Appendix A

CM-200 C* Performance Hints

This appendix describes ways to improve the performance of CM-200 C* programs. In some cases, it repeats information included in the body of this guide; in other cases (for example, the discussion of **`allocate_detailed_shape`**), it presents information not discussed elsewhere in the guide.

A.1 Declarations

A.1.1 Use Scalar Data Types

If data is scalar, declare it as a regular C variable, so that it is stored on the front end. In other words, do not store scalars in parallel variables.

A.1.2 Use the Smallest Data Type Possible

To save storage on the CM, use the smallest data types possible for parallel variables. For example, if the parallel variable is a flag, declare it as a **`bool`**. If it is to have values only from –4 to 17, declare it as a **`signed char`**.

A.1.3 Declare float constants as floats

Declaring **`float constants`** as **`floats`** (that is, with the final *f*) reduces the number of conversions that the compiler must make, thereby speeding up the program. For example,

```
float:ShapeA p1, p2;
p1 = p2 * 4.0f;
```

is better than writing the code with just “4.0”.

A.2 Functions

A.2.1 Prototype Functions

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`** will promote the argument to an **`int`**. The called function must then convert the **`int`** back to a **`char`**.

A.2.2 Use current instead of a Shape Name

If a program is to be run with safety on, it is more efficient to define a function to take a parallel variable of the current shape as an argument, rather than a parallel variable of a specified shape. In the latter case, the compiler must take the additional step of determining that the specified shape is current.

A.2.3 Use everywhere when All Positions Are Active

If a function contains statements that are to operate on all positions, regardless of the context in which they are called, you may be able to increase performance by enclosing the function's statements in an **`everywhere`** statement. The explicit use of **`everywhere`** lets the compiler use faster instructions that ignore the context.

NOTE: This technique can also work with a program's **`main`** function.

A.2.4 Pass Parallel Variables by Reference

In function calls, pass a parallel variable by reference (that is, take its address and pass the pointer) if passing the parallel variable by value is not required.

A.3 Operators

A.3.1 Avoid Parallel &&, ||, and ?: Operators Where Contextualization Is Not Necessary

As discussed in Chapter 5, the parallel versions of the `&&`, `||`, and `?:` operators perform implicit contextualization. If you do not require this aspect of the operators' behavior, your code will run faster if you can avoid using them.

For example, if `p1` and `f(p1)` are known to be 0- or 1-valued, then

```
p2 = p1 & f(p1);
```

is much more efficient than

```
p2 = p1 && f(p1);
```

The former statement avoids contextualization, and it avoids doing a logical conversion of its operands, because it assumes that the two operands have logical values.

Similarly,

```
where ( (p1 < p2) & (p2 < p3) )
```

is more efficient than a version that uses the logical AND operator. The "less-than" relational expressions have logical values; therefore, the use of the logical AND (and the resulting contextualization) is not required.

A.3.2 Avoid Promotion to ints by Assigning to a Smaller Data Type

As discussed in Chapter 5, the compiler evaluates an expression at the precision of the variable to which the expression is assigned, provided that the results are the same as if standard ANSI promotion rules were followed. Otherwise, smaller data types such as **`bool`**s and **`char`**s are promoted to **`int`**s when used in expressions. Therefore, explicitly assigning the result of an expression involving these data types to a variable of the same data type will increase performance.

A.4 Communication

To get the best performance in programs in which parallel variables send values to and receive values from other parallel variables, do the following:

1. If possible, put parallel variables that are to communicate in the same shape.
2. Use grid communication functions instead of general communication functions or the language features (like parallel left indexing) that are the equivalent of general communication functions.
3. Use send operations instead of get operations for general communication.
4. If the program has known, stable patterns of communication that use one axis more than another, use **`allocate_detailed_shape`** to weight the axes.

Some of these points are covered in more detail below.

A.4.1 Use Grid Communication Functions instead of General Communication Functions

As mentioned in Part III of this guide, grid communication is faster than general communication. Therefore, your program will run faster if parallel variables that are to communicate are in the same shape, and you use the grid communication functions for send and get operations.

A.4.2 Use Send Operations instead of Get Operations

For general communication, send operations are up to twice as fast as get operations, and use less storage. If possible, use communication functions and C* code that perform send operations rather than get operations.

In grid communication, send operations and get operations have the same cost.

A.4.3 The allocate_detailed_shape Function

Typically, programs use the C* intrinsic function **allocate_shape** to dynamically allocate shapes. If, however, your program has known, stable patterns of communication, you may be able to improve the performance of your program by using the intrinsic function **allocate_detailed_shape** instead; this function lets you weight the axes of the shape according to the relative frequency of communication along the axes. C* can then lay out the shape on the CM to optimize performance based on these weights.

Like **allocate_shape**, **allocate_detailed_shape** is overloaded. In one version, you use a variable arguments list to specify each dimension of the shape. In the other, the information about the dimensions is included in an array that is passed as an argument to the function; this format is useful if the program will not know the rank until run time.

Include the header file **<cm/cmtypes.h>** when you call **allocate_detailed_shape**.

The variable-arguments format of the function is as follows:

```
CMC_Shape_t allocate_detailed_shape (
    shape *shapep,
    int rank,
    unsigned long length,
    unsigned long weight,
    CM_axis_order_t ordering,
    unsigned long on_chip_bits,
    unsigned long off_chip_bits, ...
)
```

where:

shapep is a pointer to a shape. The remaining arguments specify this shape, and the function returns this shape.

rank specifies the number of dimensions in the shape.

length is the number of positions along axis 0.

weight is a number that indicates the relative frequency of communication along the axis. For example, weights of 1 for axis 0 and 2 for axis 1 specify that communication occurs about half as often along axis 0. Only the relative values of the **weight** arguments for the different axes matter; for example, weights of 5 for axis 0 and 10 for axis 1 specify the same communication as weights of 1 and 2, or 3 and 6. Specifying the same values for different axes indicates that they have the same level of communication.

ordering specifies how coordinates are mapped onto physical CM processors for the axis. There are three possible values: **CM_news_order**, **CM_send_order**, and **CM_fb_order**.

The value **CM_news_order** specifies the usual mapping, in which positions with adjacent coordinates are in fact represented in neighboring processors on the CM. Specifying any other order slows down grid communication considerably.

The value **CM_send_order** specifies that a position with a lower coordinate than another position also has a smaller send address. This ordering is rare, but it is used in certain applications.

Use the value **CM_fb_order** only if your shape is an image buffer and is to be moved to a framebuffer. For details, see Chapter 1 of the *Generic Display Interface Reference Manual for C**.

You can specify a different ordering for each axis.

on_chip_bits

off_chip_bits
can be used to specify the mapping of positions to physical processors *only if* the values of the **weight** argument for all axes are the same. Specify 0 for the value of each of these arguments if you use different values for the **weight** argument. For information on how to specify other values for **on_chip_bits** and **off_chip_bits**,

consult the description of the **create-detailed-geometry** instruction in the *Paris Reference Manual*.

Include values for **length**, **weight**, **ordering**, **on_chip_bits**, and **off_chip_bits** for as many axes as are specified by **rank**.

The array format of **allocated_detailed_shape** is as follows:

```
CMC_Shape_t allocate_detailed_shape (
    shape *shape_ptr
    int rank,
    CM_axis_descriptor_t axes[]
)
```

where **axes** is an array that contains descriptors for each axis in the shape to be allocated. You can fill in the information about each axis by calling the C* library function **fill_axis_descriptor**, which is defined as follows:

```
void fill_axis_descriptor (
   CM_axis_descriptor_t axis,
   unsigned long length,
   unsigned long weight,
   CM_axis_order_t ordering,
   unsigned long on_chip_bits,
   unsigned long off_chip_bits
)
```

where **axis** is an array element that corresponds to the axis being described, and the remaining arguments are defined as above.

As an intrinsic function, **allocate_detailed_shape** can be used as an initializer at file scope. Thus, you can do this:

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

shape s = allocate_detailed_shape(&s, 2, 256, 2,
          CM_news_order, 0, 0, 512, 1,
          CM_news_order, 0, 0);
```

This statement fully specifies a 256-by-512 shape **s**, for which you expect communication to occur twice as often along axis 0 as along axis 1.

A.5 Parallel Right Indexing

Parallel right indexing, as described in Chapter 7, becomes less efficient as the range of the array indexes increases.

For users familiar with Paris: The performance of parallel right indexing is comparable to **aref** and **aset** calls, rather than **aref32** and **aset32** calls.

A.6 Paris

Although generally not necessary, it may be possible to improve performance by calling Paris, the CM parallel instruction set, from within a C* program. For details on how to do this, see Chapter 2 of the *CM-200 C* User's Guide*.