QCD on the Connection Machine: Beyond *LISP

Ralph G. Brickner
Clive F. Baillie
S. Lennart Johnsson

TR-01-91

February 1991

Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

Presented at the *IMACS First International Conference on Computational Physics*, Boulder, Colorado, June 1990.

QCD ON THE CONNECTION MACHINE: BEYOND *LISP

Ralph G. BRICKNER
Los Alamos National Laboratory,
Los Alamos, NM 87545, USA

Clive F. BAILLIE
Concurrent Computation Project,
California Institute of Technology, Pasadena, CA 91125, USA

S. Lennart JOHNSSON
Thinking Machines Corporation,
Cambridge, MA 02142, USA

January 10, 1991

Abstract

We report on the status of code development for a simulation of Quantum Chromodynamics (QCD) with dynamical Wilson fermions on the Connection Machine model CM-2. Our original code, written in *Lisp, gave performance in the near-GFLOPS range. We have rewritten the most time-consuming parts of the code in the low-level programming system CMIS, including the matrix multiply and the communication. Current versions of the code run at approximately 3.6 GFLOPS for the fermion matrix inversion, and we expect the next version to reach or exceed 5 GFLOPS.

1 INTRODUCTION

This paper reports the current status of an on-going physics project[1,2,3] to run large-scale Quantum Chromodynamics (QCD) simulations on the massively parallel Connection Machine. QCD calculations are among the world's most compute-intensive: tens of thousands of hours of Cray Y-MP equivalent CPU hours are used yearly on conventional supercomputers by researchers throughout the world, and a number of special purpose QCD machines have been built and are now online. As the Connection Machine model CM-2 has the potential of being the fastest commercially available computing system for certain applications, it was natural to look towards the CM-2 for doing high-speed QCD calculations. Our initial code was written in *Lisp, a high-level language which provides around 2 GFLOPS peak performance for 32-bit arithmetic operations on a 64K CM-2. Our *Lisp code delivered nearly one GFLOPS sustained performance on the fermion matrix inversion portion of the code. However, large performance gains are possible by rewriting code in a low-level programming system, called CMIS. CMIS provides a new programming model for the machine, called the "slicewise" model. Basically this means that the machine is viewed as consisting of 2048 floating point units, rather than the 65536 bit-serial processors which the original "assembly language", Paris, deals with. Thus, CMIS allows one to program the floating point units explicitly, accessing data in 32-bit words (rather than one bit at a time from 32 bit-serial processors). In this paper, we document some of the algorithms we have implemented in our CMIS version of the code, and discuss our plans for the immediate future for this code. Note that in this paper all performance figures refer to 32-bit arithmetic, irrespective of whether we use the 32-bit or the 64-bit floating point unit on the Connection Machine model CM-2.

2 LATTICE QCD SIMULATIONS

The simulation of full QCD with dynamical Wilson fermions begins with the action for the theory:

$$E = S_G + S_F \quad , \tag{1}$$

with the pure gauge and fermionic parts being given by

$$S_G = \beta \sum_P (1 - \frac{1}{3} ReTrU_P) \tag{2}$$

and

$$S_F = \overline{\psi} M \psi \ \ , \tag{3}$$

respectively. Here,

$$U_P = U_{i,\mu} U_{i+\mu,\nu} U^\dagger_{i+\nu,\mu} U^\dagger_{i,\nu} \tag{4}$$

is a product of link matrices around an elementary square or plaquette on the lattice and

$$\begin{aligned} M[U]_{ij} \ &= \ \delta_{ij} \ + \ \kappa \sum_\mu \ [\ (\gamma_\mu - r) U_{i,\mu} \delta_{i,j-\mu} \\ &- \ (\gamma_\mu + r) U^\dagger_{i-\mu,\mu} \delta_{i,j+\mu} \ ] \end{aligned} \tag{5}$$

is the Dirac operator for Wilson fermions. γ_μ are the Dirac gamma matrices. In the following we set $r = 1$. The fermionic action is re-written in terms of the pseudo-fermion fields ϕ:

$$S_F = \phi^\dagger (M^\dagger M)^{-1} \phi \ \ . \tag{6}$$

The link matrices are 3×3 complex $SU(3)$ matrices, and the pseudo-fermions are represented by a 3×4 complex matrix associated with each site on the lattice. The Hybrid Monte Carlo Algorithm (HMCA) we use is discussed elsewhere[1,4]. A detailed discussion of the pure gauge part of the code has also been published[5,6]. Here, we note only that the algorithm requires repeated calculation of the quantity $\phi^\dagger(M^\dagger M)^{-1}\phi$. In practice, we rewrite $\chi = (M^\dagger M)^{-1}\phi$ as $(M^\dagger M)\chi = \phi$, a linear system of equations, and solve for the unknown χ. The conjugate gradient algorithm has been widely used, and it is guaranteed to converge for all quark masses. However, for the quark masses we are interested in, there are equally attractive algorithms. Currently, we use the preconditioned, over-relaxed minimal residual algorithm[1], which is much faster than the standard conjugate gradient algorithm in the quark mass regimes of interest. The basic operation in this and related algorithms is to calculate the matrix-vector product $M\chi$ or $M^\dagger(M\chi)$. In our code, this is done in the function "dslashsq", whose performance we will be discussing below. One feature of this algorithm is that we first solve for χ on the even sites, then reconstruct the solution on the odd sites by a single application of “dslashsq”. In general, “dslashsq” is coded to calculate the matrix-vector product on either even or odd sites, using data from the other sites as input. With the Wilson parameter r set to 1 (or -1), we can use the characteristics of the gamma matrices to project the 3×4 fermion matrices down to 3×2, do the communications and calculations required, and expand the result back up to 3×4. This reduces communication time by a factor of two, and the computation time by about 40%. Figure 1 is a code fragment from “dslashsq” which illustrates the basic operations. There is a separate projection and expansion function for each term in Eq. 5. For example, “*proj-xp” projects the input 3×4 matrix “chi” down to the 3×2 matrix “v2”. This corresponds to the term $(\gamma_x + r)U^{\dagger}_{i-x,x}\delta_{i,j+x}$ in Eq. 5. “*m33-32” does the complex matrix multiply, and the expand functions expand the result back up to 3×4. The 3×3 matrix “ux-ea” contains the x-direction link matrix on even lattice sites, and the negative of the adjoint of the link on the odd sites.

3 THE CONNECTION MACHINE

The Connection Machine model CM-2 from Thinking Machines Corporation (TMC) is often described as a distributed-memory, Single-Instruction Multiple-Data (SIMD) massively-parallel processor comprising up to 65536 (64K) processors[7,8]. However, these processors are simple bit-serial processors which are not used in doing floating point calculations. Instead, the Floating Point Units (FPUs) are used for applications such as QCD, and so we have come to think of the CM-2 as an eleven-dimensional hypercube of 2048 floating point units (known as “sprint nodes”). The original “assembly language” of the machine, Paris, embodies a computational model which is essentially independent of the specific nature of the hardware, but its implementation in the CM-2 leans very much towards the view of the machine as consisting of the bit-serial processors (known as the “fieldwise” view of the machine) instead of consisting of the floating point units (known as the “slicewise” view of the machine). Since all the originally provided high-level languages – *Lisp, C*, CM Fortran – compile into Paris, any code written in these languages is subject to the implementation inefficiencies inherent in

Paris.

The current implementation of Paris can impair floating point performance for the following reasons. The single-ported memory of the machine is addressed in words of 32 bits, with one bit corresponding to each of 32 bit-serial processors. In bit-serial processing, a memory reference results in the 32 bits of a word being loaded into or stored from each of the 32 bit-serial processors in parallel. However, a 32-bit floating point word is stored in memory in 32 consecutive memory locations, with the 32 bits at a given address corresponding to one bit in each of 32 processors. But the Floating Point Unit must load or store 32 bits of a single floating point word at a time. Therefore, to "glue" the bit-serial processors to the FPU, a set of transposers is provided. The transposer can be thought of as a 32 bit by 32 bit square matrix. In 32 clock cycles, the matrix is loaded row by row so that a column corresponds to a given bit-serial processor, and a row corresponds to a single bit of a floating point word corresponding to each of 32 different processors. The FPU then reads a column at a time, getting all 32 bits of a floating point word at once. For storing, the process is reversed. The set of transposers physically resides on the "sprint chip" along with several other pieces of hardware. (Most notably the bypass register, which allows feeding a 32-bit word directly from memory into the FPU.) Thus, the module consisting of 32 bit-serial processors and their memory, a sprint chip, and an FPU make up the "sprint node". Unfortunately, the use of the transposer has several negative impacts on performance. First, it requires doing all operations in the FPU in chunks of 32 words. Because the internal architecture of the Weitek chips used in the CM-2 includes only 32 registers, there is no room for storing intermediate results in the Weitek. Therefore, operations such as a complex multiply, which could employ internal registers to store the intermediate results, are forced into storing and reloading these in memory. Second, an overhead of 32 cycles is added to every load and store, since that is the time required to fill the transposer. The net effect on performance is that Paris peaks at around 2 GFLOPS for any code sequence which compiles into a series of simple multiplications and addition/subtractions. Significant improvement in performance is possible by using the chained multiply-add instructions of Paris, but the compiler needs to be sophisticated enough to detect these possibilities.

The solution to these problems is found in going to the "slicewise" model of the machine. In this model, one uses the transposers to convert data into the form required by the FPU once and for all, and thereafter bypasses the transposers when loading to or storing from the FPU (using the bypass register). After conversion, a memory address references all 32 bits of a floating point word corresponding to one bit-serial processor, with words belonging to consecutive processors arranged consecutively in memory. One is now free to program the FPU in the most efficient manner. For example, the complex matrix multiply – which is the basis for all QCD simulations – is done for each processor in turn, thereby making full use of the internal Weitek registers and the memory bandwidth. The price one pays for this freedom is that one must write virtually all the slicewise code one will need, although some Paris instructions still function on slicewise data (for example memory-to-memory moves and news communications).

Fortunately, TMC provides some rather sophisticated tools for dealing with the slicewise view of the machine. The generic name for this programming system is "CMIS" (Connection Machine Instruction Set), but there are many components to the system. A discussion of CMIS is beyond the scope of this paper (the interested reader should consult the reference manual[9] for more details), but we will state that the fundamental building blocks provided allow one to construct completely pipelined code for all the parts of the sprint node with a minimum of fuss.

To understand the algorithms we employ to do the matrix multiplies, it is necessary to understand some details of the FPU architecture. The next section discusses the currently available chips in detail.

4 WEITEK FLOATING POINT UNITS

Currently the Connection Machine model CM-2 may contain either the Weitek WTL3132 or the Weitek WTL3164 chip as its FPU. The main difference between these is that the first contains 32-bit functional units (an accumulator and a multiplier) and 32-bit internal data paths, whereas the second has 64-bit ones. However both chips read/write external data from/to the sprint chip via a single 32-bit bus. Thus double precision numbers for the WTL3164 must be passed in two cycles. We shall now describe each chip in detail and then return to comparing them.

The WTL3132, shown schematically in Figure 2, is capable of only single precision, performing the operations ADD, SUBTRACT and MULTIPLY according to IEEE standard 754. Divide is possible by means of two iterations of the Newton-Raphson algorithm using an appropriate initial value from the on-chip table. The chip does one operation every three cycles, and can be pipelined to start a new operation on every cycle. Chained MULTIPLY/ADD is available (with a latency of three cycles between MULTIPLY and corresponding ADD) but does not round correctly between the two operations, so this operation can yield answers off by as much as two least significant bits. The chip contains 32 32-bit general registers and three 32-bit temporary registers. The three temporary registers are used to store intermediate results in order to facilitate operations of the form $x = x \pm (y \times z)$. In pipelined mode, the result of an operation is generated three cycles after it is initiated; on the fourth cycle it can be returned to the register file or fed straight back into the multiplier/accumulator using these temporary registers. There is one bi-directional I/O port on the chip which can transfer a data value on every cycle. The data value can be input to either the register file or the multiplier/accumulator.

The WTL3164, shown schematically in Figure 3, can do both single and double precision and is fully IEEE compliant for ADD, SUBTRACT, MULTIPLY, DIVIDE and SQUARE ROOT. The chip has separate ADD and MULTIPLY pipes, each with a latency of two cycles. These can be chained to get IEEE-compliant MULTIPLY/ADD operations. Despite all internal data paths being 64-bit, the single I/O port to the outside world is only 32-bit wide. The chip contains 32 64-bit general registers and four 64-bit temporary registers. Two of the temporary registers – called X and Y – are connected to the I/O port so that a constant data value may be input and kept in one while varying data is input via the other. The other two temporary registers – T0 and T1 – provide necessary bandwidth for chained MULTIPLY/ADD operations. The number of operands required is four, yet the register file provides only two and the X or Y register provides the third. Therefore the fourth comes from one of the T registers which form a path from the multiplier output to the accumulator input. Two T registers are required for the chip to operate with full interruptibility during pipelined register-to-register operations. In normal use only one is used to soak up the two-cycle latency. Thus in order to do the tightest possible sum of products, for example, the multiply is executed exactly two cycles prior to the accumulator operation.

The next most important difference between the chips, after the precision difference, is that the WTL3132 has a latency of three, whereas the WTL3164's latency is two. This means that specially coded pipelines for functions like SU(3) matrix multiply must be coded differently for each chip. On the whole the WTL3164 seems to be the better chip and we have obtained higher performance from it on most of our functions.

Finally, we should mention that FPU instructions are given in two parts. The first part is a 9-bit static instruction which roughly specifies the operation to be performed. The second part is a 24-bit dynamic instruction which specifies the source operand and destination registers. Execution of a dynamic instruction causes the operation to be started on the chip. It is common to execute N of the same operation (for example ADD) by issuing one static instruction followed by N dynamic instructions.

5 MATRIX MULTIPLY ALGORITHMS

The heart of all QCD simulations is a complex matrix multiply. One operand, corresponding to the gauge fields, is a 3×3 complex matrix. The second operand and the result, corresponding to the fermion fields, is always a $3\times n$ complex matrix, but the number of columns depends on the exact discretization employed. As discussed above, our code uses projection to 3×2 matrices before the multiply. Another popular discretization method, "staggered fermions", results in 3×1 fermion matrices. To accommodate all cases, we designed our matrix multiply as a 3×3 times 3×1 multiply, and construct products for larger matrices out of this.

Matrix multiplies consist of a series of sums of products, so there is potential for simultaneously driving both accumulator and multiplier in a design such as the Weitek series of pipelined floating point units. However, the internal architecture of the chip, as well as its integration into the CM-2 architecture, can make for interesting problems in designing an efficient pipeline. The internal structure of the WTL3164 differs greatly from the original chip used in the CM-2, the WTL3132, so a completely different algorithm is necessary for the newer chip. We describe the main pipelines used to construct our matrix multiplies for both the WTL3132 and the WTL3164. In both cases, we assume the input 3×1 is already in the Weitek register file, and we assume the resulting 3×1 will be stored after the main pipelines end. Also in both cases, the input 3×3 is read from memory in the same pipeline which does the arithmetic, so its loads are always overlapped with arithmetic, and it never needs to be stored in the Weitek register file. *Lisp/CMIS was used to write the actual code for the multiplies. The primary CMIS construct used in the matrix multiplies allows one to start up, maintain, and end a pipeline which includes memory, the memory bus, a bypass register in the sprint chip, the float bus, and the FPU. It allows specification of the exact Weitek instruction on a cycle-by-cycle basis, and comes in two versions. The first version allows one to specify a single memory address and stride for consecutive loads/stores. The second version allows specification of two memory addresses and strides, so that the pipeline consists of memory accesses from the two addresses on alternating cycles.

The complex multiply for the WTL3132 is illustrated in Figure 4. It is constructed out of matrix-vector multiplications as memory-to-memory primitives. Each such operation is built out of SAXPY operations with one operand taken from memory while the remaining three operands are read from, or stored in registers on the WTL3132 chip. In this notation, $Im[U00]$ refers to the imaginary part of the element in row 0, column 0 of the input 3×3 matrix, and so on. It is stored slicewise in memory. Similarly, the input 3×1 fermion matrix is referred to as F, and it resides in the register file. The three temporary registers are denoted $T1, T2, T3$, and $r01$ refers to the second register in the Weitek's internal register file. The SAXPY part of the matrix-vector kernel uses a CMIS pipeline with a single pointer to memory. This pointer is set to either the first real or the first imaginary part of the input 3×3 matrix, and the stride is set to pick off consecutive real/imaginary parts. We associate one temporary register with each of the three rows in the result matrix. The first three cycles of the pipeline load the three words of the first column of the 3×3 matrix from memory into the multiplier, using the first element of the 3×1 matrix from the register file as the other operand. The next three cycles load the second column of the 3×3 input matrix into the multiplier, using the second element of the 3×1 matrix (resident in the register file) as the second operand, and add to this product the result of the first column product. The next three cycles multiply the third column of the 3×3 input matrix with the third element of the 3×1 matrix in the register file, and add in the previous partial sum. The result is that the sum of the three products resides in the temporary registers after 12 cycles (9 for the pipeline length plus three for the latency). The real and imaginary parts of the result are built from this basic pipeline in different ways. The imaginary part simply resets the memory pointer to the other part of the first column and runs the above pipeline again, this time using the previous partial sums which are still in the temporary registers. The result goes into the register file (in registers $r11, r13, r15$). The real part requires a sign change for its second pipe, so a static instruction must be issued to change from addition to subtraction (actually, negation then addition), as well as resetting the memory pointer. Then, the final pipeline is executed and the three words of the real part of the result end up in the register file (in registers $r10, r12, r14$), interleaved with the three words of the imaginary part. Note this memory access pattern requires the input 3×3 matrices to be stored in column major order. This is ensured when we convert the matrices to slicewise format.

The algorithm for the WTL3164 chip is illustrated in Figure 5. The basic Weitek instruction used is FMUL-FADD-CHAINED, in which the output of the multiplier is stored in the T temporary registers until it is available for input to the accumulator two cycles later. We use the CMIS pipeline with two memory addresses, and set them both to the first word of the input 3×3 matrix, with both having unit stride. For this algorithm, we have rearranged the 3×3 matrices in memory, allowing us to accumulate all the partial sums in a single pipeline. The key to doing this is loading a zero into the register file ($r31$), and then resetting the B input to the accumulator at the proper time using this zero. As the latency is two, the first two results out of the pipeline are garbage so they are dumped in a "$BASH$" register; similarly, at the end of the pipeline we must wait two extra cycles for the final results to come out, hence the NOP instructions. The imaginary part of the result (from real times imaginary plus imaginary times real) goes straight into registers $r11, r13, r15$. The real times real partial results go into registers $r10, r12, r14$ and the imaginary times imaginary partial results go into $r17, r19, r21$. The second pipeline in the algorithm computes the real part of the answer as a three step subtraction of the imaginary times imaginary parts of the result from the real times real parts of the result. We have coded this to overlap with the load of the next 3×1 input matrix, so it does not add to the cost of the multiply.

Performance of the two versions is given in Table 1, along with reference rates from Cray machines, and high-level languages on the CM-2. The CM-2 runs were done at a VP ratio of 4, except for the WTL3164 CMIS version, which was run at a VP ratio of 8 (which we expect to be using in our next set of production runs).

6 MULTIWIRE NEWS

The QCD code employs a four-dimensional lattice (x, y, z, and t) of space-time sites. The fermion matrix inversion algorithm embodies a very specific communications pattern among the sites of the lattice. The communication consists of conditional sends forward, then conditional sends backward, for each of the four dimensions of the lattice. (Sites whose self-addresses on the lattice are odd would send forward, then even sites would send backward.) In each communication a 3×2 complex matrix, or 12 elements, represented as a slicewise array, is moved between adjacent sites. Restructuring the original code allows communication to proceed in both directions in all four dimensions concurrently. As an intermediate step a code with concurrent communication in a single direction of all four dimensions has been developed. We refer to this code as the "one-direction multiwire news" code. For the bi-directional, four-dimensional concurrent communication version of the code additional restructuring is planned to eliminate local memory moves.

The data layout across processors, and within the memory of each processor is a function of the machine configuration, or geometry. The Paris geometry is different from the slicewise geometry. Transposition of data from the representation used by Paris to a slicewise representation causes an axis to have two, or sometimes three sets of strides on a sprint node. To avoid the complexity of dealing with multiple strides for each axis, memory reordering routines have been developed as part of the toolkit for the slicewise model of the Connection Machine model CM-2[10]. The memory reordering routines allow the user to specify slicewise arrays to be laid out just like standard Fortran arrays. That is, for a sub-lattice consisting of $lnx \times lny \times lnz \times lnt$ lattice sites on a sprint node (where lnx is the local number of sites in the x-direction, etc), the x-index varies fastest, then the y-index, then the z-index, and finally the t-index. All the slices for a given data structure (say a matrix) are contiguous for a lattice site, so these indices are the leftmost in the complete set of indices (say amat(3,2,16,16,16,16) for a 3×2 complex on-processor array on a $16 \times 16 \times 16 \times 16$ lattice). In each communication a face of a four-dimensional hyperparallelipiped is sent from one site to a neighbor site. This face is actually a three-dimensional parallelipiped whose size (in lattice sites) is the product of the number of lattice sites in the three dimensions other than that along which it is being sent. The amount of data to be sent is the number of slices in the array per site times the number of sites in the three-parallelipiped.

Production runs have in the past mainly been performed on 16K machines with 128 Mbytes of memory. A feasible lattice for this CM configuration is of size $16 \times 16 \times 16 \times 16$. A 16K CM-2 configuration has 512 sprint-nodes. The number of lattice sites per sprint node is 128. In order to minimize the communication requirements with an equal amount of communication in all four dimensions the segments of each of the four axes that reside on a sprint node should be as equal in length as possible. Hence, the 128 sites on a sprint node are chosen as a $4 \times 2 \times 4 \times 4$ sub-lattice. For this case, communication in the x, z, and t directions implies sending a $2 \times 4 \times 4$ cube to an adjacent sprint node, but communication in the y direction implies sending a $4 \times 4 \times 4$ cube to an adjacent sprint node. One direction requires twice the time of the other three directions, and the communication is clearly unbalanced. A communications inefficiency of 50% results for multiwire news communication. By extending the lattice a factor of two, to a $16 \times 16 \times 16 \times 32$ lattice, 256 sites reside on a sprint node, and the local sub-lattice can be chosen to be of shape $4 \times 4 \times 4 \times 4$. Then, a $4 \times 4 \times 4$ cube is communicated in every dimension. The communication efficiency for multiwire news is 100%. Moreover, it is more desirable to do physics on a lattice of this size, but a 16K CM-2 configuration with 512 Mbytes of memory is required.

The communication function needed for the one-direction multiwire news code is

(send-4d-forward x-dst y-dst z-dst t-dst x-src y-src z-src t-src)

which sends the contents of x-src, etc to x-dst, etc in the neighbors in each of the four forward directions. A corresponding function is necessary for the backward sends. The implementation of this function consists of four distinct phases. In the first phase a set of data to be communicated to adjacent sprint nodes in the four dimensions (from memory x-src etc) are moved to a "departure lounge"[11]. In phase two the data that remains on the sprint nodes is shifted "upwards" in memory for each of the four directions. In the third phase the actual communication between sprint nodes, "cubeswaps", is performed. In this phase the outgoing data in the lounge is replaced by data coming in from the backwards directions. Finally, the data in the lounge is moved to the memory locations corresponding to the incoming faces of the four-cube (x-dst etc). A large variety of design choices exist for the multiwire news in a four-dimensional, multi-slice data element geometry.

Due to the way the data indices vary, as described above, the memory-to-memory transfers need to be coded differently for each of the four directions. The objective of efficient CMIS coding is to a) maximize the operation count of each CMIS call, and b) minimize the amount of looping. Therefore, for all operations, the count is at least $nslices$, the number of words in the data element (12 for our application). For a given value of x, movements in the x-direction are implemented with a count of $nslices$, and looping must be done over the y, z, and t directions. For movements in the y-direction all the data elements for the x-loop are contiguous, so the operation count is set to $nslices \times lnx$. The z and t directions are then looped over. This process is extended to the z and t directions, until for the t-direction, all the slices are contiguous. In practice, we found that explicitly coding the loops in CMIS resulted in somewhat inefficient code, since loops ran only from 0 to 3 for our application. Therefore, we wrote a "multiwire code generator" (in Fortran) which generates completely unrolled CMIS code, with no looping whatsoever. The code generator allows one to construct any required looping logic, and to calculate operand addresses, in a high-level language. Complete CMIS code is written by the generator. This results in maximally efficient CMIS code. Moreover, by modifying a few lines of Fortran, one can generate extensive CMIS programs with very few errors.

The current production code performs conditional communication based upon whether a lattice site is even or odd (the sum of the self-address coordinates is even or odd). For a four-dimensional lattice mapped onto a one-dimensional address space, this results in a pattern which cannot be described by a single stride, or a simple set of strides. Testing each individual site for parity inside the CMIS code itself would be very inefficient. Likewise, performing unconditional communication followed by a conditional write of the saved original value would also be very inefficient. Fortunately, a modification of the multiwire code generator allows conditionals to be generated trivially. First, it is set up to generate a site-by-site set of CMIS calls, all with a count of $nslices$. A loss in efficiency might occur due to reduced operation count for the y, z, and t directions. For every piece of CMIS code to be generated by the code generator, it first checks the parity of the destination site. If it is the appropriate parity, it generates the code; if not, it does not generate the code. The combination of a) operation count inefficiency and b) moving only half the data, results in the conditional code running in 54% of the time of the optimized (maximum op count) code. This time corresponds to a speed-up of 4.4 over the conditional Paris news.

Table 2 compares multiwire "send-4d-forward" against the Paris "get-from-news" function. All times are for a 16×16×16×32 lattice on a 16K CM-2 at the Advanced Computing Laboratory of Los Alamos National Laboratory, with a Sun 4/260 front end running CM System Software release 5.2. The on-sprint-node geometry was 4×4×4×4. All code was compiled in *Lisp with safety = 1. Times are in seconds for 1000 calls to the appropriate functions. In all cases, the CM utilization was 100%.

7 CURRENT DEVELOPMENT

Table 3 shows the relative times spent in various components of "dslashsq", along with their GFLOPS rates. It is evident that a great deal of time is being spent in projecting and expanding the fermion matrices, and at a very low performance. These project and expand functions consist of some memory to memory moves, along with some simple additions and subtractions. Furthermore, they are written as unconditional versions. That is, while their results are used only on odd or even sites at a given time, they actually calculate the result for all sites. Where this would destroy data, we save the data from being overwritten by conditional Paris moves, and then restore it later. (Unconditional projects and expands would count for twice the GFLOPS rate entered in Table 3.) To alleviate the problem of unconditional projects and expands, we have adopted a strategy of segregating all the arrays in the code into contiguous sections of all even or all odd sites. This is done when the data are converted to slicewise format. Then, the projects and expands do unconditional operations on the proper half of the array. The time spent in the projects and expands is thereby cut in half. The CMIS code for the multiwire news was trivially changed to accommodate the new data layout by simply changing the function which returns the offset into the arrays. This version of the code has been written and is currently being debugged.

Overall performance of "dslashsq" for the various versions of the code is presented in Table 4. While we have not timed the segregated array version, we can confidently predict its performance from halving the times for project and expand in Table 3.

8 FUTURE WORK

Our next project is to restructure the code to allow bi-directional multiwire news. The code logic illustrated in Figure 1 was changed for the one-direction version of multiwire news to do all the calculations for the four directions at once. Then "send-4d-forward" was called. Further restructuring will allow us to complete all calculations before any news is done, so that both forward and backward sends can be done at once. Indeed, the cubeswap which lies at the heart of multiwire news is actually doing bi-directional communications even in "send-4d-forward". During this phase of code development, we will also be modifying the multiwire code generator to reduce the memory-to-memory component of news. We are confident that this version will provide another large jump in performance, and we believe we will reach very close to, or perhaps exceed, 5 GFLOPS for this version. At close to 5 GFLOPS performance, the CM-2 is approaching the performance of the special purpose QCD machines which are currently running, such as those at Columbia University[12] and IBM[13].

9 CONCLUSIONS

While the Connection Machine model CM-2 provides very high performance even with the delivered software, specific applications may have characteristics which allow one to obtain extremely high performance. The work we have done on optimizing QCD on this machine has provided very high payoffs: over a factor of five for the matrix multiply, and over a factor of four (for currently running code) for news communications. While the amount of time and effort required to obtain this performance is substantial, physics collaborations which need very large amounts of computational resources are finding it worthwhile to invest the effort. The net result is performance which can approach that of special purpose QCD machines.

ACKNOWLEDGMENTS

This work was performed under the auspices of the United States Department of Energy. We thank the Center for Applied Parallel Processing, Argonne

National Laboratory, the Northeast Parallel Architectures Center, Boston University, Thinking Machines Corporation, and the Advanced Computing Laboratory of Los Alamos National Laboratory for providing access to Connection Machines and making this work possible.

The assistance of Mike McKenna with the Paris to slicewise geometry conversion software, and of Steve Heller with the multiwire news software is greatly appreciated. Both Mike McKenna and Steve Heller are employees of Thinking Machines Corporation.

References

[1] R. Gupta, A. Patel, C.F. Baillie, G. Guralnik, G.W. Kilcup and S.R. Sharpe, *Phys. Rev.* **D40** (1989) 2072.

[2] "Hadron Spectrum from the Lattice", R. Gupta, talk presented at LATTICE 89, Capri, Italy, September 1989, to appear in *Nucl. Phys.* **B** (Proc. Supl.) (1990).

[3] R. Gupta, A. Patel, C.F. Baillie, R.G. Brickner, G.W. Kilcup and S.R. Sharpe, in preparation.

[4] "QCD with Dynamical Fermions on the Connection Machine", C.F. Baillie, R.G. Brickner, R. Gupta and L. Johnsson, in: *Proceedings of Supercomputing 89*, pp 2-9 (ACM Press, New York, NY, 1989).

[5] "Pure gauge QCD on the Connection Machine", R.G. Brickner and C.F. Baillie, in: *Proceedings of Scientific Applications of the Connection Machine*, ed: H.D. Simon, pp 234-251 (World Scientific, Singapore, 1989).

[6] "Pure gauge QCD on the Connection Machine", R.G. Brickner and C.F. Baillie, *Int. J. of High Speed Computing* **1** (1989) 303.

[7] W. Daniel Hillis, *The Connection Machine* (MIT Press, Cambridge, MA, 1985).

[8] "Connection Machine Model CM-2 Technical Summary", Thinking Machines Corporation, Cambridge, MA, 1989.

[9] "CMIS Reference Manual", Bob Lordi, Thinking Machines Corporation, Cambridge, MA, 1989.

[10] "The Shuffle and Transpose Microcode Routines", Michael McKenna, Thinking Machines Corporation, Cambridge, MA, 1989.

[11] "Multiwire Communication in the Connection Machine model CM-2", Alan Edelman, Steve Heller and Mark Bromley, private communication, 1990.

[12] "Status of the Columbia 256-node Machine", N.H. Christ, talk presented at LATTICE 89, Capri, Italy, September 1989, to appear in *Nucl. Phys.* **B** (Proc. Supl.) (1990).

[13] "Progress report on the GF11 Project", D. Weingarten, talk presented at LATTICE 89, Capri, Italy, September 1989, to appear in *Nucl. Phys.* **B** (Proc. Supl.) (1990).

Table 1. Complex 3×3 times 3×1 Execution Rates.

Machine/Compiler	GFLOPS
Cray X-MP/CFT77 (1 proc)	0.13
Cray Y-MP/CFT77 (1 proc)	0.21
CM-2/CM Fortran	1.87
CM-2/CM *Lisp	1.95
CM-2/CMIS (WTL3132)	8.50
CM-2/CMIS (WTL3164)	10.20

Table 2. Paris and Multiwire News Times (sec).

	Paris	One-dir Multiwire
Always	58.7	24.6
Even only	58.7	13.3

Table 3. Components of “dslashsq”.

Function	% Run Time	Rate (GFLOPS)
Project	19	1.5
Multiwire News	24	0.0
Matrix Multiply	24	10.2
Expand	30	1.2

Table 4. Performance of “dslashsq” versions.

Version	Rate (GFLOPS)
*Lisp	0.9
with CMIS Arithmetic	1.6
with One-dir Multiwire News	2.9
with Even/Odd Segregation	3.6