The
Connection Machine
System

CM Fortran
Optimization Notes: Slicewise Model

Version 1.0
March 1991

Thinking Machines Corporation
Cambridge, Massachusetts

First printing, March 1991

The information in this document is subject to change without notice and should not be construed as a commitment by Thinking Machines Corporation. Thinking Machines Corporation reserves the right to make changes to any products described herein to improve functioning or design. Although the information in this document has been reviewed and is believed to be reliable, Thinking Machines Corporation does not assume responsibility or liability for any errors that may appear in this document. Thinking Machines Corporation does not assume any liability arising from the application or use of any information or product described herein.

Thinking Machines Corporation
245 First Street
Cambridge, Massachusetts 02142–1264
(617) 234–1000/876–1111

Contents

Customer Support

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

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

To contact Thinking Machines Customer Support:

U.S. Mail:	Thinking Machines Corporation Customer Support 245 First Street Cambridge, Massachusetts 02142–1264
Internet Electronic Mail:	customer–support@think.com
uucp Electronic Mail:	ames!think!customer-support
Telephone:	(617) 234–4000 (617) 876–1111

About This Manual

Objectives

This manual provides hints on how to program in CM Fortran for best performance under the slicewise execution model.

Intended Audience

Readers of this manual are assumed to have programming experience in CM Fortran, including the use of the compiler directive LAYOUT.

Revision Information

This is a new manual.

Associated Documents

- *CM Fortran Reference Manual* Version 1.0
- *CM Fortran Programming Guide* Version 1.0
- *CM Fortran User's Guide* Version 1.0
- *CM Fortran Release Notes* Version 1.0

Chapter 1

The Slicewise Model

Beginning with Version 1.0, the CM Fortran compiler generates significantly improved code targeted for CM systems with the optional 64-bit floating-point accelerator. Two key improvements are that elemental computation can be speeded up by as much as a factor of two, and much less memory (and time) is used up working with compiler temporaries. The compiler generates code for this new execution model when invoked with the switch **`-slicewise`**. For convenience, we use the term *slicewise compiler* to refer to the CM Fortran compiler when it is invoked in this way.

The new execution model is source-code-compatible with the alternative model, called the *Paris* or *fieldwise* model. Thus, changing compiler output requires no action from the user other than recompiling old and new code with the appropriate switch. However, there are certain styles of usage in CM Fortran that enhance performance further under the slicewise model.

This document provides hints for writing CM Fortran programs in a way that optimizes compiler-generated slicewise code. It also notes some workarounds for performance blind spots in this early implementation of the slicewise compiler. Future versions will remove many of these deficiencies, at which point the workaround may have no effect or a negative effect. The *CM Fortran Release Notes* and future editions of this document will enhance the list of programming hints and keep you informed of the status of compiler blind spots.

Most of these hints are specific to the slicewise model, and thus confer no benefit if the program is compiled for the Paris model. (An exception is the recommendation to unwind loops that express communication on serially ordered dimensions; this practice coincidentally helps Paris output as well.)

This document is based largely on work done by Gary Sabot of the CM Fortran compiler group.

1.1 FPUs as Processors

The driving factor behind the development of the slicewise model is the performance potential of using the registers and vector-processing capabilities of the units (chips) of the 64-bit floating-point accelerator. All CM-2s are organized into *processing nodes,* each containing 32 bit-serial processors, some memory, one (optional) FPU chip, and other associated hardware. Under the Paris model, the basic processing elements (PEs) are the bit-serial processors, although the FPA and other hardware are used in some operations. Under the slicewise model, the PEs are the processing nodes themselves. Thus, a CM executing in the slicewise model is using machine-size/32 PEs — 2K PEs for a 64K CM.

When the CM Fortran compiler generates the Paris instruction set, it does not use the FPU registers to pass values between different Paris instructions. Since all Paris operations are memory-to-memory, their theoretical peak performance is limited by memory bandwidth to 1.5 Gflops (for a full-sized 64K CM-2).

Special microcode can make explicit use of the FPU registers as the source of operands and the destination of results for elemental computation, thus avoiding memory loads and stores. Also, the 64-bit FPA can be used as a vector processor — actually, a set of vector processors, each with a vector length of 4. The theoretical peak performance of such code (that is, code without loads and stores and without communication) is 14 Gflops for a full-sized CM-2. The goal of the slicewise execution model is to allow CM Fortran programs to exploit this performance enhancement.

1.2 Slicewise Execution

When invoked for slicewise, the compiler views the CM as a set of vector processors. Instead of generating Paris instructions, it generates a new RISC-like instruction set (called PEAC, for PE assembly code), which translates into microcode that executes on the FPU chips. To facilitate transfers between PE memory and FPU registers, data is stored in memory as 32-bit words — the memory of each of a node's 32 bit-serial processors holds a one-bit slice of a word, rather than the whole word (hence the term *slicewise).*

The compiler itself does not perform CM memory management or interprocessor (meaning inter-PE) communication. Instead, it calls the functions of a newly written run-time library. Because the slicewise data format is not compatible with the previously written communication microcode, all the communication functions have been reimplemented to support CM Fortran Version 1.0.

NOTE: Some CM library routines expect their input/output to be in the Paris, or fieldwise, data format. When such routines are called from a CM Fortran program that is executing slicewise, the data format needs to be transposed frequently. This transposition exacts a performance cost. Please consult the documentation for the various libraries to see which versions can be called from slicewise CM Fortran and which of these execute more slowly under slicewise than under the Paris model.

1.3 Slicewise Array Layout

The run-time system lays out arrays in CM memory differently depending on the number of PEs available to execute the program. Since array layout is not determined at compile time, you can run a CM Fortran program on any size CM system without recoding or recompiling. As with the Paris model, the (physical) PE loops over the array elements assigned to it, repeating each instruction as many times as necessary.

To lay out an array, the run-time system first takes the available PEs and organizes them into a *physical grid*. It then specifies *subgrids* of allocated memory within the PEs. The subgrids that hold a given array are all the same size and are located at the same memory address in each PE. Since the FPU's vector length is 4, the length of each subgrid must be a multiple of 4. Thus, the total number of memory locations allocated is a multiple of 4 times the number of PEs executing the program.

An array that is not a suitable size is padded up to the next multiple of 4 times number-of-PEs. For example, an array of 32,768 elements requires no padding; on a 64K machine, each subgrid contains 16 elements. An array that is 32,769 is rounded up to 40,960; on a 64K machine, each subgrid contains 20 elements. (Note that power-of-2 axis extents are not required.)

The allocated memory is described by a *machine geometry*, which specifies its physical grid and subgrids, as well as its logical rank and shape. (We use the term *virtual grid* as a convenient way to refer to the allocated memory across PEs — analogous to a Paris VP set — although a virtual grid does not exist as an object in CM memory.) The run-time system uses a complicated algorithm to determine the logical shape of subgrids and to map declared array dimensions onto the virtual grid. Features of the layout algorithm are noted in this document in connection with optimization hints that relate to them.

1.4 A Note on Precision of Numbers

Under the Paris model, where all array operations are memory-to-memory, operations on double-precision numbers are twice as expensive as operations on single-precision numbers. Under slicewise, double-precision *computation* (that is, elemental operations performed by the floating-point accelerator) does not cost any more than single-precision computation. Loads and stores to memory of double-precision quantities still cost twice as much as single-precision loads and stores. Since the slicewise model typically does fewer loads and stores to memory than the Paris model, the use of double-precision real or complex values often increases execution time by less than twice the time for single-precision values. For example, in the timed programs shown in Chapter 6, changing the values from single-precision to double-precision increased the computation time by 1.4.

Chapter 2

Elemental Code Blocks

The slicewise compiler excels when it deals with large blocks of elemental operations on one or more conformable CM arrays. Since these blocks execute entirely within the respective processing nodes, they enable the compiler to make best use of the FPU registers and pipelined vector processing.

An elemental block is broken when the compiler needs to deal with some other kind of instruction. The constructs that break up elemental code blocks are:

- Any kind of transformation (including data motion) on CM arrays.
- Any elemental operation that involves a differently shaped array from the previous elemental operation.
- Any code that executes on the front end, such as scalar assignment statements, calls to external functions, and control flow changes.

This chapter provides hints for optimizing the compiler's handling of computation within elemental code blocks.

2.1 Segregating Computation and Communication

- ♦ Group elemental operations on conformable arrays to maximize the size of elemental code blocks.

The compiler does not perform any code motion on user code to increase the size of elemental blocks, so it is up to the programmer to make them as large as possible.

For example, the following program needlessly interposes a communication operation (a vector-valued subscript, which always entails data motion) between two elemental array assignments:

```
PROGRAM BAD_1
INTEGER DATA(120000),DEST(120000),INCOMING(120000)
INTEGER UNRELATED(120000)
...
DATA      = DATA * 2
INCOMING  = DATA(DEST)
UNRELATED = UNRELATED * 2
```

You can increase the elemental block size and improve performance by moving the communication operation out from between the two computation statements:

```
PROGRAM GOOD_1
INTEGER DATA(120000),DEST(120000),INCOMING(120000)
INTEGER UNRELATED(120000)
...
DATA = DATA * 2
UNRELATED = UNRELATED * 2
INCOMING = DATA(DEST)
```

Similarly, the following program interposes a data transposition between an elemental add and an elemental multiply:

```
PROGRAM BAD_2
INTEGER M(1000,1000),N(1000,1000)
...
N = 2 * TRANSPOSE(M+1)
```

It is better to do all the computation at the source and then transpose the elements:

```
PROGRAM GOOD_2
INTEGER M(1000,1000),N(1000,1000)
...
N = TRANSPOSE( 2 * (M+1) )
```

Or, equivalently, do all the computation at the destination after the transposition:

```
N = 2 * (TRANSPOSE(M) + 1)
```

An elemental code block involves only one array shape. Hence, grouping a series of elemental operations according to shape increases block size. For example, instead of this:

```
PROGRAM BAD_3
INTEGER SIZE1A(100), SIZE1B(100)
INTEGER SIZE2A(128,128), SIZE2B(128,128)
...
SIZE1A = 0
SIZE2A = 0
SIZE1B = 0
SIZE2B = 0
```

Use this:

```
PROGRAM GOOD_3
INTEGER SIZE1A(100), SIZE1B(100)
INTEGER SIZE2A(128,128), SIZE2B(128,128)
...
SIZE1A = 0
SIZE1B = 0
SIZE2A = 0
SIZE2B = 0
```

As always, the shape to consider is the shape of the *parent* array, not of the section. Operations that involve conformable sections of nonconformable parents require communication to line up the operands in the same PEs, since the parents occupy different virtual grids. The same is true of arrays that are of the same shape but whose layouts differ because of different axis orderings or weights. For example, given the declarations above, the middle statement here breaks up the elemental code block even though the operands are the same shape:

```
SIZE1A = 0
SIZE1A = SIZE1B + SIZE2A(1, 1:100)
SIZE1B = 0
```

You could avoid communication in this case only by aligning the two vectors with the first row of **SIZE2A** (using the compiler directive **ALIGN**).

2.2 Communication Temporaries: A Compiler Blind Spot

- ♦ Introduce temporaries for communication operations and place them so as to increase the size of elemental blocks.

The slicewise compiler routinely removes communication from within elemental expressions. It creates a temporary array in memory and then performs the communication operation, using the temporary as the destination. Then, using the temporary in place of the communication, the compiler evaluates the now fully elemental expression.

A blind spot in the current compiler is that the communication is performed immediately before the statement. The compiler never moves statements earlier than this to avoid breaking up an elemental code block, even though this is often possible. In effect, the compiler transforms this program:

```
PROGRAM BAD_4
INTEGER M(1000,1000),N(1000,1000),P(1000,1000)
...
P = M
N = TRANSPOSE(M) * 10
```

into this program, which contains two elemental code blocks:

```
PROGRAM BAD_4_INTERNAL
INTEGER M(1000,1000),N(1000,1000),P(1000,1000)
INTEGER COMPILERTEMP(1000,1000)
...
P = M
COMPILERTEMP = TRANSPOSE(M)
N = COMPILERTEMP * 10
```

You can reveal a single elemental block to the compiler by declaring a temporary yourself and performing the communication before the two elemental operations:

```
PROGRAM GOOD_4
INTEGER M(1000,1000),N(1000,1000),P(1000,1000)
INTEGER USERTEMP(1000,1000)
...
USERTEMP = TRANSPOSE(M)
P = M
N = USERTEMP * 10
```

For the Future

In a future release, the compiler will be able to perform code motion itself, and it may be less constrained to separate communication from computation via temporaries. At that point, the approach of the original program, **BAD_4**, will be preferred to that of **GOOD_4**. For the short term (perhaps two years), **GOOD_4** gives better performance.

2.3 Limit on Elemental Block Size: A Compiler Blind Spot

- ♦ Split up elemental code blocks that reference more than about 18 CM arrays (or serial dimension coordinates).

To identify each CM array to be used in an elemental code block, the front end broadcasts a *parallel memory address* to the CM, where it is stored in an address register in the sequencer during the execution of the block. A separate parallel memory address is provided for each specified coordinate in a serial axis. For example, the following loop uses two parallel memory addresses:

```
      INTEGER A(10,1000), I
CMF$LAYOUT    A(:SERIAL,:NEWS)
      ...
      DO I=1,10
        A(I,:) = A(I+1,:) + 1
      ENDDO
```

Excessive Parallel Addresses

When too many parallel memory addresses are used in a code block, the CM sequencer runs out of address registers and is forced to spill the addresses to memory. Normally, the sequencer can receive an address broadcast from the front end in less time than it takes to spill and then reload an address register. Therefore, it is usually better to broadcast a value redundantly (that is, use it again in a separate code block) than to allow the sequencer to run out of registers in a single code block. The overall gain from avoiding unnecessary spill/reloads is on the order of 5 to 10 percent.

Redundant address broadcasting is preferable *as long as* the elemental code block is large enough to keep the CM busy while the front end races ahead. As long as the front end is operating ahead of the CM, it can have the addresses ready at the CM when needed. If the CM is not kept busy, the front end might not be far enough ahead to have the needed addresses ready, and the performance advantage is lost as the CM waits for them.

Optimal Number of Parallel Addresses

The optimal number of parallel memory addresses per elemental code block is between 15 and 19, depending on some internal considerations. This number is small enough to avoid spilling sequencer registers, and large enough to indicate that the code block probably contains enough operations to keep the CM busy.

Reducing the Number of Parallel Addresses

Elemental code blocks can be split by reordering your code to violate the restrictions described in Section 2.1. Moving a communication operation, or an operation on a nonconformable array, into the middle of an elemental code block will split the block, as will any scalar operation or control flow statement. Also, you should probably stop unwinding loops on serially ordered dimensions once about 18 parallel memory addresses are used (see Section 5.5).

Some Caveats on Splitting Code Blocks

Splitting up an elemental code block is an optimization only if its pattern of CM array usage allows you to produce blocks that use disjoint, smaller sets of arrays. If a large block uses 30 parallel memory addresses, it does not make sense to split it into two halves that each use 28 addresses. If each half uses 18 addresses, however, splitting the block may improve performance.

It is often convenient to use a scalar assignment to break up an elemental block. However, if the scalar variable that is assigned is never used, the optimizer may kill the apparently useless statements. You must use the variable in a way that tricks the optimizer into not eliminating the block-breaking statements:

```
      PROGRAM BREAKBLOCK
C     ... declarations
      INTEGER BREAKBLOCK

      ARRAY = ... parallel code

C Break the block
      BREAKBLOCK = 1

      ARRAY = ... more parallel code

C Break the block again, using previous value
      BREAKBLOCK = BREAKBLOCK + 1

      ARRAY = ... still more parallel code

C Use the variable so the optimizer does not remove it.
      CALL USE_VARIABLE( BREAKBLOCK )
      END

      SUBROUTINE USE_VARIABLE( X )
      END
```

For the Future

Eventually, the compiler may be able to break code blocks itself to avoid overloading the sequencer address registers. At that point, the user's explicit block-breaking efforts will become a pessimization. The compiler will have better information about register usage and exact spill/reload costs than will the user, so it will be more effective at picking the optimal breaking points. Alternatively, if the sequencer's spill/reload performance improves, the compiler may choose to avoid breaking blocks in some cases.

Future release notes and updates to this manual will provide information on the status of this compiler blind spot.

Chapter 3

Temporary Arrays

An important innovation of slicewise compilation, compared with the Paris model, is the compiler's use of FPU registers. The compiler may need to allocate temporaries when evaluating expressions, but, as long as the temporary is not the destination of a communication operation, it is held in an FPU vector register. In contrast, compiler temporaries created under the Paris model are held in memory.

Compiler temporaries created for use in elemental *computation* thus make fewer demands on system resources (memory and memory bandwidth) under the slicewise model than they do under the Paris model. Even if the compiler needs to spill a register to memory, it spills only number-of-nodes * 4 values (that is, the number of PEs executing the program times the internal vector length), rather than the full array size, which may be much larger.

Temporaries created for use in *communication* do reside in memory under the slicewise model, as they do under Paris (see Section 2.2).

3.1 User Temporaries

♦ Write large expressions to avoid unnecessary temporary arrays in computation.

In the following program, the array **TEMP** is not really necessary. This program needs to allocate space in memory to hold 120,000 extra integer values and needs to perform three extra writes to memory:

```
PROGRAM BAD_5
INTEGER A(120000), TEMP(120000)
...
```

```
TEMP = A + 1
TEMP = TEMP/2
A = TEMP
```

Instead, write this:

```
PROGRAM GOOD_5
INTEGER A(120000)
...
A = (A+1)/2
```

The **GOOD_5** program allocates memory only for array **A** (the literal constants are provided as immediate operands), and it requires only one write to memory. If, under the worst case, the compiler is forced to spill a register to memory, it needs to write only 8192 values (2048*4, assuming a full-sized CM) rather than 120,000 values (plus padding, as described in Chapter 4).

3.2 Communication Temporaries: A Compiler Blind Spot

- ♦ Reduce memory usage by sharing your temporaries with the compiler for communication.

The compiler allocates memory temporarily to isolate communication operations from computation. A blind spot in the current implementation is that the compiler uses only its own temporaries for this purpose: it never "borrows" an array created by the program, even though this is occasionally possible. In effect, the compiler transforms this program:

```
PROGRAM OKAY_6
INTEGER M(1000,1000), N(1000,1000)
...
N = TRANSPOSE(2 * (M+1))
```

into this program:

```
PROGRAM OKAY_6_INTERNAL
INTEGER M(1000,1000), N(1000,1000)
INTEGER COMPILERTEMP(1000,1000)
...
```

```
COMPILERTEMP = 2 * (M+1)
N = TRANSPOSE(COMPILERTEMP)
```

You could restructure this program to eliminate the compiler temporary by borrowing **N**:

```
PROGRAM BETTER_6
INTEGER M(1000,1000), N(1000,1000)
...
N = 2 * (M+1)
N = TRANSPOSE(N)
```

This restructuring saves memory by working around a blind spot of the compiler, which currently never uses a user array to hold compiler temporaries.

Naturally, there would be no memory savings if you created a *new* temporary and then moved communication results into it. In fact, creating a new temporary for this purpose might be a pessimization, because the compiler currently does a better job of deallocating its own temporaries as early as possible than it does with user temporaries.

For the Future

In the future the compiler will be able to do this transformation (borrowing user memory) itself. It may also begin to overlap communication with computation, which would reduce the need for temporary memory in performing communication. At that point, the approach of program **OKAY_6** will be preferred to that of **BETTER_6**. For the near term (say two years), **BETTER_6** is more space-efficient.

Chapter 4

Effects of Array Size

Both the slicewise and Paris execution models place certain requirements on the amount of CM memory allocated to hold a Fortran array. Fortran arrays can be any arbitrary size, but virtual grids (and Paris VP sets) must meet certain constraints. In both models, an array that is not an acceptable size for a grid in CM memory is "padded" up to an acceptable size, and the CM may operate on the padding elements as well as on the elements containing user data. Naturally, the padding translates into both wasted memory and wasted processing time.

At the same time, both execution models operate more efficiently on large arrays than on small arrays, because a processing element better amortizes its start-up overhead over a large number of loop iterations than over only a few. The efficiency of computations — given here in Flops rates, although integer operations also benefit — rises with the number of subgrid iterations (or VP ratio) until it hits a peak, and then stays about the same as subgrid length increases further.

This chapter provides information on the performance effects of array padding and suggests ways of discovering the array size at which performance peaks. As in the CM Fortran language manuals, we use the term *array size* to mean the product of its axis extents.

4.1 Padded Arrays

Under the slicewise model the best performance occurs on a much broader class of arrays than under the Paris model. Because the constraints on virtual grid size are less severe, array dimensions are less likely to be padded, and are usually padded less heavily under slicewise than under Paris.

Computation on Padded Arrays

- ♦ The effect of array padding on computational efficiency is trivial except for very small arrays.

An array of size *s* that is padded up to size *p* takes as long to process as if it were size *p* to begin with. Under the Paris model, where each dimension that is not a power of 2 is padded up to the next power of 2, the worst-case performance degradation can be a factor of 2 on every dimension. This fact leads performance-minded programmers to write their code in unnatural ways to avoid non-power-of-2 arrays.

Under the slicewise model, padding is needed only to make the *product* of the dimension extents (for non-serial axes) equal a multiple of 4 times the number of PEs executing the program. There is no length constraint on any particular axis.

The padded size *p* is determined by an algorithm that seeks to minimize the amount of memory used (as well as to place the same number of elements for a given dimension in each of the PEs). This fact places a simple upper bound on the total amount of padding that is much lower than the potential padding under Paris. For a 1-dimensional array (that is, ignoring the second constraint above), the maximum amount the dimension can be padded is (n * 4) –1 elements, where *n* is the number of PEs executing the program. This amounts to only 8191 elements for a 64K CM. Thus, the difference in efficiency between the best and worst array sizes is much smaller with slicewise execution than with Paris.

> NOTE: Padding may exceed (n * 4) –1 elements for multidimensional arrays because of the need to give each PE the same number of elements from a given array dimension. That is, a multidimensional array may sometimes, depending on its shape, be padded beyond the *next* multiple of (n*4). The total padding is still comparatively small, however. You can use the CM Fortran utility procedure `CMF_DESCRIBE_ARRAY` to determine how much a particular array is padded. This procedure prints information about the array geometry and the machine geometry (the shape of the virtual grid), among other things, to `stdout`. Any difference between their sizes is padding.

Because the amount of padding added to arrays is fairly small, its effect on computational efficiency is significant only for very small arrays. For example, for a 1-dimensional array with 200,000 elements, the maximum possible padding is around 4 percent, which is probably not worth worrying about. In most cases, of course, padding will be less than the maximum possible. For example, if a 1-dimensional array is of size 120,000, the padded size is 122,880 (60 * 2048, assuming a full-size CM), so that padding makes up only 2.3 percent of the virtual grid. Thus, except for arrays that are much smaller than (or not much larger than) 4 times the number of PEs, the slicewise model removes the programmer's incentive to warp the natural sizes of arrays to improve computational performance.

Communication on Padded Arrays

- ♦ Try to avoid padding on arrays that are used heavily in communication operations.

Communication operations on an array of size *s* that is padded to size *p* is slower than it would be if the array were size *p* to begin with. It does not matter which dimensions are padded or which dimensions are used in the communication.

The slicewise model, which does not have a notion of virtual processors, does not associate as much state information with each array element as does the Paris model. Under Paris, each array element is mapped to a virtual processor that has a state bit (called the context flag) whose setting indicates whether that processor is to participate in the next instruction. In the slicewise model, all the elements participate, but some may be ignored.

This approach causes no problem for elemental computation: it is faster to process an element and later ignore the results than it is to have a virtual processor check its state before each instruction. (The same rationale underlies the fast **`_always`** instructions in Paris.) The *garbage data* in such an element does no harm — *unless* a communication operation allows it to get out.

Therefore, when an array has *any* padding at all, a communication function must check *every* element before moving data from it, and skip the communication if the element contains garbage data. This checking slows the algorithm down, by different proportions for the different communication functions. For example, the **`SUM`** intrinsic is slower by about a factor of 2 when applied to a padded array.

Because the library of run-time communication functions is new with Version 1.0 of CM Fortran, little comparative timing information is available. Also, timings are likely to change frequently, since performance tuning of the run-time library is a current focus of the development effort.

The safest approach to communication for performance-minded programmers is to declare arrays such that they will not need to be padded. An array whose declared size (that is, the product of its non-serial dimension extents) equals a multiple of 4 times the number of PEs executing the program is never padded.

CSHIFT Wrapping Performance

- **CSHIFT** takes more time on a padded array dimension than on a nonpadded dimension, but the time difference is less under slicewise than under Paris.

Dimension padding has a particular effect on the intrinsic function **CSHIFT** because the function wraps the last element(s) in the shift direction around to the beginning. If the dimension is not padded *and if* its extent is a power of 2, the wrapping occurs in hardware (that is, the last element on a power-of-2 virtual grid axis is connected to the first element on that axis). In this situation, wrapping takes no more time than simply discarding the last element (as **EOSHIFT** does). If the dimension contains garbage elements, however, *or if* its extent is not a power of 2, the wrapping entails moving data more than one "hop" in hardware.

Under the Paris model, **CSHIFT** may use an expensive general communication instruction (a Paris **send**) to perform the wrap. Under slicewise, the wrapping is always performed with one or more power-of-2 NEWS operations. At most seven NEWS hops are required to do the wrapping for any array on the CM. Most **CSHIFT** operations require fewer than seven, but even seven NEWS operations can be completed faster than one **send**.

Because arrays are less likely to be padded at all under slicewise than under Paris, there are more cases when the best **CSHIFT** performance is possible. And, because of the difference in implementing wrapping, the best and worst cases of **CSHIFT** performance are closer together.

4.2 Subgrid Looping

A major effect of array size on performance arises from the length of the subgrid that the PE iterates over. Every elemental code block or communication operation involves some overhead — for instance, the time required for the CM to receive addresses and data from the front end. The overhead is incurred at the beginning of the code block, before the subgrid loop begins to execute. The larger the subgrid, the more iterations in the loop and thus the greater the time over which to amortize the overhead.

- Peak CM efficiency is achieved with smaller array sizes under slicewise than under Paris.

It is immediately obvious that the slicewise model has an efficiency advantage over Paris because the iteration count is 32 times larger for a given array size. For example, an array of size 128K on a 64K CM translates under Paris to a VP ratio of 2, since each bit-serial processor holds 2 elements. Under slicewise, the same array translates to a subgrid length of 64, since the number of PEs is 64K/32. In fact, 64 loop iterations is enough to reach peak efficiency for even the worst-behaved elemental code, whereas the timing of the 2 iterations that occur under Paris is still heavily influenced by the start-up overhead.

An important question to users who can vary the size of their data sets is what subgrid length (that is, data set size) they must operate on to use a particular size of CM efficiently. The best answer comes from timing your code with various array sizes on an actual CM, and noting at what point the Flops rate stops rising with increasing array size. This section gives some hints for estimating what the optimal subgrid length might be for a particular elemental code block.

The discussion that follows assumes, for convenience, that no padding is present on the array in question. If padding is a significant concern (say, if one-quarter of the virtual grid is padding), then the fact that available computation time is lost to operating on garbage data is a more important source of inefficiency than insufficient amortization of overhead. If the subgrid is large enough to achieve peak efficiency, then it is likely that the padding is insignificant (at least for elemental computation).

Array Size and Computation

Given a particular size of CM, the peak Flops rate of a block of elemental code is a fixed value. For inspiration, you can calculate the peak rate from information contained in the **.peac** intermediate file (see Appendix A). The peak can be achieved, however, only if the overhead required to start execution of the code block is amortized over a sufficiently large amount of computation. Once efficiency peaks, it remains about the same if array size is increased further.

For example, suppose the peak for a piece of code is reached at subgrid size s. Our preliminary timings indicate that doubling the array size so that the subgrid size is $2s$ does not measurably improve performance. A subgrid size of $s/2$, however, reduces performance to about 70 percent of the peak, and a subgrid size of $s/4$ reduces performance to about 30 percent of the peak.

- ♦ If you can vary array size, try to achieve a subgrid length of at least 32 on a Sun-driven CM and at least 64 on a VAX-driven CM.

In some cases, an even smaller subgrid length (about 16 for CM/Sun or 32 for CM/VAX) is sufficient to reach peak efficiency for an elemental code block. The key factor determining the array size needed to amortize overhead for a code block is the ratio of floating-point (or integer) operations in the block to the number of values (including CM array addresses) received from the front end.

- ▪ The number of floating-point operations in a code block can be determined from the **.peac** intermediate file, which gives a Flops count per line of the block and totals them at the end.
- ▪ The number of values received from the front end can be determined from the source code: it is the number of scalar variables, constants, and CM arrays that the elemental code block uses. (When a serially ordered dimension of a CM array is referenced with scalar subscripts, each such coordinate should be counted as a separate array, since it is sent to the CM as a separate parallel memory address.)

The significance of these factors is that a block that uses a large number of values once each (for example, `A=B+C+D+E+F`) has higher overhead and thus requires a larger subgrid size to reach its peak Flops rate than does a block that performs the same number of operations on a smaller number of values (for example, `A=A+A+A+A+A`).

Preliminary timings on a CM with a Sun front end indicate that if the Flops/values ratio of an elemental code block is near 1, the peak Flops rate is first reached at a subgrid size of 32 — hence the recommendation at the beginning of this subsection. However, for blocks with a Flops/values ratio of 3, the peak is reached at subgrid size 16.

Preliminary timings on a CM with a VAX front end indicate that this configuration reaches its peak efficiency with about twice as many subgrid elements as does a Sun-driven CM. Thus, for a Flops/values ratio of 3, a VAX-driven CM comes near its peak with a subgrid size of 32 (compared with 16 for the Sun-driven CM). Since the same CM hardware is involved, the peak rates are the same. The Sun-driven CM is able to achieve them on smaller array sizes because the Sun is able to send the CM a given set of values in less time than a VAX.

- ♦ If possible, time some elemental code blocks with various array sizes to determine more precisely what size array is needed to reach peak speed on your CM system.

The suggestions for target array sizes given different Flops/values ratios are merely rules of thumb based on preliminary timings. Also, we have not yet isolated all the factors that might affect slicewise performance on a particular CM system configuration. For best performance, you might want to conduct some further timing experiments on your own CM system. Also, it is only by timing experiments that you can optimize your entire program, including both its computation and its communication.

Array Size and Communication

The time required for communication is very dependent upon the particular function and the particular pattern involved. The execution time of a communication function can be thought of as having a constant base cost plus a linear cost per subgrid element that moves off the PE. (That is, the cost of communicating between elements stored on different PEs is added to the base cost of overhead plus communicating among elements that all reside on the same PE.) The number of elements that move off the PE can be determined from the description of an array's corresponding machine geometry given by the CM Fortran utility library procedure **CMF_DESCRIBE_ARRAY**. See the *CM Fortran User's Guide.*

The base cost can be much larger than the linear cost. For example, the function that implements **CSHIFT** of an unpadded integer array takes a constant 475 microseconds plus about 19 microseconds per subgrid element that moves off the PE. This means that a much larger subgrid size is needed to effectively amortize the overhead than is the case for elemental code. With a subgrid size of 32, a **CSHIFT** might take a millisecond: almost half is due to the overhead.

- ♦ If possible, time your communication operations with various array sizes to determine what size array is needed to reach peak efficiency.

We do not yet have systematic information on the timings of the communication functions. At present, we urge you to time the communication parts of your programs with various array sizes on a fixed-size CM to determine what array size optimizes their performance.

Chapter 5

Effects of Array Layout

The layout of a CM Fortran array in CM memory is determined by a run-time system algorithm that considers the array's shape (that is, its rank and dimension extents) in relation to machine size, along with the array's dimension orderings and weights. (The latter two factors might be affected by the compiler directive LAYOUT.) This chapter examines some performance factors that relate to array layout.

Under the Paris model, an array's shape also determines the amount of padding added to it — a major factor in Paris execution speed. Under the slicewise layout strategy, the need for padding is determined by the array's total size, not by its shape. The performance effects of array size and padding are described in Chapter 4.

5.1 Array Shapes in Elemental Computation

- ♦ Avoid needless communication by declaring similarly shaped arrays in exactly the same shape (or by aligning them).

The slicewise compiler assumes that CM arrays share a machine geometry *only if* the arrays have exactly the same layout, and this requires that they have exactly the same shape. If you use two or more nearly conformable arrays together in elemental computations, you are incurring a communication cost.

Under the Paris model, it is often easy to see that two or more nearly conformable arrays do in fact share a VP set, and therefore that they can be used together without incurring a communication cost. Users can rely on the fact that each padded dimension extent is the next power of 2 beyond its declared extent, and that arrays that are rounded up to the same shape are in the same VP set. For example, the following assignment does not require communication, since **SMALL** and **BIG** are in the same Paris VP set.

```
PROGRAM PARIS
INTEGER SMALL(2047)
INTEGER BIG(2048)
...
SMALL = BIG(1:2047)
```

Under the slicewise model, however, each distinct array shape is given its own *array geometry*, which the run-time system later relates to a machine geometry that describes the memory allocated. The compiler cannot predict the machine geometry because it cannot predict how the run-time system will pad arrays or lay out their subgrids. Although it is likely in an example as simple as this that the two machine geometries will be identical, the compiler does not make this assumption. Therefore, the assignment statement requires communication to move data from one area in memory (virtual grid) to the other.

If you want to use **SMALL** and **BIG** together without communication, make them exactly the same shape. When performing communication, you will need to mask out the element you added to **SMALL** to prevent garbage data from escaping. There is no need to mask the element for elemental computations, since the garbage data causes no problems.

Better still, you could use the compiler directive **ALIGN** to force a smaller array to have the same machine geometry as a larger one. In this case, the run-time communication functions do the necessary masking of the garbage elements.

5.2 Array Rank and Virtual Grid Rank

Under the slicewise model, the rank of a virtual grid is the same as the rank of the array(s) mapped onto it. If the compiler needs to pad an array up to a legal size, it does so by extending one or more of the existing array dimensions, not by extending an additional hidden dimension as in the Paris model.

This information is useful only when using CM libraries that do not have a CM Fortran interface — that is, when using procedures that operate on lower-level data structures. Under the Paris model, the programmer needs to be aware that the rank of a VP set is one greater than the rank of the arrays it holds. Under the slicewise model, arrays and their virtual grids have the same rank.

Virtual grid axes are numbered from 0 rather than from 1 in the machine geometry that describes the grid. In the description of machine geometry generated by the CM Fortran utility procedure **CMF_DESCRIBE_ARRAY**, the grid axis numbered 0 corresponds to array dimension 1, axis 1 corresponds to dimension 2, and so on. However, in the Utility Library procedures that take array dimensions as arguments, the dimensions are numbered from 1, as they are in CM Fortran itself.

5.3 Dimension Weights and Communication Speed

♦ When using weights to influence the layout of parallel dimensions, declare those dimension extents as powers of 2.

Communication speed along an array dimension is influenced by the relative number of on-node and off-node elements on that dimension. That is, the system needs extra time to communicate between elements that reside on different PEs. By default, no array dimension is particularly favored for communication under the slicewise layout strategy: any or all parallel (NEWS-ordered or send-ordered) dimensions might contain some on-node and some off-node elements.

The compiler directive **LAYOUT** allows you to specify that certain dimensions will be used more heavily than others in communication operations by giving them higher weights. The run-time system responds as follows:

- A parallel dimension *tends* to be laid out such that communication along that axis requires less movement off-node.

- Serial dimensions are *always* laid out entirely on-node. Therefore, weights applied to a serial dimension are ignored.

- There is an advantage in declaring all parallel dimensions in power-of-2 extents. In this case, weights have the same effect as in the Paris model: they cause the higher-weighted axes to be favored for communication, although not necessarily in proportion to their respective weights.

- If *any* parallel dimension extent in an array is not a power of 2, weights are not guaranteed to affect that array's layout. The run-time system might find that higher-priority goals, such as minimizing the amount of memory allocated (and thus reducing padding), completely determine the array's layout, and that it has no latitude to factor in an effect from weights.

5.4 Declaring Serial Dimensions: A Compiler Blind Spot

- Declare all serial dimensions of an array before any parallel dimension.

CM Fortran gives best performance on an array with serially ordered dimensions when all such dimensions are declared to the left of any of the parallel dimensions.

This on-going restriction is permanent under the Paris model, since it is enforced by the Paris run-time system. Under slicewise, this restriction will be removed in a future release, and performance will not be affected by the placement of serial dimensions in the array declaration. The *CM Fortran Release Notes* and future editions of this document will alert you when this restriction has been removed.

5.5 Communication on Serial Dimensions: A Compiler Blind Spot

A serial array dimension has the distinctive property that it is always allocated entirely within — never across — PEs. Even for very small arrays, the run-time system configures a virtual grid, determining subgrid layout and adding padding if necessary, by considering *only* the parallel dimensions of an array. After the machine geometry is determined, the system extends the subgrids to accommodate any serial dimensions (always unpadded).

This strategy optimizes elemental computation on sections of an array that are referenced with scalar subscripts along a serial dimension. Such operations are guaranteed to be entirely on-node.

For communication operations, however, the slicewise CM Fortran compiler shares a deficiency with previous and current releases of the Paris model. The compiler fails to see that a serial dimension is a special case: it handles communication on a serial dimension as if it were any other dimension. Thus, it creates communication temporaries in memory and performs needless moves within a PE's memory to initialize them.

This section provides hints on working around this compiler blind spot.

Using Loops

- ♦ Use serial loops to express communication along serial dimensions.

You can prevent the compiler from creating temporaries to perform communication on a serial dimension by explicitly coding such operations as serial loops. This action saves memory and saves the time that would otherwise be spent writing to and reading from that memory.

For example, consider this shift operation on a serial dimension:

```
      PROGRAM OKAY_7
      INTEGER A(10,1000)
CMF$LAYOUT A(:SERIAL,:NEWS)
      ...
      A(1:9,:) = A(2:10,:) + 1
```

The compiler transforms the program into this:

```
      PROGRAM OKAY_7_INTERNAL
      INTEGER A(10,1000)
      INTEGER COMPILERTEMP(10,1000)
CMF$LAYOUT A(:SERIAL,:NEWS)
CMF$LAYOUT COMPILERTEMP(:SERIAL,:NEWS)
      ...
      COMPILERTEMP(1:9,:) = A(2:10,:)
      A(1:10,:) = COMPILERTEMP(1:10,:) + 1
```

You can prevent the transformation by coding the shift as a serial loop:

```
      PROGRAM BETTER_7
      INTEGER A(10,1000), I
CMF$LAYOUT A(:SERIAL,:NEWS)
      ...
      DO I=1,9
         A(I,:) = A(I+1,:) + 1
      ENDDO
```

A caveat for using loops to perform communication: As with any serial in-place algorithm for array movement, the loop direction must match the direction of data. In **BETTER_7**, if the right-hand side had used `I-1` instead of `I+1`, the loop would have had to gone from 9 to 1 rather than 1 to 9.

Unwinding Loops

- ♦ Unwind the loops that express communication along serial dimensions.

You can improve communication performance further on serial dimensions by unwinding the serial loops. This action increases elemental code block size and thus helps amortize the start-up overhead for the subgrid loop. (Recall from Section 2.3, however, that you should probably not unwind beyond about 18 serial coordinates, since using too many coordinates might cause a register spill.)

```
      PROGRAM BEST_7
      INTEGER A(10,1000), I
CMF$LAYOUT A(:SERIAL,:NEWS)
      ...
      DO I=1,9,5
        A(I,:)   = A(I+1,:) + 1
        A(I+1,:) = A(I+2,:) + 1
        A(I+2,:) = A(I+3,:) + 1
        A(I+3,:) = A(I+4,:) + 1
        A(I+4,:) = A(I+5,:) + 1
      ENDDO
```

The reason for the improved performance is that the optimizer is currently unable to move the serial axis address calculations out of the inner loop. If you unwind the loop, the address calculations can be shared across the whole computation, which reduces the workload on the front end.

Note that in unwinding a loop, you may need to add cleanup code to handle any leftover elements. For example, if the serial dimension in program **BETTER_7** were size `11` and the program moved data from `1:11` instead of from `1:10`, you would need an extra assignment statement after the unwound loop to handle the leftover element 11.

For the Future

A future release of CM Fortran will implement communication along serial dimensions more efficiently. At that point, the original approach, shown in program **OKAY_7**, will become preferable to both **BETTER_7** and **BEST_7**.

Another Example

Here is another example of coding communication along serial axes as a loop to improve performance. This program sums along the serial dimension of a 2-dimensional array, producing a 1-dimensional output array:

```
      PROGRAM SLOW_SUM
      INTEGER A(16,10), B(16,10), C(10)
CMF$LAYOUT A(:SERIAL,:NEWS), B(:SERIAL,:NEWS), C(:NEWS)
      ...
      C = SUM(A*B,DIM=1)
```

Left to its own devices, the compiler will calculate A*B into a temporary and then perform the summation. However, you can use a serial loop to perform the same operation without the temporary:

```
      PROGRAM FAST_SUM
      INTEGER A(16,10), B(16,10), C(10), I
CMF$LAYOUT A(:SERIAL,:NEWS), B(:SERIAL,:NEWS), C(:NEWS)
      ...
      C=0
      DO I=1,16
        C = C + A(I,:) * B(I,:)
      ENDDO
```

Finally, unwind the loop for even better performance:

```
      PROGRAM FASTEST_SUM
      INTEGER A(16,10), B(16,10), C(10), I
CMF$LAYOUT A(:SERIAL,:NEWS), B(:SERIAL,:NEWS), C(:NEWS)
      ...
      C=0
      DO I=1,16,4
        C = C + A(I,:)   * B(I,:)
        C = C + A(I+1,:) * B(I+1,:)
        C = C + A(I+2,:) * B(I+2,:)
        C = C + A(I+3,:) * B(I+3,:)
      ENDDO
```

Computation on Serial Dimensions

- Avoid using serial loops to express computation on serial dimensions.

The compiler excels at performing elemental computation (as opposed to communication) along serial dimensions — this is the rationale not only for serial dimensions but for the slicewise model itself. It is a pessimization to use serial loops to express such computation. For example, this non-looping program:

```
      PROGRAM FAST_COMPUTE
      INTEGER A(16,10), B(16,10)
CMF$LAYOUT A(:SERIAL,:NEWS), B(:SERIAL,:NEWS)
      ...
      A = A * B
```

executes significantly faster than this looping program:

```
      PROGRAM SLOW_COMPUTE
      INTEGER A(16,10), B(16,10), I
CMF$LAYOUT A(:SERIAL,:NEWS), B(:SERIAL,:NEWS)
      ...
      DO I=1,16
        A(I,:) = A(I,:) * B(I,:)
      ENDDO
```

Chapter 6

Example: Complex Matrix Multiply

This chapter contains an example of elemental code that illustrates some of the performance factors described in this manual. This subroutine multiplies, within each PE, a 3 x 3 matrix of complex numbers by a vector of 3 complex numbers. The computation occurs in **AXISLEN** positions on a parallel dimension; no communication takes place. The program performs this operation 1000 times and prints out the overall time. It repeats this timing operation 3 times.

This chapter shows a plain and optimized version of the source program, as well as the timings for these programs under a variety of conditions: single-precision and double-precision complex values, varying array sizes (that is, different values for **AXISLEN**), and slicewise versus Paris compilation.

6.1 Source Programs

Here is the "plain" program. Since a complex add takes two operations and a complex multiply takes six, the timed section of this program performs 1000 * 8 * 3 * 3 = 72,000 floating-point operations for each of the **AXISLEN** elements.

Example 1. Subroutine **MM_PLAIN** source code

```
      SUBROUTINE MM_PLAIN
      IMPLICIT NONE
      INTEGER N,AXISLEN,TIMES,I,J,K
      PARAMETER (N=3)
      PARAMETER (AXISLEN=16384)
      COMPLEX A(N,N,AXISLEN), B(N,AXISLEN), C(N,AXISLEN)
```

```
CMF$LAYOUT A(:SERIAL,:SERIAL,:NEWS)
CMF$LAYOUT B(:SERIAL,:NEWS)
CMF$LAYOUT C(:SERIAL,:NEWS)

      FORALL(I=1:3, J=1:3) A(I,J,:) = I+1000*J
      FORALL(I=1:3) B(I,:) = I
      C = 0
      DO TIMES =1,3
        CALL CM_TIMER_CLEAR(1)
        CALL CM_TIMER_START(1)

        DO I= 1,1000
           DO J=1,3
              DO K=1,3
                 C(J,:) = C(J,:)+A(J,K,:)*B(K,:)
              ENDDO
           ENDDO

C Without the following, the optimizer would move the
C computations out of the timing loop, or even eliminate
C them, since their results are never used.

           CALL TOUCH( C,A,B )
        ENDDO

        CALL CM_TIMER_STOP(1)
        CALL CM_TIMER_PRINT(1)
      ENDDO
      RETURN
      END

      SUBROUTINE TOUCH( C,A,B )
      INTEGER C,A,B
      RETURN
      END
```

The following subroutine performs the same operations and produces the same results as the one above. The difference is that the serial loops are unwound. This program's inner core also performs 72,000 floating-point operations per parallel element.

Example 2. Subroutine **MM_UNWOUND** source code

```
      SUBROUTINE MM_UNWOUND
      IMPLICIT NONE
      INTEGER N,AXISLEN,TIMES,I,J,K
      PARAMETER (N=3)
      PARAMETER (AXISLEN=16384)
      COMPLEX A(N,N,AXISLEN), B(N,AXISLEN), C(N,AXISLEN)
CMF$LAYOUT A(:SERIAL,:SERIAL,:NEWS)
CMF$LAYOUT B(:SERIAL,:NEWS)
CMF$LAYOUT C(:SERIAL,:NEWS)
      FORALL(I=1:3, J=1:3) A(I,J,:) = I+1000*J
      FORALL(I=1:3) B(I,:) = I
      C = 0
      DO TIMES =1,3
         CALL CM_TIMER_CLEAR(1)
         CALL CM_TIMER_START(1)

         DO I= 1,1000
            C(1, :) = A(1,1,:) * B(1,:)    + A(1,2,:) * B(2,:)
     $                + A(1,3,:) * B(3,:) + C(1,:)
            C(2, :) = A(2,1,:) * B(1,:)    + A(2,2,:) * B(2,:)
     $                + A(2,3,:) * B(3,:) + C(2,:)
            C(3, :) = A(3,1,:) * B(1,:)    + A(3,2,:) * B(2,:)
     $                + A(3,3,:) * B(3,:) + C(3,:)
            CALL TOUCH(C,A,B)
         ENDDO

         CALL CM_TIMER_STOP(1)
         CALL CM_TIMER_PRINT(1)
      ENDDO
      RETURN
      END
```

6.2 Peak Flops Rates

This section shows the peak Flops rates for the timed sections of the two programs, calculated with the technique shown in Appendix A. When compiled with single-precision complex values, the cycle counts for the respective elemental code blocks are:

```
MM_PLAIN,  elemental code block total:   101 cycles
MM_UNWOUND,elemental code block total:   607 cycles
```

The elemental block from **MM_PLAIN** performs 8 vector operations, each of length 4. The block from **MM_UNWOUND** code performs 72 vector operations. Assuming a clock rate of 150 nanoseconds and a 64K CM (2048 PEs), the peak Flops rate:

```
MM_PLAIN:   ((2048 * 4 * 8)  / (101 * 150))   =  4.326 Gflops
MM_UNWOUND:((2048 * 4 * 72) / (607 * 150))   =  6.478 Gflops
```

When the programs are recompiled using double-precision complex values, the peak Flops rates are:

```
MM_PLAIN_DBL,    elemental code block total:149 cycles
MM_UNWOUND_DBL,  elemental code block total:875 cycles
```

```
MM_PLAIN_DBL: ((2048 * 4 * 8)  / (149 * 150)) = 2.932 Gflops
MM_UNWND_DBL: ((2048 * 4 * 72) / (875 * 150)) = 4.494 Gflops
```

6.3 Timings

The timings shown below for the two programs are for a 64K CM with a VAX front end. The programs were executed on an 8K CM and the results extrapolated (the extrapolation is linear).

These tables allow you to compare timings for the plain versus unwound loops, for single-precision and double-precision complex values, for various array sizes, and for the Paris and slicewise execution models.

The array sizes shown reflect **AXISLEN** values that are large and small, as well as power-of-2 and non-power-of-2. Sizes smaller than 8K for the parallel dimension would be unfair to the Paris model, since Paris would begin to leave some of the FPUs unused while the slicewise model would continue to use all the FPUs until the **AXISLEN** shrank below 256. (Slicewise would thus outperform Paris by an absurdly large number).

The Flops ratings are in Gflops. The first Gflop rate in each pair is based on CM busy time; the second is for wallclock time.

Table 1. Single-precision Gflops for
complex 3 x 3 matrix-vector multiply

AXIS				**Slicewise**
LENGTH	**Paris Plain**	**Paris Unwound**	**Slicewise Plain**	**Unwound**
8192	.923/.340	1.15/.763	2.89/2.26	6.03/6.03
10000	.720/.380	1.03/.902	3.39/3.30	5.88/5.88
16384	1.36/.660	1.74/1.48	3.98/3.98	5.99/5.99
32768	1.69/1.31	2.18/2.18	4.03/4.03	6.01/6.01
65536	1.89.1.87	2.35/2.35	4.05/4.05	6.00/6.00
80000	.985/.980	1.44/1.43	4.01/4.01	5.89/5.89
Calculated Kernel Peak	?	?	4.33	6.48

Table 2. Double-precision Gflops for
complex 3 x 3 matrix-vector multiply

AXIS				**Slicewise**
LENGTH	**Paris Plain**	**Paris Unwound**	**Slicewise Plain**	**Unwound**
8192	.689/.302	.837/.623	1.15/.740	2.42/2.27
10000	.467/.347	.618/.607	1.21/.86	2.48/2.48
16384	.852/.601	1.03/1.00	2.05/1.89	4.30/4.30
32768	.965/.965	1.08/1.08	2.83/2.83	4.32/4.32
65536	1.0/1.0	1.11/1.11	2.86/2.86	4.33/4.33
80000	.561/.550	.670/.647	2.81/2.81	4.28/4.28
Calculated Kernel Peak	?	?	2.93	4.49

Appendix A

Calculating Peak Flops Rate

To calculate the peak Flops rate of a section of elemental code, begin by compiling using the switches **-S**, **-pecode**, and **-slice**. The generated **.peac** file contains compiled routines for the PEs (PECODE procedures) corresponding to each section of elemental code. The PECODEs have names of the form **$_pname_pe_code_n** where **pname** is the user's procedure name and **n** is a number. You can use the **.s** file, which contains line number information, to figure out which PECODE your elemental code block has been compiled into. (For a small program, the relationship between source and compiled code may be obvious in the **.peac** file.)

Locate the PECODE in the **.peac** file. Each line has a vector Flops count and cycle count, and they are totalled at the end of the PECODE (do not be misled by the totals for the POPs that come closer to the beginning).

The peak Flops rate can be calculated from TOTAL-FLOPS, TOTAL-CYCLES, and the number of FPUs executing the program:

```
Flops rate = number-FPUs * Flops-rate-per-FPU

Flops-rate-per-FPU = vector-length
                     * TOTAL-FLOPS/(TOTAL-CYCLES * cycle-time)
```

where:

```
number-FPUs = number-processors/32
vector-length = 4
cycle-time = 150ns
```

Thus, if the **.peac** file indicates that an elemental code block contains 72 vector Flops and is 607 cycles in length, the Flops rate for a 64K CM is:

```
2048 * 4 * 72 / (607 * 150)   =  6.478 Gflops
```

Index

Index