of the executable file.

% rsh *machine-name* **cmattach** [*options*] *path/executable-filename* [*arguments*]

For example, the following command line causes **my-program** to be executed from the CM front end **other-machine**. The command line invokes **pwd** to find the path of a program that resides in the user's present working directory:

```
%rsh other-machine cmattach -w 'pwd'/my-program
Attaching the Connection Machine system...
coldbooting... done.
Attached to 8192 physical processors

[program output, if any]

Detaching... done.
%
```

8.4 To Execute Interactively

If **cmattach** is invoked without an executable filename, it attaches and cold boots the hardware and then spawns a subshell in which programs may be executed. It does not detach the hardware until the subshell is exited, even if no program is executing.

```
% cmattach [options]
Attaching the Connection Machine system...
coldbooting... done.
Attached to 8192 physical processors

Entering CMATTACH subshell. Type "exit" or control-D
to detach the CM. . .

% my-program-1
[program output, if any]

% my-program-2
[program output, if any]

% exit
Detaching... done.
%
```

The major recommended use of the **cmattach** subshell is in running shell-level utilities, such as the debugger **dbx** or the run-time safety checker, **cmsetsafety**. The procedure is described in Chapter 10.

Chapter 9

Programming Utilities

This chapter describes three utilities that are useful in developing C/Paris programs:

- The run-time safety utility
- The debugger **dbx**
- The Paris timer

9.1 Run-Time Safety

The run-time safety utility checks for certain errors and inconsistencies in user programs. When it detects a user error, the utility aborts program execution and prints information about the error. The user should of course expect reduced execution speed when safety checking is enabled, but it can be a helpful tool in program development and debugging.

The utility has two states, on and off. When enabled, the utility checks the following conditions:

- Whether the field-id's passed as arguments to Paris instructions refer to fields in the current vp-set
- Whether the field-id's passed as arguments to Paris instructions are valid field-id's (although not all invalid field-id's are caught)
- Whether the length specifiers passed to Paris instructions exceed the lengths of the respective field operands

This utility is intended for use with C/Paris programs only. It is not recommended for use in run-time checking of compiler output of the higher level CM languages.

From within a Program

The safety utility is available as a Paris instruction, **CM_set_safety_mode**, which takes an unsigned integer argument. Any non-zero value enables the utility; a zero argument disables it.

From the Shell

The shell-level command **cmsetsafety** performs the same action as the Paris instruction **CM_set_safety_mode**.

```
% cmsetsafety [on] [off]
```

By using the command rather than the Paris instruction, we can execute a program either with or without safety checking without changing the source file. However, the shell command does not permit safety checking of only selected parts of a program.

The command **cmsetsafety** is executed from within a **cmattach** subshell. For example:

```
% cmattach
Attaching the Connection Machine system... coldbooting... done.
Attached to 8192 physical processors

Entering CMATTACH subshell. Type "exit" or control-D
to detach the CM. . .

% cmsetsafety on

% my-program
[program output, if any]

% cmsetsafety off

% my-program
[program output, if any]

% exit
%
```

Safety is initially off in a subshell. If **cmsetsafety** is executed with the option **on**, all programs are then executed with safety on until the safety mode is changed or the subshell is exited.

Changing Default Safety Behavior

By default, all CM programs are executed with safety off, whether execution is interactive or in batch. To enable safety by default for all CM program execution, set the environmental variable **CM_DEFAULT_SAFETY** to **on** or **ON** in the **.cshrc** file:

```
setenv CM_DEFAULT_SAFETY on
```

If the variable is not set, or if it is set to any other value, safety is off for batch execution and initially off in an interactive **cmattach** subshell. Safety can, of course, be enabled at any time within a subshell by invoking by command **cmsetsafety**, as noted above.

It is often convenient to set the defaults such that safety is off for batch execution but on in an interactive **cmattach** subshell. In particular, safety should be enabled when using the interactive debugger **dbx** from within a subshell, as described in the next section. To have safety be initially on in a subshell but off for batch execution, add the following line to the **.cshrc** file:

```
if ($?CMDEVICE) cmsetsafety on
```

9.2 The Debugger dbx

Like any C program, a C/Paris program can be debugged interactively by means of the debugger **dbx**. As noted in the previous section, it is strongly recommended that the Paris run-time safety utility be enabled whenever **dbx** is in use.

Invoking the Debugger

The debugger is activated from within a **cmattach** subshell. The procedure is:

```
% cmattach
Attaching the Connection Machine system...
coldbooting... done.
Attached to 8192 physical processors

Entering CMATTACH subshell. Type "exit" or control-D
to detach the CM. . .
```

```
% dbx my-program

dbx> [dbx commands such as run, s, and n]
dbx> q

% exit
%
```

Locating Paris Errors

Locating an error in a C/Paris program is made somewhat difficult by the fact that the CM executes asynchronously with respect to the front end. After sending an instruction over the bus to the CM, the front end continues to execute other code. If the CM signals an error, it is not immediately obvious which Paris call is at fault:

- Since the queue for the instruction bus can accommodate over 200 instructions, the CM may not execute an erroneous instruction until the front end is much farther along in the program.
- CM error signals are not sent back to the front end as they occur. Instead, for reasons of system efficiency, the CM holds the error signals until the next time the front end reads data from the CM.

The standard method for debugging programs is to insert breakpoints at various places before the point where the error is reported. For debugging a C/Paris program, we can force synchronization between the two machines at each breakpoint by interactively calling a Paris instruction that reads data from the CM. (**CM_global_logior_context** is commonly used for this purpose because it is fast and requires no arguments.)

With each call to an instruction that reads CM data, any pending CM error messages are "piggybacked" to the front end. By using this method with a binary search strategy, we can usually isolate the offending Paris call quickly.

Be aware that Paris functions are not linked into a program unless the program references them (directly or indirectly). To make sure that instructions used in debugging, such as **CM_global_logior_context**, are always linked, it is advisable to write a dummy function that calls all such instructions and place the dummy function is a separate file, say, **debug.c**. Then, link **debug.o** into any program that is still under development.

Examining CM Data

Paris does not extend **dbx** to support examining values on the CM. The user can write procedures that print CM values in any desired format. An example of such a procedure, one that calls **CM_*type*_read_from_processor_1L** to display the values in any specified number of processors, is shown in Chapter 7.

All such debugging routines can be made available in a program by including them in the file **debug.c**, mentioned in the previous section, and linking the program with the associated object file.

9.3 The Paris Timer

The Paris timer is a facility for recording program execution time—both the total elapsed time and the time during which the CM was active. The timer consists of a set of Paris instructions; it can be used only from within a program.

The Timing Functions

The timing facility consists of three functions:

CM_start_timer	begins accumulating timing information
CM_stop_timer	stops accumulating timing information and records (optionally, prints) the information
CM_reset_timer	erases accumulated timing information

The information recorded is:

- The CM's active time (in seconds) since the call to **CM_start_timer**
- Real time (in seconds) elapsed since the call to **CM_start_timer**
- Percentage machine utilization, calculated as CM time divided by real time

The functions **CM_start_timer** and **CM_stop_timer** can be inserted in a program to span the portion of the code for which timing information is desired. The program can also make successive calls to these two functions, much like starting and stopping a

stop watch. In this case, the information recorded is the cumulative time over the successive calls. To erase the accumulated time at any point, call **CM_reset_timer**.

The exact behavior of the functions is as follows:

CM_start_timer *verbose*

Begins accumulating times. If the integer argument *verbose* is true (non-zero) and if there is a pause of 5 seconds or more while the timer is being calibrated, the instruction writes an explanation of the delay to the stream **stderr**. This message can appear only the first time the instruction is called in a program.

CM_timeval_t *CM_stop_timer *verbose*

Stops accumulating times and returns a pointer to a structure of type **CM_timeval_t**, where the updated times are stored. If the integer argument *verbose* is true (non-zero), the function writes the timing information to the stream **stderr**.

The structure contains the members **cmtv_real** and **cmtv_cm**, both of type **double**. The structure is stored in static space; it must be copied if it is to be saved.

CM_reset_timer

Erases accumulated timing information. The function does not restart the timer. (It is not necessary to call this function before the first call to **CM_start_timer**.)

Interpreting Timer Output

To use the C/Paris timer effectively, it is helpful to understand something of how it is implemented.

In the present release of Paris, the timer proceeds by counting the number of idle CM cycles during the code segment in question, rather than the number of cycles during which the CM is active. Idle cycles are those during which the CM sequencer is waiting to receive an instruction from the front end.

Since idle cycles are all of the same length, the CM's active time can be computed by:

1. Measuring elapsed real time with the front end's real-time clock
2. Multiplying the number of CM idle cycles by the (constant) time per cycle
3. Subtracting total CM idle time from elapsed real time

With UNIX front ends, two potential problems arise:

- The UNIX real-time clock has lower resolution (on the order of 1 millisecond) than CM cycle time (on the order of 1 microsecond). Since the reported CM active time is computed directly from the real time as measured on the front end, distortions are introduced when timing code segments whose total elapsed time is under about 1 second.

- UNIX machines typically have some degree of multiprocessing activity, even when only one user is logged in. The real time that the front end is measuring is not the virtual time of the Paris program's process; instead, it includes the time consumed by other processes.

  Such "interference" from other processes can lead to timing variations on the order of 15 percent even when the load on the front end is relatively light.The longer the code segment being timed, the more likely it is that timing distortions will be introduced by other processes.

The implementation of the Paris timer suggests some rules of thumb for using it most effectively:

- Use a front end that is as unloaded as possible.

- Select or manipulate the code segment being timed so that the elapsed time is between 1 and 5 seconds.

- Run the code segment at least five times and use the minimum value reported.

Some caveats are also in order, both resulting from the timer's definition of CM idle time. Idle time includes only those cycles during which the CM is waiting for an instruction from the front end. Consequently, CM active time includes not only those cycles during which the CM is performing computations, but also those during which the CM is waiting for arguments to an instruction it has received.

The caveats are:

- Expect slightly different CM active times on different front-end models for code segments that do not keep the CM 100 percent active. The time the CM spends waiting for data to appear is counted as active, but front-end models differ in the speed with which they can move data over the bus interface to the sequencer.

- Avoid stopping a process that is being timed. If the process stops while the CM is waiting for an instruction, then all is well since the time spent stopped is subtracted from the total real time in computing CM active time. However, if the process stops while the CM is waiting for data, the time spent stopped is counted as active time, which artificially increases the CM execution time recorded.

Part V
Example Programs

Appendix A

Game of Life

```
/*
  This program is a cellular automata model of life and
  death that works with the following rules:

      The cells are set up on a 2-d grid.

      Each cell starts in a randomly chosen life or death
      state.

      Cells compute life and death at every time step n
      for the state at time step n+1.

      Each cell gathers information about its 8 adjacent
      neighbors. North, south, east, west, northwest,
      northeast, southwest, and southeast.

      Each cell uses its neighbor information to determine
      whether or it will live or die depending on the number
      of its neighbors that are alive

      A cell "lives" if it is alive and it has two alive
      neighbor OR if it is dead and has three live
      neighbors.  Otherwise it dies.

  *This parallel version was adapted from a program by
   Craig Reese, IDA/Supercomputing Research Center,
*/
```

Example 24. Conway's game of life: **life.c**

```
#include <stdio.h>
#include <cm/paris.h>

#define PERCENT_ON 40
#define NORTH   0,CM_upward
#define SOUTH   0,CM_downward
#define EAST 1,CM_upward
#define WEST 1,CM_downward
#define DEFAULT_GRID_SIZE 512

int grid_size = DEFAULT_GRID_SIZE;


/*************************************************************/
main(argc, argv)
    char *argv[];
{
    int generation;

    CM_geometry_id_t  life_2d_geometry;
    CM_vp_set_id_t    life_vp_set;
    unsigned       dimensions[2];
    CM_field_id_t
      alive,
      my_sum,neighbor_sum,
      temp;


/*************************************************************/
    if (argc-1)
      sscanf(argv[1],"%d",&grid_size);

    printf("Warm booting the CM ..."); fflush(stdout);
    CM_init();
    printf("Done\n");
```

```
/*************************************************************/
/* Set up the vp-set in which the the game of life will be
   played */

    dimensions[0] = grid_size;
    dimensions[1] = grid_size;
    life_2d_geometry = CM_create_geometry(dimensions,2);
    life_vp_set = CM_allocate_vp_set(life_2d_geometry);
    CM_set_vp_set(life_vp_set);

/*************************************************************/
/* allocate storage */

    /* set up the storage in each cell VP */
    alive = CM_allocate_heap_field(1);
    my_sum = CM_allocate_heap_field(4);
    neighbor_sum = CM_allocate_heap_field(4);
    temp = CM_allocate_heap_field(4);

/*************************************************************/
/* Randomly choose an initial state */

    CM_set_context();
    CM_u_random_1L(alive,1,2);

/*************************************************************/
/* The Main loop */

    for (generation = 0; generation < 1000; generation++) {

     /* start with everybody */
     CM_set_context();

     /*initialize fields*/
     CM_u_move_zero_1L(my_sum, 4);
     CM_u_move_zero_1L(neighbor_sum, 4);
     CM_u_move_zero_1L(temp, 4);

     /* N neighbor */
     CM_get_from_news_1L(temp, alive, NORTH, 1);
     CM_u_add_3_3L(my_sum, temp, alive, 4, 4, 1);
```

```
/* S neighbor */
CM_get_from_news_1L(temp, alive, SOUTH, 1);
CM_u_add_2_1L(my_sum, temp, 2);

/* Share results so far.
   Notice that after this call, each cell's east
   neighbor has life information about the cells
   northeast and southeast neighbor encode in
   neighbor_sum.  The same is true for the west
   neighbor with respect to each cell northwest and
   southwest neighbors.
 */
CM_u_move_1L(neighbor_sum, my_sum, 2);

/* E neighbor */
CM_get_from_news_1L(temp, neighbor_sum, EAST, 2);
CM_u_add_2_1L(my_sum, temp, 3);

/* W neighbor */
CM_get_from_news_1L(temp, neighbor_sum, WEST, 2);
CM_u_add_2_1L(my_sum, temp, 4);

CM_u_subtract_3_3L(my_sum, my_sum, alive, 4, 4, 1);

/* whoever has two or three live neighbor lives */
/* everyone else dies*/
CM_u_move_zero_1L(temp,4);

CM_load_context(alive);
CM_u_eq_constant_1L(my_sum, 2, 4);
CM_store_test(temp);

/* test for three live neighbors */
/* inclusive or the result with alive and */
/* place back into alive */
CM_set_context();
CM_u_eq_constant_1L(my_sum, 3, 4);
CM_logior_test(temp);
CM_store_test(alive);

/* now only those processors who have alive set */
/* survived this generation. */
```

```
        CM_load_context(alive);

        /* output of the grid to your favorite display goes here */

      }
  }
```

Appendix B

Include File: Macros and Constants

```
/*
  This header file defines macros and constants that will
  be used throughout programming examples in the appendixes
  to "Introduction to Programming in C/Paris.".  Please
  note that these example macros are not supported as part
  of the CM System Software and Thinking Machines
  Corporation does not warrant them as such.
*/
```

Example 25. A file included in later examples: **macros-and-constants.h**

```
/* Constant Macros */

#define SLEN    23
#define ELEN    8
#define IEEE_TOTAL_LENGTH 32
#define FLEN          IEEE_TOTAL_LENGTH
#define FLENS      SLEN,ELEN

/*********************************************************/
/* Macros with arguments */

#define MIN(a, b) (((a)<(b))?(a):(b))
#define MAX(a, b) (((a)>(b))?(a):(b))
#define POWER_OF_TWO(a) (1<<(MAX((a),0)))
```

```
/*************************************************************/
/* Macros that map front-end types onto the CM */

/* returns the length of a type in bits */
#define CM_TYPELEN(type) (unsigned)(8 * sizeof(type))

/* allocates enough room on the CM stack for a type */
#define CM_ALLOCATE_TYPE_ON_STACK(type) \
  CM_allocate_stack_field(CM_TYPELEN(type))

/* allocates enough room on the CM heap for a type */
#define CM_ALLOCATE_TYPE_ON_HEAP(type) \
  CM_allocate_heap_field(CM_TYPELEN(type))

/* this expression comes from the example idioms of */
 /* the ANSI C draft for determining the offset of a */
 /* slot in a struct */
#define STRUCT_SLOT_OFFSET(type,slotname) \
  ((unsigned)&(((type *)0)->slotname))

/* determines the offset, in bits, of a struct slot type */
#define CM_STRUCT_SLOT_OFFSET(type,slotname) \
  (8*STRUCT_SLOT_OFFSET(type,slotname))

/* returns the CM_field_id_t of the subfield slotname */
#define CM_STRUCT_SUBFIELD(obj,type,slotname) \
  (CM_field_id_t)CM_add_offset_to_field_id\
    ((obj),CM_STRUCT_SLOT_OFFSET(type,slotname))
```

Appendix C

Transposing an Array

```
/*
  This program transposes a 2-D matrix that is stored in
  the Connection Machine across the processors. The
  storage is such that each virtual processor has one
  element of the array.  After some initial set-up and
  send-address calculation, the send instruction actually
  moves each datum to its transposed location.
*/
```

Example 26. Transposing an array: **transpose.c**

```
#include <stdio.h>
#include <cm/paris.h>

#define FIELD_LENGTH 8

/*************************************************************/
main(argc, argv)
    char *argv[];
{

    CM_field_id_t
      matrix_value, temp, send_address, x_news, y_news;
    CM_geometry_id_t  matrix_2d_geometry;
    CM_vp_set_id_t    matrix_vp_set;
    unsigned
      matrix_dimensions[2],
      send_address_length,
      news_coordinate_length,
```

```
      row, column;
    CM_sendaddr_t fe_send_address;

/*************************************************************/

    printf("Warm booting the CM ..."); fflush(stdout);
    CM_init();
    printf("Done\n");

/*************************************************************/
/* Set up geometry and vp-set of matrix */

    matrix_dimensions[0] = 256;
    matrix_dimensions[1] = 256;
    matrix_2d_geometry =
      CM_create_geometry(matrix_dimensions,2);
    matrix_vp_set = CM_allocate_vp_set(matrix_2d_geometry);
    CM_set_vp_set(matrix_vp_set);

/*************************************************************/
/* Allocate storage */

    /* activate all processors */
    CM_set_context();

    /* Determine how much storage is needed for the send
       address */
    send_address_length =
      CM_geometry_send_address_length(matrix_2d_geometry);

    /* Determine how much storage is needed for the news
       coordinates. We simplify matters by having only one
       length because we know we are dealing with a square
       matrix. In general, we need one length value per
       axis */
    news_coordinate_length =
      CM_geometry_coordinate_length(matrix_2d_geometry, 0);
```

```
    matrix_value =
      CM_allocate_heap_field(FIELD_LENGTH * 2 +
               send_address_length +
               news_coordinate_length * 2);
    temp =
      CM_add_offset_to_field_id(matrix_value, FIELD_LENGTH);
    send_address =
      CM_add_offset_to_field_id(matrix_value,
            FIELD_LENGTH*2);
    x_news = CM_add_offset_to_field_id(send_address,
                    send_address_length);
    y_news =
      CM_add_offset_to_field_id(x_news,
            news_coordinate_length);


/*************************************************************/

/*Initialize fields*/
    CM_u_move_zero_1L(matrix_value,
                FIELD_LENGTH * 2 +
                send_address_length +
                news_coordinate_length * 2);


    CM_my_news_coordinate_1L(x_news, 0,
              news_coordinate_length);
    CM_my_news_coordinate_1L(y_news, 1,
              news_coordinate_length);


    /* Seed the triangle where x > y with 1 */
    CM_u_gt_1L(x_news, y_news, news_coordinate_length);
    CM_logand_context_with_test();
    CM_u_move_constant_1L(matrix_value, 1, FIELD_LENGTH);
    /* reset processor mask */
    CM_set_context();


    /* Seed the triangle where x < y with 255 */
    CM_u_gt_1L(y_news, x_news, news_coordinate_length);
    CM_logand_context_with_test();
    CM_u_move_constant_1L(matrix_value, 255, FIELD_LENGTH);
    /* reset processor mask */
    CM_set_context();
```

```
/*************************************************************/
/* Display via text the upper portion of the matrix */

    printf("A 10x10 region of the matrix before transpose\n");
    for (row=0; row < 10; row++) {
     for (column=0; column < 10; column++) {
         fe_send_address =
           CM_fe_make_news_coordinate(matrix_2d_geometry,
                 0, /* axis */
                 column/*news coord*/
                 );
         fe_send_address =
         CM_fe_deposit_news_coordinate(matrix_2d_geometry,
                    fe_send_address,
                    1, /* axis */
                    row /* news coord*/);
         printf("%5d",
             CM_u_read_from_processor_1L
               (fe_send_address, matrix_value,
                FIELD_LENGTH));
     }
     printf("\n");
    }
    printf("\n");

/*************************************************************/
/* Build send address
   Notice that to make the address the transpose address
   we need only to switch the news coordinates in the
   send address. The part of the send address that is
   normally for x (in this example) is being filled by
   y_news, namely axis 0. The same is done for the normal
   place for the y coordinate, it is being filled with
   the value of x_news */
    CM_make_news_coordinate_1L(matrix_2d_geometry,
                send_address, 0, y_news,
                news_coordinate_length);

    CM_deposit_news_coordinate_1L(matrix_2d_geometry,
              send_address, 1, x_news,
              news_coordinate_length);
```

```
/*************************************************************/
/* We must do the send to the temp field because it is
   illegal to do a send from and to the same field. We put
   temp into matrix_value after the send. */

    CM_send_with_overwrite_1L(temp, send_address, matrix_value,
            FIELD_LENGTH, CM_no_field);

    CM_u_move_1L(matrix_value, temp, FIELD_LENGTH);

/*************************************************************/
/* Display via text the upper portion of the matrix */

    printf("A 10x10 region of the matrix after transpose \n");
    for (row=0; row < 10; row++) {
     for (column=0; column < 10; column++) {
         fe_send_address =
           CM_fe_make_news_coordinate(matrix_2d_geometry,
                 0, /* axis */
                 column/*news coord*/
                 );
         fe_send_address =
         CM_fe_deposit_news_coordinate(matrix_2d_geometry,
                    fe_send_address,
                    1, /* axis */
                    row /* news coord*/);
         printf("%5d",
             CM_u_read_from_processor_1L
               (fe_send_address, matrix_value,
                FIELD_LENGTH));
     }
     printf("\n");
    }
    printf("\n");

/*************************************************************/
/* Deallocate the matrix fields, the vp-set, and geometry */

    CM_deallocate_heap_field(matrix_value);
    CM_deallocate_vp_set(matrix_vp_set);
    CM_deallocate_geometry(matrix_2d_geometry);
}
```

Appendix D

Computing a Histogram

```
/*
  This file contains two functions.

  The first function, accumulate_votes, computes the
  histogram of a field that is part of a 2-d vp_set.  It
  uses a send with add to do this after it has calculated
  the send address of the destination.

  The second function sets up the 2-d field to be
  histogrammed.  To demonstrate one function of the
  histogram, we use a field in which every processor has
  the same value.  This will result in a histogram with
  all the votes in one cell.

  To demonstrate the use of two differently shaped vp_sets,
  we have a vp_set for the image (2-d field) and a vp_set
  for the histogram.  The method of communication between
  the two is CM_send_with_u_add_1L.
*/
```

Example 27. Computing a histogram: **histogram.c**

```
#include <stdio.h>
#include <cm/paris.h>
#include "macros-and-constants.h"

#define TARGET_VALUE 0
```

```
/* ======================================================*/

     accumulate_votes(accumulator_field, src_field,
          accumulator_length, src_length)
    CM_field_id_t accumulator_field, src_field;
    unsigned accumulator_length, src_length;
{
    CM_geometry_id_t dest_geometry;
    CM_field_id_t send_address, vote;
    unsigned send_address_length;

/*======================================================*/

/* Garner information about what is passed in to be used
   later */

    dest_geometry =
      CM_vp_set_geometry(CM_field_vp_set(accumulator_field));

    send_address_length =
      CM_geometry_send_address_length(dest_geometry);

    CM_set_vp_set(CM_field_vp_set(src_field));

/*======================================================*/

/* Allocate temporary fields */

    vote =
      CM_allocate_stack_field(accumulator_length +
                send_address_length);

    send_address =
      CM_add_offset_to_field_id(vote, accumulator_length);

/*======================================================*/

/* Initialize values for the temp fields */
    /*
      We can set vote one and zero out send_address at the
      same time by setting vote to one and specifying a
      length argument that is the sum of accumulator_length
```

```
      and send_address_length.  This is possible only
      because these fields, by virtue of their allocation
      using field offsets, are contiguous neighbors inside
      a larger field.
    */
    CM_u_move_constant_1L(vote, 1, accumulator_length +
                    send_address_length);

/*=======================================================*/

/* Calculate the send address of the votes.

  We use the src_field as a basis from which to construct
  an address into another (the histogram) vp_set.  The
  histogram vp_set is, by definition, 1-d.  Hence, there is
  only one component is the construction of the address.
*/

    CM_make_news_coordinate_1L(dest_geometry,
                send_address,
                /* axis = */ 0,
                /* coord = */ src_field,
                src_length);

/*=======================================================*/

/* Send the votes to there respective accumulator cells.

  The CM_no_field indicates that we do not want the send
  mechanism to waste time notifying us when a message (in
  this case a vote) cannot be delivered due to collision.
  The use of CM_send_with_u_add_1L explicitly indicates the
  method by which collisions are handled.  If we used a
  CM_send_1L (the generic form) we would want to be
  notified in a field when our messages didn't make it.

*/
    CM_send_with_u_add_1L (accumulator_field,
            send_address,
            vote,
            accumulator_length,
            CM_no_field);
```

```
/* =====================================================*/

/* Deallocate temp fields */

    CM_deallocate_stack_through(vote);
}

/* =====================================================*/
/* =====================================================*/

/* The function main does little more than set up image
   and histogram vp_sets and call accumulate_votes.
*/
main ()
{
/* =====================================================*/

    unsigned int image_dimensions[2];
    CM_vp_set_id_t image_vp_set;
    CM_geometry_id_t image_geometry;
    CM_field_id_t image;
    unsigned int coordinate_length, image_length;

    unsigned int accumulator_dimensions[1];
    CM_vp_set_id_t accumulator_vp_set;
    CM_geometry_id_t accumulator_geometry;
    CM_field_id_t tally;
    unsigned int tally_length;
    CM_sendaddr_t send_address;

/* =====================================================*/

    printf("Warm booting the CM ..."); fflush(stdout);
    CM_init();
    printf("Done\n");

/* =====================================================*/

    /* allocate the image vp-set and fields*/
    image_dimensions[0] = 256;
    image_dimensions[1] = 256;
    image_length = 8;
    coordinate_length = 16;
```

```
    image_geometry = CM_create_geometry(image_dimensions, 2);
    image_vp_set = CM_allocate_vp_set(image_geometry);
    CM_set_vp_set(image_vp_set);
    image = CM_allocate_heap_field(image_length);
    CM_u_move_zero_1L (image, image_length);

/* ========================================================*/

    /* allocate the accumulator vp-set and fields*/
    accumulator_dimensions[0] = CM_physical_processors_limit;
    tally_length = 32;
    accumulator_geometry =
      CM_create_geometry(accumulator_dimensions, 1);
    accumulator_vp_set =
      CM_allocate_vp_set(accumulator_geometry);
    CM_set_vp_set(accumulator_vp_set);
    tally = CM_allocate_heap_field(tally_length);
    CM_u_move_zero_1L (tally, tally_length);


/* ========================================================*/

    /* Main section */

    /* note we are in the image_vp_set for accumulation */
    CM_set_vp_set(image_vp_set);
    CM_set_context();

    /* set image to a constant value */
    CM_u_move_constant_1L(image, TARGET_VALUE, image_length);

    accumulate_votes(tally, /* accumulator_field */
              image,  /* src_field */
              tally_length, /* accumulator_length */
              image_length /* src_length */
              );

    /* now we switch to the accumulator_vp_set */
    CM_set_vp_set(accumulator_vp_set);

    send_address =
      CM_fe_make_news_coordinate(accumulator_geometry,
```

```
            /* axis = */ 0,
            /* coord = */TARGET_VALUE);
    printf("Total in proc[%d] = %d: Total image size %d\n",
        TARGET_VALUE,
        CM_u_read_from_processor_1L(send_address,
                    tally,
                    tally_length),
        image_dimensions[0]*image_dimensions[1]);


}
```

Appendix E

Particle Problem

```
/*
  This file contains a collection of functions that model
  a very simple Newtonian physics particle system in
  parallel. The purpose is to demonstrate how to set up and
  use more than one vp-set on the Connection Machine using
  C/Paris. It also demonstrates a method for generating
  send addresses from computed data.

  The physics of this toy problem is the ultra-simple
  portion.  What we loosely attempt to model is the
  behavior of particles under the laws of Newton.  We
  roughly approximate the interaction of acceleration and
  velocity on the position of a particle over time.  We
  divide time up into discrete steps and answer the
  question, What happens to each particle in this time
  step?

  The implementation of this particle system consists
  primarily of two entities: a vp-set to hold the
  information about the particles and a vp-set to hold the
  display of those particles.

  The particle vp-set has each processor contain all the
  information about one particle. This information
  consists of position, velocity, and acceleration vectors
  and a color. The particles are initialized with
  contained random values. The particle vectors are then
  updated according to some arbitrary rules (roughly
  Newtonian--see update_part for details) for every time
  step.
```

```
  The image vp-set, which holds the plotted positions of
  each particle, can used to display the entire set of
  particles at every time step.

  The event loop is as follows:
  -Initialize particle
  -for every time step
    - draw current position of the particles
    - update particle information
*/
```

Example 28. A Newtonian particle problem: **cm-particles.c**

```
#include <stdio.h>
#include <cm/paris.h>
#include "macros-and-constants.h"

/*********************************************************/
/* Constant Definitions */

#define NUM_TIME_STEPS   2000
#define DEFAULT_SCREEN  1024

/*********************************************************/
/*
  This structure is mapped into the memory of every
  processor that contains a particle.  Throughout this
  file we will use accessor functions found in the header
  file "macros-and-constants.h".  For example,

  CM_STRUCT_SUBFIELD(particles, Particle, x)

  returns the field-id for the slot 'x' in the particles
  field.
*/
```

```
typedef struct {
   char
     color;
   float
     x, y,   /* position */
     vx, vy, /* velocity */
     ax, ay; /* acceleration */
} Particle;

/* the size of one side of the 2-d image. */
int image_edge_size = DEFAULT_SCREEN;

/*************************************************************/
/*************************************************************/

/*
  This function takes the current position and color of
  each particle (in parallel, of course) and plots each
  particle's color at the x and y location specified.  A
  send is used after a send address is generated from the
  current location information.
*/

void
draw_particles(image, particles)

    CM_field_id_t image, particles;
{
    CM_field_id_t sx, sy;
    unsigned len;
    CM_geometry_id_t geometry;

/*************************************************************/
    /* figure out the size of the coordinate field */
    /* notice we add 1 for the sign bit */
    geometry = CM_vp_set_geometry(CM_field_vp_set(image));
    len = MAX(CM_geometry_coordinate_length (geometry, 0),
          CM_geometry_coordinate_length (geometry, 1)) +
      1;
```

```
/*********************************************************/
/*
  Allocate temp fields to hold the integer values of the
  current x and y locations
*/
    sx = CM_allocate_stack_field( 2 * len);
    sy = CM_add_offset_to_field_id(sx, len);
    CM_u_move_zero_1L(sx, 2*len);

/*********************************************************/
/*
  Convert the floating-point values to signed integers.
  They are needed in this form for the next call.
*/
    CM_s_f_floor_2_2L(sx,
                CM_STRUCT_SUBFIELD(particles,
                  Particle,x),
                len, FLENS);
    CM_s_f_floor_2_2L(sy,
                CM_STRUCT_SUBFIELD(particles,
                  Particle,y),
                len, FLENS);

/*********************************************************/
/* Call draw_point.

  The source code for draw_point resides in draw-point.c (on-
  line) and is described in Chapter 7 of this manual.
*/

    draw_point(image, sx, sy,
            CM_STRUCT_SUBFIELD(particles,
                               Particle,color),
            len, CM_TYPELEN(char));
}

/*********************************************************/
/*********************************************************/

/*
  This function sets up the initial state of the particles.
  All it does is put the particle in the middle of the
  screen (position) and specify that the velocity will be
```

```
  between +- image_edge_size/32 and the acceleration will
  be between +- image_edge_size/128 and the color to 1.
*/
void
seed_parts(particles)
    CM_field_id_t particles;
{
    CM_field_id_t
      tx, ty, tvx, tvy, tax, tay, tcolor;

    /*
      Make simple references to each subfield.
     */
    tx = CM_STRUCT_SUBFIELD(particles,Particle,x);
    ty = CM_STRUCT_SUBFIELD(particles,Particle,y);
    tvx = CM_STRUCT_SUBFIELD(particles,Particle,vx);
    tvy = CM_STRUCT_SUBFIELD(particles,Particle,vy);
    tax = CM_STRUCT_SUBFIELD(particles,Particle,ax);
    tay = CM_STRUCT_SUBFIELD(particles,Particle,ay);
    tcolor = CM_STRUCT_SUBFIELD(particles,Particle,color);

/*************************************************************/
    /* initialize start location to center of screen */
    CM_f_move_constant_1L (tx, (double)(image_edge_size>>1),
            FLENS);
    CM_f_move_constant_1L (ty, (double)(image_edge_size>>1),
            FLENS);

/*************************************************************/
    /* the velocity will be between +- image_edge_size/32 */
    CM_f_random_1L(tvx, FLENS);
    CM_f_subtract_constant_2_1L(tvx, (double) 0.5,
            FLENS);
    CM_f_multiply_constant_2_1L(tvx,
            (double)(image_edge_size>>5),
            FLENS);

    CM_f_random_1L(tvy, FLENS);
    CM_f_subtract_constant_2_1L(tvy, (double) 0.5,
            FLENS);
    CM_f_multiply_constant_2_1L(tvy,
            (double)(image_edge_size>>5),
            FLENS);
```

```
/*************************************************************/
    /* the acceleration will be between
       +- image_edge_size/128 */
    CM_f_random_1L(tax, FLENS);
    CM_f_subtract_constant_2_1L(tax, (double) 0.5,
            FLENS);
    CM_f_multiply_constant_2_1L(tax,
            (double)(image_edge_size>>7),
            FLENS);

    CM_f_random_1L(tay, FLENS);
    CM_f_subtract_constant_2_1L(tay, (double) 0.5,
            FLENS);
    CM_f_multiply_constant_2_1L(tay,
            (double)(image_edge_size>>7),
            FLENS);

/*************************************************************/
    CM_u_move_constant_1L(tcolor, 1, CM_TYPELEN(char));
}

/*************************************************************/
/*************************************************************/
/*
  This function updates the particles as follows:

  x = x + vx x(n) is vx(n-1)*t + x(n-1) where t=1
  y = y + vy ditto
  vx = vx + ax  vx(n) is ax(n-1)*t + vx(n-1) where t=1
  vy = vy + ay  ditto
  ax = ax*0.66  arbitrarily scale back the acceleration.
  ay = ay*0.66  ditto
  color++;   color indicates age

  For every particle that has gone off the edge of the image,
  the context is set and seed_parts is called.

*/
void
update_part(particles)
    CM_field_id_t particles;
{
    CM_field_id_t
```

```
        tx, ty, tvx, tvy, tax, tay, tcolor,
        saved_context, reseed;

/*************************************************************/
    /* make simple references to each subfield. */
    tx = CM_STRUCT_SUBFIELD(particles,Particle,x);
    ty = CM_STRUCT_SUBFIELD(particles,Particle,y);
    tvx = CM_STRUCT_SUBFIELD(particles,Particle,vx);
    tvy = CM_STRUCT_SUBFIELD(particles,Particle,vy);
    tax = CM_STRUCT_SUBFIELD(particles,Particle,ax);
    tay = CM_STRUCT_SUBFIELD(particles,Particle,ay);
    tcolor = CM_STRUCT_SUBFIELD(particles,Particle,color);

/*************************************************************/
    /* update components */
    CM_f_add_2_1L(tx, tvx, FLENS);
    CM_f_add_2_1L(ty, tvy, FLENS);
    CM_f_add_2_1L(tvx, tax, FLENS);
    CM_f_add_2_1L(tvy, tay, FLENS);
    CM_f_multiply_constant_2_1L(tax, (double) 0.66,
            FLENS);
    CM_f_multiply_constant_2_1L(tay, (double) 0.66,
            FLENS);
    CM_u_add_constant_2_1L(tcolor, 16, CM_TYPELEN(char));

/*************************************************************/
    /* check to see if any particles are out of bounds */
    /* if they are reseed them and proceed */
    saved_context = CM_allocate_stack_field(2);
    reseed = CM_add_offset_to_field_id(saved_context, 1);
    CM_u_move_zero_1L(saved_context, 2);

    CM_store_context(saved_context);

    CM_f_lt_constant_1L(tx, (double)0, FLENS);
    CM_logior_test(reseed);
    CM_store_test(reseed);

    CM_f_lt_constant_1L(ty, (double)0, FLENS);
    CM_logior_test(reseed);
    CM_store_test(reseed);
```

```
    CM_f_gt_constant_1L(tx, (double)image_edge_size,
         FLENS);
    CM_logior_test(reseed);
    CM_store_test(reseed);

    CM_f_gt_constant_1L(ty, (double)image_edge_size,
         FLENS);
    CM_logior_test(reseed);
    CM_store_test(reseed);

/*************************************************************/
/* set context and call seed_parts to reseed those who */
 /* are off the edge */

    CM_load_context(reseed);
    seed_parts(particles);
    CM_load_context(saved_context);

    CM_deallocate_stack_through(saved_context);

}

/*************************************************************/
/*************************************************************/

main(argc, argv)
    int argc;
    char **argv;
{
    CM_field_id_t particles, image;
    CM_geometry_id_t image_geometry;
    CM_vp_set_id_t image_vp_set, particle_vp_set;
    unsigned dimensions[2], gen, color_length;

/*************************************************************/
/* allocate the particles field */

    particle_vp_set = CM_current_vp_set;
    particles = CM_ALLOCATE_TYPE_ON_HEAP(Particle);
    CM_u_move_zero_1L(particles, CM_TYPELEN(Particle));
```

```
/*************************************************************/
/* allocate the image field */

    color_length = CM_TYPELEN(char);
    dimensions[0] = image_edge_size;
    dimensions[1] = image_edge_size;
    image_geometry = CM_create_geometry(dimensions, 2);
    image_vp_set = CM_allocate_vp_set(image_geometry);

    CM_set_vp_set(image_vp_set);
    image = CM_allocate_heap_field(color_length);
    CM_u_move_zero_1L (image, color_length);

/*************************************************************/
/*The main loop */

    CM_set_vp_set(particle_vp_set);
    seed_parts(particles);
    draw_particles(image, particles);

    for (gen = 0 ; gen < NUM_TIME_STEPS ; gen++) {
     update_part(particles);
     draw_particles(image, particles);
    }
}
```

Appendix F

Lines of Sight

```
/*
  This program computes a line of sight from an arbitrary
  point in space to each point of a slice of terrain to
  calculate which of the points are in view. We consider
  the 2-d case:

  ---- *
    |
    |
    |   --------   X                    XXX X          XX
    |     |          X          XXX      X   X XX       X
    |     |           XX      XX   X    X        XXXX  X
^   |     |             XX XXX      XXXX             XX
|   |     |               X
He  | | Ht(n)
|   | |
V   | |
    | |
-------------------------------------------------------------
                           1         2         3
         <--De    0123456789012345678901234567890123456
       |----------|       <-- n -->                    ... |

  *      - Eye point
  X      - Terrain point along n
  He     - Height of the eye
  Ht(n)  - Height of the terrain at point n.
  De     - Displacement of the eye relative to the first point.
           This value is negative if the eye is to the left
           of the eye point.
```

```
In this case, we calculate the angle a(n) in the following
triangle for every point X(n) such that:

  angle a(n) = arctan( (n - De) / (He - Ht(n)) )

            *   -- eye point
            ..
            . .
            .  .
            .   .
            .\ / .
            . a(n).
 He - Ht(n) .      .
            .       . d(n)
            .        .
            .         .
            .          .
            ._b(n) c(n)_.
            ..|........|.X(n)  -- terrain point

                n - De

We consider a terrain point X(n) to be out of view when
the angle a(n) is less than the largest angle
encountered thus far between a(0) and a(n).

   a(n) < MAX( a(0)..a(i) ) where 0 < i <= n

*/
```

Example 29. Lines of sight: **line-of-sight.c**

```
#include <stdio.h>
#include <cm/paris.h>
#include "macros-and-constants.h"

#define EYE_DISPLACEMENT -1000.0  /* De */
#define EYE_HEIGHT          1500.0   /* He */
```

```
#define TERRAIN_HEIGHT_LIMIT750.0  /* limit random */
                    /* height */

/*************************************************************/

typedef struct {
    float
      angles,
      heights,
      displacements,
      side_ratios,
      running_maxs;
    short unsigned int
      news_coords;
} Terrain;

/*************************************************************/

main ()
{
    CM_field_id_t
      terrain_vars,      /* all the variables for
             * the sight calculation */
      angle,
      height,
      displacement,
      side_ratio,
      news_coord,
      running_max,

      in_sight;       /* a 1-bit variable to mark
             * which processor is in
             * sight and which isn't
*/

/*************************************************************/
/* Begin Code */

    CM_init();        /* always the first thing!*/
    CM_set_context();
```

```
/*********************************************************/
/* allocate the sight_vars field */

    terrain_vars = CM_ALLOCATE_TYPE_ON_HEAP(Terrain);
    CM_u_move_zero_1L(terrain_vars, CM_TYPELEN(Terrain));

    in_sight = CM_allocate_heap_field(1);
    CM_u_move_zero_1L(in_sight, 1);

/*********************************************************/
/* initialize locations */

    /* first we clear out the Terrain struct on the CM */
    CM_u_move_zero_1L(terrain_vars, CM_TYPELEN(Terrain));

    /* now for convenience sake we provide aliases for
     * the fields in terrain_vars. Spare us some typing */
    angle =
      CM_STRUCT_SUBFIELD(terrain_vars,Terrain,angles);
    height =
      CM_STRUCT_SUBFIELD(terrain_vars,Terrain,heights);
    displacement=
      CM_STRUCT_SUBFIELD(terrain_vars,Terrain,
          displacements);
    side_ratio =
      CM_STRUCT_SUBFIELD(terrain_vars,Terrain,side_ratios);
    running_max =
      CM_STRUCT_SUBFIELD(terrain_vars,Terrain,running_maxs);
    news_coord =
      CM_STRUCT_SUBFIELD(terrain_vars,Terrain,news_coords);

    /* assign the coordinate each processor */
    CM_my_news_coordinate_1L(news_coord, 0,
              CM_TYPELEN(short));

    /* make the news coordinate usable in a float form.
     * this will serve as the displacement away from the
     * beginning of the terrain */
    CM_f_u_float_2_2L(displacement, news_coord,
               CM_TYPELEN(short), FLENS);
```

```
    /* give us some random scale factor for height */
    CM_f_random_1L(height, FLENS);

    /* scale height with random value */
    CM_f_multiply_constant_2_1L(height,
            (double)TERRAIN_HEIGHT_LIMIT,
            FLENS);


/*********************************************************/
/* calculate angle */
/*
         / (terrain.displacement - EYE_DISPLACEMENT) \
 angle = arctan | ------------------------------------- |
         \     (EYE_HEIGHT - terrain.height)          /
*/

    CM_f_subtract_constant_2_1L(displacement,
            (double) EYE_DISPLACEMENT,
            FLENS);
    CM_f_subfrom_constant_2_1L(height,
               (double) EYE_HEIGHT,
               FLENS);
    /* since we constrain height to be non-zero by our choice
       of constants we don't need to worry about division by
       zero */
    CM_f_divide_3_1L(side_ratio, displacement, height,
              FLENS);

    CM_f_atan_2_1L(angle, side_ratio, FLENS);


/*********************************************************/
/* scan with maximum */

    CM_scan_with_f_max_1L(running_max, angle, 0, FLENS,
           CM_upward, /* direction */
           CM_inclusive, /* include self */
           CM_none, /* do not segment scan */
           CM_no_field /* no segment bit */
           );
```

```
/*********************************************************/
/* determine which points are in view */

    /* set in_sight for those processors whose angle is
       equal to the running max */
    CM_f_eq_1L(angle, running_max, FLENS);
    CM_store_test(in_sight);
}
```

Appendix G

Drawing Lines

```
/*
For each active processor, draw_line draws a line into the
image field between the locations specified in the start
vector (sx, sy) and the end vector (ex, ey). This example is
a simplified version of the line drawing function
CMSR_f_draw_line found in the *Render parallel rendering
library.

THE ALGORITHM
=============
The algorithm we use to is a variant of the Bresenham
algorithm, redesigned slightly to work on the Connection
Machine. In short, the program simply takes the starting
point of the line and uses a Digital Differential Analyzer
(DDA) to the compute the location of the pixels on the line
to the end pixel. A DDA is real nothing more than a process
that, given some initial conditions, allows us to to perform
the same steps on all lines to interpolate the in-between
locations of the pixels. This is the type of situation that
we strive for when programming the CM because we have broken
the problem into very small pieces and use the scan
mechanism to calculate in parallel.

The DDA dictates that we have one axis act as the discrete
independent variable. Or in English, one variable must be be
incremented by one and the other by a differential form.
Since we are drawing straight lines, the form is nothing
more than the slope of the line.  We will use x as the
discrete independent variable and increment it by one and
increment y by the slope.  This is only possible after we
apply the proper initial conditions.
```

The conditions we must have are:

1) dx, the difference in x, must be larger than dy.
2) dx must be positive.

Obviously this restricts us to a limited set of lines if we do not provide a method to make all lines conform to this specification. As it turns out, conforming requires little more than a few comparisons and a few one-bit flags to keep track of things.

THE IMPLEMENTATION

One important aspect of CM programming is keeping all the processors busy. In many cases it pays to look at a problem in terms of decomposition. In this case we can decompose the lines we want to draw into constituent pixels. Once we have the lines decomposed into pixels we are able to operate on them in parallel performing the necessary interpolations. Once interpolated, each point is rendered independently.

When we enter draw_line we enter it in what is assumed to be the vp-set that contains the start and end vertices. From these vertices, we compute the other information we need such as dx, dy, slope, and various flags as mentioned above.

Once this information is calculated, we need to create a place for the pixels to be calculated. We accomplish this by creating a new vp-set with size np, where np is the total number of pixels in all the lines. This allows us to associate one virtual processor with each point. The pixel vp-set will be divided as a contiguous one-dimensional array of line segments. It will be zero-based so that line 0, which has, say, 100 pixels, will take up the first 100 pixel vp's (0-99), line two which has, say, 30 pixels, will be in the next 30 vp's (100-129), and so on.

Next we transfer the data to the pixel vp-set from the line vp-set through a send instruction. We only need one send because the necessary data will be in one big field divided up into separate little fields. When at all possible, this is the preferred method. Reducing the number of sends enhances program performance.

Before we can do a send, however, we must first calculate the send address. Since we have contained dx > dy (with exceptions kept track of with flags) and use a DDA to calculate the intermediate, we know that the exact number of pixels between the start and the end point is dx + 1. In this program we will simply use dx**. If we scan with add the values in dx we have running sum of values that also serve as the send address. To make this a general purpose algorithm for drawing lines, we allow a hypothetical user to call this paris function in an arbitrary vp-set of any legal dimension. Before we do the scan we must temporarily change the shape of the vp-set to make it 1-d. Then do the scan so that each send address we have calculated corresponds to a real send address and that all send addresses, with segment length included, are contiguous. Accordingly the pixel vp set must be in send ordering.

Now we interpolate. In order to do that we must copy the sent value of the starting point to the rest of the processors in the segment with scan with copy. We then scan with add the values of the slope and the unity (a temp field with all ones). Then we add the the scanned slope values to y and the scanned unity values to x.

We now have a computed point in each processor and can thus call draw_point with these values to update the image field.

We are done.

** Using dx as opposed to dx + 1 differs only in that the end point will not be drawn (useful if one desires connected lines). To include the end point, simply add 1 to dx after the slope calculation. As an exercise the reader can change the code so that it will draw the end point optionally based on the true/false condition of either a front-end value specified as an argument to the function or specified in a one bit field passed in as an argument. In either case be careful to maintain the conditions for the DDA and not change the line that is drawn. The only pixel that should change is the end pixel and no other. Or you can use the Connection Machine graphics library *Render.

*/

Example 30. Drawing lines: **draw-line.c**

```
#include <stdio.h>
#include <cm/paris.h>
#include "macros-and-constants.h"

/*************************************************************/

/*
the following structure will be used to transfer all the
necessary information between the line vp-set and the
pixel vp-set
*/
typedef struct {
    float
      tsx_s, tsy_s, slope_s;
    unsigned char
      tcolor_s, flags_s;
} Line_Fields_Packet;

typedef struct {
    unsigned long int
      send_address_s;
    float
      dx_s, dy_s, tex_s, tey_s, temp_x_s, temp_y_s;
} Temp_Fields;

/*************************************************************/
unsigned
round_to_nearest_virtual_machine_size(num_of_vps)
    unsigned num_of_vps;
{
    unsigned exp;

    exp = CM_physical_cube_address_length;
    while (POWER_OF_TWO(exp) < num_of_vps)
      exp++;

    return POWER_OF_TWO(exp);
}
```

```
/*************************************************************/
void
draw_line(image, sx, sy, ex, ey, color, color_length)

    CM_field_id_t
        image,  /* The image field the lines render into.
            Assumed to have at least the number of
            bits as color_length */
        sx, sy, /* The starting coords (integers) */
        ex, ey, /* The ending coords (integers) */
        color;  /* The field which has a value for the
            color of the line */
    unsigned color_length;

{
/*************************************************************/
/* local variables and fields*/
    /* local to the current vp-set when function called */

    CM_vp_set_id_t
      line_vp_set; /* the vp-set upon entering */

    /* these two geometries are for temporarily changing the
       line vp-set geometry to a one-dimensional and back */
    CM_geometry_id_t
      saved_geometry, one_d_line_geometry;

    CM_field_id_t
      line_packet,

      tsx,tsy,      /* the info from the lines. Declared */
      tex,tey,      /* locally so we can bash them. */
      tcolor,

      slope,     /* The slope of each line */

      reverse_p,    /* Set to TRUE if the coords have been
                reversed (X for Y) */
      /* these fields will be part of flags_s */
      saved_context, /* the context bit upon invocation */
      ends_sw,      /* set true if the start and end points
                of the lines are switched */
```

```
      segment_start_bit, /* this field is used to mark the
             beginning of a segment. in the
             line vp-set it is set to one*/

      temp_packet,  /* packets for the above structs */
             /* that will live on the CM */
      dx,dy,     /* the difference along each axis. */

      send_address, /* a pointer to the starting processor
                of the allocated segment for each
                line */

      temp_x, temp_y;

    unsigned
      pixel_sum,   /* the total number of pixels */
         /* for all the lines */
      line_procs;  /* the total number of processors in
             the line vp-set */

/*************************************************************/
/* local variables and fields */
    /* local to a vp-set allocated by this function */

    CM_field_id_t   /* these are all the fields allocated
                in the allocated vp-set
                they mirror the ones above */
      alloc_line_packet,
      alloc_x,
      alloc_y,
      alloc_color,
      alloc_slope,
      alloc_reverse_p,
      alloc_saved_context,
      alloc_ends_sw,
      alloc_segment_start_bit,
      alloc_temp;

    /* these variables are needed to create a new vp-set */

    /* describes the only axis for the new vp-set */
    CM_axis_descriptor_t
```

```
      allocated_descriptor_array[1];
    struct CM_axis_descriptor allocated_descriptor;

    CM_geometry_id_t /* the allocated vp-set's geometry*/
      allocated_geometry;

    CM_vp_set_id_t
      allocated_vp_set;  /* the allocated vp-set */

    /* This variable holds the number of bits that will be
       copy scanned in the new vp-set */
    unsigned copy_size;
/* END DECLARATION*/
/*************************************************************/

/*************************************************************/
/* BEGIN CODE */
    /* save the current vp-set */
    line_vp_set = CM_current_vp_set;

/*************************************************************/
    /* Allocate space and initialize*/

    line_packet =
     CM_allocate_stack_field(CM_TYPELEN(Line_Fields_Packet));

    CM_u_move_zero_1L(line_packet,
              CM_TYPELEN(Line_Fields_Packet));

    tsx = CM_STRUCT_SUBFIELD (line_packet,
              Line_Fields_Packet,tsx_s);
    tsy = CM_STRUCT_SUBFIELD (line_packet,
              Line_Fields_Packet,tsy_s);
    slope = CM_STRUCT_SUBFIELD (line_packet,
            Line_Fields_Packet,
            slope_s);
    tcolor = CM_STRUCT_SUBFIELD (line_packet,
             Line_Fields_Packet,
             tcolor_s);
    reverse_p = CM_STRUCT_SUBFIELD (line_packet,
                Line_Fields_Packet,
                flags_s);
```

```
    /* since reverse_p is actually 8 bits (=sizeof(char))
       we can use the extra space for more 1-bit flags */
    ends_sw =
      CM_add_offset_to_field_id(reverse_p,1);

    saved_context =
      CM_add_offset_to_field_id(reverse_p,2);

    segment_start_bit =
      CM_add_offset_to_field_id(reverse_p,3);

/*************************************************************/
    temp_packet =
      CM_allocate_stack_field(CM_TYPELEN(Temp_Fields));

    CM_u_move_zero_1L(temp_packet,
              CM_TYPELEN(Temp_Fields));

    dx = CM_STRUCT_SUBFIELD (temp_packet,Temp_Fields,dx_s);
    dy = CM_STRUCT_SUBFIELD (temp_packet,Temp_Fields,dy_s);
    tex = CM_STRUCT_SUBFIELD (temp_packet,Temp_Fields,tex_s);
    tey = CM_STRUCT_SUBFIELD (temp_packet,Temp_Fields,tey_s);
    temp_x = CM_STRUCT_SUBFIELD (temp_packet,
                Temp_Fields,temp_x_s);
    temp_y = CM_STRUCT_SUBFIELD (temp_packet,
                Temp_Fields,temp_y_s);
    send_address = CM_STRUCT_SUBFIELD (temp_packet,
                   Temp_Fields,
                   send_address_s);

    /* save context */
    CM_store_context (saved_context);

/*************************************************************/
/* copy start and end into local values */

    CM_f_move_1L (tsx, sx, FLENS);
    CM_f_move_1L (tsy, sy, FLENS);
    CM_f_move_1L (tex, ex, FLENS);
    CM_f_move_1L (tey, ey, FLENS);
    CM_u_move_1L (tcolor, color, color_length);
```

```
/*************************************************************/
/* determine lengths */
    CM_f_subtract_3_1L (dx, tex, tsx, FLENS);
    CM_f_subtract_3_1L (dy, tey, tsy, FLENS);

/*************************************************************/
/* if |dy| greater than |dx| reverse the coords
   and set reverse_p */
    CM_f_abs_2_1L (temp_x, dx, FLENS);
    CM_f_abs_2_1L (temp_y, dy, FLENS);
    CM_f_gt_1L (temp_y, temp_x, FLENS);
    CM_logand_context_with_test();
    CM_u_move_constant_1L (reverse_p, 1, 1);
    CM_swap_2_1L (dx, dy, FLEN);
    CM_swap_2_1L (tsx, tsy, FLEN);
    CM_swap_2_1L (tex, tey, FLEN);
    CM_load_context(saved_context);

/*************************************************************/
/* Compute the slope.*/
/* The slope will always be between [-1,1] because dx >= dy
   This will only work for lines where dx is non-zero. */

    CM_f_divide_3_1L(slope, dy, dx, SLEN, ELEN);

/*************************************************************/
/* Make sure dx is positive. */
/* It will be used to figure out how many vp's to allocate.
   Thus, if it is less than zero we negate it.
   If we negate dx we must also negate the slope and set a
   flag so we know where we have turned things around. */

    CM_f_lt_zero_1L (dx, FLENS);
    CM_logand_context_with_test();
    CM_f_negate_1_1L (dx, FLENS);
    CM_f_negate_1_1L (slope, SLEN, ELEN);
    CM_store_context (ends_sw);
    CM_load_context (saved_context);
```

```
/*********************************************************/
/* Convert dx to integer value from float. */

    CM_s_f_floor_2_2L(temp_x, dx, FLEN, FLENS);
    CM_u_move_1L(dx, temp_x, FLEN);

/*********************************************************/
/* Initialize send_addresss to the processors to be
   allocated

   To make this function truly general we will allow it
   to called with any valid geometry. To make our scheme
   of having a pointer in each line processor that
   points to where it's segment in the pixel vp-set
   begins work, we must
   1) save the current geometry of the line vp-set.
   2) figure out how big it is
   3) create a new geometry of the same size BUT with
   only one dimension (so the scan will work)
   4) change the geometry of the line vp-set to the new
   one-d geometry
   5) perform an scan with unsigned add
   6) restore the original geometry of the line vp-set
*/

    saved_geometry = CM_vp_set_geometry(line_vp_set);
    line_procs =
      CM_geometry_total_processors(saved_geometry);
    one_d_line_geometry = CM_create_geometry(&line_procs,1);

    CM_set_vp_set_geometry(line_vp_set, one_d_line_geometry);
    CM_scan_with_u_add_1L (send_address, dx, /* axis */ 0,
            FLEN,
            CM_upward, CM_exclusive,
            CM_none, CM_no_field);

    CM_set_vp_set_geometry(line_vp_set, saved_geometry);
```

```
/*********************************************************/
/* Allocate a vp-set for the pixels

   Allocate a segment of processors for each line to be
   drawn so that there is one virtual processor per pixel.
   Thus length of each segment equals the number of pixels
   in the line and the total number of processors needed
   is the total number of pixels.

   Since we have contained the slope in the interval [-1,1]
   the number of processors we need to allocate for each
   line is in dx. Thus a global sum of dx determines how
   many total processors we need.
*/
/*********************************************************/
/* Compute the total length which is the size of the
   new vp-set. */

    pixel_sum = CM_global_u_add_1L (dx, FLEN);

/*********************************************************/
/* Allocate the vp-set with a detailed geometry

   We need a detailed geometry because it is the only way
   to specify that we want send address ordering.
   We want send address order because we are going to
   send the line information from the line vp-set to
   this vp-set using an address computed by a running sum.
   If it were news ordering, which is
   the default, then we would have to first deposit the
   news coordinate into the send address. Instead we do
   less overhead on the CM and more on the front end
   which in this case makes more sense.
   The function round_to_nearest_virtual_machine_size is
   used because the vp mechanism requires a vp-set that is
   a power of two in size. The function is defined above.

   In this situation these are the slots in the
   CM_axis_descriptor structure that we need to fill
   with something non-zero.
*/
```

```
    allocated_descriptor.ordering = CM_send_order;
    allocated_descriptor.length =
      round_to_nearest_virtual_machine_size(pixel_sum);

    /* these slots are cleared because they need to be */
    allocated_descriptor.weight = 0;
    allocated_descriptor.on_chip_bits = 0;
    allocated_descriptor.on_chip_pos = 0;
    allocated_descriptor.off_chip_bits = 0;
    allocated_descriptor.off_chip_pos = 0;
    allocated_descriptor.vp_ratio = 0;
    allocated_descriptor.vp_ratio_multiplier = 0;
    allocated_descriptor.address_length = 0;
    allocated_descriptor.virtual_bitmask = 0;

    allocated_descriptor_array[0] =
      &allocated_descriptor;

    allocated_geometry =
      CM_create_detailed_geometry
     (allocated_descriptor_array, 1);

    allocated_vp_set =
      CM_allocate_vp_set(allocated_geometry);

/*************************************************************/
/* Initialize the allocated processors PART I */

    CM_set_vp_set (allocated_vp_set);
    CM_set_context();

    alloc_line_packet =
     CM_allocate_stack_field(CM_TYPELEN(Line_Fields_Packet));

    CM_u_move_zero_1L(alloc_line_packet,
               CM_TYPELEN(Line_Fields_Packet));

    alloc_x = CM_STRUCT_SUBFIELD (alloc_line_packet,
              Line_Fields_Packet,
              tsx_s);
```

```
    alloc_y = CM_STRUCT_SUBFIELD (alloc_line_packet,
              Line_Fields_Packet,
              tsy_s);
    alloc_slope = CM_STRUCT_SUBFIELD (alloc_line_packet,
                  Line_Fields_Packet,
                  slope_s);
    alloc_color = CM_STRUCT_SUBFIELD (alloc_line_packet,
                  Line_Fields_Packet,
                  tcolor_s);
    alloc_reverse_p =
      CM_STRUCT_SUBFIELD (alloc_line_packet,
           Line_Fields_Packet,
           flags_s);
    alloc_ends_sw =
      CM_add_offset_to_field_id(alloc_reverse_p,1);

    alloc_saved_context =
      CM_add_offset_to_field_id(alloc_reverse_p,2);

    alloc_segment_start_bit =
      CM_add_offset_to_field_id(alloc_reverse_p,3);

/*********************************************************/
/* Create one temporary variable for various and sundry
   purposes */

    alloc_temp =
      CM_allocate_stack_field(FLEN);
    CM_u_move_zero_1L (alloc_temp, FLEN);

/*********************************************************/
/* Set the context flag in just those processors that
   will really represent pixels. The rest turn off and
   save the result in alloc_saved_context.
*/
    CM_my_send_address_1L(alloc_temp);
    CM_u_lt_constant_1L (alloc_temp, pixel_sum, FLEN);
    CM_logand_context_with_test();
    CM_store_context(alloc_saved_context);
```

```
/*********************************************************/
/* Initialize each segment start processors

  The values in the field send_address points to the first
  processor in each line segment of the allocated pixel
  vp-set. We therefore use it as a send address to move the
  computed line data to the pixel vp-set. Since the data is
  constituent contiguous fields of a larger field, we only
  have to perform one send.  This is a major performance
  gain.

*/

    CM_set_vp_set (line_vp_set);

    /* set the segment start bit on in all processors that
       will send line values so we know were the segments
       start in the pixel vp-set. */
    CM_u_move_constant_1L(segment_start_bit, 1, 1);

    CM_send_1L (alloc_line_packet, send_address, line_packet,
         CM_TYPELEN(Line_Fields_Packet), CM_no_field);

/*********************************************************/
/* Spread the data from the starting processor in each
   processor to the rest in the segment.

  Notice the use of the CM_start_bit to indicate that this
  will be a segmented scan.
  Notice also that since this is a copy we can optimize
  performance by only performing one scan instead of
  several, one per field.

  we want to copy all but the alloc_segment_start_bit
  and the alloc_saved_context bits because they have
  already been set for each processor. Since they are
  adjacent to each other in CM memory we only need one
  move instead of two.
*/
```

```
    CM_set_vp_set (allocated_vp_set);

    CM_u_move_1L(alloc_temp, alloc_saved_context, 2);

    CM_scan_with_copy_1L (alloc_line_packet,
           alloc_line_packet,
           0, CM_TYPELEN(Line_Fields_Packet),
           CM_upward, CM_inclusive,
           CM_start_bit,
           alloc_segment_start_bit);

    CM_u_move_1L(alloc_saved_context, alloc_temp, 2);

/*************************************************************/
/* Compute the location of each pixel. */

/*************************************************************/
/* Set the increment for the x coordinate to 1 or -1 at each
   pixel depending on ends_sw */

    CM_f_move_constant_1L (alloc_temp, (double)1, FLENS);

    CM_load_context(alloc_ends_sw);
    CM_f_move_constant_1L(alloc_temp, (double)-1, FLENS);
    CM_load_context(alloc_saved_context);

/*************************************************************/
/* Compute the increment for the x coordinate at each
   processor in the field alloc_temp. Since we have
   constrained dx >= dy, we can assume that it is the basis of
   the DDA and thus can be increment by 1 or -1 depending */

    CM_scan_with_f_add_1L (alloc_temp, alloc_temp, 0, FLENS,
           CM_upward, CM_exclusive,
           CM_start_bit,
           alloc_segment_start_bit);
```

```
/*********************************************************/
/* Clean up spillover left by the start-bit segmented scan.
   Unfortunately, the segmented scan with add trashes the
   value in the first processor of the next segment. The
   good news is that we can use the segment bit field as a
   source to load into the context to clean up the field */

    CM_load_context (alloc_segment_start_bit);
    CM_f_move_zero_1L (alloc_temp, FLENS);
    CM_load_context(alloc_saved_context);

/*********************************************************/
/* Add the increment to the x coordinate */

    CM_f_add_2_1L (alloc_x, alloc_temp, FLENS);

/*********************************************************/
/* Compute the increment for the y coordinate at each
   processor in the field alloc_temp. Since we have
   constrained dx >= dy we can assume that it is the basis of
   the DDA and thus can be increment by 1 or -1 depending */

/* Increment y by the slope */

    CM_scan_with_f_add_1L (alloc_slope, alloc_slope, 0,
            SLEN, ELEN,
            CM_upward, CM_exclusive,
            CM_start_bit,
            alloc_segment_start_bit);

/*********************************************************/
/* Clean up spillover left by the start-bit segmented scan.
   Unfortunately, the segmented scan with add trashes the
   value in the first processor of the next segment. The
   good news is that we can use the segment bit field as a
   source to load into the context to clean up the field */

    CM_load_context (alloc_segment_start_bit);
    CM_f_move_zero_1L (alloc_slope, SLEN, ELEN);
    CM_load_context(alloc_saved_context);
```

```
/*********************************************************/
/* Add .5 to slope so we can floor it and
   get the integer value */

    CM_f_add_constant_2_1L (alloc_slope,(double) 0.5,
              SLEN, ELEN);

/*********************************************************/
/* Add the increment to the y coordinate after converting
   it to an integer value */

    CM_f_add_2_1L (alloc_y, alloc_slope, FLENS);

/*********************************************************/
/* Reverse X and Y if necessary */

    CM_load_context (alloc_reverse_p);
    CM_swap_2_1L (alloc_x, alloc_y, FLEN);
    CM_load_context (alloc_saved_context);

/*********************************************************/
/* Draw the data after converting the coordinates to ints */

    CM_s_f_floor_2_2L(alloc_temp, alloc_x,
              CM_TYPELEN(short), SLEN, ELEN);
    CM_s_move_1L(alloc_x, alloc_temp, CM_TYPELEN(short));

    CM_s_f_floor_2_2L(alloc_temp, alloc_y,
              CM_TYPELEN(short), SLEN, ELEN);
    CM_s_move_1L(alloc_y, alloc_temp, CM_TYPELEN(short));

    draw_point (image, alloc_x, alloc_y, alloc_color,
         CM_TYPELEN(short), color_length);

/*********************************************************/
/* Clean up */
    CM_set_vp_set (line_vp_set);
    CM_deallocate_stack_through (line_packet);
    CM_deallocate_vp_set (allocated_vp_set);
    CM_deallocate_geometry (allocated_geometry);
```

```
}
```

```
/*The following file is the main file that uses draw-line in a
real program and displays the results on the Connection Machine
graphics display system.  Essentially, it creates two vp-sets,
one for the line vertices and one for the resultant image.
*/
```

Example 31. Displaying the results of the line-drawing procedure: **draw-line-main.c**

```
#include <stdio.h>
#include <cm/paris.h>
#include <cm/cmfb.h>
#include "macros-and-constants.h"

#define IMAGE_DEPTH (unsigned)8

/*************************************************************/
#define DISPLAY_IMAGE(display,buffer,image,arg1,arg2,clear) \
{ \
    CM_vp_set_id_t current, image_vp_set; \
    current = CM_current_vp_set; \
    image_vp_set = CM_field_vp_set(image); \
    CM_set_vp_set (image_vp_set); \
    CMFB_write_always(display,buffer,image,arg1,arg2); \
    if (clear) \
      CM_u_move_zero_1L (image, clear); \
    CM_set_vp_set(current); \
}

/*************************************************************/
/*************************************************************/

main(argc,argv)
    char *argv[];
{
    unsigned line_dimensions[1], image_dimensions[2];
```

```
    CM_vp_set_id_t line_vp_set, image_vp_set;
    CM_geometry_id_t line_geometry, image_geometry;

    CM_field_id_t
      image_field,
      sx, sy, ex, ey, color,
      fsx, fsy, fex, fey,
      send_address;
    unsigned
      coord_length, image_edge, sal, nl, zoom;

    struct CMFB_display_id display;

/*************************************************************/
    if (argc > 1)
      image_edge = atoi(argv[1]);
    else
      image_edge = 256;

/*************************************************************/

    printf("Warm booting the CM ..."); fflush(stdout);
    CM_init();
    printf("Done\n");

/*************************************************************/
/* set up the framebuffer for use */
    printf("Attaching and Initializing the Display ... Rainbow
palette ...");
    fflush(stdout);
    zoom = 1024/image_edge - 1;
    CMFB_attach_display(NULL,&display);
    CMFB_initialize_display(&display,IMAGE_DEPTH,1);
    CMFB_set_zoom(&display,zoom,zoom,0);
    CMFB_set_color_table_rainbow (&display,
              (double) 1.0     /* red_freq */,
              (double) 1.0     /* green_freq */,
              (double) 1.0     /* blue_freq */,
              (double) 0.0     /* red_phase */,
              (double) 0.33333    /* green_phase */,
              (double) 0.66666    /* blue_phase */,
              (double) 1.0     /* red_ampl */,
              (double) 1.0     /* green_ampl */,
```

```
                (double) 1.0     /* blue_ampl */,
                (unsigned) 0 /*include_first_index*/);
    printf(" Done\n");

/*************************************************************/
/* Initialize image regalia */

    image_dimensions[0] = image_edge;
    image_dimensions[1] = image_edge;
    image_geometry = CM_create_geometry(image_dimensions, 2);
    image_vp_set = CM_allocate_vp_set(image_geometry);
    CM_set_vp_set (image_vp_set);

    image_field = CM_allocate_stack_field(IMAGE_DEPTH);
    CM_u_move_zero_1L(image_field, IMAGE_DEPTH);

/*************************************************************/
/* Initialize line regalia to be the size of the physical machine
*/

    line_dimensions[0] = CM_physical_cube_address_limit;
    line_geometry = CM_create_geometry(line_dimensions, 1);
    line_vp_set = CM_allocate_vp_set(line_geometry);
    CM_set_vp_set (line_vp_set);

    /* Determine the length in bits of a coordinate in our image
       vp-set. We need to add one for a sign bit */

    coord_length = CM_geometry_coordinate_length(image_geometry,
                          0) + 1;

    /* Determine the size of the send address length in the line
        vp-set */

    sal = CM_geometry_send_address_length(line_geometry);

    sx = CM_allocate_stack_field(coord_length);
    sy = CM_allocate_stack_field(coord_length);
    ex = CM_allocate_stack_field(coord_length);
    ey = CM_allocate_stack_field(coord_length);
```

```
    fsx = CM_allocate_stack_field(FLEN);
    fsy = CM_allocate_stack_field(FLEN);
    fex = CM_allocate_stack_field(FLEN);
    fey = CM_allocate_stack_field(FLEN);

    color = CM_allocate_stack_field(IMAGE_DEPTH);

    send_address = CM_allocate_stack_field(sal);

/*********************************************************/
 /* set the vertices of each line to be radial.  The starting
point of each */
 /* line is the center of the screen (actually the 2-d image vp-
set).  The */
 /* end point of each is a unique point along the perimeter of
the screen. */

    set_radial_lines(sx, sy, ex, ey, color, send_address,
              coord_length, IMAGE_DEPTH, sal,
              image_dimensions[0]);

    CM_f_u_float_2_2L(fsx,sx,coord_length, FLENS);
    CM_f_u_float_2_2L(fsy,sy,coord_length, FLENS);
    CM_f_u_float_2_2L(fex,ex,coord_length, FLENS);
    CM_f_u_float_2_2L(fey,ey,coord_length, FLENS);

/*********************************************************/
/* Draw lines one at a time */

    printf("Draw lines one at a time\n");
    for (nl=0; nl< image_dimensions[0]<<2; nl+=image_dimen-
sions[0]>>4) {
     CM_set_context();
     CM_u_eq_constant_1L (send_address, nl, sal);
     CM_logand_context_with_test();
     draw_line (image_field, fsx, fsy, fex, fey, color,
            IMAGE_DEPTH);
     DISPLAY_IMAGE(&display,CMFB_current_buffer(&display),
               image_field,0,0,IMAGE_DEPTH);
    }
```

```
/*************************************************************/
/* Draw lines in parallel */

    printf("Draw lines in parallel\n");
    CM_set_context();
    CM_u_lt_constant_1L (send_address, 4*image_dimensions[0],
sal);
    CM_logand_context_with_test();
    draw_line (image_field, fsx, fsy, fex, fey, color,
            IMAGE_DEPTH);
    DISPLAY_IMAGE(&display,CMFB_current_buffer(&display),
            image_field,0,0,IMAGE_DEPTH);

/*************************************************************/
    CM_deallocate_stack_through(sx);

}

/*************************************************************/
/*************************************************************/
/* This function sets the first 4*image_size processors to have
   start vertices in the center of the screen and the end points
to
   the perimeter of the rectanglar area defined by
   image_sizeximage_size. the result will draw radial if the
   vertices are passed to draw_line. color is also set up. tmp is
   paseed in because it is useful outside the this procedure */

set_radial_lines(sx, sy, ex, ey, color, tmp,
          coord_length, color_length, tmp_length, image_size)
    CM_field_id_t sx, sy, ex, ey, color, tmp;
    unsigned coord_length, color_length, tmp_length, image_size;
{

    CM_set_context();

    CM_u_move_zero_1L (sx, coord_length);
    CM_u_move_zero_1L (sy, coord_length);
    CM_u_move_zero_1L (ex, coord_length);
    CM_u_move_zero_1L (ey, coord_length);
    CM_u_move_zero_1L (tmp, tmp_length);
```

```
    CM_my_news_coordinate_1L(tmp,0,tmp_length);

/*************************************************************/
    /* initialize color */
    CM_u_add_3_1L(color,tmp,tmp,color_length);
    CM_u_eq_constant_1L(color,0,color_length);
    CM_logand_context_with_test();
    CM_u_move_constant_1L(color,1,color_length);

    CM_set_context();

/*************************************************************/
    /* set starting points to middle of screen */
    CM_u_move_constant_1L (sx, image_size>>1, coord_length);
    CM_u_move_constant_1L (sy, image_size>>1, coord_length);
            /* set end points of 135 thru 45 deg lines */
    CM_u_lt_constant_1L (tmp, image_size, tmp_length);
    CM_logand_context_with_test();
    CM_u_move_1L (ex, tmp, coord_length);
    CM_u_move_constant_1L (ey, 0, coord_length);
/*************************************************************/
    /* set 44 to 315 end points  */
    CM_set_context();
    CM_u_ge_constant_1L (tmp, image_size, tmp_length);
    CM_logand_context_with_test();
    CM_u_lt_constant_1L (tmp, 2*image_size, tmp_length);
    CM_logand_context_with_test();
    CM_u_move_constant_1L (ex, image_size - 1, coord_length);
    CM_u_move_1L (ey, tmp, coord_length);
    CM_u_subtract_constant_2_1L(ey,image_size,coord_length);
/*************************************************************/
    /* set 314 to 225 end points   */
    CM_set_context();
    CM_u_ge_constant_1L (tmp, 2*image_size, tmp_length);
    CM_logand_context_with_test();
    CM_u_lt_constant_1L (tmp, 3*image_size, tmp_length);
    CM_logand_context_with_test();
    CM_u_move_constant_1L (ey, image_size - 1, coord_length);
    CM_u_move_1L (ex, tmp, coord_length);
    CM_u_subtract_constant_2_1L(ex,2*image_size,coord_length);
    CM_u_subfrom_constant_2_1L(ex, image_size - 1,coord_length);
/*************************************************************/
    /* set 224 to 136 end points   */
    CM_set_context();
```

```
    CM_u_ge_constant_1L (tmp, 3*image_size, tmp_length);
    CM_logand_context_with_test();
    CM_u_lt_constant_1L (tmp, 4*image_size, tmp_length);
    CM_logand_context_with_test();
    CM_u_move_constant_1L (ex, 0, coord_length);
    CM_u_move_1L (ey, tmp, coord_length);
    CM_u_subtract_constant_2_1L(ey, 3*image_size,coord_length);
    CM_u_subfrom_constant_2_1L(ey, image_size - 1,coord_length);

    CM_set_context();
}
```

Index

Index

G

H

I

L

N

O

P

R

S

T

U

V