POLYSHIFT Communications Software for the Connection Machine System CM-200

WILLIAM GEORGE[1,2], RALPH G. BRICKNER[1], AND S. LENNART JOHNSSON[3,4]

[1]*Los Alamos National Laboratory, Los Alamos, NM 87545*
[2]*Department of Computer Science, Clemson University, Clemson, SC 29634*
[3]*Thinking Machines Corporation, Cambridge, MA 02142*
[4]*Division of Applied Sciences, Harvard University, Cambridge, MA 02138*

ABSTRACT

We describe the use and implementation of a polyshift function PSHIFT for circular shifts and end-offs shifts. Polyshift is useful in many scientific codes using regular grids, such as finite difference codes in several dimensions, and multigrid codes, molecular dynamics computations, and in lattice gauge physics computations, such as quantum chromodynamics (QCD) calculations. Our implementation of the PSHIFT function on the Connection Machine systems CM-2 and CM-200 offers a speedup of up to a factor of 3–4 compared with CSHIFT when the local data motion within a node is small. The PSHIFT routine is included in the Connection Machine Scientific Software Library (CMSSL). © 1994 John Wiley & Sons, Inc.

1 INTRODUCTION

Efficient and minimal data motion is critical for high performance in most computer architectures. The polyshift function presented in this article addresses this issue. The impact of the data motion on performance depends on the memory architecture of the system. Memory systems have been slower than processors, almost as long as electronic computers have been built. Although the technological reasons for this fact have changed over time, it is expected to be the case also for the foreseeable future. Memory hierarchies (registers, cache, main memory, etc.) and parallel memories (banked and interleaved memories) have been used extensively for a long time to achieve a desired level of performance at an acceptable price. The efficiency of these architectures depends critically on locality of reference.

Massively parallel supercomputer architectures achieve the required memory bandwidth by using thousands of processing units with local memories. We refer to a processor, its local memory, and associated communications circuitry as a node. A communications system interconnects the nodes. Preserving locality of reference assumes several new characteristics in distributed memory architectures. Data placement among memory modules affects the lower bounds for latency and bandwidth. The routing disciplines determine how close to the bounds the actual data motion time is.

The goal in allocating data to the memory units is to make most data references be references to local memory, yet achieve good load balance. Whenever references must be nonlocal, then the placement should be such that the communication time is minimized with a good (optimal) rout-

Received February 1992
Accepted January 1994

Scientific Programming, Vol. 3, pp. 83–99 (1994)
CCC 1058-9244/94/010083-17

ing strategy. Ideally, data is mapped to nodes such that nonlocal references always are references to adjacent nodes. The ability to accomplish this task depends both on the data reference pattern and the network topology (mesh, binary cube, tree, ring, etc.). The access time to data in nonlocal memory depends both on the network topology and the routing mode (e.g., circuit switched, packet switched, or wormhole routing [1]).

Many problems in the natural and mathematical sciences and in engineering can be solved by discretizing the governing equations onto a regular grid (lattice) in two, three, or several dimensions. One such example is quantum chromodynamics (QCD) calculations, which use a four-dimensional space-time regular lattice. The computational requirements for QCD are enormous. The desired lattice sizes are of the order of 100 million grid points. For each such grid a range of parameter values must be covered. Each set of parameters, known as a configuration, requires 10^{14} to 10^{15} floating-point operations [2]. Clearly, a high efficiency in utilizing the computational and communication resources in highly desirable. An early implementation of a QCD code on the Connection Machine system CM-2 resulted in a performance of 0.9 Gflop/s in 32-bit precision for a 2,048 node configuration. Code restructuring and other optimizations improved the performance by close to a factor of six to 5.2 Gflop/s.

A large fraction of the performance enhancement in the QCD application was due to code restructuring to allow for concurrent bidirectional communications in each of four dimensions simultaneously and to avoid extraneous local memory moves. The polyshift function described here is a generalization of the communications routines developed for the QCD application. The polyshift function is included in the Connection Machine Scientific Software Library (CMSSL) [3, 4] as the routine PSHIFT.

The PSHIFT routine is critical for the performance of many scientific programs based on finite difference techniques, multigrid techniques, as well as molecular dynamics applications. In this article, we describe this software, along with features of the Connection Machine system CM-200 that support it, and give performance numbers and analyses. The PSHIFT software technology is also used in a special compiler known as the stencil and convolution compiler, now available in CMSSL for the CM-2 and CM-200. A prototype version of this compiler was described by Bromley et al. [5]. The stencil compiler will be described elsewhere.

In Section 2, we describe the programming model of the CM-200 used by the compilers and the run-time system, as well as the hardware features that are used for the implementation of PSHIFT. Section 3 describes the software architecture of the PSHIFT routine and Section 4 discusses its implementation in some more detail. Section 5 describes the interface of the PSHIFT library routine to Connection Machine Fortran [6, 7], a subset of Fortran 90 [8] with extensions. Calling sequences and supported functionality are reported. Section 6 presents some performance measurements and a performance model. We conclude with a section summarizing our experience from developing and using the PSHIFT library routine, and discussing possible future enhancements and generalizations.

2 THE CONNECTION MACHINE MODEL CM-200

2.1 Data Allocation

The CM-2 and CM-200 support a programming model with a global address space. (For the remainder of this article, unless otherwise stated, descriptions of the CM-200 hardware and software will also apply to the CM-2.) Data arrays declared in any of the supported languages are by default distributed evenly over all memory units. The default allocation of arrays to memory units is entirely based on array shape. This allocation is known as a canonical layout, and is determined by the geometry manager at run-time. For each array, the nodes are configured as an array of nodes with the same rank as the data array. Thus, for one-dimensional data arrays the nodes form a linear array; for a two-dimensional data array, the nodes form a two-dimensional array of nodes, etc.

The geometry manager also decides which elements are mapped into the same node, and to which node each aggregate of data is mapped. On the Connection Machine systems, a set of consecutive elements [9] along each axis are mapped into the same memory unit. If the number of elements along an axis is not evenly divisible by the number of nodes assigned to that axis, then some nodes may not be assigned any elements. Of the nodes that are assigned elements, all but one receive the same number of elements. Thus, for a

one-dimensional data array of M elements mapped to N nodes, $\lceil \frac{M}{N} \rceil$ successive elements are assigned to the first $\lfloor M/\lceil \frac{M}{N} \rceil \rfloor$ nodes. On nodes that receive fewer than $\lceil \frac{M}{N} \rceil$ elements, memory is nevertheless allocated for this number of elements. For rank two or higher arrays, one or more axes may be padded in this manner. Nonvalid data elements are identified by setting bits of a garbage mask. This mask has one bit for each data element.

Other mappings of interest with respect to either load balance (such as in LU decomposition [9–12]) or communications requirements (such as in FFT computations [13, 14] are cyclic mapping [9], and combinations of consecutive and cyclic mappings, such as block cyclic mappings [10, 15–17]. The proposed languages Fortran D [18] and High Performance Fortran [19] support such data allocations. However, the current Connection Machine languages do not support cyclic mappings, or combinations thereof with the consecutive mapping.

The consecutive allocation defines which elements are assigned to the same node, but does not specify how aggregates of elements are assigned to nodes. On the CM-200, the default assignment is such that a pair of successive indices along any axis are either mapped into the same memory unit, or into the memory units of adjacent nodes. This mapping, known as NEWS order, uses a binary-reflected Gray code [20, 21] for the encoding of node addresses. The standard binary encoding is referred to as SEND order. In the default NEWS order, allocation blocks i and $i + 1$ are assigned to adjacent nodes, while in the SEND order allocation block i is assigned to node i. In a SEND ordered assignment, blocks $N/2 - 1$ and $N/2$ are assigned to nodes at a distance of $\log_2 N$ apart. The LAYOUT compiler directive is available to individually choose which order is to be used for the axes of a given data array.

The default strategy for assigning nodes to axes of multidimensional data arrays is to make the local length of each of the data array axes the same, if possible. For example, in two dimensions the geometry manager attempts to make the local data array approximately square, and in three dimensions it attempts to make the local array approximately a cube. This strategy minimizes the average number of remote references per local reference when the references along the different array axes are equally frequent for all array elements. In other words, this minimizes the surface area of a subgrid for a given volume. For example, a 32×32 array mapped onto a four-dimensional binary cube would result in subarrays on each node of size 8×8, i.e., each instance of an axis being assigned a two-dimensional subcube. However, because of certain low level details of the architecture, the geometry manager does not always succeed in creating subarrays with axes of equal lengths, even when that should be possible.

The shape of the local arrays, but not their size, is controlled by assigning weights to the various axes using the LAYOUT compiler directive. The number of local array elements is not affected by the LAYOUT directives as long as the array size is such that the lengths of the axes of the selected processing array shape divide the corresponding data array axes. A high weight for an axis relative to the weight of other axes increases its local length at the expense of the length of the other axes. The SEND and NEWS order specifiers have no effect on the local array shape. They only affect the node to which subarrays are assigned.

Further control of axes layouts can be obtained by using the SERIAL specifier in the LAYOUT compiler directive. This specifier forces an axis to be entirely local to a memory unit, so that a distinct copy of the entire axis resides in every memory unit. Finally, the allocation of an array can also be controlled by the ALIGN compiler directive. With this directive one array is aligned with another array in a specified way. The run-time system maintains information about array type, location, shape, and layout in an array descriptor for each array.

2.2 Data Motion Primitives

Fortran 90 and CM Fortran provide the intrinsic functions CSHIFT and EOSHIFT for circular shift and end-off shift. These intrinsic functions shift an entire Fortran array in a given dimension by a given amount. CSHIFT is a circular shift, whereas EOSHIFT is an end-off shift, with incoming boundary data specified as literal data, or a scalar or array-valued variable. Because these functions are the array-syntax expressions of offset array indices, their use is very common in scientific programming.

The PSHIFT routine allows a user to specify, in a single statement, data motion equivalent to one or more calls to CSHIFT and EOSHIFT. The CM-2 and CM-200 implementations of PSHIFT also offer enhanced performance, compared with calls to

CSHIFT and EOSHIFT through an efficient implementation of multiple communication operations.

In the CM-200 systems, an exchange of data between adjacent nodes requires the same time as a one-way communication between a pair of adjacent nodes. Moreover, communication along several axes can be performed concurrently. This ability for concurrent communication between nodes is exploited by PSHIFT to efficiently perform multiple shifts when compared to CSHIFT. The specific node hardware that supports the concurrent exchange of data is described next.

2.3 Node Architecture

The CM-200 can have up to 2,048 nodes, each of which is equipped with a 32- or 64-bit (internal) floating-point unit (FPU), 4 MBytes of dynamic random access memory (DRAM) operated in page mode, and a network interface. There is also a set of 32 bit-serial processors, and some associated hardware for translating data between the bit-serial representation and the 32-bit wide representation required by the FPU. Sixty-four-bit operations are supported by the 64-bit FPU, but the data paths external to the FPU are 32 bits wide. This requires consecutive loads and stores of the two 32-bit halfwords associated with each 64-bit floating-point word. During memory loads, a DRAM page fault adds one cycle. That is, striding out of a 4k (32-bit) word DRAM page causes a single-cycle stall. Stores always require close to two cycles. The clock frequency is 10 MHz (7 MHz for the CM-2).

The bit-serial processors are physically arranged on two "CM Chips," each of which is the terminus for 12 communications wires. One wire on each CM Chip is connected to the other CM Chip on the node, the others go to other nodes. Because of this doubling of the CM Chips and their associated communications wires, the machine is effectively interconnected as a binary hypercube with up to 11 dimensions, depending on machine size, and with connections between adjacent nodes consisting of a pair of 1-bit wide channels. Figure 1 gives a block diagram of a node and Figure 2 shows how a three-dimensional cube of CM Chips can be thought of as a two-dimensional cube with a pair of channels between each pair of nodes. The CM-200 is described in detail [22].

The basic communications operation between a pair of adjacent nodes is an exchange, i.e., a pair of nodes can exchange data in the time required for one of the nodes to send data to the other node. Moreover, it is possible to concurrently exchange data on all channels of a node. In a 2,048-node system, up to 22 (2×11) data elements can be exchanged concurrently. The network interface contains three register files of thirty-two 32-bit registers each, known as "transposers" A, B, and C. The transposers function also to convert data between the 1-bit wide bit-serial representation (employed by the bit-serial CM processors), and the 32-bit wide representation required by the FPU. In this case, each transposer converts a block of 32 bit-serial words of 32 bits, to a 32-bit parallel word, and vice versa. When functioning as part of the communications system, the transposers serve as the source and destination of the data exchanged between a pair of adjacent nodes. In a typical operation, transposer A is loaded with two 32-bit data elements for each adjacent node to which data shall be sent. After the exchange, transposer B contains the data received—two 32-bit data elements from each adjacent node. "Slots" 1 through 11 in the

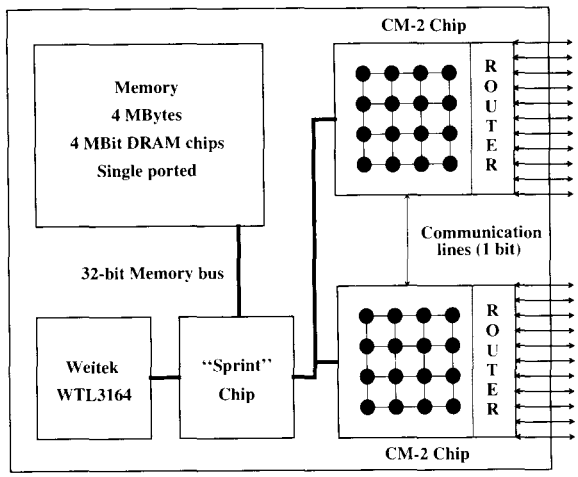

FIGURE 1 A CM-2/200 floating-point processor node.

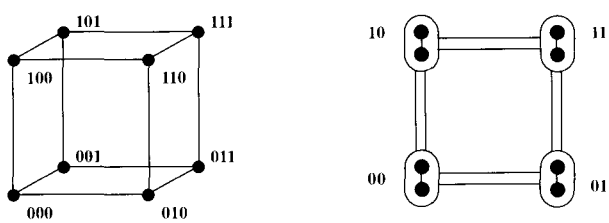

FIGURE 2 Eight nodes can be thought of as a three-dimensional hypercube, or as a double two-dimensional hypercube with connections internal to a node.

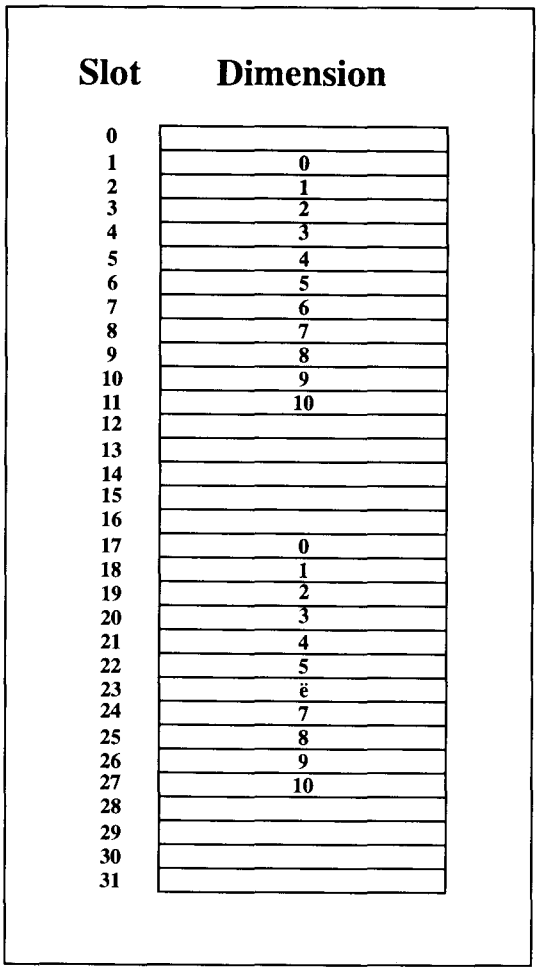

FIGURE 3 Transposer slots and hypercube dimensions.

transposer register file correspond to one set of eleven 1-bit channels, and slots 17 through 26 correspond to the other set of eleven 1-bit channels. The correspondence between transposer slots and cube dimensions is illustrated in Figure 3.

2.4 Virtual Machine Model

The Connection Machine systems implement a virtual machine model in which one data element (e.g., a CM Fortran array element) is assigned to one virtual processor. Conceptually, the user data arrays are fully distributed amongst the virtual processors, and all operations take place fully in parallel. However, to actually implement a data distribution that assigns one data element to one physical processor would result in highly inefficient use of the hardware, and also the inability to run different sized problems on the same system configuration. The virtual machine model allows the distribution of a large number of data elements onto a smaller number of physical processors, so that multiple data elements are assigned to a single physical processor, alleviating these problems. (Details of this distribution were discussed above.)

Now consider a typical shift operation in CM Fortran. The operation `CSHIFT(A, 1, 1)` implies that all references to `A(I, :, :)` after the shift operation reference element `A((I+1)mod N, :, :)` prior to the shift, where N is the length of the shifted axes. In the current implementation of `CSHIFT` all elements of A are moved precisely as stated in the call to the function `CSHIFT`. `EOSHIFT` is implemented analogously, and so is `PSHIFT`. But, because, in general, there are many array elements assigned to each node, the virtual machine model results in a large number of memory moves local to a node. As the size of a subarray increases, the time for local memory moves becomes comparable to, and eventually exceeds, the time for communication. Thus the performance advantage of `PSHIFT` decreases with increasing size of the local subarrays.

However, the local memory moves could be avoided by suitable manipulation of pointers, i.e., by appropriately modifying the address calculation in referencing array elements. The CM-200 compilers do not currently maintain a set of pointers. In the `PSHIFT` function, the data motion between nodes and the support of the virtual machine model are implemented as separate modules. Both modules are required for consistency with the memory model used by the CM Fortran compiler.

3 PSHIFT SOFTWARE ARCHITECTURE

The polyshift operation is supported by the three routines `PSHIFT`, `PSHIFT_SETUP`, and `DEALLOCATE_PSHIFT_SETUP`. The `PSHIFT` routine is further divided internally into modules corresponding to

1. The data motion between nodes
2. Local data motion to support the virtual machine model
3. The boundary conditions

We now describe the routines in more detail.

The purpose of the PSHIFT_SETUP function is to carry out operations that are the same for each call to the routine PSHIFT, which performs the shift operations, and thus can be performed once and for all. Largely, the PSHIFT_SETUP function performs address calculations and generates a custom routine to accomplish the requested set of shifts. On the CM-200, each array axis is assigned to a subcube of nodes. For a data array mapped to the nodes in NEWS order, traversing a given data array axis corresponds to traversing the binary subcube to which it is allocated. For a shift operation it is necessary to determine, for each node, the dimensions for incoming and outgoing data in the positive and negative axis direction. This information is held in communication tables on the CM. One table, of at most four 32-bit words, is required for each shift. The fact that different cube dimensions are used for an axis is clearly seen in Figure 2.

In order to perform the required calculations the PSHIFT_SETUP function must gather information about the mapping of the array to the nodes and the array data type from the array descriptor. Although the axis encoding (NEWS or SEND) the data type and the array rank are all known at compile-time, neither the actual array extents nor the machine size is known until runtime. Using these final pieces of information, the geometry manager determines the actual layout of the arrays. It is only at this point that PSHIFT has enough information to perform the actual setup functionality required by the given arrays. All data arrays having the same shape, same data type, and same layout directives are mapped to the CM-200 nodes in the same way. Thus, one call to the setup routine suffices for a given set of shifts for all such arrays.

The PSHIFT routine performs the actual data motion. The data movement is performed in three steps:

1. Perform all of the on-node memory-to-memory moves (in support of the virtual machine model)
2. Perform the required exchange of data between nodes
3. If needed, move the boundary data for EOSHIFT to the appropriate nodes

The first step performs any local memory moves that are needed. If the source and destination arrays are the same for any shift, then the source is first copied to a temporary array to avoid overwriting data needed in step 2. No local memory moves are required when the shift distance is greater than the length of the local axis segment for an axis. If this is not the case, local memory moves can only be avoided by code restructuring in which the boundary is extracted from the original array.

The second step performs the data motion required to exchange data between nodes. When data is needed from an adjacent node, a single exchange across the hypercube channels is needed. PSHIFT can handle any shift distance up to twice the subgrid extent along the shift axis and shifts of distance $S = L2^k$ for $k \geq 1$, where L is the length of the subgrid along the shift axis. Shifts of length $S = L2^k$ are referred to as power-of-2 shifts. For such shifts, all array elements are moved a distance of two nodes (require two exchanges). This fact is due to the properties of the binary-reflected Gray coding used in distributing the array elements.

The third step moves boundary data for EOSHIFT into the appropriate nodes. This step is only required for end-off shifts with array or scalar variable boundaries. For end-off shifts, PSHIFT also allows either a constant 0 or a constant 1 to be specified for the boundary values. These boundaries can often be handled without requiring this third step. This aspect of PSHIFT is discussed further in the next section.

The subfunction DEALLOCATE_PSHIFT_SETUP deallocates the data structures allocated in the PSHIFT_SETUP routine. The only CM memory allocated by PSHIFT_SETUP is for the communication tables, because these tables are different for each processor node. Temporary arrays, if needed by PSHIFT, are allocated and deallocated on each call to PSHIFT.

4 PSHIFT IMPLEMENTATION

In this section we describe the implementation details of the PSHIFT routines. The desired level of control over data motion required that much of the code for the CM-200 be expressed in Connection Machine Instruction set (CMIS). The remainder of the polyshift code was written in C. Lying somewhere between assembly language and microcode, CMIS allows low-level control of the features of a Connection Machine system without a need for the programmer to be concerned with the lowest level details, such as setting up pipelines in the node. The CMIS functionality includes mem-

ory-to-memory transfers, memory-FPU pipelines, and special instructions for concurrent communication on multiple channels of a node.

The PSHIFT_SETUP routine accepts a set of arguments that specify a prototypical array and a set of shifts to be performed on arrays of the same size and layout as the prototype. This information is used to generate a custom (internal macro procedure [IMP] consisting of CMIS and IMP instructions) that will perform the shifts requested. Communication tables are computed for each node. These tables are used by the IMP to load and unload the correct transposer slots for each exchange of data across the hypercube connections. The IMP is written to a file, assembled, and then loaded into the static random access memory (SRAM) of the sequencer(s) for the CM-200 configuration being used. The PSHIFT_SETUP function returns an `integer` ID, which is used as an argument to the PSHIFT routine to identify the previously geneated IMP. The same ID can be used for all calls to PSHIFT for the same set of shifts on any arrays with the same layout and data type as the prototype array passed to PSHIFT _SETUP. Different calls to the PSHIFT_SETUP function and different IDs are needed for different array layouts and/or a different set of shifts.

The IMPs generated by PSHIFT_SETUP perform the following steps:
Repeat until all exchanges are completed

1. Load transposer A, indirectly with communication tables, from the source arrays
2. Exchange data with adjacent nodes
3. Unload transposer B, indirectly with communication tables, to the destination arrays

The IMPs generated by the PSHIFT_SETUP function perform all of the communications required by the shifts, except for assigning the array, scalar, and in some cases, constant 1.0 and 0.0 values to the boundaries. These boundary values are assigned with calls to Connection Machine run-time library routines after the IMP has completed. Boundaries assigned the constant 0.0 are handled in the IMPS by loading 0s into transposer A slot 0 and modifying the communication tables to read from transposer B slot 16 for nodes that require the boundary elements. Each boundary node in effect sends the appropriate constant to itself. This communications path is an artifact of the original CM-2 bit-serial architecture and was used to communicate between the pair of CM-2 Chips found on each node (see Fig. 1). The same technique is used if all end-off shifts specity a constant boundary value of 1.0 (loading a 1.0 instead of a 0.0 into transposer A slot 0). In the case that some end-off shifts specify a constant 1 boundary and some specify a constant 0.0, then the constant 0.0 boundaries are handled in the IMP and the constant 1.0 is broadcast to those boundary elements requiring the value, after the IMP has completed.

Due to limited space in SRAM, and for performance reasons, there are actually two setup routines. The PSHIFT_SETUP routine generates IMPs in which all loops are completely unrolled. The PSHIFT_SETUP_LOOPED routine generates IMPs in which the loops are unrolled only slightly, e.g., four data exchanges are performed in each iteration of a loop. The PSHIFT_SETUP routine generates faster routines due to the high overhead of loop control statements in IMPs (especially on the CM-2) at the expense of larger IMPs and higher setup times. In fact, the PSHIFT_SETUP routine can fail in some cases if it produces an IMP that is too large to fit into the available SRAM. In these cases the PSHIFT_SETUP_LOOPED routine must be used. The calling convention is the same for PSHIFT_SETUP and PSHIFT_SETUP_LOOPED. Other restrictions and performance considerations are discussed in more detail in the next two sections.

For shifts that cannot be handled by PSHIFT, PSHIFT_SETUP sets an internal flag, causing the shift to be performed through calls to the Connection Machine run-time library. If none of the requested shifts can be handled by IMPs then the performance of PSHIFT will be approximately the same as for the equivalent set of calls to the intrinsic routines CSHIFT and EOSHIFT. For a shift along any axis a, with subgrid axis extent L, and a shift distance S, a shift will be handled by an IMP if the following restrictions are met:

1. Axis a is not padded.
2. Axis a is distributed in NEWS order.
3. $|S| \leq 2L$ or $S = L(2^k)$ for $k \geq 1$.

Also, no more than two shifts are allowed per axis.

5 PSHIFT CM FORTRAN INTERFACE

PSHIFT is accessed from CM Fortran using the cals described in Figure 4. The shift-type arguments are predefined constants defined in the CMSSL header file "cmssl-cmf.h". They may

```
      include '/usr/include/cm/cmssl-cmf.h'

      integer setup_id

      setup_id = PSHIFT_SETUP (n, cm_array, ier,
     &    shift_type_1, dim_1, dist_1,
     &    shift_type_2, dim_2, dist_2,
            .
            .
            .
            .
     &    shift_type_n, dim_n, dist_n)

      CALL PSHIFT (n, setup_id, ier,
     &    shift_type_1, dst_array_1, src_array_1, dim_1, dist_1[, bdry_1],
     &    shift_type_2, dst_array_2, src_array_2, dim_2, dist_2[, bdry_2],
            .
            .
            .
            .
     &    shift_type_n, dst_array_n, src_array_n, dim_n, dist_n[, bdry_n])

      CALL DEALLOCATE_PSHIFT_SETUP (setup_id)
```

FIGURE 4 PSHIFT calling sequences.

be one of CMSSL_CSHIFT, CMSSL_EOSHIFT_SCALAR, CMSSL_EOSHIFT_ARRAY, CMSSL_EOSHIFT_0, or CMSSL_EOSHIFT_1, in any combination. Boundary values for end-off shifts may be the constant values 0.0 or 1.0, a front-end scalar variable, or a CM array (or rank one less than the source and destination arrays).

To compare the use of CSHIFT and PSHIFT, we present in Figure 5 code fragments representing the calculation of a five-point, two-dimensional stencil. Notice that in using PSHIFT, temporary storage must be managed by the user, whereas in the corresponding code using CSHIFT, the compiler manages temporary storage. In the example, the temporary arrays are named N, E, W, and S. The advantage of PSHIFT lies in the implementation of the required communication. The library routine PSHIFT performs the specified communications concurrently, whereas the CM-200 compilers do not instantiate multiple CSHIFT or EOSHIFT operations concurrently. Thus, in our example, the CM Fortran code requires four communications instead of one.

An optimized CM Fortran compiler would not only perform the required communication concurrently, but also avoid unnecessary local data motion, and optimize the register usage in the floating-point unit. A prototype compiler of this nature has been implemented for stencils applied to two-dimensional arrays. This compiler is known as a stencil or convolution compiler [5]. A generalization of this prototype compiler has now been completed for the CM-200 and is part of CMSSL 3.1 for the CM-2 and CM-200. A CM-5 version is now under development. The CM-2/200 version of the stencil compiler uses a module of the PSHIFT function for data motion between nodes.

For the QCD computations mentioned earlier PSHIFT replaces eight communications performed sequentially by one concurrent communication exchanging data along all four axes at

```
C.... Stencil with CSHIFTs:

      .
      .
      .
C.... Do the communications and computations in one statement
      dst =
     &  cN * CSHIFT (src, 1, -1) +
     &  cE * CSHIFT (src, 2,  1) +
     &  cW * CSHIFT (src, 2, -1) +
     &  cS * CSHIFT (src, 1,  1) +
     &  c0 *          src

      .
      .
      .



C.... Stencil with PSHIFTs:

      .
      .
      .

C.... Define local news arrays
      real*8, array (NX, NY) :: N, E, S, W
cmf$  layout N (:news, :news), E (:news, :news)
cmf$  layout W (:news, :news), S (:news, :news)

      .
      .
      .

C.... Do multiple communications
      call pshift (4, id, ier,
     & CMSSL_CSHIFT, N, src, 1, -1,
     & CMSSL_CSHIFT, E, src, 2,  1,
     & CMSSL_CSHIFT, W, src, 2, -1,
     & CMSSL_CSHIFT, S, src, 1,  1)

C.... Do the computation
      dst = cN*N + cE*E + cW*W + cS*S + c0*src

      .
      .
      .
```

FIGURE 5 Comparison of CSHIFT and PSHIFT usage.

```
      .
      .
      .

      REAL, ARRAY (3,2,NX,NY,NZ,NT) :: XP_SRC, XP_DST
      REAL, ARRAY (3,2,NX,NY,NZ,NT) :: XM_SRC, XM_DST
      REAL, ARRAY (3,2,NX,NY,NZ,NT) :: YP_SRC, YP_DST
      REAL, ARRAY (3,2,NX,NY,NZ,NT) :: YM_SRC, YM_DST
      REAL, ARRAY (3,2,NX,NY,NZ,NT) :: ZP_SRC, ZP_DST
      REAL, ARRAY (3,2,NX,NY,NZ,NT) :: ZM_SRC, ZM_DST
      REAL, ARRAY (3,2,NX,NY,NZ,NT) :: TP_SRC, TP_DST
      REAL, ARRAY (3,2,NX,NY,NZ,NT) :: TM_SRC, TM_DST

CMF$  LAYOUT XP_SRC (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT XP_DST (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT XM_SRC (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT XM_DST (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT YP_SRC (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT YP_DST (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT YM_SRC (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT YM_DST (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT ZP_SRC (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT ZP_DST (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT TP_SRC (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT TP_DST (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT TM_SRC (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)
CMF$  LAYOUT TM_DST (:SERIAL,:SERIAL,:NEWS,:NEWS,:NEWS,:NEWS)

      .
      .
      .

      CALL PSHIFT (8, ID, IER,
     &  CMSSL_CSHIFT, XP_DST, XP_SRC, 3, +1,
     &  CMSSL_CSHIFT, XM_DST, XM_SRC, 3, -1,
     &  CMSSL_CSHIFT, YP_DST, YP_SRC, 4, +1,
     &  CMSSL_CSHIFT, YM_DST, YM_SRC, 4, -1,
     &  CMSSL_CSHIFT, ZP_DST, ZP_SRC, 5, +1,
     &  CMSSL_CSHIFT, ZM_DST, ZM_SRC, 5, -1,
     &  CMSSL_CSHIFT, TP_DST, TP_SRC, 6, +1,
     &  CMSSL_CSHIFT, TM_DST, TM_SRC, 6, -1)

      .
      .
      .
```

FIGURE 6 Four-dimensional communications from QCD using PSHIFT.

once. The use of PSHIFT in a fragment of a QCD application is shown in Figure 6.

A final example is the calculation of a 27-point, three-dimensional stencil, given as a complete subroutine in Figure 7. This example also illustrates the reuse of intermediate communications results in a complicated communications pattern, as indicated by the appearance of several PSHIFT destination arguments as both later PSHIFT source arguments, and as operands in the arithmetic expression.

6 PERFORMANCE

6.1 Timings

In this section we present performance data for the PSHIFT routines. The goal of PSHIFT is to provide improved performance over the intrinsic functions CSHIFT and EOSHIFT. The PSHIFT routine supports the virtual machine model, like CSHIFT and EOSHIFT. For small subarrays of high rank the speedup is expected to be relatively high, whereas for large subarrays of low rank the on-node memory moves will dominate, and PSHIFT is not expected to offer much improve-

```
      subroutine stencil_27 (mx, my, mz, a, x, y)

      include '/usr/include/cm/cmssl-cmf.h'

C.... Parameters
      integer mx, my, mz
      real, array (27, 0:mx, 0:my, 0:mz) :: a
      real, array (    0:mx, 0:my, 0:mz) :: x, y
cmf$  layout a (:serial, :news, :news, :news)
cmf$  layout x          (:news, :news, :news)
cmf$  layout y          (:news, :news, :news)

C.... Local scalars
      integer id1_27,  id2_27,  ier
      logical first_call_27
      data    first_call_27 /.true./
      save id1_27,  id2_27,  first_call_27

C.... Local arrays

      real, array(0:mx, 0:my, 0:mz) ::
     *  xs__0__0__1, xs__0__0_m1,
     *  xs__0__1__0, xs__0__1__1,
     *  xs__0__1_m1, xs__0_m1__0,
     *  xs__0_m1__1, xs__0_m1_m1,
     *  xs__1__0__0, xs__1__0__1,
     *  xs__1__0_m1, xs__1__1__0,
     *  xs__1__1__1, xs__1__1_m1,
     *  xs__1_m1__0, xs__1_m1__1,
     *  xs__1_m1_m1, xs_m1__0__0,
     *  xs_m1__0__1, xs_m1__0_m1,
     *  xs_m1__1__0, xs_m1__1__1,
     *  xs_m1__1_m1, xs_m1_m1__0,
     *  xs_m1_m1__1, xs_m1_m1_m1

C.... If this is the first call, do the setups

      if (first_call_27) then

        first_call_27 = .false.
        id1_27 = pshift_setup (6, X, ier,
     *    CMSSL_CSHIFT, 1, +1,
     *    CMSSL_CSHIFT, 1, -1,
     *    CMSSL_CSHIFT, 2, +1,
     *    CMSSL_CSHIFT, 2, -1,
     *    CMSSL_CSHIFT, 3, +1,
     *    CMSSL_CSHIFT, 3, -1)
        id2_27 = pshift_setup (2, X, ier,
     *    CMSSL_CSHIFT, 3, +1,
     *    CMSSL_CSHIFT, 3, -1)

      endif
```

FIGURE 7 27-point, three-dimensional stencil using PSHIFT.

```
C.... Always do the shifts and the calculation

      call pshift (6, id1_27, ier,
     *  CMSSL_CSHIFT, xs__1__0__0, X,           1, +1,
     *  CMSSL_CSHIFT, xs_m1__0__0, X,           1, -1,
     *  CMSSL_CSHIFT, xs__0__1__0, X,           2, +1,
     *  CMSSL_CSHIFT, xs__0_m1__0, X,           2, -1,
     *  CMSSL_CSHIFT, xs__0__0__1, X,           3, +1,
     *  CMSSL_CSHIFT, xs__0__0_m1, X,           3, -1)
      call pshift (6, id1_27, ier,
     *  CMSSL_CSHIFT, xs__1__0__1, xs__0__0__1, 1, +1,
     *  CMSSL_CSHIFT, xs_m1__0__1, xs__0__0__1, 1, -1,
     *  CMSSL_CSHIFT, xs__1__1__0, xs__1__0__0, 2, +1,
     *  CMSSL_CSHIFT, xs__1_m1__0, xs__1__0__0, 2, -1,
     *  CMSSL_CSHIFT, xs__0__1__1, xs__0__1__0, 3, +1,
     *  CMSSL_CSHIFT, xs__0__1_m1, xs__0__1__0, 3, -1)
      call pshift (6, id1_27, ier,
     *  CMSSL_CSHIFT, xs__1__0_m1, xs__0__0_m1, 1, +1,
     *  CMSSL_CSHIFT, xs_m1__0_m1, xs__0__0_m1, 1, -1,
     *  CMSSL_CSHIFT, xs_m1__1__0, xs_m1__0__0, 2, +1,
     *  CMSSL_CSHIFT, xs_m1_m1__0, xs_m1__0__0, 2, -1,
     *  CMSSL_CSHIFT, xs__0_m1__1, xs__0_m1__0, 3, +1,
     *  CMSSL_CSHIFT, xs__0_m1_m1, xs__0_m1__0, 3, -1)
      call pshift (6, id1_27, ier,
     *  CMSSL_CSHIFT, xs__1_m1__1, xs__0_m1__1, 1, +1,
     *  CMSSL_CSHIFT, xs_m1_m1__1, xs__0_m1__1, 1, -1,
     *  CMSSL_CSHIFT, xs_m1__1_m1, xs_m1__0_m1, 2, +1,
     *  CMSSL_CSHIFT, xs_m1_m1_m1, xs_m1__0_m1, 2, -1,
     *  CMSSL_CSHIFT, xs__1__1__1, xs__1__1__0, 3, +1,
     *  CMSSL_CSHIFT, xs__1__1_m1, xs__1__1__0, 3, -1)
      call pshift (2, id2_27, ier,
     *  CMSSL_CSHIFT, xs_m1__1__1, xs_m1__1__0, 3, +1,
     *  CMSSL_CSHIFT, xs__1_m1_m1, xs__1_m1__0, 3, -1)

      y (:,:,:) =
     *  a ( 1,:,:,:) * xs_m1_m1_m1 + a ( 2,:,:,:) * xs__0_m1_m1 +
     *  a ( 3,:,:,:) * xs__1_m1_m1 + a ( 4,:,:,:) * xs_m1__0_m1 +
     *  a ( 5,:,:,:) * xs__0__0_m1 + a ( 6,:,:,:) * xs__1__0_m1 +
     *  a ( 7,:,:,:) * xs_m1__1_m1 + a ( 8,:,:,:) * xs__0__1_m1 +
     *  a ( 9,:,:,:) * xs__1__1_m1 + a (10,:,:,:) * xs_m1_m1__0 +
     *  a (11,:,:,:) * xs__0_m1__0 + a (12,:,:,:) * xs__1_m1__0 +
     *  a (13,:,:,:) * xs_m1__0__0 + a (14,:,:,:) * x           +
     *  a (15,:,:,:) * xs__1__0__0 + a (16,:,:,:) * xs_m1__1__0 +
     *  a (17,:,:,:) * xs__0__1__0 + a (18,:,:,:) * xs__1__1__0 +
     *  a (19,:,:,:) * xs_m1_m1__1 + a (20,:,:,:) * xs__0_m1__1 +
     *  a (21,:,:,:) * xs__1_m1__1 + a (22,:,:,:) * xs_m1__0__1 +
     *  a (23,:,:,:) * xs__0__0__1 + a (24,:,:,:) * xs__1__0__1 +
     *  a (25,:,:,:) * xs_m1__1__1 + a (26,:,:,:) * xs__0__1__1 +
     *  a (27,:,:,:) * xs__1__1__1

      end
```

FIGURE 7 Continued.

ment over CSHIFT or EOSHIFT. Furthermore, the advantage of PSHIFT is the highest when the lengths of the axes of the local subarray are the same. For instance, if the local subarray is of shape 100×10, then only 10 element exchanges along the two axes are overlapped for a shift distance of one. Then, 90 elements must be exchanged without any concurrency. In the default layout targeted by the geometry manager, the local segments of the array axes are of as equal length as possible. Thus, in such a layout, a maximum overlap between communications along different axes is achieved. The timings reported in Tables 1 through 4 were designed to verify the performance behavior relative to the intrinsic functions.

All timings were performed using the version of PSHIFT that is include in CMSSL 3.0. The timings were carried out at the Advanced Computing

Table 1. Timings on a Connection Machine System CM-200 for Calls to PSHIFT and CSHIFT with a Shift Distance of ±1 on Rank One Subarrays of Type REAL*8

Axes Length	CSHIFT Elapsed (msec)	PSHIFT Elapsed (msec)	Elapsed Speedup	CSHIFT CM-Time (msec)	PSHIFT CM-Time (msec)	CM-Time Speedup
4	0.221	0.058	3.8	0.147	0.041	3.6
8	0.230	0.059	3.9	0.151	0.045	3.4
16	0.223	0.064	3.5	0.159	0.051	3.1
32	0.228	0.064	3.6	0.176	0.064	2.8
64	0.226	0.099	2.3	0.208	0.099	2.1
128	0.333	0.234	1.4	0.305	0.210	1.5
256	0.744	0.610	1.2	0.571	0.456	1.3
512	1.289	1.029	1.3	1.058	0.904	1.2
1024	1.909	1.792	1.1	1.902	1.791	1.1
2048	3.566	3.460	1.0	3.566	3.458	1.0
4096	6.794	6.689	1.0	6.794	6.689	1.0
8192	13.451	13.346	1.0	13.451	13.346	1.0
16384	26.767	26.662	1.0	26.767	26.662	1.0

Table 2. Timings on a Connection Machine System CM-200 for Calls to PSHIFT and CSHIFT with a Shift Distance of ±1 Along Both Axes of Rank Two Subarrays of Type REAL*8

Axes Length	CSHIFT Elapsed (msec)	PSHIFT Elapsed (msec)	Elapsed Speedup	CSHIFT CM-Time (msec)	PSHIFT CM-Time (msec)	CM-Time Speedup
2×2	0.511	0.124	4.1	0.414	0.107	3.9
4×4	0.517	0.173	3.0	0.517	0.166	3.1
8×8	1.005	0.470	2.1	1.005	0.469	2.1
16×16	2.289	1.400	1.6	2.289	1.399	1.6
32×32	5.867	4.465	1.3	5.867	4.462	1.3
64×64	24.248	19.699	1.2	24.205	19.655	1.2
128×128	86.965	67.190	1.3	86.757	67.064	1.3

Table 3. Timings on a Connection Machine System CM-200 for Calls to PSHIFT and CSHIFT with a Shift Distance of ±1 Along All Three Axes of Rank Three Subarrays of Type REAL*8

Axes Length	CSHIFT Elapsed (msec)	PSHIFT Elapsed (msec)	Elapsed Speedup	CSHIFT CM-Time (msec)	PSHIFT CM-Time (msec)	CM-Time Speedup
2×2×2	0.722	0.205	3.5	0.722	0.205	3.5
4×4×4	2.357	0.911	2.6	2.330	0.910	2.6
8×8×8	9.948	4.954	2.0	9.923	4.954	2.0
16×16×16	49.044	30.389	1.6	49.044	30.379	1.6

Table 4. Timings on a Connection Machine System CM-200 for Calls to PSHIFT and CSHIFT with a Shift Distance of ±1 Along All Four Axes of Rank Four Subarrays of Type REAL*8

Axes Length	CSHIFT Elapsed (msec)	PSHIFT Elapsed (msec)	Elapsed Speedup	CSHIFT CM-Time (msec)	PSHIFT CM-Time (msec)	CM-Time Speedup
2×2×2×2	2.516	0.758	3.3	1.576	0.504	3.1
4×4×4×4	11.872	4.659	2.5	11.192	4.659	2.4
8×8×8×8	106.644	53.486	2.0	106.478	53.451	2.0

Laboratory of Los Alamos National Laboratory on a Connection Machine system CM-200 with a Sun-4 front-end. The system was operated using the Connection Machine System Software Version 6.1 and Version 2.1 of the CM Fortran compiler. In order to get accurate timings, each call was repeated a number of times, and the time measured for all calls. The number of calls was chosen such that the total time was approximately the same for the different cases. Thus, for example 10,000 calls were used for shifting a one-dimensional array, whereas 100 calls were used for shifting four-dimensional arrays along all four axes. The times given in Tables 1 through 4 are given in milliseconds per call.

The time required for the PSHIFT_SETUP calls varied from approximately 0.1 to 4 or 5 seconds. There was no correlation of setup time to the array size or shifts specified. The large amount of disk I/O required for writing, assembling, and loading the IMPs accounts for the majority of the setup time.

Arrays of type REAL*8 were used for all timings. The shape of the subarrays was chosen to measure the maximum relative performance gain of PSHIFT over the intrinsic functions. Thus, in two dimensions square subarrays were used and in three dimensions cubic subarrays were used. This is the default layout on the Connection Machine systems.

All the data in Tables 1 through 4 are for shift distances of 1 along all axes of the arrays. The elapsed time shown is the total time including time in which the front-end is busy while the CM is idle. The CM-time is the time that the CM was busy. The CM-time will always be less than or equal to the elapsed time.

The speedup of PSHIFT over CSHIFT is summarized in Figure 8. The speedups are computed using the elapsed times. As expected, the speedup decreases with the size of the local subarrays. For large subarrays, the execution time is dominated by on-node memory moves to support the virtual machine model, and the performance of PSHIFT and CSHIFT is practically the same. For small subarrays the time for a shift is dominated by exchanging data with neighboring nodes. In the one-dimensional case the expected speedup of PSHIFT over CSHIFT for a shift in both directions of the axis is two. However, the speedup is actually higher due to different implementation techniques. This fact is most notable for the front-end time. For four-dimensional subarrays the expected speedup of PSHIFT over CSHIFT for shifts in both directions of all four axis is eight. But, in this case the measured speedup is significantly less. Part of the reason is that although the actual exchange of data between a node and its neighboring nodes is fully concurrent, all memory operations in a node are serial. A more careful analysis of the expected performance is given next.

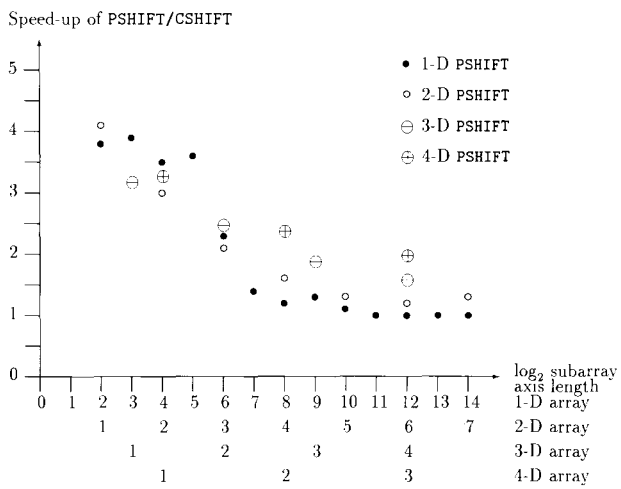

FIGURE 8 The speedup of PSHIFT compared to CSHIFT for some arrays and shift distances of ±1.

6.2 Performance Model

From the architecture of PSHIFT we can derive the following model for the time to execute a single call to PSHIFT for a k-dimensional subarray of shape $L_0 \times L_1 \times \cdot L_{k-1} = V$:

$$T_{total} = T_0 + N_{sl}2T_{mm}\sum_{i=0}^{k-1}\frac{V}{L_i}(L_i - S_i) + N_{sl}2T_{mt}\sum_{i=0}^{k-1}\frac{V}{L_i}S_i + N_{sl}2T_{tm}\sum_{i=0}^{k-1}\frac{V}{L_i}S_i + T_{ex}\max_i\left\lceil\frac{N_{sl}V\,S_i}{L_i 2}\right\rceil \quad (1)$$

where L_i is the length of subgrid axis i, k is the dimension (rank) of the subgrid, N_{sl} is the number of slices (one slice is 32 bits) per element for the given data type, S_i is the shift distance for axis i, T_0 is a startup time, independent of k and L_i, T_{mm} is the time for the memory-to-memory transfer of a single slice of data, T_{mt} is the time to transfer a slice of data from memory to a transposer, T_{tm} is the time to transfer a slice of data from a transposer to memory, T_{ex} is the time to do one exchange. The ceiling function occurs in the last term because the exchange operation swaps two slices at a time; if the number of slices to be sent over the wires is not even, the number of exchanges required is the next multiple of two.

For a shift distance of one and $L_i = L$, $0 \le i < k$, combining terms yields

$$T_{total} = a + bL^{(k-1)} + cL^k. \quad (2)$$

The $L^{(k-1)}$ term is the one associated with the actual between-processor communications; the L^k term is associated with on-processor memory-to-memory traffic. Thus depending on the relative values of the coefficients b and c, the PSHIFT execution time can become dominated by either term for a given subgrid extent and array rank.

We can obtain approximate values for the time of an exchange and the time for a memory transfer from the rank one array data. In this case, there is only one exchange per call to PSHIFT regardless of the subgrid length. Thus, any dependence of the timing on subgrid length comes entirely from the memory-to-memory portion of the code. For the smallest subgrid (length 4), the PSHIFT CM-time is totally dominated by the single exchange, because one array element is sent in each direction, and only six array elements are moved within the processor's memory. (We are anticipating here that an exchange takes much longer than a memory transfer.) Similarly, for the largest rank one subgrid, the time is dominated by the memory transfer time. The timings indicate the CM-time for a single call to PSHIFT with a rank one array and a subgrid length four requires 41 microseconds.

To obtain the time per memory transfer, we take the timing in Table 1 for a subgrid length of 16,384, subtract the 41 microseconds corresponding to the subgrid length four shift, and divide the remainder by twice 16,380 (the total number of memory-to-memory transfers not accounted for by the subgrid length four times). From this we obtain a time of .81 microseconds per slice moved. As noted above, the time per exchange is much greater than the time for a memory-to-memory transfer. However, as the subgrid size increases, the execution time will eventually be dominated by on-processor memory-to-memory time.

Figure 9 gives the time to execute PSHIFT as a function of the rank one subgrid length. An interesting feature is the kink in the curve occurring at a subgrid length of 64. This can be understood in terms of the Connection Machine memory architecture, wherein memory locations are addressed in pages. Changing the address of consecutive memory accesses by more than one page will require an extra step of resetting the page address; hence the effective time of memory-to-memory transfers depends on whether the source and target locations are in the same page or not. The slope of the curve increases for this case.

Figures 10 and 11 give the results of fitting the above model to the rank two and rank three data. As can be observed, the fit is quite good for the rank two data. For the rank three data, we have only four data points (so a cubic function of L can be fit exactly too the data points).

7 CONCLUSIONS AND DISCUSSION

PSHIFT gives the best performance gain over the CM Fortran intrinsics for

1. Subarrays with axes of equal lengths
2. Bidirectional shifts
3. Arrays of high dimensionality
4. Small subarrays

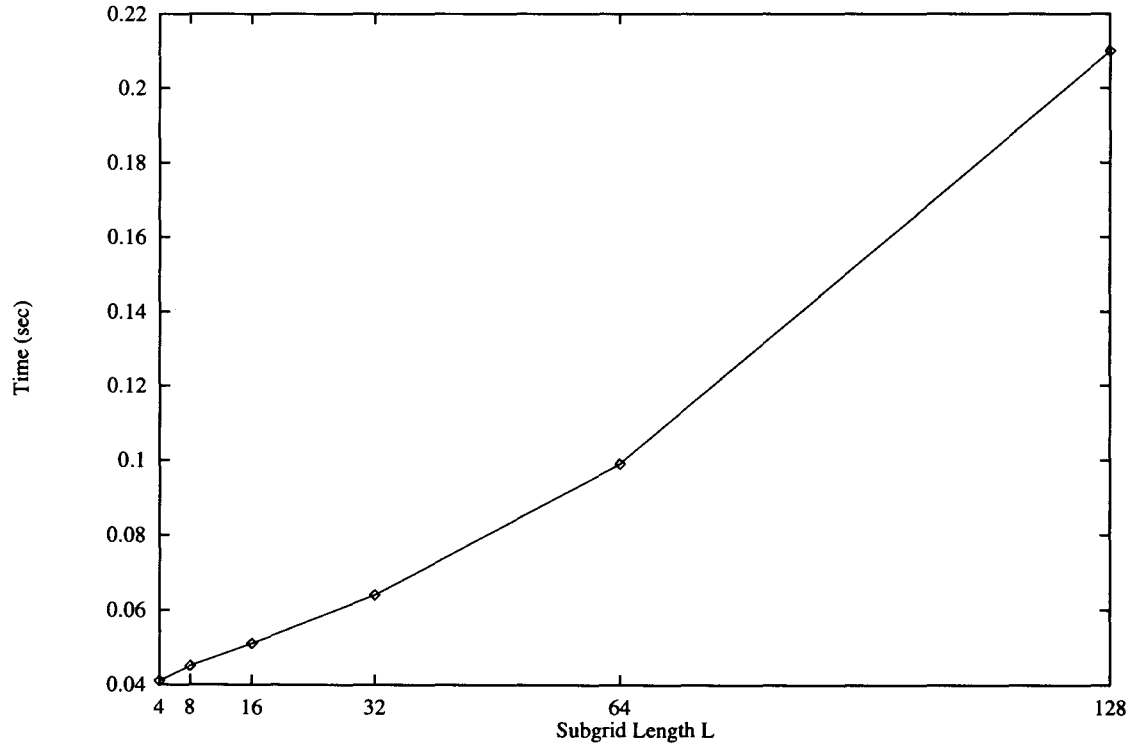

FIGURE 9 Timings for PSHIFT with shift distance ±1 on rank one REAL-8 arrays.

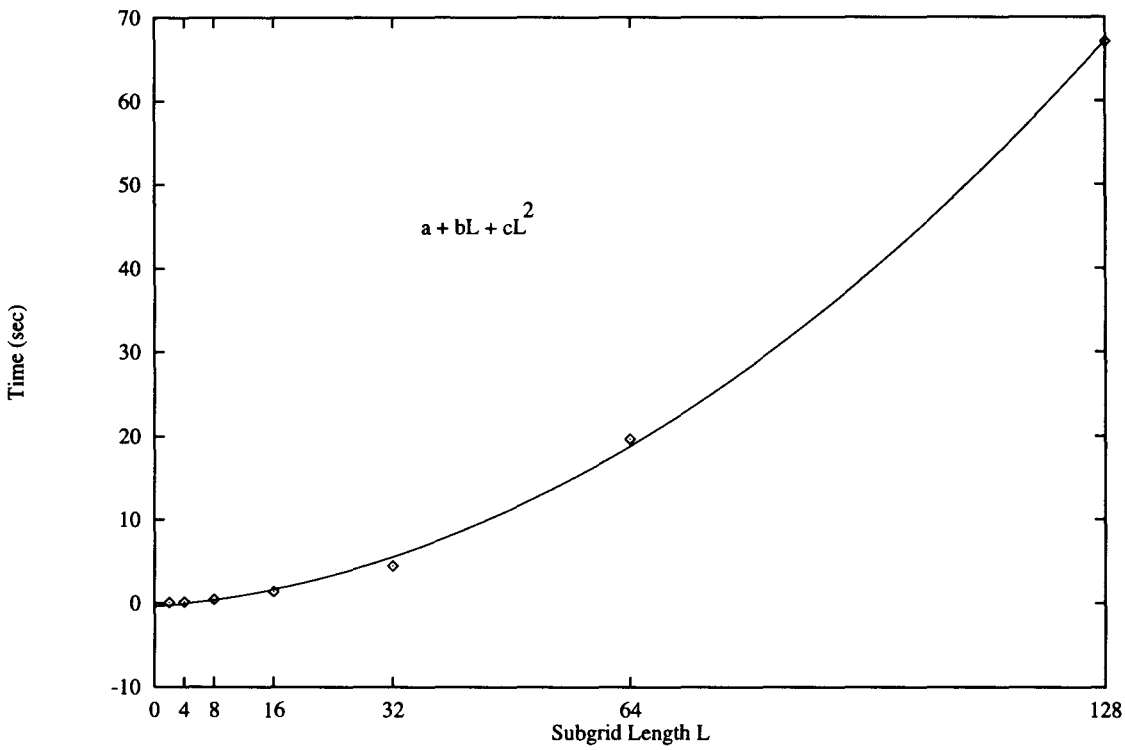

FIGURE 10 Timings for PSHIFT with shift distance ±1 on rank two REAL-8 arrays.

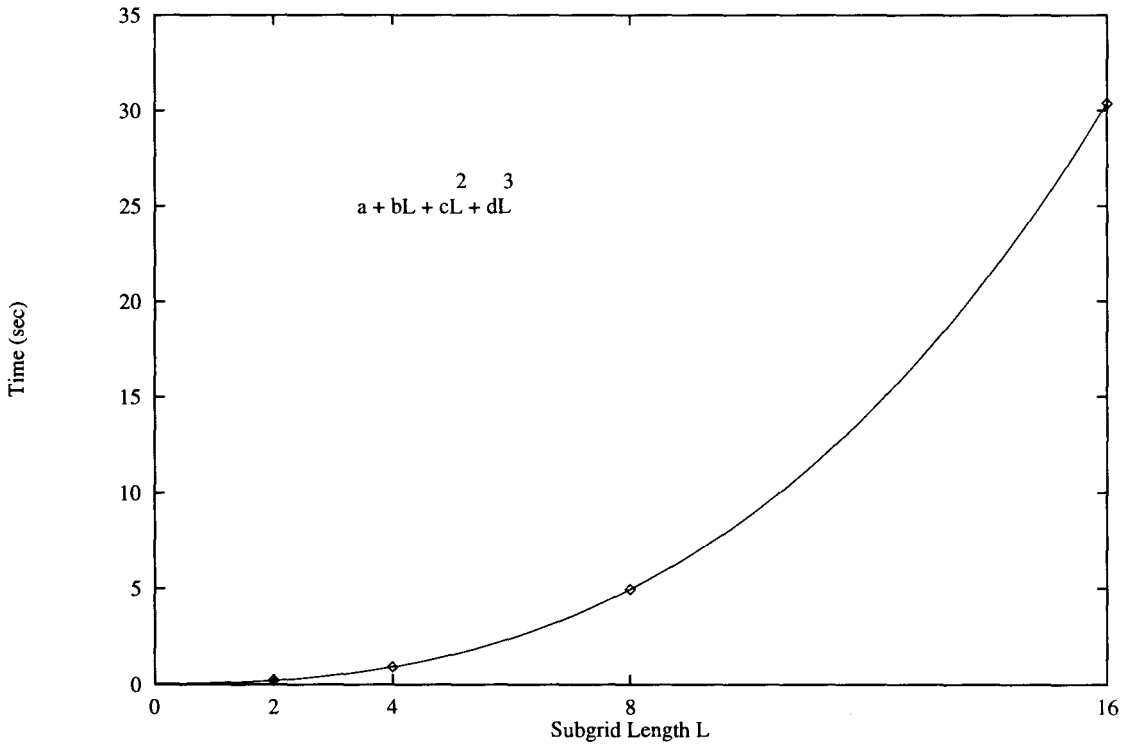

FIGURE 11 Timings for PSHIFT with shift distance ±1 on rank three REAL-8 arrays.

The first three rules simply result from the observation that parallelism is lost when they are violated. The fourth comes about because, as the subarray size increases, the execution times of both CSHIFT and PSHIFT become dominated by the on-node memory-to-memory movement of the data. There is no remedy for this situation as long as the virtual machine model is fully supported. However, it would be possible for a compiler to analyze the context of the communications pattern to determine what is the ultimate destiny of the shifted data. In the cases where this analysis succeeds, it would be possible to generate code so that unshifted data is used within a node, thereby eliminating the unnecessary local memory moves. Such a technology is not currently present in the CM Fortran compiler. Compiler features such as these are a research topic at this time, and are being considered by Rice and Syracuse Universities in developing a compiler for Fortran D [18], as well as at Thinking Machines Corporation.

Another approach towards reducing or eliminating the on-node memory-to-memory moves, is to identify classes of calculations that use patterns of shifts and consume the output locally, and provide library routines or a special-purpose compiler to generate optimized code for these classes. Both these approaches have been taken by Thinking Machines Corporation. The Stencil Library is an example of the former approach, and the Stencil Compiler [5] is an example of the second approach.

ACKNOWLEDGMENTS

The work reported herein was partially supported under DOE contract W-7405-ENG-36. We thank the Advanced Computing Laboratory of Los Alamos National Laboratory for access to the CM-200 located there. Valuable assistance and advice were given by a number of people during the course of this work, notably Steve Heller, Mark Bromley, Mike McKenna, Bob Lordi, Kapil K. Mathur, Woody Lichtenstein, Doug MacDonald, and Robert L. Krawitz of Thinking Machines Corporation. We thank John McGhee of Los Alamos National Laboratory for providing the 27-point stencil example.

REFERENCES

[1] W.J. Dally and C.L. Seitz, "Deadlock free message routing in multiprocessor interconnection networks," *IEEE Trans. Comput.*, vol. C-36, pp. 547–553, 1987.

[2] R. Gupta, *Computational requirements for QCD*, 1990, private communication.

[3] Thinking Machines Corp., *CMSSL for CM Fortran, Version 3.0*, 1992.

[4] Thinking Machines Corp., *CMSSL Release Notes, Version 3.0*, 1992.

[5] M. Bromley, S. Heller, T. McNerny, and G. Steele, *Proceedings of ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation.* New York: ACM Press, 1991.

[6] Thinking Machines Corp., *CM Fortran Reference Manual, Version 2.1*, 1993.

[7] Thinking Machines Corp., *CM Fortran Release Notes, Version 2.1*, 1993.

[8] M. Metcalf and J. Reid, *Fortran 90 Explained.* Oxford, England: Oxford Scientific Publications, 1991, pp. 1–315.

[9] S.L. Johnsson, "Communication efficient basic linear algebra computations on hypercube architectures," *J. Parallel Distrib. Comput.*, vol. 4, pp. 133–172, 1987.

[10] S.L. Johnsson, "Fast banded systems solvers for ensemble architectures," Technical Report YALEU/DCS/RR-379, Department of Computer Science, Yale University, March 1985.

[11] G. Li and T.F. Coleman, "A parallel triangular solver for a distributed memory multiprocessor," *SIAM J. Sci. Stat. Comput.*, vol. 9, pp. 485–502, 1988.

[12] G. Li and T.F. Coleman, "A new method for solving triangular systems on a distributed memory message-passing multiprocessor," *SIAM J. Sci. Stat. Comput.*, vol. 10, pp. 382–396, 1989.

[13] S.L. Johnsson, C.-T. Ho, M. Jacquemin, and A. Ruttenberg, *Advanced Algorithms and Architectures for Signal Processing II.* San Diego, CA. Bellingham, WA: Society of Photo-Optical Instrumentation Engineers, 1987, vol. 826, pp. 223–231.

[14] C. Tong and P.N. Swarztrauber, "Ordered fast fourier transforms on a massively parallel hypercube multiprocessor," *J. Parallel Distrib. Comput.*, vol. 12, pp. 50–59, 1991.

[15] S.L. Johnsson, *Algorithms, Architecture, and the Future of Scientific Computation.* Austin, TX: University of Texas Press, 1985, pp. 195–216.

[16] W. Lichtenstein and S.L. Johnsson, "Block cyclic dense linear algebra," *SIAM J. Sci. Comput.*, vol. 14, pp. 1257–1286, 1993.

[17] R.A. van de Geijn, "Massively parallel LINPACK benchmark on the Intel Touchstone Delta and iPSC/860 systems," Technical Report, The University of Texas at Austin, July 1991.

[18] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, "Fortran D language specification," Technical Report TR90-141, Department of Computer Science, Rice University, December 1990.

[19] High Performance Fortran Forum, "High Performance Fortran/Journal of Development," *Scientific Programming*, vol. 2/Nos. 1 & 2, Spring and Summer '93 (entire volume).

[20] E.N. Gilbert, "Gray codes and paths on the *n*-cube," *Bell Systems Tech. J.*, vol. 37, pp. 815–826, 1958.

[21] E.M. Reingold, J. Nievergelt, and N. Deo, *Combinatorial Algorithms.* Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 172–179.

[22] Thinking Machines Corp., *CM-200 Technical Summary*, 1991.