Chapter 1

The Compilation and Execution Models

This guide explains how C* programs are compiled and executed, and describes how to increase the performance of programs written in C* for execution on a CM-5 using the vector units (VUs). Chapter 6 describes how to reduce memory usage in a CM-5 C* program.

This chapter describes the basic model that the CM-5 C* compiler uses in generating code, and how to generate and analyze the assembly-language files you can use in trying to improve your code. It also discusses the PN (processing node) code blocks that perform parallel computation, and how you can control construction of these code blocks.

In the manual there are occasional examples that show generated DPEAC assembly code. We illustrate the assembler code so that no more than a rudimentary knowledge of these assembly languages is necessary to understand the examples. The appendix goes into more detail about the DPEAC assembly code.

Where specific examples of generated code or specific performance numbers are given, keep in mind that these may change with releases of the compiler, and that they are provided only as general guides to understanding performance issues. In addition, the advice given in this guide may not be applicable to programs compiled with future versions of the compiler.

1.1 The Compilation Model

The C* compiler, like an ordinary C compiler, transforms the program from the high-level language in which it is written into machine code. For the CM-5, the compiler-generated machine code consists of two parts:

- SPARC assembler code, which is executed on the partition manager; we also refer to this as *scalar code*.
- DPEAC assembler code, which is executed on the PNs; we also refer to this as *parallel code*.

Section 1.2.1 describes how to examine these assembly-language files.

For C* code that consists only of ordinary scalar C operations, the compiler generates scalar code that is similar to the code an ordinary C compiler would generate.

For C* code that performs parallel communication operations (that is, operations that require transferring data between processors), the scalar code includes calls to an internal run-time system that implements the operations.

For C* code that performs parallel computation operations (that is, operations that take place independently in each processor), the compiler generates two kinds of code:

- routines that execute simultaneously on all the PNs; these are called *PN code blocks*
- the scalar calls that initiate these PN code blocks

All parallel computation is performed in the PN code blocks. The vector units are used for both integer and floating-point operations.

Figure 1 diagrams the CM-5 C* compilation model.

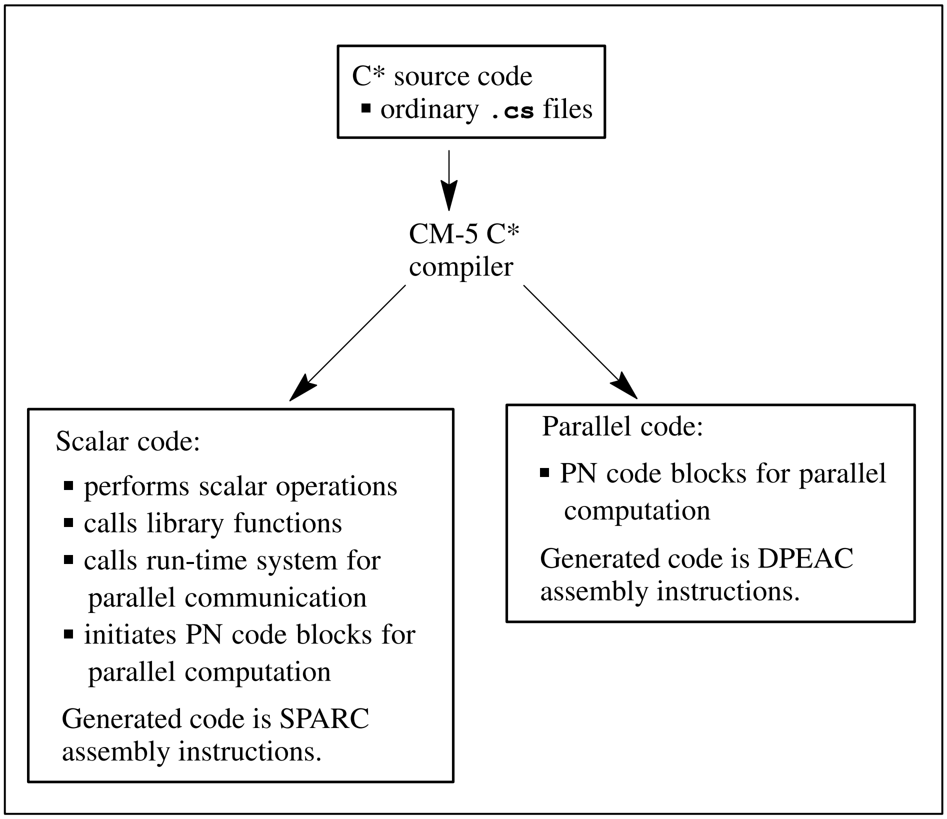

Figure 1. The CM-5 C* compilation model.

1.2 The Compilation Process

The C* compiler generates two separate assembly-language output files for its scalar and parallel code.

The scalar code is compiled into a SPARC assembly-language file with a name of the form *program-name*`.s`.

The production of the parallel code actually involves two steps:

1. First, the compiler produces a file containing DPEAC code. DPEAC code is a mixture of SPARC assembler instructions and instructions that perform operations using the vector units. This file has a name of the form *program-name*`.pe.dp`.

2. The compiler (via the **dpas** assembler) then translates the DPEAC code into SPARC instructions; when executed, these SPARC instructions initiate the vector-unit operations on the VUs. This file has a name of the form *program-name* **.pe.s**.

The two .s files are then assembled via **as** into object files, which are then linked via the **cmld** linker into the executable program.

Figure 2 outlines the compilation process.

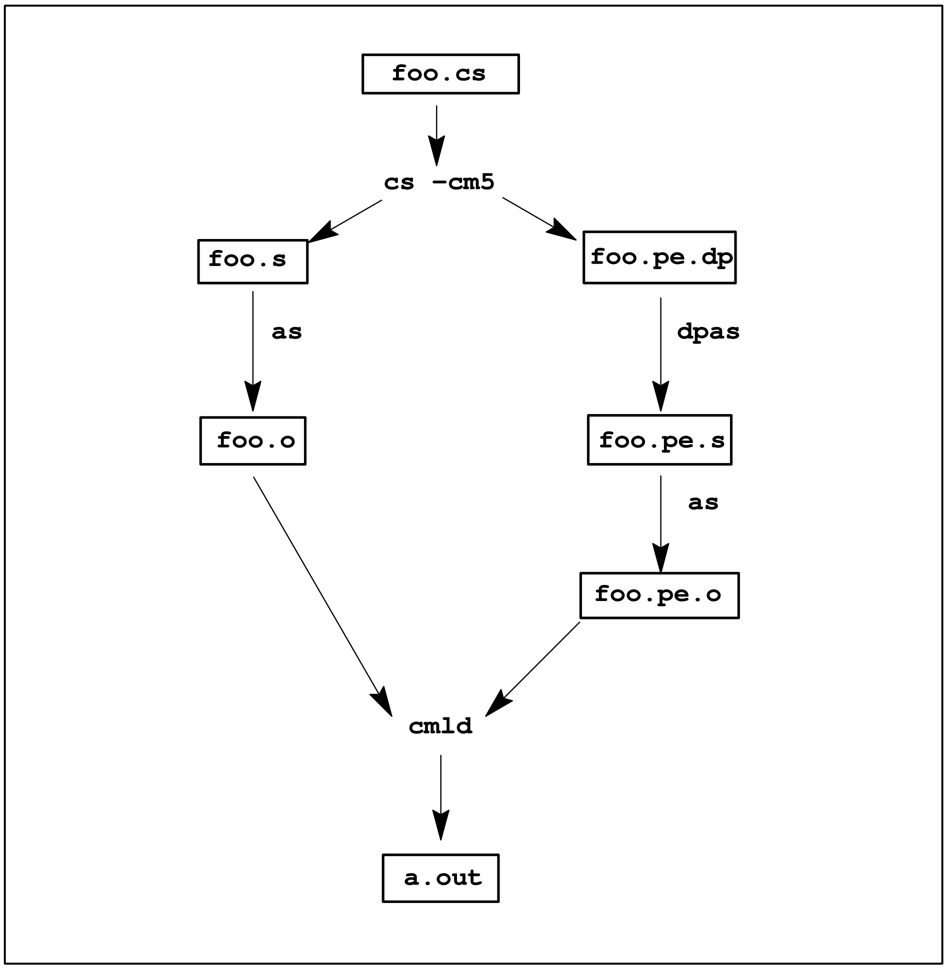

Figure 2. The CM-5 C* compilation process.

1.2.1 Examining Assembly-Language Files

Normally the compiler creates intermediate files in a temporary location, and deletes them after they are no longer needed. Use the **-keep s** compiler option to tell the compiler to retain the **.s** and the **.pe.s** assembly-language files in the current directory. For example:

```
% cs -cm5 -vu -keep s fishcake.cs
```

Use the **-keep dp** option to tell the compiler to retain the **.pe.dp** DPEAC file. For example:

```
% cs -cm5 -vu -keep dp fishcake.cs
```

Generally you will be most interested in the DPEAC file, since this gives the clearest view of how the program is going to execute parallel computation on the PNs. For information on DPEAC, consult the *DPEAC Reference Manual*. See also the example in the Appendix.

1.3 PN Code Blocks

As we discussed earlier, the compiler generates PN code blocks to perform parallel computation operations.

The compiler-generated PN code blocks are simple routines that are executed simultaneously on all the processing nodes. They include DPEAC instructions that each PN broadcasts simultaneously to all four of its vector units.

When a program creates a shape (either through the declaration of a fully specified shape variable or through dynamic allocation), the C* run-time system dynamically determines a regular mapping of the positions of the shape onto the VUs in the partition. See Appendix B of the *C* Programming Guide* (May 1993 edition) for more information. The details of how that mapping works are irrelevant to PN code blocks except for one point: A shape always maps a certain number of elements onto every vector unit. This number is called the *subgrid size* and is always a multiple of eight. The subgrid size is roughly the number of positions in the shape divided by the number of vector units in the partition; it is rounded up in some cases to meet the constraints of the layout mechanism.

A PN code block consists of a single loop called the *subgrid loop*. The subgrid loop iterates over the number of elements on each VU, eight at a time. In each iteration, the subgrid loop uses vector operations to perform the appropriate computation on eight subgrid elements per VU, or 32 elements per PN (since there are four VUs per PN). See Section 1.4 for an example of a simple code block.

1.3.1 How PN Code Blocks Are Invoked

The scalar portion of the program, running on the partition manager, causes a PN code block to be executed simultaneously on all PNs by calling an internal run-time system function. Through this call, the following information needed to invoke the PN code block is broadcast to the PNs: the address of the code block itself, the subgrid size, the memory addresses of the parallel variables used in the code block, and the values of scalar variables used in the code block.

1.3.2 Understanding Costs in PN Code Blocks

The amount of time spent executing a PN code block is equal to the startup time plus the time spent in the subgrid loop.

The startup time includes:

- the time spent packing up the information to be passed from the partition manager to the PNs
- the time spent broadcasting this information to the PNs (this typically dominates the startup time)
- the time spent in the prologue of the PN code block itself

The startup time is incurred once whenever the code block is invoked. It depends on the amount of information passed to the PN code block, but not upon the subgrid size.

The loop time depends upon two things:

- the number of iterations spent in the loop (that is, the subgrid size divided by eight); this depends in turn upon the size of the current shape
- the amount of code in the loop body

Note that parallel computation performance does not depend upon the rank or other layout characteristics of the current shape, only upon the shape's size.

The PN code block startup time is pure overhead, since useful computation is not performed during startup. When the startup time dominates your program's execution time, the program is running inefficiently. Two situations can cause this:

- Your subgrid sizes are too small.
- Your code is executed with many smaller code blocks rather than few larger ones.

You can address the first problem by choosing a shape size that is large relative to the machine size. Doing this may require reconsidering how your problem is mapped onto the machine. The subgrid size you need to amortize code block startup costs will vary, depending upon how much code is in your code blocks and how much information is passed to the code block. As a rough rule of thumb, aim for a subgrid size of at least 64 — that is, a shape size that has at least 64 times as many positions as **`positionsof(physical)`**, the number of vector units in the partition.

1.4 A Simple C* Routine

The C* routine below performs both ordinary scalar C operations and parallel operations:

```
#include <stdio.h>
void fishcake(int x, int:current a, float:current b)
{
    float sum;
    x = x + 2;
    printf("The value of x is: %d\n", x);
    b = b * 17.2f + a * x;
    sum = += b;
    printf("The sum of b is: %f\n", sum);
}
```

The statement

```
x = x + 2;
```

and both calls to the **`printf`** function are generated just as an ordinary C compiler would generate them; all are executed on the partition manager.

The statement

```
b = b * 17.2f + a * x;
```

performs parallel computation. The compiler generates a PN code block, and a call in the generated scalar code initiates that code block on all of the PNs after the first call to **`printf`**.

The statement

```
sum = += b;
```

performs a form of parallel communication — in this case, all of the values of **`b`** are summed to a single value. The compiler generates a call in the scalar code to a run-time routine that performs this reduction.

Thus, the generated scalar code for the routine is structured as shown in Figure 3.

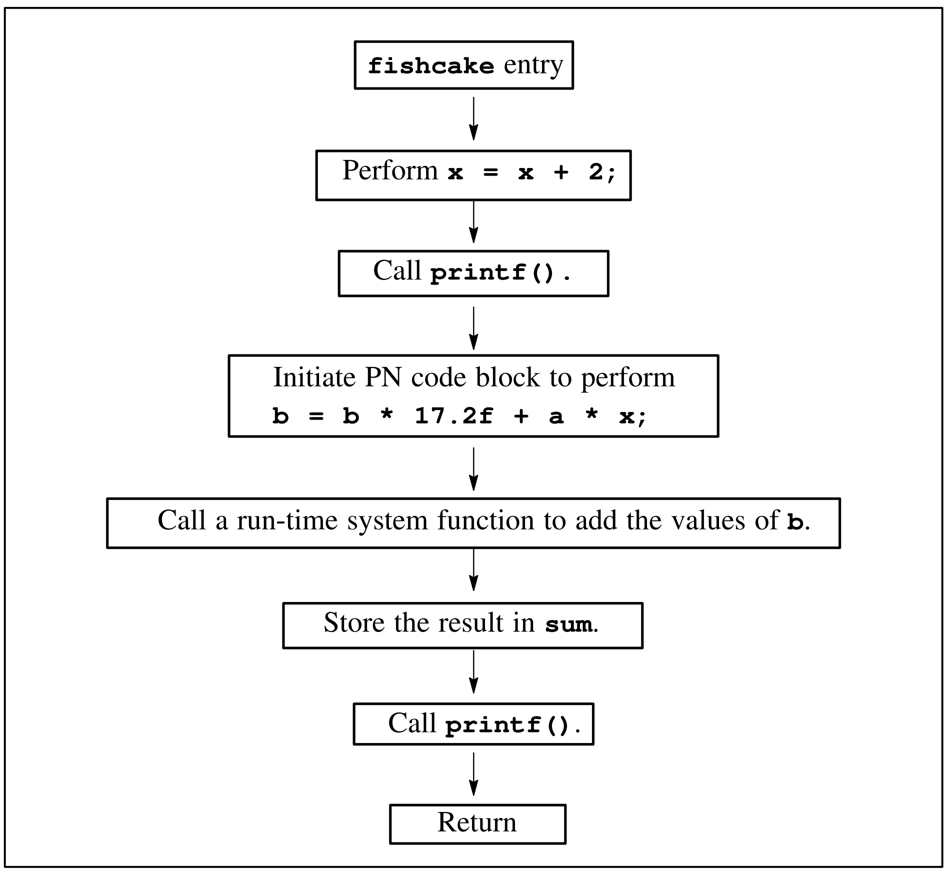

Figure 3. Structure of a C* routine.

1.4.1 The Structure of the PN Code Block

Our simple routine contains one PN code block that performs the computation

```
b = b * 17.2f + a * x;
```

where **b** and **a** are parallel variables and **x** is a scalar variable. Figure 4 shows the structure of the PN code block.

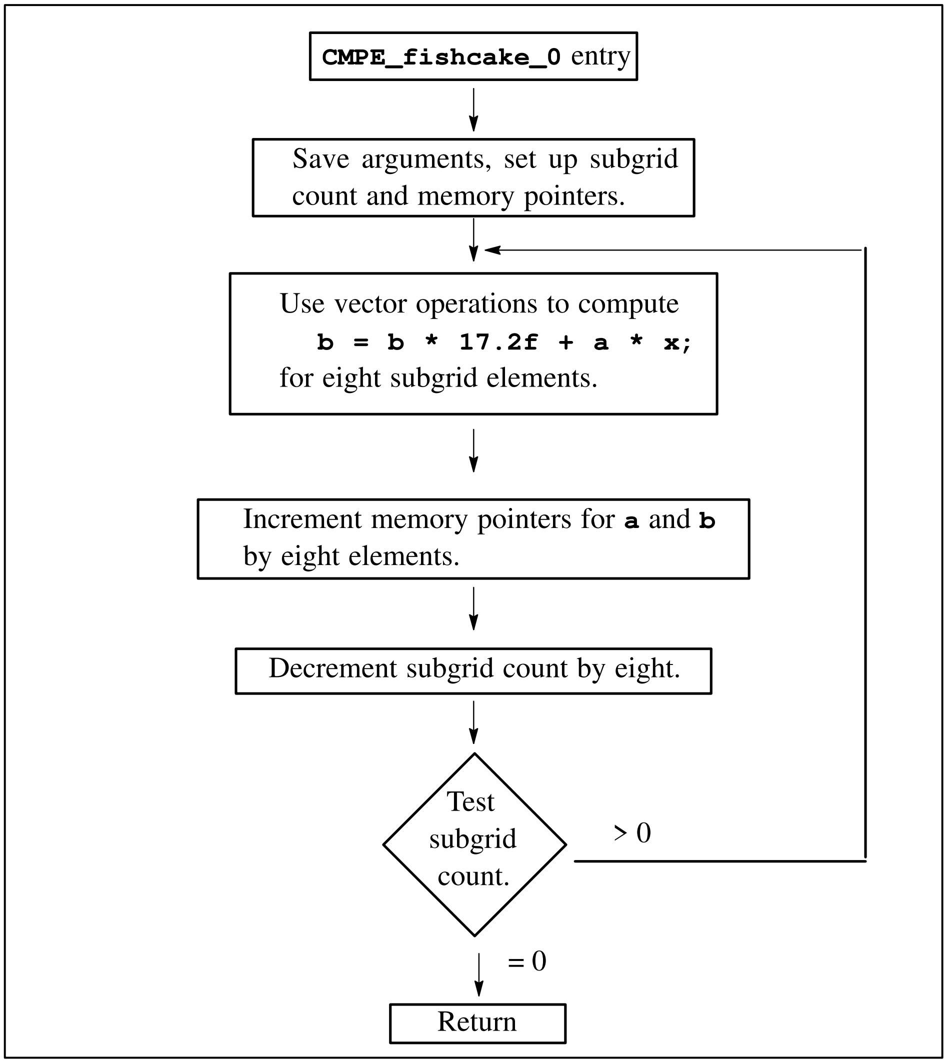

Figure 4. Structure of a PN code block.

In this example, the startup time for the PN code block is about 25 μs., and the loop time is about 6 μs. per loop iteration, or 0.8 μs. per subgrid element, since each loop iteration operates on eight subgrid elements. (Your timings may vary.) See Chapter 2 for an example of how to measure these times.

Since the startup cost is significantly larger than the cost of one loop iteration in this code block, we will not be computing results efficiently unless our shape size is large enough that the loop makes several iterations.

Suppose our subgrid size is 32 elements per VU. (On a 32-PN CM-5 partition, this requires a shape with 4096 elements.) In this case, our loop executes four iterations. The total time spent executing the PN code block is about 50 μs., of which 25 are the overhead of starting the code block and 25 are spent doing actual computation in the loop body.

1.5 Controlling the Construction of PN Code Blocks

In general, a PN code block may perform computation for several statements or even a fraction of a statement. The compiler merges parallel computation into a single PN code block only when the computation is not interrupted by an operation that “breaks code blocks.” Since extra code blocks means extra overhead, it is generally beneficial to write code that produces fewer code blocks. To increase performance, you therefore need to understand what sorts of operations cause the compiler to break code blocks. Section 1.5.1 goes into detail about these operations. Here we consider a simple example:

```
#include <stdio.h>
void fishhead(float:current a, float:current b)
{
    a = a * 17.2 + 3;
    b = b / 4 + a;
    printf(”The sum of a is %f\n”, += a);
    printf(”The sum of b is %f\n”, += b);
}
```

In this case, two consecutive statements perform parallel computation, and the compiler produces a single PN code block to perform the computation for both statements. However, suppose we had written the code in a slightly different order:

```
#include <stdio.h>

void fishhead(float:current a, float:current b)
{
    a = a * 17.2 + 3;
    printf("The sum of a is %f\n", += a);
    b = b / 4 + a;
    printf("The sum of b is %f\n", += b);
}
```

In this case, the compiler emits two PN code blocks, one for the statement

```
a = a * 17.2 + 3;
```

and the other for the statement

```
b = b / 4 + a;
```

1.5.1 Operations that Cause Code Blocks to Be Broken

This section discusses operations that cause a code block to be broken when they appear in the middle of parallel computation. Note that future versions of the compiler may remove this limitation for many of these operations.

Flow Control

Any operation that performs flow control causes code blocks to be broken. For example, parallel code in the body of an **`if`**, **`while`**, **`for`**, **`do`**, or **`switch`** statement is always compiled into a code block that is separate from those of the surrounding code. See Section 4.5 for a discussion of a way of expanding code blocks by unrolling loops.

(Keep in mind that the **`||`**, **`&&`**, and **`?:`** operators result in flow control when their operands are scalar types.)

Function Calls

Any call to a function causes code blocks to be broken. This includes calls to scalar functions, functions in the C* communication library, and even simple math functions.

Note that calls to the CM timer functions themselves can break code blocks. Introduction of these calls inside what would otherwise be a single code block can therefore hurt the performance of the code you are timing. See Chapter 2 and the *CM-5 C* User's Guide* for information on the CM timer functions.

Contextualization

The body of a **`where`** or **`everywhere`** statement is always compiled into code blocks that are separate from those of surrounding code.

Furthermore, the **`&&`**, **`||`**, and **`?:`** operators cause code blocks to be broken when their operands are parallel, because these operators perform implicit contextualization. See the *C* Programming Guide* for an explanation of implicit contextualization through these operators.

A future version of the compiler may remove this limitation in many cases.

See Section 4.2 for more information on contextualization.

Parallel Communication Operations

Statements that perform parallel communication operations, in the form of left-indexing to perform send, get, or grid communication operations, break code blocks.

Scalar Left Indexing

Scalar left indexing to access particular positions of a parallel variable breaks code blocks.

Parallel Reductions

Reducing a parallel value to a scalar value — through the use of the **+=**, **-=**, ***=**, **/=**, **&=**, **|=**, **<?=**, or **>?=** unary operators with parallel operands, or through an explicit cast of a parallel value to a scalar value — breaks code blocks.

with Statements

The body of a **with** statement is always compiled into one or more code blocks that are separate from those of surrounding code.

Extra Levels of { } Braces Forming Compound Statements

Introducing extra levels of **{ }** braces in C* code (these are called *compound statements*) causes the body of the braces to be in a separate code block. A future version of the compiler may remove this limitation.

Comma Operator

The use of the comma operator in parallel expressions causes code blocks to be broken.

For example, the statement in:

```
int:current a, b, c, d;
a = b, c = d;
```

can be rewritten as:

```
a = b;  c = d;
```

to avoid this problem.

A future version of the compiler may remove this limitation.

Multiple Assignments

Multiple assignments in a single statement cause code blocks to be broken in some cases.

For example, the assignment in:

```
int:current a, b, c, d;
a = b = c = d;
```

can be rewritten as:

```
c = d;
b = c;
a = b;
```

to avoid this problem.

A future version of the compiler may remove this limitation.

Compiling C* with the –g Option

Using the **-g** option to compile C* code always forces separate statements to emit separate code blocks. This can dramatically affect your program's performance. You should be aware of this when examining the compiler output or timing code.

Chapter 2

Timing and Profiling Code

There are two mechanisms available for analyzing the performance of a C* program:

- You can use the **CM_timer** facility to insert calls in your program; these calls start and stop timers as the programs run, and report information about how much time was spent in specific portions of the program. This facility is described in the *CM-5 C* User's Guide*.

- You can compile your program with the **-cmprofile** option, and then display information about the program's performance within the Prism programming environment. Prism performance analysis is described in the *Prism User's Guide*.

In general, using Prism is more convenient, since it doesn't require changing your source code; it also provides an easy-to-read graphical interface to the results. However, the current implementation of Prism performance analysis has several restrictions that can lead to inaccurate or misleading results. The **CM_timer** facility gives you more control over exactly what gets timed and lets you perform computations on the results. You may find the Prism performance analysis facility more useful for assessing the overall performance of an application and the **CM_timer** facility more useful for timing specific pieces of code.

Note these other general points:

- For both Prism performance analysis and the **CM_timer** facility, compiling with the **-g** option will distort the results, since it forces each statement to be in a separate code block.

- It is possible to use both Prism and the **CM_timer** facility to analyze a program's performance at the same time. However, there are complications if you use timers that have numbers greater than 4 in the program.

See the *Prism User's Guide* for more details. In addition, note that Prism will time the programmer's **CM_timer** routines, and the programmer's **CM_timer** routines will time Prism's timing activity. This will usually have a greater effect on the programmer's **CM_timer** results, since Prism usually inserts more timing calls than the programmer.

2.1 Using the CM_timer Facility

This section gives an example of using the **CM_timer** facility, and discusses how to analyze the results. It assumes you are familiar with the **CM_timer** calls, as described in the *CM-5 C* User's Guide*.

2.1.1 Timing Computation

Here are some tips to keep in mind when timing computation:

- Read the *elapsed* time to report how long the program spent performing the operation. The elapsed time reports the time that the process spent executing, and is not actual wall-clock time. The *CM busy* time reports only how long the PNs were performing computation; when it differs significantly from the elapsed time, it is because the program is busy executing operations that do not involve the PNs.

- Keep in mind that interrupting code blocks with the **CM_timer** calls can change performance. Because calls such as **CM_timer_start** and **CM_timer_stop** are ordinary function calls, they can cause the compiler to break code blocks; see Section 1.5.1. If you are concerned about this, avoid putting these calls in the middle of code that would ordinarily be a single code block.

- When timing code that performs parallel I/O operations via the CMFS interface, the **CM_timer** facility will not report accurate results, because its timings do not include the time spent by the operating system performing the I/O operations. You should use the UNIX timers (such as **gettimeofday**) to obtain actual wall-clock times.

- When timing small portions of code, time the code executing several times in a loop to get more accurate timing results.

Suppose we wish to time the computation performed in the following piece of code:

```
double:current a[5], sum;

/* ... */

everywhere
{
    sum = a[0] + 2 * a[1] + 3 * a[2] + 4 * a[3] + 5 * a[4];
}
```

The simplest approach is to introduce calls **CM_timer_start** and **CM_timer_stop** around the computation. However, this is a small fragment of code, and the time measurement from a single execution might not accurately reflect the code's performance, particularly with small subgrid sizes in which the execution time is less than 100 µs. To measure the performance more accurately, we will introduce a loop that performs the computation 100 times.

The computation in this example is performed in a single PN code block. We can measure both the PN code block's startup cost and the time spent executing its subgrid loop by measuring the time required to execute the code using shapes that have widely varying subgrid sizes. For the smallest subgrid sizes, we expect the startup cost to dominate the elapsed time; the time to exercise a single cycle of the subgrid loop should be negligible. For very large subgrid sizes, most of the time is spent in the subgrid loop, and relatively little time is spent in the code block startup.

Timing actual applications need not be this complicated, of course. It is sufficient to just insert **CM_timer** calls around code that is being timed, particularly when the code in question is larger than a single statement. The technique shown here is useful for measuring the specific overheads in a code block, but for a larger application you may be interested more in aggregate performance.

The complete program we'll use is below:

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

#define TRIALS 100

void fishcake(void)
{
    int trial;
    double:current a[5], sum;
```

```
    /* ... */

    everywhere
    {
        /*
         * Time our expression, looping to execute it 100 times:
         */
        CM_timer_start(0);
        for(trial = 0; trial < TRIALS; ++trial)
        {
            sum = a[0] + 2 * a[1] + 3 * a[2] + 4 * a[3] +
                   5 * a[4];
        }
    }    CM_timer_stop(0);
}

void time_subgrid_size(int size)
{
    /*
     * Call the above routine using a current shape that has the
     * specified subgrid size.
     */
    shape [size * positionsof(physical)]S;
    double t;

    CM_timer_clear(0);
    with(S)
    {
        fishcake();
    }
    t = CM_timer_read_elapsed(0);

    printf("subgrid size: %d\n", size);
    printf("total time: %f sec\n", t);
    printf("time per trial: %f us.\n", t * 1000000.0 / TRIALS);
    printf("time per subgrid element: %f us.\n\n",
           t * 1000000.0 / (TRIALS * size));
}

main()
{
    /*
     * Time our example using specific subgrid sizes.
     */
```

```
    time_subgrid_size(8);
    time_subgrid_size(16);
    time_subgrid_size(32);
    time_subgrid_size(64);
    time_subgrid_size(128);
    time_subgrid_size(1000);
    time_subgrid_size(10000);
}
```

Because the program scales the size of its test shapes with the size of the partition (by using **positionsof(physical)**), it produces timings that are independent of the machine size being used.

Here is a sample output of this program:

```
subgrid size: 8
total time: 0.002656 sec
time per trial: 26.559697 us.
time per subgrid element: 3.319962 us.

subgrid size: 16
total time: 0.002661 sec
time per trial: 26.609091 us.
time per subgrid element: 1.663068 us.

subgrid size: 32
total time: 0.003355 sec
time per trial: 33.546364 us.
time per subgrid element: 1.048324 us.

subgrid size: 64
total time: 0.004807 sec
time per trial: 48.074242 us.
time per subgrid element: 0.751160 us.

subgrid size: 128
total time: 0.008152 sec
time per trial: 81.522727 us.
time per subgrid element: 0.636896 us.

subgrid size: 1000
total time: 0.050639 sec
time per trial: 506.393030 us.
time per subgrid element: 0.506393 us.

subgrid size: 10000
```

```
total time: 0.501246 sec
time per trial: 5012.457879 us.
time per subgrid element: 0.501246 us.
```

Remember that the actual subgrid size selected by the run-time system (when the program is compiled for VU execution) is always a multiple of eight. If we had called **`time_subgrid_size`** with 1, a layout with a subgrid size of 8 would have been used; see Section 1.3. The function **`CMC_shape_subgrid_size`**, described in the *C* Programming Guide*, returns a shape's actual subgrid size.

Determining Subgrid Loop Costs

With a subgrid size of eight, the time to execute each trial (which invokes our code block exactly once) is about 27 µs. This is approximately the startup time of this code block. The startup time is constant for each trial, independent of the subgrid size. As we increase the subgrid size, and the startup time becomes a negligible fraction of the time to execute each trial, the time per subgrid element decreases to reach about 0.50 µs. Thus, each iteration of the subgrid loop, which performs computation for eight subgrid elements, executes in about 4.0 µs. in this example.

Counting Flops

The expression we are timing performs eight floating point operations (four adds and four multiplies) for each position of the current shape. With a subgrid size of 1000, each PN performed the computation for 4000 positions (because each PN has four VUs) in 506 µs. We can compute the number of floating operations per second for each PN as follows:

$$\frac{8 \text{ ops/element} \cdot 4 \cdot 1000 \text{ elements/PN}}{506 \cdot 10^{-6} \text{ s.}} = 63 \text{ Mflops/PN}$$

Thus, on a 32-node partition, the computation with this subgrid size is performed at about 2.0 Gflops.

Assessing Computation Performance

The CM-5 vector units can perform computation at a rate of up to 128 Mflops (million floating point operations per second) per PN. They can also perform integer computation at this rate. This performance figure assumes that the VUs are continuously executing vector operations that are performing chained operations (these are typically operations that do both a multiply and an add as a single operation). The example above is computing an expression that allows the compiler to use these operations, and thus achieves a high Flops rate. In practice, it is not possible for the compiler to emit chained operations for some expressions, and an application that is performing at 30 or 40 Mflops per PN may be doing as well as it can do, even though this figure is far below the theoretical peak Flops rate of the machine.

2.1.2 Timing Communication

You can measure the performance of code that performs parallel communication similar to the way you measure the performance of code that performs computation, by using the **`CM_timer`** calls. Because communication operations are not performed within code blocks, inserting calls to the **`CM_timer`** functions among communication operations does not affect performance by breaking code blocks.

Assessing the Performance of General Communication

The performance of send and get operations is usually characterized in terms of the bandwidth achieved for each PN. With the assumption that each position is sending data to or getting data from a position on another PN, the bandwidth per PN is the total amount of data sent (or gotten), divided by the number of PNs, divided by the amount of time required for the operation.

Table 1 roughly describes the send and get performance you can expect when using a random communication pattern and reasonably large subgrid sizes.

Table 1. Send and Get bandwidth estimates (Mbytes/s/PN).

Data Size	4-byte	8-byte	16-byte
Send	1.2	2.2	3.2
Get	0.6	1.0	1.7

Assessing the Performance of Grid Communication

The performance of grid communication is more difficult to characterize than that of general communication (send and get operations). Grid communication performance depends very much on the rank of the current shape, the shape's size and layout, and the axis along which the communication is performed.

There are two components to the implementation of grid communication operations. One moves data within the VUs (called *on-VU movement*), and one moves data between the VUs (called *off-VU movement*). Movement within the VUs is limited both by how fast the VUs can access memory and by how efficiently the data motion can be vectorized. Movement between the VUs is limited by the memory bandwidth between VUs on the same PN and the data router network bandwidth between PNs. (See the discussion of **`allocate_detailed_shape`** in Appendix B of the *C* Programming Guide* for further discussion of how shape layout affects grid communication performance.)

When working with 1-dimensional shapes, the performance characteristics of grid communication operations are easier to quantify. A nearest-neighbor grid operation (such as **`from_grid_dim`** called with a distance of 1) performs on-VU movement to move the entire subgrid, and off-VU movement to move exactly one element per VU, of which one element per PN moves between PNs. The on-VU movement can theoretically be performed at a rate up to the peak memory bandwidth of the node (about 256 Mbytes/s/PN), but in practice will run slower than this except with very large subgrid sizes, because of high startup costs in grid communication. The off-PN movement occurs at the rate at which data movement between PNs can be sustained. In practice, this is about 1 to 3 Mbytes/s/PN, depending upon the subgrid size and the shape's layout.

2.2 Using Prism to Analyze Performance

This section briefly describes aspects of using Prism to analyze the performance of a C* program. For more information, see the *Prism User's Guide*.

As mentioned above, you must compile your program with the **-cmprofile** option to collect performance data. Within Prism, you run your program with collection turned on. When the program has finished execution, you can display the data.

Prism provides three levels of performance data:

- *Resources* — Prism provides data on the program's overall use of individual CM-5 resources: for example, general communication, grid communication, and PN computation (referred to as node CPU) time.

- *Functions* — Prism provides data on the use of a given resource by each function in the program. The data is available in both flat and call-graph mode. Flat mode displays each function's total use of the resource, regardless of where the function was called. Call-graph mode displays the dynamic call graph of the functions, and the use of the resource for each individual call.

- *Source lines* — Finally, Prism provides data on the use of a given resource for each source line within a function.

The data is available both as seconds (or microseconds) of elapsed time, or as a percentage of the total elapsed time.

Prism performance analysis data is most useful for determining bottlenecks in a program. Where is a program spending its time? What resource is it depending on most?

When using Prism to analyze the performance of a CM-5 C* program, you should be aware of the restrictions listed below. The restrictions will be removed in future compiler and Prism releases.

- Prism does not account for the time spent executing functions that were not compiled using **-cmprofile**. *This includes all C* library functions, in particular the functions in the communication library.*

- Prism reports only some of the time spent performing parallel I/O operations (via the CMFS interface). The time reported does not include the time spent re-ordering the parallel data for the I/O operation.

- The time attributed to scalar computation includes time spent waiting for PNs to complete operations. Because of this, the time reported as **PM cpu time** can be misleadingly large.

Figure 5 shows one view of the performance data for the sample program we used in the previous section. Note these points:

- At the top left is the resource information. The reported **PM cpu** times are the times spent performing scalar operations. As noted above, the user time is misleading, since this is mostly time spent waiting for PN operations to complete.

- At the top right is the flat-mode per-function information for the **Node cpu (user)** resource; this is the amount of CPU time on the nodes each function spent. Essentially all of the time was spent in the function **`fishcake`**.

- At the bottom of the window is the source-line data for **`fishcake`**'s use of the **Node cpu (user)** resource. This shows that all of the time for this resource is spent in the **`sum`** statement.

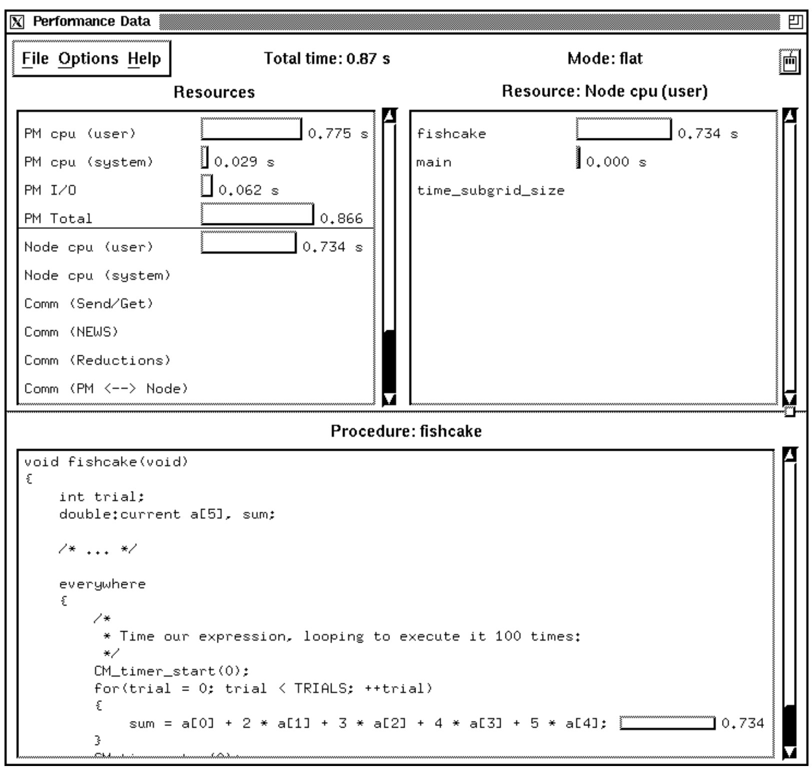

Figure 5. Performance data for a C* program.

It may be more useful to look at this program in call-graph mode. This will show the time spent in each call to fishcake. Figure 6 shows the data in call-graph mode; we use microseconds instead of seconds for measurement units.

The top-right pane displays the beginning of the call graph: **`main`** calls **`time_subgrid_size`** repeatedly. If we were to click on one of the **`time_subgrid_size`** entries, we would see its call to **`fishcake`**; all the **node CPU (user)** time attributed to **`time_subgrid_size`** is accounted for by the code in **`fishcake`**.

The source-line data at the bottom of the window in Figure 6 also shows the time allocated to each **`time_subgrid_size`** call from within **`main`**. In this case, it shows the argument to **`time_subgrid_size`** that resulted in the different times.

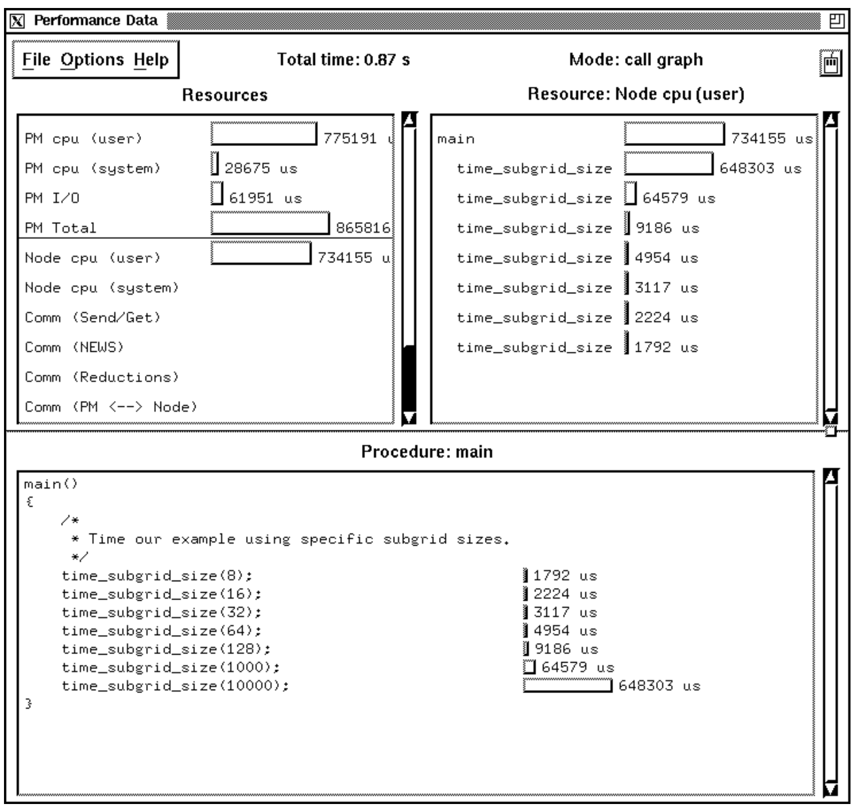

Figure 6. Call-graph display of performance data.

One note about the source-line data: Prism actually calculates usage per PN code block, not per source line. If there are multiple source lines in the code block, Prism divides the usage equally among these source lines for display purposes. These equal values for source lines can be an indication that the source lines belong to the same code block.

Chapter 3

General Performance Tips

This chapter contains some general advice about increasing the performance of your program.

3.1 Think about How to Map Your Problem onto the Machine

Before you even begin coding, it is important to consider the best way to map your problem onto the CM-5. Keep in mind these major points:

- *Communication is expensive.* Lay out your data to avoid communication if you can. If you have to do communication, use grid communication or scan operations rather than general communication if possible. See Chapter 5 for more information.

- *Keep subgrid sizes sufficiently large.* This amortizes the startup cost for executing a PN code block. See Section 1.3 for more information.

- *Avoid parallel computation or communication when few positions are active.* It is inefficient to perform parallel computation or communication when only a few positions are active. See Section 4.3. In particular, avoid situations where a program slowly iterates until no positions are active; see Section 5.1.7.

3.2 Prototype Your Algorithm and Measure Performance from the Start

It is useful at the start of coding to write a small prototype of the most performance-critical part of your program and measure its performance. Even better, try several different prototypes and time each. This may uncover performance problems before you become too committed to a particular approach. You may then be able to revise your algorithm without having to revise an enormous amount of code.

3.3 Write Scalar Code in C, not C*

If you have large chunks of code that are entirely scalar, put them in a `.c` file and use your C compiler. The C* compiler is targeted for the efficient compilation of parallel code; although the C* compiler handles scalar code, a C compiler will probably do a more efficient job.

Chapter 4

Performance Tips for Parallel Computation

The performance tips in this chapter apply specifically to optimizing parallel computation. They are discussed roughly in the order of their importance.

4.1 Avoid Parallel Computation that Uses Small Integers

The CM-5 vector units do not directly support vector load and store operations of 1-byte and 2-byte quantities. The C* compiler supports parallel **`bool`**, **`char`**, and **`short`** data types, but loading and storing these parallel values to and from memory is (very roughly) 10 times as expensive as loading and storing other data types. (Note that parallel **`bool`** values in the CM-5 compiler are stored in memory as 1-byte quantities.)

It is acceptable to use parallel **`bool`**, **`char`**, and **`short`** types to conserve memory, or in areas of the program where performance is not critical. Often it is simple to rewrite code using **`int`** temporaries so that small integer values are not directly loaded or stored in performance-critical sections.

Another useful strategy is to write parallel code that stores several small integer values into a 4-byte **`int`** type by using shift and bitwise logical mask operations.

4.2 Avoid Contextualization

Context manipulation through use of the **where** statement is often an expensive operation in the current version of the CM-5 C* compiler. The operation that the compiler emits to compute and update its notion of context can be surprisingly inefficient. You should avoid the use of **where** statements in sections of code where performance is critical. (We expect that future versions of the C* compiler will address many of the performance problems with **where** operations.)

A **where** statement not only breaks code blocks for its body; it also always emits a separate code block that computes the context condition and updates the current context. And the process used to store the new context is itself slow.

Consider this code fragment:

```
int:current a, b, c;
/* ... */

a = b * 11 + c;
where(a > 3)
    b *= c;
```

Compiling this code emits three PN code blocks. The first performs the statement

```
a = b * 11 + c;
```

The second computes the new context for

```
where(a > 3)
```

And the third computes

```
b *= c;
```

The code block that computes the context is nontrivial. Because the context is stored in a special packed format, the code block has to load and unpack the original context from memory, compute the new context, and store it back into the packed format in memory.

However, there is a simple way we can rewrite this code to avoid all of this overhead. Taking advantage of the fact that the expression **a > 3** evaluates to 1 in the positions where the condition is true and 0 elsewhere, we can replace:

```
where(a > 3)
    b *= c;
```

with the equivalent statement:

```
b *= c * (a > 3);
```

This not only allows us to avoid the **where** statement, it also allows this expression to be merged into the same code block as the previous expression. The compiler generates a single code block to compute:

```
a = b * 11 + c;
b *= c * (a > 3);
```

The resulting code is much faster. In this case, the subgrid loop body executes about 60% faster than the ones generated using the **where** statement, and the startup overhead of the single subgrid loop is about one quarter of what the **where** statement generates.

4.2.1 Use everywhere to Avoid Context Overhead

The C* compiler generally assumes that it must perform all computation using the current context. This assumption causes extra code to be introduced into every code block; this code is used to load the context values from memory. The overhead adds to the code block's startup and loop costs, but does not increase with the amount of code in the code block. Even in cases where you might expect the compiler to be able to detect that contextualization is not necessary, this overhead is usually present.

The use of an **everywhere** statement around parallel operations where the context does not matter usually prevents the compiler from emitting code that performs contextualization in PN code blocks. However, since the contextualization overhead does not increase with the size of the code block, the additional overhead of contextualization is usually less important in code blocks where a substantial amount of computation is performed.

4.2.2 Avoid Contextualization through &&, ||, and ?: Operators

The **&&**, **||**, and **?:** operators perform contextualization when used with parallel operands, and add the same sorts of overhead that the **where** expression produces. See the *C* Programming Guide* for a fuller discussion of these operators.

It is often easy to rewrite expressions that use these operators so that they don't perform contextualization. For example, you can rewrite expressions that use the **&&** and **||** operators by substituting the bitwise logical **|** and **&** operators when the operands are known to be values that are 1 or 0.

Consider the code fragment:

```
int:current a, b, c, d;
a = (b > c) && (d < c);
```

The statement produces three code blocks, and has the same overhead problems that we discussed in the **where** example in Section 4.2. In this case, we can rewrite the expression equivalently as:

```
a = (b > c) & (d < c);
```

This creates only one code block, which is much more efficient.

In cases where an operand of **&&** or **||** is not known to be 1 or 0, we can introduce a test for nonzero; for example, we can replace:

```
a = b && (d < c);
```

with:

```
a = (b != 0) & (d < c);
```

4.2.3 Don't Use Context to Create Masks

Don't use context to create masks. For example, don't write:

```
where(pcoord(0) == 0)
{
    border = 1;
}
else
{
    border = 0;
}
```

or:

```
border = (pcoord(0) == 0) ? 1 : 0;
```

Both of these perform contextualization. Instead, write:

```
border = (pcoord(0) == 0);
```

4.3 Inactive Positions Do Not Increase Computation Performance

The time required to execute parallel computation is the same regardless of how many positions are active.

In the vector-unit computation model used by the C* compiler, context is implemented by masking vector store operations so that they store only in active positions. The same code is always executed on the PNs regardless of how many positions are active.

Note that this also means that performing computations in a shape where relatively few positions are active can be inefficient, since it takes the same amount of time as performing the computation on all positions in the shape. See Section 5.1.7 for a possible way of improving performance under these conditions.

4.4 Rearrange Code to Form Fewer Code Blocks

Sometimes a simple rearrangement of code around statements that perform flow control increases performance by allowing PN code blocks to be merged.

For example, suppose we have parallel statements both inside and outside the body of an **`if`** statement:

```
int:current a, b, c, d;
int x;

/* ... */

a = b * 17 + c;
if(x > 2)
{
    d *= a;
}
```

```
else
{
    d -= a;
}
```

In this case, the compiler generates three code blocks, the first for the statement

```
a = b * 17 + c;
```

and one for each branch of the **`if`** statement. Two code blocks are executed, since only one branch of the **`if`** is taken. We can rewrite this code to merge the first statement into each of the branches:

```
if(x > 2)
{
    a = b * 17 + c;
    d *= a;
}
else
{
    a = b * 17 + c;
    d -= a;
}
```

The code is equivalent, but only one code block is executed instead of two.

Since communication breaks code blocks, it is advantageous to rearrange code so that communication operations do not unnecessarily interrupt computation. Consider this example:

```
double:current a, b, c, d, e;

/* ... */

a += b * 17.2;
c = [(. + 1) %% dimof(current, 0)]a;
d = a / b;
e = [(. - 1) %% dimof(current, 0)]d;
```

This generates two code blocks. The first performs

```
a += b * 17.2;
```

and the second performs

```
d = a / b;
```

Because the second expression does not depend upon the previous communication operation, we can easily rearrange this code to be:

```
a += b * 17.2;
d = a / b;
c = [(. + 1) %% dimof(current, 0)]a;
e = [(. - 1) %% dimof(current, 0)]d;
```

This allows the compiler to perform both computation statements in a single code block.

4.5 Lengthen Code Blocks by Unrolling Loops

Loop constructs break code blocks by performing flow control, as discussed in Section 1.5.1. This often hinders performance of code that performs a parallel operation in a loop.

Consider this example, which sums the elements of a parallel array:

```
int i;
int:current a[5], sum;
/* ... */
everywhere
{
    sum = 0;
    for(i = 0; i < 5; ++i)
    {
        sum += a[i] * (i + 1);
    }
}
```

Although this is a clean way to express the algorithm, the body of the **for** loop will be in a separate code block; this means that executing this code will execute one small PN code block to initialize **sum**, and then another small PN code block five times, once for each loop iteration. We incur the overhead of six invocations of a code block. Furthermore, each loop iteration stores a result to **sum**, and we would like to be able to avoid those stores, particularly since they are 4-byte stores that are particularly expensive. See Section 4.7.

The solution is to unroll our **`for`** loop and write a single expression that computes **`sum`**:

```
everywhere
{
    sum = a[0] + 2 * a[1] + 3 * a[2] + 4 * a[3] +
          5 * a[4];
}
```

The revised expression runs almost four times as fast as our original code.

Unrolling loops like the one above is a very useful strategy for improving code performance. Usually unrolling a loop like this just a few times will suffice to gain most of the performance benefit. Even when the process of loop unrolling is more difficult than it is in our example, it will often be worth the effort.

4.6 Avoid Breaking Computation into Small Statements

It is common style among C programmers to break up large expressions into smaller parts, sometimes using temporary variables to store intermediate results. Unfortunately, this strategy can hurt performance with the current version of the CM-5 C* compiler. For every assignment that is performed, the C* compiler emits code that stores values to memory. In some cases you would expect the compiler to simply reuse a value we assigned previously (without actually storing the value to memory), but the compiler currently does not do this.

For example, consider the code:

```
double:current a, b, c, d, e, f, f3;

everywhere
{
    f3 = f * f * f;
    a = b * 17;
    a += c * d * 3 / f3;
    a *= e + f3;
}
```

This generates a single PN code block that is reasonably efficient but not optimal, using 14 vector operations in the body of the subgrid loop. Each statement results in a **`dustorev`** instruction that stores the result to memory; these values are reloaded the next time they are used. These stores and loads are often unnecessary, but the compiler does not detect this.

When using **float** or **int** data types, the incentive to eliminate extra stores is much greater, since stores of 4-byte parallel data types are more expensive than 8-byte stores; see Section 4.7.

Our first attempt at rewriting this code computes **a** with two expressions instead of four:

```
everywhere
{
    f3 = f * f * f;
    a = (b * 17 + c * d * 3 / f3) * (e + f3);
}
```

The resulting code has 11 vector operations in the subgrid loop body instead of 14. But we can do even better than this. We've written the **f3** temporary so that we don't have to write that computation twice, but it turns out that the compiler is smart enough to reuse its value if we do simply write it twice in one expression. And by doing that, we eliminate an extra load and store. So we write:

```
everywhere
{
    a = (b * 17 + c * d * 3 / (f * f * f)) *
         (e + (f * f * f));
}
```

and the compiler generates a code block that performs the calculation in 9 vector operations:

```
L2$_CMPE_meatcake2_0:
!   a = (b * 17 + c * d * 3 / (f * f * f)) * (e + (f * f
         * f));
    dfmulv V6, V6, V8; duloadv [%i5 + %o2]:8, V6;
    dfmulv V8, V6, V8; duloadv [%i3 + %o2]:8, V2;
    dfmulv V2, V4, V2; duloadv [%i4 + %o2]:8, V4;
    dfmulv S2:0, V2, V2; duloadv [%g7 + %o2]:8, V12;
    dfdivv V2, V8, V2; memnop;
    dfmadtv S4:0, V10, V2, V10; duloadv [%i2 + %o2]:8,
    V10;
    dfaddv V12, V8, V12; memnop;
    dfmulv V10, V12, V10; memnop;
    fnopv; dustorev V10, [%o1 + %o2]:8;
    add %o2, 64, %o2
    subcc %i0, 8, %i0
```

```
bnz L2$_CMPE_meatcake2_0
nop
```

4.7 Avoid Assigning to Parallel ints and floats

The CM-5 vector units perform vector stores of 4-byte data much more slowly than they perform stores of 8-byte data, or loads of either 4-byte or 8-byte data. (Four-byte stores take 3.5 times as long as the other operations.) This means that assigning to parallel **`int`** and **`float`** data types is disproportionately expensive. The actual performance impact will depend on how many stores the code is performing relative to other operations.

Aside from the cost of store operations, **`float`** and **`double`** parallel data types perform computation more or less equally fast. Thus, it may be useful to use parallel **`double`** types instead of **`float`** types.

4.8 Use Explicit float Constants to Avoid Unnecessary Conversions

ANSI C specifies that floating-point constants by default have **`double`** type. When such a constant is used in an expression, it can cause unwanted type conversions to a **`double`** type, which can add extra overhead to the code. Writing the constant with a trailing **`f`** is C syntax that forces it to have a **`float`** type. This can eliminate the extra conversions.

For example, suppose we are compiling the code:

```
int x;
int:current a;
float:current b;

/* ... */

everywhere
    b = b * 17.2 + a * x;
```

Although **b** has a **float** type, multiplying it by 17.2 (which has **double** type) causes it to be converted to a **double** type; furthermore, the result must then be converted back to a **float** type when it is stored. We see this in the subgrid loop body produced by this code:

```
L2$_CMPE_fishcake_0:
    imulv S2:0, V2, V2; uloadv [%i3 + %g7]:4, V2;
    itodfv V2, V4; uloadv [%i2 + %g7]:4, V6;
    ftodfv V6, V8; memnop;
    dfmadtv V8, S4:0, V4, V8; memnop;
    dftofv V8, V8; memnop;
    fnopv; ustorev V8, [%i2 + %g7]:4;
    add %g7, 32, %g7
    subcc %i0, 8, %i0
    bnz L2$_CMPE_fishcake_0
    nop
```

The **ftodfv** instruction converts **b** to a **double** type before multiplying it by the constant. The **dftofv** instruction converts the result of the expression to a **float** type before storing it in **b**.

In this case we can correct the problem by writing **17.2f** instead of **17.2**:

```
everywhere
    b = b * 17.2f + a * x;
```

The code produced for this statement is simpler:

```
L2$_CMPE_cheesecake_0:
    imulv S2:0, V2, V2; uloadv [%i3 + %g6]:4, V2;
    itofv V2, V2; memnop;
    fmadtv S4:0, V4, V2, V4; uloadv [%i2 + %g6]:4, V4;
    fnopv; ustorev V4, [%i2 + %g6]:4;
    add %g6, 32, %g6
    subcc %i0, 8, %i0
    bnz L2$_CMPE_cheesecake_0
    nop
```

The number of vector instructions in the second case is reduced from six to four.

4.9 Prototype Functions

As with most compilers for Standard C, using ANSI function prototyping speeds up a program by reducing the number of conversions. For example, a call to an unprototyped function with a **`char`** argument will promote that argument to an **`int`**. The called function must then convert the **`int`** back to a **`char`**.

This applies to functions passing parallel arguments as well. Prototyping functions can avoid extra code blocks and extra computation within code blocks.

In general, using prototyping is good programming practice. By declaring prototyped functions before they are called, you can avoid many common argument-type mismatch errors.

4.10 Avoid Passing Parallel Arguments by Value

In function calls, passing a parallel argument by value requires the compiler to use a temporary in which to store the value, and incurs the overhead of copying the parallel value into this temporary. Where this is a problem, you can avoid the overhead by writing functions that pass by reference instead; you do this by passing a pointer to a parallel variable rather than the parallel value itself.

Under certain circumstances, you can avoid the overhead of passing a parallel variable by value without passing it by reference; you do this by declaring the parallel variable as a **`const`** in the function definition. For example:

```
void foo (const int:current x)
{
    int sum;
    /*...*/
    sum = += x;
    printf ("%d\n", sum)
}
```

The caller doesn't have to create a temporary for the parallel argument in this case, if all of these points are true:

- The function is prototyped.
- The parallel argument is passed by value.

- The argument is declared as **`const`** in the prototype.
- The caller of the function passes a simple variable of the same type (that is, it doesn't pass a structure member, an array element, or a dereferenced pointer).
- The variable passed to the function is local (that is, not global or file static).
- The variable's address is never taken.

If the constraints in the callee are not met, the program will still work properly, but will incur the ordinary pass-by-value overhead.

4.11 Avoid Unnecessary Calls to Parallel Library Functions

If possible, avoid calls to simple parallel library functions. For example **`abs`** and **`fabs`** incur the standard function call overheads: they break code blocks, use parallel temps for arguments and return values, and generate code to assign to those temps.

Thus, instead of writing:

```
float:current a,b;
a = fabs(b);
```

we can write:

```
a = b >? -b;
```

This takes advantage of the C* binary maximum operator to compute the absolute value.

In other cases, it may be possible to rewrite a program to reduce the number of calls to such functions, even if they cannot be altogether eliminated.

A future version of the compiler may remove the performance penalty for calling some of these parallel library functions.

4.12 Parallel Array Indexing and Table Lookup Functions

Indexing a parallel array with a parallel variable is much more efficient with CM-5 C* than it is with CM-200 C*, since these operations are supported by the CM-5 vector unit hardware. This provides a very powerful and useful programming construct for C* programmers. However, parallel array references are about two to four times slower than ordinary array references (indexing a parallel array with a scalar variable).

The shared table lookup utility (described in Appendix D of the *C* Programming Guide*, May 1993 edition) provides an alternative for a specific case of array referencing through the use of **`CMC_lookup_shared_table`**. It is not possible to use a parallel index with a scalar array, and when you want to do this sort of operation, you must either replicate the entire scalar array in a parallel array in the current shape or use the shared table lookup utility.

The advantage of using the shared table lookup utility for this operation is that a smaller amount of memory is used to store the table in parallel memory. (Only one copy of the table is made in each VU, rather than one copy per subgrid element in each VU.) However, shared table lookups are roughly two to four times as slow as parallel array references.

Any array references or shared table lookup operations that involve data types whose sizes are not multiples of four bytes are significantly slower, just as with ordinary load and store operations; see Section 4.1.

The overloading of **`CMC_lookup_shared_table`** that takes an element size is much slower than the other overloadings in C* Version 7.1. This will be fixed in a later release of C*. For now, avoid using this function when its performance is important.

Chapter 5

Performance Tips for Parallel Communication

The performance tips in this chapter apply specifically to parallel communication. In general, the advice applies whether the communication is expressed in the syntax of the language, or via functions in the communication library.

5.1 Tips for Increasing General Communication Performance

5.1.1 Use Send Operations instead of Gets

If possible, write your program to use a send operation — for example:

```
int:current index, dest, source;
[index]dest = source;
```

instead of a get operation — for example:

```
dest = [index]source;
```

The CM-5 hardware does not directly implement get operations; instead, the runtime system performs a send for the request and a send for the reply. Therefore, a get operation is roughly twice as expensive as a send.

5.1.2 Use Send and Get Patterns that Avoid Excessive Collisions

Although send and get performance can depend upon particular routes taken through the data router (such things are beyond the control of C* programmers), the most common problems with send and get performance are caused by patterns of communication that involve excessive collisions. A collision occurs when more than one position sends to, or gets from, a single position. The basic rule is therefore: *avoid excessive collisions*. Thus:

- Avoid sending data to relatively few positions of a destination.
- Avoid getting data from relatively few positions of a source.

This rule applies even if you use a combiner in your send or get function. In all cases, even with a combiner, the nodes receive the data serially; no combining takes place in the network. This means that performance is bound by the maximum amount of data going to a single node.

A good pattern is therefore one in which the data is evenly distributed among the destinations of a send or the sources of a get, with no one position receiving much more data than any other.

The best way to handle the effect of collisions on performance is to come up with an algorithm that avoids the collisions in the first place. If that isn't possible, try reducing the values going to a position to a single value, then send the single value. The reduction may be less expensive than the cost of the collisions. Here is a method for doing this (for a send):

1. Sort the source positions by the position to which they are sending.
2. Identify the groups of positions that are sending to the same destination positions and execute a **`scan`** or similar function to combine the values in each group.
3. Send the resulting value for each group to its destination position.

You can follow a similar procedure for a get.

Below are two routines that use the **`rank`** function in this manner to combine data before sending it. In practice routines like these help only if there are many collisions, since **`rank`** is an expensive function. However, if you can arrange your data so that it is sorted (or mostly sorted) and use scan operations to copy data, you can do similar things without as much overhead. Note also that when possible you should use CMSSL's support for fast rank operations, as described in Section 5.2.4, to achieve the maximum performance for this function.

The routine **sorted_send**, below, implements the equivalent of

```
[index]dest = source;
```

but sorts the data to avoid collisions.

```
#include <stdlib.h>
#include <cscomm.h>

int:current sorted_send(int:current dest, int:current source,
                        int:current index)
{
    int:current index_rank;
    int:current sorted_source, sorted_index;

    /*
     * Rank the index values.
     */
    index_rank = rank(index, 0, CMC_upward, CMC_none, CMC_no_field);

    /*
     * Use the rank to sort the source and index by the index values.
     */
    [index_rank]sorted_source = source;
    [index_rank]sorted_index = index;

    /*
     * Perform the send operation only once for each unique rank value.
     */
    where(sorted_index != from_torus(&sorted_index, 1))
    {
        [sorted_index]dest = sorted_source;
    }

    return dest;
}
```

The function **sorted_combining_send**, shown below, uses a similar method, but implements a combining send, equivalent to

```
[index]dest += source;
```

To do this, we sort the data, identify segments that have identical destinations, and use a scan operation to combine data in each segment before sending it.

```
int:current sorted_combining_send(int:current dest, int:current source,
                                  int:current index)
{
    int:current index_rank;
    int:current sorted_source, sorted_index;
```

```
    bool:current segment;
    int:current combined_source;

    /*
     * Rank the index values:
     */
    index_rank = rank(index, 0, CMC_upward, CMC_none, CMC_no_field);

    /*
     * Use the rank to sort the source and index by the index values:
     */
    [index_rank]sorted_source = source;
    [index_rank]sorted_index = index;

    /*
     * Identify the segments that have identical indices:
     */
    segment = (sorted_index != from_torus(&sorted_index, -1));

    /*
     * Combine the sorted source data in each segment using a scan
     * operation:
     */
    combined_source = scan(sorted_source, 0,
                           CMC_combiner_add, CMC_downward,
                           CMC_segment_bit,
                           &segment, CMC_inclusive);

    /*
     * Send only the combined values to the destination:
     */
    where(segment)
    {
        [sorted_index]dest += combined_source;
    }

    return dest;
}
```

5.1.3 Inactive Elements Can Increase Send and Get Performance

Although inactive elements do not increase computation performance (see Section 4.3), they can increase the performance of send and get operations somewhat. In the CM-5, when fewer messages need to be sent through the network, the send or get operation can complete more quickly. The relationship is not proportional, however: for example, if half your positions are inactive, send and get time is not cut in half. By reducing the number of active elements, it is possible to save up to about three-fourths of a send cost or two-thirds of a get cost.

5.1.4 Use Get Operations to Get Data from a Much Larger Shape

The time spent performing a send or get operation depends both upon the amount of data being sent or gotten (which depends upon how many positions are active) and upon the total number of positions (active or inactive) in the current shape. When the source shape is much larger than the destination shape, it can be faster to use a get operation to get selected elements from the larger shape than it is to send the selected elements.

Consider the following example. We have variables in a 2-dimensional shape and a 1-dimensional shape, and we want to extract a row from the 2-dimensional variable and assign the values to the 1-dimensional variable. We could implement this with a send as follows:

```
shape [1000][1000]s1;
shape [1000]s2;

int:s1 m;
int:s2 v;

with(s1)
{
    /* Send column 3 of m to v: */
    where(pcoord(1) == 3)
    {
         [pcoord(0)]v = m;
    }
}
```

In this case, the time required to perform the send is dominated by the number of positions in the current shape (**s1**), even though most of those positions are inactive. Suppose we implement the same thing using a get operation instead:

```
with(s2)
{
    /* Get column 3 from m: */
    v = [pcoord(0)][3]m;
}
```

In this case, the current shape is much smaller, and the time required to do the operation is not dominated by the large current shape. The get operation here is much more efficient.

The second case is also more memory-efficient, since the compiler-generated temporaries used for send address calculations (in the current shape) are smaller.

An exception to this rule arises when the data being sent is not a data type that is a multiple of four bytes. In this case, as discussed in Section 5.1.5, the compiler generates temporaries for both the source and destination of a send or a get. In either the send or get case, we incur the overhead of the manipulation in the larger shape.

5.1.5 Avoid Communication Operations on Data that Is Not a Multiple of Four Bytes

All communication operations require extra overhead if they involve data that is not a multiple of four bytes. This overhead occurs because the compiler must perform all communication operations with word-sized data. When data is not a multiple of four bytes, the data is copied to word-sized temporaries, operated upon, and copied back. This adds memory overhead and costs additional time. You should avoid communication involving parallel **`short`**, **`char`**, and **`bool`** types to avoid this additional overhead. For example:

- Simply declare the parallel variable as an **`int`** in the first place.
- Create your own word-sized temporaries around several communication operations, to avoid creating temporaries for each operation.

Similarly, communication involving structures that are multiples of four bytes is less expensive than communication involving structures that are not multiples of four bytes.

5.1.6 Package Your Data into Structures to Avoid Extra Communication Operations

When performing identical communication operations on several different parallel variables, it is often more efficient to package up these parallel variables into a single structure, allowing a single communication operation to be performed for all of the data.

For example, in the code below, the parallel variables have the same send pattern:

```
int:current a, b, c, d;
int:current i;

/* ...initialize data... */

/* send c to a, and d to be using identical
   coordinates */

[i]a = c;
[i]b = d;
```

We can rewrite this code as follows to assign the parallel variables to structure members, thereby saving a send operation:

```
typedef struct {int x, y} pair;

pair:current e, g;

/* package the source and dest into a structure: */

e.x = a;
e.y = b;
g.x = c;
g.y = d;

/* send the entire structure: */

[i]e = g;

/* unpack the result: */

a = e.x;
b = e.y;
```

In this case, the cost of packing and unpacking the parallel variables is less than the cost of the extra send.

5.1.7 Repack Your Shape When Too Many Positions Are Inactive

As we discussed in Section 4.3, the cost of performing parallel computation in a shape is the same, regardless of how many positions are active. This means that performing the computation when relatively few positions are active can be inefficient. To avoid this, it may be more efficient to repack your data into a new shape containing only the active positions.

The benefits of this strategy depend upon how many positions are inactive, how much computation is being performed on the active positions, and how easily the data can be moved to a smaller shape. The time saved by performing the computation in a smaller shape must be greater than that required to move the data between shapes.

For example, it is sometimes more efficient to do this:

1. Identify the active positions.
2. Create a smaller shape containing only these positions.
3. Send the data to this shape.
4. Perform the computation there.
5. Send the resulting data back.

In some cases, the cost of the two sends is less than the cost of operating on the inactive elements.

We provide two examples. In both cases, the function **`foo`** does a nontrivial parallel computation. (It's not important what that computation is.) It is passed and it returns parallel values in the current shape. In both cases, we are calling **`foo`** to perform computation on a small fraction of the data in a large shape. The examples demonstrate more efficient ways to do this computation by sending the active data to a smaller shape.

The first example shows an easy case, where you know when writing the code which positions are going to be active. In this case, we create a smaller shape, do a get to fetch the data into the smaller shape, perform the computation, and send the result back.

```
#include <stdlib.h>
#include <math.h>

int:current foo(int:current a)
{
    double:current x, y;

    x = prand();
    y = log(x) + a;
    return ceil(y / 17.2);
}
```

```
main()
{
    shape [1000][1000]s1;
    shape [1000]s2;
    int:s1 a, b;

    with(s1)
    {
        a = pcoord(1);

        /* Compute only on a small section of the shape: */
        where(pcoord(0) == 3)
        {
            b = foo(a);
        }
    }


    with(s2)
    {
        int:current a2, b2;

        /* Instead, move the active data to a variable in a different
           shape, perform the computation, and move it back. */

        a2 = [3][pcoord(0)]a;
        b2 = foo(a2);
        [3][pcoord(0)]b = b2;
    }
}
```

A more difficult situation arises when you cannot easily express the active elements in the shape — that is, they cannot be known until run time. In that case, you can:

1. Count the active positions.
2. Create a smaller shape with enough positions to hold all of the active ones.
3. Enumerate the active positions to determine where they will be sent to.
4. Get the data to the smaller shape.
5. Perform the computation in the smaller shape.
6. Send the results back.

It's the same idea, only a little more complicated.

In the example below, the variable **e** is used to compute the coordinates of the data in **s2**. Then we send the source coordinates to a variable in **s2** and use this to fetch the data we need from **s1** to **s2**. We then perform the computation in the smaller shape and send the results back.

```
#include <cscomm.h>
#include <stdlib.h>
#include <math.h>

int:current foo(int:current a)
{
    double:current x, y;

    x = prand();
    y = log(x) + a;
    return ceil(y / 17.2);
}

main()
{
    shape [1000000]s1;
    int:s1 a, b, c;
    int:s1 e;
    int num_active;

    with(s1)
    {
        a = pcoord(1);
        c = prand() % 1000;

        /* Compute only on a small section of the shape: */
        where(c == 3)
        {
            b = foo(a);
        }

        /* Send the active data to a smaller shape to perform the
           computation: */
        where(c == 3)
        {
            /* Count the active elements, and enumerate them to determine
             their position in the smaller shape. */

            num_active = += (int:current) 1;
            e = enumerate(0, CMC_upward, CMC_exclusive,
                          CMC_none, CMC_no_field);

            {
                shape [num_active]s2;
```

```
                int:s2 a2, b2, coord;

                /* Send the source coordinates to a variable in the
                   smaller shape. */

                [e]coord = pcoord(0);

                /* Use those source coordinates to fetch the data, then
                   perform the computation and send it back. */

                with(s2)
                {
                    a2 = [coord]a;
                    b2 = foo(a2);
                    [coord]b = b2;
                }
            }
        }
    }
}
```

5.1.8 Use the Aggregate Versions of General Communication Functions

The overloadings of the general communication functions **get**, **send**, **write_to_pvar**, and **read_from_pvar** for aggregate data types (that is, those that take a pointer to a parallel variable of any length as an argument) are sometimes faster than the overloadings for simple data types, because they don't require the argument-passing overhead of the latter. (Note, however, that the aggregate versions may not be faster in all cases.) When performance of these functions is critical, you may want to time both versions to determine which is faster for your application.

5.1.9 Using collision_mode Doesn't Increase Performance

Version 7.1 of the CM-5 C* compiler ignores the **collision_mode** argument to the **get** function. Thus, you can't improve performance via this argument.

5.2 Tips for Increasing Grid Communication Performance

As we noted at the beginning of the chapter, the advice here applies to grid communication expressed either via syntax or via functions in the communication library.

5.2.1 Use Torus Rather Than Grid Functions

The torus functions are much faster than the grid functions in the current version of the compiler. This applies to all variations of these functions:

```
from_torus
from_torus_dim
to_torus
to_torus_dim
from_grid
from_grid_dim
to_grid
to_grid_dim
```

5.2.2 Use from_ Rather Than to_ Functions

The **`from_torus`**, **`from_torus_dim`**, **`from_grid`**, and **`from_grid_dim`** functions are much faster in the current version of the compiler than the corresponding **`to_torus`** and **`to_grid`** functions, except when the functions appear in an **`everywhere`** block; in that case, their speed is about the same.

Note that this rule is different from the rule for send and get operations (see Section 5.1.1). The torus functions that have get-like semantics are faster than those that have send-like semantics.

5.2.3 Use the Aggregate Versions of Grid Communication Functions

As with general communication functions, the overloadings of grid communication functions for aggregate data types (that is, those that take a pointer to a parallel variable of any length as an argument) are sometimes faster than the versions for simple data types, because they don't require the argument-passing overhead of the latter. (Note, however, that the aggregate versions may not be faster in all cases.) When performance of these functions is critical, you may want to time both versions to determine which is faster for your application.

5.2.4 Use the CMSSL Version of the rank Function

The CMSSL library provides support for performing rank operations as much as 15 times faster than ordinary C*. CM-5 C* users can take advantage of this simply by calling the C* **rank** function and linking their programs specially so that the CMSSL support is used.

To use the CMSSL rank support, you must:

- Use a system on which the CMSSL library is installed.
- Compile your C* program for execution on the CM-5 using the vector units (that is, with the **-vu** switch).
- Add the following options to the command that links your program:

```
-lcmssl_dash_opt_sp -Zcmld "-u _CMCOM_u_rank"
```

5.2.5 Performing Diagonal Moves in a Single Function Doesn't Save Time

Although it may appear to be more efficient to move data along more than one dimension in a single function call, the current compiler implementation does not make this any more efficient than moving the data along one dimension at a time in multiple function calls. Thus,

```
dest = from_grid(&source, fill, -1, 2);
```

is about as fast as

```
dest = from_grid_dim(&source, fill, 0, -1);
dest = from_grid_dim(&source, fill, 1, 2);
```

This can be important when considering the costs of various communication patterns.

See the next section for an application of this point to convolution operations.

5.2.6 Consider Communication Patterns when Doing Convolution Operations

The example below demonstrates three ways of computing the average of the values of each point in a 2-dimensional grid and its eight nearest neighbors. The most straightforward method is also the most inefficient; this points out the need to carefully consider the costs involved when performing such operations.

In the first method, we use eight **`from_torus`** operations to fetch a value from each of the eight neighbors, as shown in Figure 7.

The example computes **`smooth_image`** as the unweighted average of each position of **`image`** and its eight nearest neighbors.

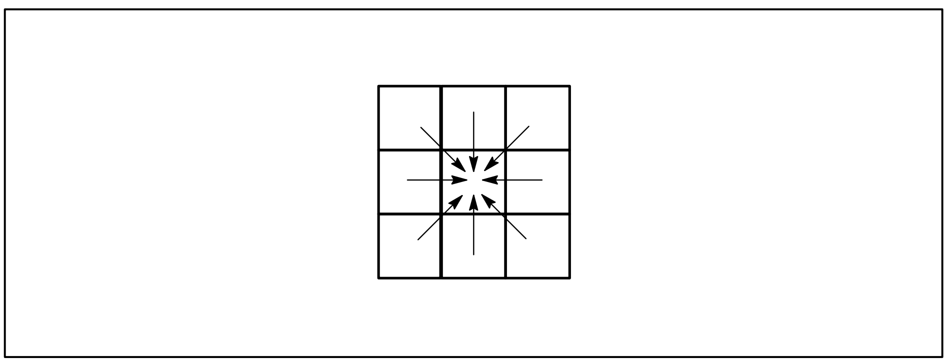

Figure 7. A straightforward convolution operation.

```
float:current image, smooth_image;

smooth_image = (image +
          from_torus(&image, -1, -1) +
          from_torus(&image, -1, 0) +
          from_torus(&image, -1, 1) +
          from_torus(&image, 0, -1) +
          from_torus(&image, 0, 1) +
          from_torus(&image, 1, -1) +
          from_torus(&image, 1, 0) +
          from_torus(&image, 1, 1)) / 9.0;
```

Note that four of the **from_torus** operations are along diagonals. As discussed in Section 5.2.5, these are each the equivalent of two **from_torus_dim** operations. Including the four horizontal and vertical **from_torus**, calls, the entire convolution requires the equivalent of 12 **from_torus_dim** operations.

A faster method — one that avoids doing the **from_torus** calls along the diagonals — involves accumulating the results at each neighbor in turn before sending them to the central point. See Figure 8. This still involves eight **from_torus** operations, but it is the equivalent of only eight **from_torus_dim** operations.

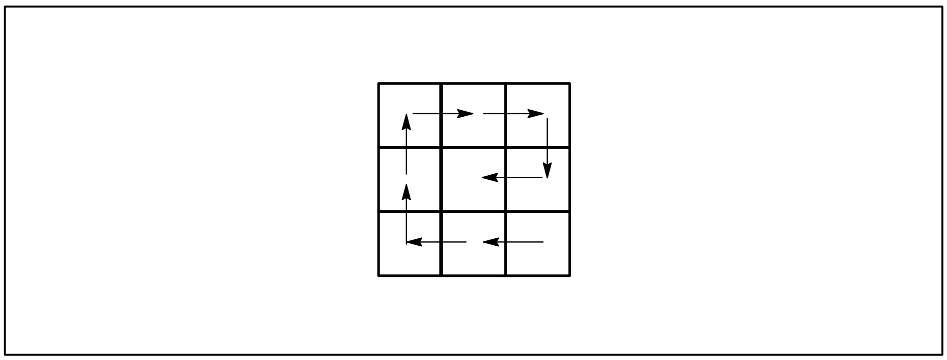

Figure 8. A faster convolution operation.

```
float:current image, smooth_image;

smooth_image = from_torus(&image, 1, 0) + image;
smooth_image = from_torus(&smooth_image, 1, 0) + image;
smooth_image = from_torus(&smooth_image, 0, 1) + image;
smooth_image = from_torus(&smooth_image, 0, 1) + image;
smooth_image = from_torus(&smooth_image, -1, 0) + image;
smooth_image = from_torus(&smooth_image, -1, 0) + image;
smooth_image = from_torus(&smooth_image, 0, -1) + image;
smooth_image = (from_torus(&smooth_image, 1, 0) + image) / 9.0;
```

A still faster method is first to combine values along axis 0, then to combine the results along axis 1. This requires only four **from_torus** operations. Since there is no diagonal movement involved, this is the equivalent of four **from_torus_dim** operations. This method relies on the fact that we are combining all the neighbors with equal weight. See Figure 9.

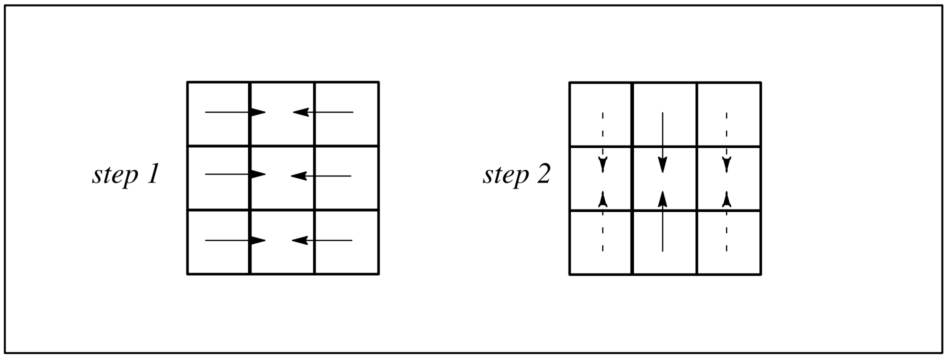

Figure 9. A still faster convolution operation.

```
float:current image, smooth_image;

smooth_image = image +
     from_torus(&image, -1, 0) +
     from_torus(&image, 1, 0);
smooth_image = (smooth_image +
     from_torus(&smooth_image, 0, -1) +
     from_torus(&smooth_image, 0, 1)) / 9.0;
```

Chapter 6

Reducing Memory Usage

This chapter explains how memory on the CM-5 partition manager and processing nodes is used by this implementation of C*, and gives some hints for reducing memory usage.

6.1 How C* Uses Memory

In the C* programming model, there are two distinct address spaces, scalar memory and parallel memory. On the CM-5, scalar memory is instantiated on the partition manager, and parallel memory is instantiated on the PNs.

6.1.1 Scalar Variables

All scalar variables (including shapes and pointers to parallel variables) take up **`sizeof(`***var***`)`** bytes of scalar memory on the partition manager, just as they would in an ordinary C program. Shapes may additionally take up parallel heap space; see Section 6.1.3. In general, the compiler uses scalar memory on the partition manager just as an ordinary C compiler uses memory in a UNIX environment. This chapter does not attempt to explain the use of scalar memory in detail.

6.1.2 Parallel Variables

Parallel variables occupy an amount of parallel memory in each vector unit that is equal to the variable's size times its shape's subgrid size. For shapes that have a large number of positions relative to the number of vector units, the subgrid size can be thought of as approximately the number of positions in the shape divided by the number of VUs. However, for smaller shapes it is important to realize that the subgrid size may be larger than this, and in particular that it is never smaller than eight. See Section 1.3 for more discussion of subgrid size.

Parallel variables also use a few bytes of scalar memory for bookkeeping.

The next sections describe the kinds of memory allocated for different kinds of parallel variables.

6.1.3 Parallel Memory Spaces

Memory on the CM-5 nodes is divided into several different memory spaces. In addition to the standard text, data, and BSS segments provided for the SPARC processors, the compiler uses two special segments of memory to implement parallel variables. These segments are the *parallel stack* and *heap*; they are allocated on the PNs such that they can be used with the vector units. This section describes how C* programs use these PN memory segments.

Parallel Stack Memory

Parallel stack memory is used by C* programs for:

- Automatic parallel variables (that is, those that are declared at function scope and are not static).
- Parallel temporaries created by the compiler. Section 6.2.3 discusses some of the situations in which parallel temporaries are introduced.

Parallel stack memory is allocated when a parallel variable is declared (or when a parallel temp is first needed by the compiler) and deallocated when the enclosing block is exited.

Parallel Heap Memory

Parallel heap memory is used by C* programs to store:

- all file-scope parallel variables
- all parallel variables declared as **`static`**
- parallel memory allocated via **`palloc`**

Parallel heap memory is also used by the current compiler for some temporaries (see Section 6.2.3). The temporaries are allocated when the compiler first needs them and freed at the end of the function. These parallel heap temporaries are reused in many circumstances to limit the number needed.

SPARC Memory Segments

Memory segments available to the SPARC processors on the PNs but not to the VUs are used for these purposes:

- to store the text and data portions of the program that reside on the PNs (this includes the VU instructions generated by the compiler)
- to store some bookkeeping information, particularly layout information that is associated with shapes
- to provide local memory for the SPARC processors; this is used by PN code blocks for scratch memory, and by internal PN functions in the run-time system when performing communication operations

Memory Used by Shapes

A shape, strictly speaking, is a scalar data object, represented with four bytes of memory. However, the C* compiler's run-time system allocates and deallocates additional information to represent the shape's layout when the program is run. This information is allocated:

- before the first use of the shape, for fully specified file-scope and static shapes
- when a shape's scope is entered, for fully specified block-scope shapes
- when **`allocate_shape`** and **`allocate_detailed_shape`** are called

The current implementation of the compiler's run-time system uses some parallel heap memory for shape allocation. In CM-5 C*, a shape is allowed to have a number of positions that is not necessarily an exact multiple of the number of VUs in the current partition. The run-time system accomplishes this by using an internal layout with axis dimensions that may be larger than those of the shape. When this happens, a *garbage mask* is constructed in heap memory on each node to represent which positions are masked out. This mask consists of one bit per subgrid position on each VU, rounded up to an integral number of words on each VU. Shapes with the same extents and layout share a single garbage mask. See the discussion of **`allocate_detailed_shape`** in the *C* Programming Guide* for more information.

In the current implementation of the run-time system, the heap space used by the garbage mask is never released. You should be aware that this can accumulate parallel memory when many different shapes are allocated. A later version of the run-time system will ensure that all parallel memory is freed when a shape is deallocated.

6.1.4 Lifetimes of Parallel Variables

To use parallel variables wisely, it is important to understand their lifetimes: when they are allocated and when they are deallocated. We discuss this with reference to the following code fragment (with line numbers added):

```
#include <stdlib.h>
#include <stdio.h>

shape [1024]S;

int:S a;
static float:S b;

main()
{
    static char:S c[10];
    double:S d;

    int:S *p;

    with(S)
    {
        int:S e;
```

```
19                static short:S f;
20
21                /* ... */
22
23                p = palloc(current, sizeof(int));
24
25                /* ... */
26
27                pfree(p);
28
29                /* ... */
30            }
31            /* ... */
32        }
```

All file-scope parallel variables (**a** and **b** in the code fragment) and all static parallel variables (**b**, **c**, and **f**) have lifetimes that are the duration of the program. These variables are allocated in parallel heap memory by the C* run-time system. Their allocation can occur at any time between program startup and when the variable is first used. The memory is deallocated when the program completes.

Automatic parallel variables (**d** and **e**) have lifetimes that extend to the end of their enclosing block. These variables are allocated in parallel stack memory and deallocated at the end of the block. Thus, **d** is deallocated at line 32, and **e** is deallocated at line 30.

Parallel data that is allocated using the **palloc** function is allocated in parallel heap memory and deallocated only when a call to **pfree** is made. Thus, the data allocated at line 23 is deallocated at line 27.

6.1.5 C* Memory and cmps Output

You can find out a program's memory usage by issuing the **cmps** command. Here is sample output from this command:

```
% cmps
32 PN System, 21440K mem. free, 4976K VU mem. free, 1 procs, TS-6/30/93-15:54
(CMOST 7.2 Beta 2) Daemon up: 15:30

USER      PID    CMPID  TIME   TEXT   ILH   ILS    IGS    IGH    VUS    VUH   COMMAND
wavin   *22214  1      0:11   384K  116K  48K    0K     4K   1088K  1044K  a.out
```

This shows the user Wavin running the command **a.out**.

For a C* program compiled with **-vu**, the **VUS** and **VUH** numbers report the number of bytes per VU used by the parallel stack and heap, respectively, by this program. The **TEXT**, **ILH**, and **ILS** numbers report the number of bytes per PN used by the SPARC PN memory segments. The **IGS** and **IGH** spaces are generally not used substantially by C* programs compiled for the vector units.

For more information, see the **cmps** man page.

6.2 Minimizing Memory Use

6.2.1 Using Parallel Variables

Every parallel variable declared in your program will actually cause parallel memory to be allocated. The compiler does not ever eliminate or overlap storage for parallel variables. Careful use of parallel variables can therefore reduce memory usage. For example:

- Declare parallel variables only when necessary.
- Reuse variables already declared but no longer needed.
- Avoid using parallel variables as temporary values in expressions when you can fold the computation into a single expression. (This helps computation performance, too. See Section 4.6.)
- Limit the lifetime of automatic parallel variables; see Section 6.1.4. Unlike scalar variables, the C* compiler deallocates parallel variables that are declared in an inner block of a function at the end of that block. Thus, you can limit the lifetimes of automatic parallel variables by introducing enclosed blocks. For example,

```
int sum;
int:current x;

/* lots of code not involving x */

foo(&x, ...);
sum = += x;
```

```
/* more code not involving x */
```

can be turned into:

```
int sum;

/* lots of code not involving x */

{
    int:current x;
    foo(&x, ...);
    sum = += x;
}

/* more code not involving x */
```

Recall, however, that introducing braces breaks code blocks; see Section 1.5.1.

- You can limit the lifetimes of global and file **`static`** parallel variables by replacing them with pointers to parallel variables and dynamically allocating the parallel memory (using **`palloc`**) only for the duration that the variables are used. Once again, see Section 6.1.4 for more information.
- Always free all heap memory as soon as it is no longer needed. A common reason for running out of memory in C and C* is dynamic memory leakage (that is, allocating heap memory without freeing it).
- Recursive functions that declare local automatic parallel variables will allocate this data once for each level of recursion performed. When this is undesirable, either avoid this kind of recursion or use **`static`** parallel variables instead, so that fewer automatics are used in the recursion.

6.2.2 Parallel Heap Fragmentation

Parallel variables are never relocated in memory after they are allocated. It is possible to accumulate free memory in the parallel heap that cannot be used by the program because the free memory blocks are too small. Consider the following scenario: there are 50 total words of memory, and the program emits a sequence of allocate/free requests:

```
p1 = alloc(10);
p2 = alloc(30);
```

```
p3 = alloc(10);
free(p1);
free(p3);
p1 = alloc(20);
```

This last allocate request should succeed, since there are 20 free words. However, the heap has been fragmented so that the free words are in blocks of 10 on either end of the still-allocated 30 words; under these circumstances, the allocation system can't return a block of 20 contiguous words because it can't relocate the memory used by **`p2`**.

It is possible (although somewhat unusual) to encounter this situation with parallel heap memory allocated with **`palloc`**. When the situation occurs, you can sometimes remedy it by rearranging the order in which variables are allocated and freed.

6.2.3 Parallel Compiler Temporaries

The rest of this section focuses on parallel temporaries that are created by the compiler. These are often the culprit when a program that should have enough memory to run doesn't in fact run. The discussion is not meant to explain all cases in which the compiler introduces temporaries, but rather to cover the common cases and explain how to avoid the temporaries in some cases.

In almost all cases, the compiler should use stack temporaries instead of heap temporaries, since they're slightly quicker to allocate and deallocate; this has a minimal effect on the performance of parallel operations. However, the compiler currently allocates some temporaries on the heap, as noted below. These temporaries are deallocated after they are used, in any case.

Within Code Blocks

The compiler generally does *not* use parallel temporaries for computing expressions within a code block. When temporaries are needed for this, they usually take the form of vector registers or scratch memory for registers, and do not occupy much memory. Preventing code from being broken into multiple code blocks can therefore help reduce parallel compiler temporaries. See Section 1.5.

Parallel Arguments and Return Values

Functions that pass parallel variables as arguments require parallel stack temporaries allocated by the calling function; these temporaries hold the values passed to the function. (You can think of these temporaries as local variables that represent the function arguments.) These temporaries live for the duration of the function call.

You can eliminate this memory overhead by passing parallel arguments by reference instead (that is, by passing pointers to parallel variables). It is also possible to eliminate this overhead in some cases by declaring parallel arguments as **`const`**. See Section 4.10.

Calling a function that returns a parallel value also causes a parallel stack temporary to be allocated to hold the return value. You can avoid this by instead writing the function to pass a pointer to a parallel value into which the result is stored.

Temporaries Introduced by a where Statement

The **`where`** statement causes these parallel stack temporaries to be allocated:

- an integer-sized temporary to evaluate the condition
- a parallel bitmask (one word per 32 subgrid elements) for storing the context

These temporaries have the lifetime of the **`where`** statement. See Section 4.2 for ways to eliminate **`where`** statements.

The **`everywhere`** statement does not allocate parallel temporaries.

Communication Temporaries

Communication operations are the main source of compiler temporaries in many programs. Each communication operation must be performed by a separate runtime call in the scalar code (see Section 1.1). This implies that statements generating multiple communication operations, such as

```
int:current a, b, i, j;
[i]a = [j]b;
```

require temporaries for intermediate results (in this case, a single temporary is needed for the result of **[j]b**). Most of these communication temporaries are allocated on the stack, but some may be allocated on the heap.

Communication operations used in computation expressions also require temporaries to store the result of the computation outside the code block. For example,

```
int:current a, b, c, i;
c = [i]a + b;
```

requires a temp for **[i]a**.

Likewise, computation expressions used in communication operations also require temporaries. For example,

```
int:current a, c, i, j;
c = [i+j]a;
```

requires a temp for **i+j**.

Note that a scalar promoted to parallel for a communication operation (for example, the 7 in **[i]a = 7;**) is a special case of such a computation expression, and also requires a parallel temporary.

Most of these temporaries for computation expressions used in communication operations are allocated on the heap. However, it is again possible to explicitly assign these values to automatic user variables so that stack memory is used instead of heap memory; this will have a minimal effect on performance.

Temporaries for Types that Aren't a Multiple of Four Bytes

The compiler's internal run-time system implements communication operations only for data sizes that are multiples of four bytes (one word) and for data that is word-aligned; see Section 5.1.5. Communication operations involving **char**, **bool**, and **short** types, and some structures and unions containing these types, require parallel temporaries. This is true for calls to communication library functions as well. These temporaries are allocated on the parallel stack. In general, using types whose size is a multiple of four bytes is both faster (since the vector units are built to support this size) and requires less temporary memory space.

Common Subexpressions

One source of parallel heap temporaries is a compiler optimization done between PN code blocks. When a given subexpression is guaranteed to have the same value in two different expressions, it is called a *common subexpression* or CSE. In the following code involving all parallel variables, **`((a + b) / (x * y))`** is a CSE:

```
int:current a, b, c, d, i, x, y;

c = ((a + b) / (x * y)) / c; /* first PN code block */
[i]c = c;                    /* communication */
d = ((a + b) / (x * y)) * d; /* second PN code block */
```

Rather than compute the CSE twice (once in each PN code block), the compiler stores the resulting value in a heap temporary in the first PN code block, then simply loads the value and reuses it in the second code block. (As mentioned above, a stack temp would be preferable, but the compiler uses a heap temp instead.)

You can obviate the need for these temporaries by making the CSEs occur in the same PN code block. In this case, exchanging the second and third lines of the example causes the CSE to exist within a single PN code block, since the communication operation has been moved out of the way (see Section 1.5.1). CSEs within a single code block take up only a single vector register or a vector-sized amount of memory, rather than a whole subgrid, on each node.

Where it is not possible to place the CSEs in a single code block, introducing an automatic variable to hold the CSE will at least cause the value to be held on the stack instead of the heap.

Appendix

Examining Generated Assembly Code

This appendix goes through the process of creating and examining the assembly code files for a simple C* program, and gives some advice about how to understand DPEAC code.

We use the same simple C* routine that we used in Chapter 1:

```
include <stdio.h>
void fishcake(int x, int:current a, float:current b)
{
    float sum;
    x = x + 2;
    printf("The value of x is: %d\n", x);
    b = b * 17.2f + a * x;
    sum = += b;
    printf("The sum of b is: %f\n", sum);
}
```

As mentioned in Chapter 1, we can instruct the compiler to leave the intermediate assembler file around by using the **-keep** option. The **-keep s** option produces SPARC assembler files for both the scalar and PN code. However, when examining the PN code, we would like to look at the DPEAC code before it is processed by **dpas** (see Section 1.2). To do this, we use the **-keep dp** option as well. Thus, the command:

```
% cs -cm5 -vu -keep s -keep dp fishcake.cs
```

causes the compiler to produce both of the assembler intermediate files, **fishcake.s** and **fishcake.pe.s**, as well as the DPEAC PN file, **fishcake.pe.dp**. We are interested in the scalar **.s** file and the PN **.pe.dp** file.

A.1 Examining the Generated Scalar Code

Looking at a **.s** file, you will notice that the generated code is somewhat more complicated than the assembler code generated by ordinary C compilers. Some of this complexity is due to extra work the C* compiler does — for example, to initialize its own run-time system and allocate parallel memory. Some of it is also due to inefficiencies in the current version of the C* compiler. (The compiler was built primarily to produce efficient parallel code, and is not as efficient as ordinary C compilers for producing scalar code.)

The C* compiler inserts comments into the generated assembler code that show lines of code from the source program. These comments show roughly which source lines the assembler code is derived from. Although the comments are not always exact, they are very helpful for understanding the generated code.

The assembler code is long, so we just excerpt some of the more interesting parts below. The entry point for the function **fishcake** begins:

```
!void fishcake(int x, int:current a, float:current b)
_fishcake:
    sethi     %hi(-272), %g1
    or        %g1, %lo(-272), %g1
    save      %sp, %g1, %sp
```

This is the code that the compiler has generated to allocate local data on the stack. Following this is code that initializes some registers and makes calls to two internal functions in the lines:

```
    call      __CMC_init, 0
    [...]
    call      _CMRT_allocate_physical_stack_field, 1
```

The first call is used to guarantee that C*'s run-time system is initialized, and the second call performs bookkeeping used for allocating parallel memory.

Below this is code that implements the scalar arithmetic:

```
!   x = x + 2;
    ld        [%i6+68], %l0 ! Line 8
    add       %l0, 2, %l0
    st        %l0, [%i6+68]
```

followed by a call to **printf** in the line:

```
    call      _printf, 2
```

followed by code that initiates our code block to perform the operation

```
b = b * 17.2f + a * x;
```

The code that initiates the code block is complicated because a significant amount of information must be packaged up and handed to the mechanism that broadcasts information to the PNs. The important part to recognize is the call to **`CMRT_funcall`**, the run-time mechanism that initiates code blocks. This is preceded by the setting of the code block's address (**`_CMPE_fishcake_0`**) in the SPARC **`%o0`** register:

```
sethi     %hi(_CMPE_fishcake_0), %l0
add       %l0, %lo(_CMPE_fishcake_0), %o0
add       %l7, Lt7a-4096, %o1
call      _CMRT_funcall, 2
```

(The names of PN code blocks always begin with **`_CMPE_`** and encode the name of the function, as well as a number that makes this code block name unique, since there may be several code blocks for one function. Thus, the code block generated in our example is named **`_CMPE_fishcake_0`**.)

The statement

```
sum = += b;
```

is compiled into a call to another internal run-time system function that computes the sum:

```
call      _CMRT_global_sum_real4, 2
```

(The call is preceded by code that computes information that is handed to this routine.)

The result is stored into the **`sum`** variable:

```
fdtos     %f0, %f0
st        %f0, [%i6+sum$a42]
```

A call to **`printf`** is made to print the result:

```
call      _printf, 3
```

And finally the function exits after again calling a run-time routine that performs more parallel-memory bookkeeping.

A.2 Examining the Generated Parallel Code

A.2.1 Understanding DPEAC Code

DPEAC code is a mixture of SPARC assembler instructions and instructions that perform operations using the vector units. The **`dpas`** assembler translates the vector-unit instructions into SPARC instructions that, when executed, initiate these vector-unit operations on the VUs.

Full understanding of the generated DPEAC code requires understanding the information in the *DPEAC Reference Manual*. But even without understanding all of the details of the vector units, it is not difficult to read the compiler-generated DPEAC code and have a basic understanding of what is going on.

A DPEAC operation typically contains two parts: an arithmetic (or ALU) operation and a memory operation. Both operations are triggered as one instruction. The result of a memory load is usually available before the arithmetic is performed, even though they are not written in that order on the line. Either the memory or the ALU operation may be a no-op (written as **`memnop`**, **`fnops`**, or **`fnopv`**).

Consider this DPEAC instruction:

```
imulv S2:0, V2, V2; uloadv [%i3 + %g6]:4, V2;
```

This performs two operations: a vector load operation and a vector integer multiply operation. Each vector consists of eight elements.

The **`imulv`** specifies the vector integer-multiply operation (**i**nteger-**mul**tiply-**v**ector). It uses the scalar register **`S2`** as one source operand, the vector register **`V2`** as the other source operand, and **`V2`** as the result. **`imuls`** would specify an **i**nteger-**mul**tiply-**s**calar operation, which operates on a single element instead of eight.

The **`uloadv`** (**u**nsigned-**load**-**v**ector) loads an unsigned vector 4-byte word from the base address given by the SPARC **`%i3 + %g6`** values, loading consecutive values 4 bytes apart in memory into the vector register **`V2`**.

Since the result of the load is available for the multiply, the entire instruction loads eight integer values, multiplies each value by the value in **`S2`**, and leaves the results in **`V2`**.

As a rule of thumb, compiler-generated DPEAC vector instructions execute in 16 SPARC cycles each, and DPEAC scalar instructions execute in 2 SPARC cycles each. Thus, the instruction above executes both the load and the multiply for eight subgrid elements (and for all four vector units) in 16 SPARC cycles. On the CM-5 the PN clock frequency is 33 MHz (0.03 µs. per cycle), so this instruction executes in 0.48 µs., or 0.06 µs. per subgrid element.

The most common exception to this rule of thumb is that vector stores of 4-byte quantities (either **`int`** or **`float`** data types) execute in 56 SPARC cycles — that is, 3.5 times as slowly. Similarly, scalar stores execute in 7 SPARC cycles.

Ordinary SPARC instructions execute in one SPARC cycle each. Since DPEAC code mixes vector, scalar, and SPARC instructions together, it is important to remember that the vector instructions are much more expensive. In some cases, SPARC instructions can execute while VU instructions are completing, amortizing the cost of the SPARC instructions.

NOTE: While the SPARC cycle counts above are a good rule of thumb, they ignore several potential factors: instruction issuing time when it is not successfully pipelined, SPARC cache misses, DRAM page faults, and pipeline bubbles that are caused by some DPEAC instructions. Refer to the *DPEAC Reference Manual* for a more complete analysis.

A.2.2 Examining the PN Code

This code is generated for the PN code block in our example:

```
_CMPE_fishcake_0:
    sub %g0, 96, %g2
    save %sp, %g2, %sp
    dpregs %g1=, %g4=, %g2

L1$_CMPE_fishcake_0:
    dpwrt *, %i5, S2
!     b = b * 17.2f + a * x;
    load 0f17.2000008     , %g5
    dpwrt *, %g5, S4
    load 0, %g6
    load 0, %g7

L2$_CMPE_fishcake_0:
!     b = b * 17.2f + a * x;
    imulv S2:0, V2, V2; uloadv [%i3 + %g6]:4, V2;
    load -4, %o0
```

```
    and %g7, %o0, %o0
    and %g7, 3, %o1
    sll %o1, 3, %o1
    add %g7, 1, %g7
    dpwrt *, %o1, S8
!     b = b * 17.2f + a * x;
    itofv V2, V2; uloadv [%i4 + %o0], S6:0;
    fmadtv S4:0, V4, V2, V4; uloadv [%i2 + %g6]:4, V4;
    ushrs S6, S8, S6; memnop;
    ldvm S6
!     b = b * 17.2f + a * x;
    fnopv; ustorev V4, [%i2 + %g6]:4; vmmode:condmem;
    add %g6, 32, %g6
    subcc %i0, 8, %i0
    bnz L2$_CMPE_fishcake_0
    nop

L3$_CMPE_fishcake_0:
    ret
    restore
```

The first part of our code block is the entry point, which begins:

```
_CMPE_fishcake_0:
    sub %g0, 96, %g2
    save %sp, %g2, %sp
    dpregs %g1=, %g4=, %g2
```

The first two instructions allocate local stack space, following the SPARC convention. The **dpregs** line is a directive for **dpas** that tells it which SPARC registers are reserved for its use.

The code in the section labeled **L1$_CMPE_fishcake_0** initializes values in SPARC registers and VU scalar registers. The VU scalar register **S2** here is initialized with the value of **x**, and the **S4** register is initialized with the value 7.2.

The code labeled **L2$_CMPE_fishcake_0** is the body of the subgrid loop. It is actually performing the operation

```
b = b * 17.2f + a * x;
```

Consider for the moment only the vector operations in this section, which are:

```
    imulv S2:0, V2, V2; uloadv [%i3 + %g6]:4, V2;
    itofv V2, V2; uloadv [%i4 + %o0], S6:0;
```

```
fmadtv S4:0, V4, V2, V4; uloadv [%i2 + %g6]:4, V4;
fnopv; ustorev V4, [%i2 + %g6]:4; vmmode:condmem;
```

The first instruction is just what we discussed above. It is loading eight values of the parallel integer **a** into **V2**, and multiplying that by the scalar value of **x** that is stored in the **S2** register, storing the result in **v2**.

The second instruction converts the integer values in **V2** (the result of **a * x**) to single-precision floating-point values, storing the result in **V2**. (Ignore for the moment the memory operation in this instruction.)

The third instruction, like the first, both loads values from memory into a vector register and uses these values in computation. First, eight single-precision floating-point values are loaded into **v4**. Then these values are multiplied by the scalar value (17.2) in **S4** and added to the values of **a * x** in **V2**. The entire right-hand side of the expression has been computed, and the result is left in **V4**.

The fourth instruction stores the result of our computation into **b**. This completes the operation for eight elements.

What do all the other instructions do? Most of them are performing contextualization. Remember that in C* parallel operations are performed only in positions where the current shape's context is active. The compiler implements this by storing the current context as a bitmask in memory, and loading the bitmask values into the VU "vector mask" register that is used to perform conditionalization. The instructions below are performing contextualization in our example:

```
load -4, %o0
and %g7, %o0, %o0
and %g7, 3, %o1
sll %o1, 3, %o1
add %g7, 1, %g7
dpwrt *, %o1, S8
uloadv [%i4 + %o0], S6:0;
ushrs S6, S8, S6; memnop;
ldvm S6
```

The **uloadv** instruction (actually half of the **itodfv** instruction discussed above) loads the vector mask from memory into a scalar register; the **ushrs** instruction selects the appropriate bits in the mask; and the **ldvm** instruction loads the vector mask register itself. The other instructions are ordinary SPARC instructions, and although they add five lines to the code, they add relatively little cost.

Note that, as discussed in Section 4.2.1, the use of **everywhere** in the C* code can eliminate this overhead, when contextualization is not required.

The final code in the body of our subgrid loop is:

```
add %g6, 32, %g6
subcc %i0, 8, %i0
bnz L2$_CMPE_fishcake_0
nop
```

This increments the value used for referencing memory, decrements the subgrid count, and branches back to the start of the loop body, forming the loop.