A HIGHLY OPTIMIZED VECTORIZED CODE FOR MONTE CARLO SIMULATIONS OF SU(3) LATTICE GAUGE THEORIES

D. BARKAI
CONTROL DATA CORPORATION
INSTITUTE FOR COMPUTATIONAL STUDIES AT CSU
FORT COLLINS, COLORADO

K. J. M. MORIARTY
DEPARTMENT OF MATHEMATICS, STATISTICS
AND COMPUTING SCIENCE
DALHOUSIE UNIVERSITY
HALIFAX, NOVA SCOTIA, CANADA
AND
DEPARTMENT OF MATHEMATICS
ROYAL HOLLOWAY COLLEGE
ENGLEFIELD GREEN, SURREY, U.K.

AND

C. REBBI
DEPARTMENT OF PHYSICS
BROOKHAVEN NATIONAL LABORATORY
UPTON, NEW YORK

A Highly Optimized Vectorized Code for Monte Carlo Simulations of SU(3) Lattice Gauge Theories*

D. Barkai
Control Data Corporation
at the
Institute for Computational Studies at CSU
P.O. Box 1852
Fort Collins, Colorado 80522

K.J.M. Moriarty
Department of Mathematics, Statistics
and Computing Science
Dalhousie University
Halifax, Nova Scotia B3H 4H8, Canada
and
Department of Mathematics
Royal Holloway College
Englefield Green, Surrey TW20 OEX, U.K.

and
C. Rebbi
Department of Physics
Brookhaven National Laboratory
Upton, New York 11973

Abstract

New methods are introduced for improving the performance of the vectorized Monte Carlo SU(3) lattice gauge theory algorithm using the CDC CYBER 205. Structure, algorithm and programming considerations are discussed. The performance achieved for a 16^4 lattice on a 2-pipe system may be phrased in terms of the link update time or overall MFLOPS rates. For 32-bit arithmetic it is 36.3 μsec/link for 8 hits per iteration (40.9 μsec for 10 hits) or 101.5 MFLOPS.

September 1983

*Talk presented at the International Conference "Parallel Computing 83" at the Free University of Berlin, 26-28 September 1983 and at the Joint NASA/Goddard-CDC Symposium on CYBER 205 Applications, held at Lanham, Maryland, 11-12 October 1983.

1. *Introduction*

Many important results for quantum field theories in general and, in particular, for the gauge theory of strong interactions known as Quantum Chromodynamics (QCD) have been obtained by formulating the dynamics on a space-time lattice. The lattice version of a quantized gauge field theory, as proposed by Wilson [1], has the properties of introducing an ultraviolet cut-off independently of any perturbative expansion and of preserving manifest gauge invariance. It permits a variety of investigations by non-perturbative techniques, strong-coupling expansions [2] and Monte Carlo (MC) simulations [3] being the most notable ones. Monte Carlo simulations, indeed, have probably produced the most important results for QCD, being able to probe the structure of the theory in the domain where the transition between the strong-coupling behavior at large distances and the asymptotically-free behavior at small separation takes place.

Numerical methods must be used to explore the vary crucial domain of intermediate couplings, since there are no known analytical techniques for solving or even efficiently approximating gauge theories throughout that region. On the other hand the fact that quantum fluctuations on a finite lattice extending for n sites in four dimensions are given by integrals of a dimensionality $4n^4n_g$ (n_g is the number of independent parameters in group space), which can easily exceed 2,000,000, leaves importance sampling, i.e. Monte Carlo simulations, as the only calculational possibility.

Monte Carlo calculations are of a numerical nature, and quite demanding on computational resources. The simulation of a system with SU(3) gauge group (i.e. the system underlying QCD) on a lattice extending for n sites in each of the four space-time dimensions requires storage of $4n^4$ link variables, i.e. $4n^4$ SU(3) matrices, and the systematic

replacement, or "upgrading", of each of these matrices with new, updated values, for several hundred or several thousand sweeps of the whole lattice. One MC iteration is defined as a sweep of the lattice, i.e., one upgrade per link variable. A computation involving M MC iterations thus implies $4Mn^4$ individual upgrades of SU(3) matrices. The upgrading of each SU(3) matrix requires approximately 4,150 elementary arithmetic operations and 180 table look-ups (if 10 attempts at changing the link variable are made for each upgrade). For a lattice large enough for obtaining physically meaningful results, the amount of computation needed for a Monte Carlo simulation of QCD becomes extremely high.

Because of the aforementioned difficulties, Monte Carlo simulations of QCD have been generally limited to lattices of rather small extent, a lattice of 8^4 sites already representing a large lattice with respect to the scale of most calculations. On the other hand, with the progress in the field, it has become apparent that one must definitely analyze larger systems to develop confidence in the numerical results. This need may be understood on physical grounds. If 2 GeV is considered as a universal energy for the effects of asymptotic freedom to begin manifesting themselves, one would like the lattice spacing to be smaller than $(2\text{GeV})^{-1}$ (and the corresponding ultraviolet cut-off larger than 2GeV) i.e. smaller than 0.1fm. Conversely, if the goal of the computations is to determine hadronic structure, the extent of the lattice should be larger than the typical size of a hadron. Taking this size to be (minimally) 1 fm, it becomes apparent that the parameter n ought to be larger, if possible substantially larger, than 10. With, e.g., n = 16 and M = 1000 the calculation of a MC simulation requires more than 10^{12} operations not a small task even for the largest machines currently available.

The number of the data elements involved, and the amount of computations needed for manipulating this data, makes it worth while to investigate ways for vectorization of the code.

The purpose of this article is to illustrate the vectorization and implementation on the CDC CYBER 205 of a code for Monte Carlo simulations of the SU(3) lattice gauge theory. (For previous implementations of vectorized code see Ref.4.) As will be discussed in more detail in the final section of this paper, the characteristics and performance are such that 1 MC iteration of a 16^4 lattice can be done in 10.72 seconds (corresponding to an upgrade time of 40.9 μsec per SU(3) link variable). Thus, 16^4 and larger lattices can be considered for meaningful simulations of QCD. While we describe in this article the program for the basic Monte Carlo algorithm, we are currently using it, together with other vectorized codes, for a reevaluation on a large lattice, of several quantities of theoretical and phenomenological interest in QCD. The results of these investigations will be presented separately [5]. Here we proceed with a description of the computational algorithm and an outline of its vectorization in Sect. 2, with a more detailed account of the program in Sect. 3 and a summary of performance data in Sect. 4.

2. The Monte Carlo Algorithm

We consider a hypercubical lattice of n_s sites in each of the three spatial directions and n_t sites in the temporal one. The dynamical variables of the SU(3) gauge theory are 3×3 unitary-unimodular complex matrices, which are associated with the $4n_s^3n_t$ links of the lattice. We denote by U_x^μ the matrix associated with the

oriented link coming out of the lattice site of (integer) coordinates $x \equiv (x_1,x_2,x_3,x_4)$ in the direction μ (μ=1,2,3,4). The goal of the Monte Carlo algorithm is to produce a stochastic sequence of configurations of the system $C^{(i)}$, (a configuration being defined as the collection of all U_x^μ), such that the probability $P(C)$ of encountering any configuration C in the sequence approaches, after a reasonable equilibriation time, the distribution

$$P(C) \propto \exp\{-S(C)\} \quad , \tag{2.1}$$

where S is the action of the configuration C in that sequence. S is given by a sum over plaquette variables p, a plaquette being an oriented square of the lattice defined by the origin x and two directions μ and ν :

$$S = \sum_p S_p = \beta \sum_p (1 - \frac{1}{3} \text{Re Tr } U_p) \quad , \tag{2.2}$$

where

$$U_p \equiv U_x^{\mu\nu} = U_x^{\nu +} U_{x+\hat{\nu}}^{\mu +} U_{x+\hat{\mu}}^{\nu} U_x^{\mu} \quad , \tag{2.3}$$

β is the coupling parameter and $\hat{\mu}$, $\hat{\nu}$ stand for unit lattice vectors in the μ and ν directions, respectively. When Eqn. 2.1 is satisfied, quantum mechanical expectation values of observables $\mathcal{O}$, defined rigorously as averages over all possible configurations, namely

$$\langle \mathcal{O} \rangle = Z^{-1} \int \prod_{x,\mu} dU_x^\mu \, \mathcal{O}(U) \exp[-S(U)] \qquad (2.4)$$

with

$$Z = \int \left(\prod_{x,\mu} U_x^\mu \right) \exp[-S(U)] \quad , \qquad (2.5)$$

can be approximated by averages taken over the configurations generated by the Monte Carlo algorithm:

$$\langle \mathcal{O} \rangle \simeq \frac{1}{N} \sum_{i=N_0+1}^{i=N_0+N} \mathcal{O}(C^{(i)}) \quad . \qquad (2.6)$$

N_0 represents the number of initial configurations discarded in order to allow for the stochastic sequence to reach equilibrium.

In our code we implement the MC algorithm following the method of Metropolis et al [6]. Each individual dynamical variable U_x^μ is replaced by a new one $\tilde{U}_x^\mu$ according to the following procedure:

i) a new candidate matrix $U_x^{\mu\,\prime}$ is obtained from U_x^μ by group multiplication:

$$U_x^{\mu\,\prime} = R_k U_x^\mu \quad ,$$

where R_k is an $SU(3)$ matrix randomly selected from a prepared set $\{R_1, \ldots, R_M\}$ of M matrices, to be discussed later.

ii) the change in action, ΔS induced by the variation $U_x^\mu \to U_x^{\mu\,\prime}$ is calculated:

$$\Delta S = S(U_x^{\mu\,\prime},\ldots)-S(U_x^\mu,\ldots); \qquad (2.7)$$

iii) a pseudorandom number r with uniform distribution between 0 and 1 is generated and

$$\tilde{U}_x^\mu = U_x^{\mu\,\prime} \text{ if } r < \exp(-\Delta S) \ ,$$

$$\tilde{U}_x^\mu = U_x^\mu \text{ otherwise.}$$

The steps i) to iii) define what is called a "hit" on one of the link variables. These steps are repeated N_h (number of hits) times. This completes the upgrading of one (link) variable U_x^μ. One MC iteration (or one sweep of the lattice) is executed when all the variables have been processed in this manner.

A crucial consideration for the whole algorithm and also for its vectorization is that the calculation of the variation of the action ΔS involves only a few of the dynamical variables apart from U_x^μ itself, namely those defined on the remaining links of the six plaquettes which share the link between x and $x+\hat{\mu}$. It is convenient to be slightly detailed at this point and to introduce some terminology. Given the link from x to $x+\hat{\mu}$ there are three "forward" plaquettes incident on it, namely those with vertices

$$x,\ x+\hat{\mu},\ x+\hat{\mu}+\hat{\nu} \text{ and } x+\hat{\nu} \ ,$$

(ν taking the three values $\neq \mu$) and three "backward" plaquettes, namely those with vertices

$$x, \quad x+\hat{\mu}, \; x+\hat{\mu}-\hat{\nu} \text{ and } \quad x-\hat{\nu} \quad ,$$

(see Fig. 1).

We shall define as the "force" acting on U_x^μ the sum of the expressions

$$F_{f,x}^{\mu\nu} = U_{x+\hat{\mu}}^{\mu\dagger} \, U_{x+\hat{\nu}}^{\mu} \, U_x^{\nu} \tag{2.8}$$

(corresponding to the forward plaquettes) and

$$F_{b,x}^{\mu\nu} = U_{x+\hat{\mu}-\hat{\nu}}^{\nu} U_{x-\hat{\nu}}^{\mu} U_{x-\hat{\nu}}^{\nu\dagger} \tag{2.9}$$

(corresponding to the backward plaquettes) over the three values of $\nu \neq \mu$

$$F_x^\mu = \sum_{\nu\neq\mu} (F_{f,x}^{\mu\nu} + F_{b,x}^{\mu\nu}) \quad . \tag{2.10}$$

One can easily convince oneself that of the terms contributing to the action in Eqn. 2.2 all those containing U_x^μ can be written in the form

$$\beta[1 - \tfrac{1}{3} \, \mathrm{ReTr}(F_x^{\mu\dagger} U_x^\mu)] \quad , \tag{2.11}$$

and therefore

$$\Delta S = -\frac{\beta}{3}\,\mathrm{ReTr}[F_x^{\mu\dagger}(U_x^{\mu\prime} - U_x^{\mu})] \quad . \tag{2.12}$$

Thus, we become aware of two fundamental facts:

i) once the force F_x^{μ} is calculated, the N_h subsequent hits on the link variable U_x^{μ} can be done without any further recourse to the values of other U variables.

ii) several upgradings can be done in parallel, provided only that the forces F_x^{μ} required for the computation do not involve any of the U_x^{μ} variables that are simultaneously upgraded.

While point i) is relevant for any MC simulation, point ii) acquires particular importance if one wants to write a vectorized code. Indeed, as we shall show, all U_x^{μ} variables with fixed μ can be separated into two sets such that the forces for one set only involve elements of the other. Then, all the U_x^{μ} variables belonging to one set can be grouped together in an array and upgraded simultaneously. Finally one proceeds to upgrade the elements of the other set (the red-black or checkerboard algorithm [4]). We will see in the next section that the ability to separate the link variables into two independent sets is a key to efficient vectorization.

3. The Vectorized Implementation of the Algorithm

The previous discussion has demonstrated that Monte Carlo lattice gauge theories are worthy candidates for vector processing. Until recently, however, people were doubtful as to whether the vector capabilities of current

supercomputers can be effectively utilized for such applications. The main source for this skepticism is the inherent conflict between random access to data, an integral part of a Monte Carlo process, and the strict order of data elements required for pipelined computations. In other words, unless data can be "gathered" at rates comparable to computation rates no efficient vectorization can be achieved.

One of the major strengths of the CDC CYBER 205, and what makes it a particularly powerful Monte Carlo machine, is the ability to order a random collection of data by means of a vector instruction, namely, the "Gather" instruction. This instruction is equivalent to a series of random, or, indirect "load" operations on a serial computer. The Gather instruction uses a vector of integers as an "index-list" pointing to the elements to be fetched. These elements are stored in the order they have been encountered into an output vector. The result rate for the Gather operation is one element every 1.25 cycles (a cycle, or clock-period on the CDC CYBER 205 is 20 nonoseconds). For a comparison, note that the floating-point arithmetic rate, excluding division, is one element every cycle per pipe for 64-bit operands. The CYBER 205 hardware also supports 32-bit operations with twice the result rate for vector floating-point operations. For example, on a two pipe machine 32-bit arithmetic is performed at a rate of 5 nsec per result, or 200 MFLOPS.

The effective utilization of the computational tools build into the vector processor is closely related to the data structure, as are most of the important algorithmic decisions. It is, therefore, appropriate, at this point, to discuss the memory requirements. A 3×3 complex matrix is represented by 18 real numbers. The constraints

of being unitary and unimodular reduce the number of independent parameters to 8, but such a minimal representation of the U_x^μ variables implies a substantial increase in the computational complexity. To obtain optimal performance it is useful to keep all the 18 values representing the real and imaginary parts of the elements of U_x^μ . For a lattice with $n_s = n_t = 16$ a configuration will be defined by $18 \times 4 \times 16^4 =$ 4.718592 million values, which may be more than can be put in the fast memory of many computer systems. Fortunately, the sequential nature of the MC algorithm suggests that only a fraction of the variables need to be in memory at any one time. The others can be kept on disk. The factors which determine an optimal size for the partition between variables in memory and on disk are the following:

i) the partition should not make the code unnecessarily complicated;

ii) the I/O operations should not take longer than the actual computations;

iii)sufficiently long vectors should be available.

On the basis of the above requirements we decided to upgrade one space at a time, i.e. to upgrade all the $4n_s^3$ variables U_x^μ with fixed time coordinate x_4 , and then to proceed to the next x_4 etc. We shall refer to this procedure as time-slicing and to the collection of variables with fixed time coordinate x_4 as one time-slice of the system. If the variables with a given $x_4 = t$ are being upgraded, the calculation of the force requires knowledge of the U_x^μ with $x_4 = t-1$, $x_4 = t$ and $x_4 = t+1$. Thus 3 time slices need to be in memory throughout this stage of the calculation. As a matter of fact, since I/O operations can proceed independently from CPU operations, it

is possible to achieve concurrency of I/O and CPU operations if extra memory buffer space is allocated for holding the $x_4 = t-2$ slice (to be written out), and the $x_4 = t+2$ slice (to be read in). The conventional way of implementing concurrent I/O is to allocate space for two more slices. The resulting five slices in memory act as a circular buffer as shown in Fig. 2. However, the virtual memory hardware on the CDC CYBER 205, and the supporting software provide the capability to swap data between disk and memory. Hence, the memory area of one slice only is needed to write out the $x_4 = t-2$ slice, and read in the $x_4 = t+2$ slice. Consequently, the total memory requirements for the link variables are thus $4 \times n_s^3 \times 4 \times 18$ locations. Allowing for some additional work-space we find that lattices with $n_s = 16$ can be considered in a machine with 2m words (16m bytes) in full precision (64-bit words) and $n_s = 20$ in half precision (32-bit words). The length in time does not constitute a problem any longer and lattices with any n_t may be simulated.

With the slicing mechanism in place we now turn to vectorization aspects of the code. In Sec. 2, the Red-Black ordering was introduced. The motivation for this choice merits some discussion. The computation involves, mainly, matrix multiplications. This operation is easily vectorized, but the matrices concerned are 3×3 matrices, and the resulting vectors are going to be 3 elements long. For efficiently vectorized code one needs to seek longer vectors. This results from the observation that the timing formula for a vector instruction may be written as

$$(\text{Start-up} + \alpha \cdot N) \text{ cycles} \qquad (3.1)$$

where the start-up time is a constant, independent of the vector length. It amounts to aligning the input and output streams, filling up the pipelines up to the point where the first result is available and storing the last result. The start-up time is also independent of the number of pipelines and whether 64-bit or 32-bit arithmetic is performed. On the CDC CYBER 205 it amounts to about 50 cycles, or 1 μsec. The "$\alpha \cdot N$" term is known as the "stream time". N is the number of elements in the vector, so that the stream time is proportional to the vector length. α is a constant associated with the number of pipelines and the arithmetic mode. Table 3.1 contains the α values for some relevant circumstances. It is now obvious that high performance is achieved by minimizing the number of "start-ups" as a consequence of using longer vectors, or, increasing N for each vector operation.

The SU(3) matrices are too small as an object for vectorization; however, there are n_s^3 such matrices in every time slice. One cannot use all of these link values simultaneously because -

i) updating each link requires all its immediate neighbors, and

ii) the correct convergence of the Metropolis process depends upon using "new" values as soon as they are available.

The Red-Black (checker-board) ordering resolves this apparent recursive relationship. The separation of the U_x^μ variables into two sets, for each value of μ and at fixed x_4, is achieved by putting in the two sets all the variables belonging to links originating from odd and even sites, respectively, i.e. with $x_1 + x_2 + x_3 = 1$ or 0 (mod 2). This assures the independence of the forces F_x^μ from the variables U_x^μ being upgraded. On a lattice with $n_s = 16$ the above separation gives a vector length of $n_s^3/2 = 2048$, sufficiently large to insure almost optimal

performance (in fact, 91% and 95% in 32-bit and 64-bit arithmetic, respectively). The calculation of the force F_x^μ requires knowledge of the U_x^μ variables associated with links neighboring the one under consideration. Because of boundary conditions, which we take to be periodic, the variables which enter the calculation of F_x^μ will not, in general, have a simple location-index relative to U_x^μ in the array of dimension $n_s^3/2$. This is easily remedied by the introduction of auxiliary integer-valued arrays, where the indices of the various neighbors of U_x^μ are prestored. The Gather instruction plays a crucial role in the way these index arrays are used. When F_x^μ is evaluated, all the needed variables are gathered into temporary arrays, so that the indices of all elements entering into the computation of F_x^μ are the same, and this proceeds in a fully vectorized manner.

Once the F_x^μ's are determined the algorithm for the upgrading of all the U_x^μ (in the same set) is straightforward and completely vectorizable. The matrices R which are used for finding the new candidates $U_x^{\mu\,\prime}$, are Gathered according to an array of indices extracted at random from a table. The table contains M $SU(3)$ matrices which have a distribution centered around the identity of the group and are obtained in the following fashion. For each value of i between 1 and $M/2$ (M must be even) an eight component vector V_k with approximately gaussian distribution and $\langle V_k^2 \rangle = 1$ is pseodoramdomly generated. The fourth-order approximation to R_i is given by

$$R_i^0 \simeq 1 + iA - \frac{A^2}{2} - \frac{iA^3}{3!} + \frac{A^4}{4!} \simeq \exp(iA) \quad , \tag{3.2}$$

where

$$A = b \sum_k V_k \lambda_k \quad . \tag{3.3}$$

λ_k are Gell-Mann's matrices (i.e., a set of generators of the Lie algebra of SU(3)) and b is a real parameter specifying the spread of the distribution. The final value for R_i is obtained by normalizing R_i^0 to a unitary-unimodular matrix. In general, if we denote the three columns of an SU(3) matrix by $\vec{r}_1, \vec{r}_2$ and $\vec{r}_3$ the constraint of being unitary and unimodular is expressed by

$$\begin{aligned} |\vec{r}_1|^2 &= |\vec{r}_2|^2 = 1 \\ \vec{r}_1 \cdot \vec{r}_2^* &= 0 \\ \text{and} \qquad \vec{r}_3 &= (\vec{r}_1 \times \vec{r}_2)^* \quad . \end{aligned} \tag{3.4}$$

Given a matrix R^0 with the first two columns $\vec{r}_1^0$ and $\vec{r}_2^0$, with $\vec{r}_1^0 \times \vec{r}_2^0 \neq 0$, we shall define as the normalized form of R^0 the matrix R with columns

$$\begin{aligned} \vec{r}_1 &= \vec{r}_1^0 / \sqrt{|\vec{r}_1^0|^2} \\ \vec{r}_2 &= \{\vec{r}_2^0 - \vec{r}_1 (\vec{r}_1^*, \vec{r}_2^0)\} / \sqrt{|\vec{r}_2^0 - \vec{r}_1 (\vec{r}_1^* \cdot \vec{r}_2^0)|^2} \\ \text{and} \qquad \vec{r}_3 &= (\vec{r}_1 \times \vec{r}_2)^* \end{aligned} \tag{3.5}$$

The reason for the nonmenclature is due to the fact that, if R^0 differs slightly from a unitary-unimodular matrix, e.g. as a consequence of roundoff errors, then R is an $SU(3)$ matrix close in value to R^0 Thus, the approximately unitary-unimodular matrix R_i^0 obtained by truncated exponentiation in Eqn. 3.2 is converted to a proper SU(3) matrix R_i by normalization. The last $M/2$ matrices are obtained by

$$R_{\frac{M}{2}+i} = R_i^{+} \qquad (1 \leq i \leq \frac{M}{2}) \tag{3.6}$$

so as to insure that, together with any given matrix R_i , the inverse should also belong to the table.

The procedure for normalizing the $SU(3)$ matrices of the random table, as described above, is also applied, every few iterations, to the link matrices. This is done to insure that the group symmetry of the matrices is preserved regardless of rounding errors which may be introduced by the hardware after many arithmetic operations. This renormalization process is particularly important when the computations are performed using low precision arithmetic. It gives us confidence, which was also tested and verified, in using 32-bit arithmetic for our calculations on the CDC CYBER 205.

Once $U_x^{\mu'}$ is determined, using the table of random $SU(3)$ matrices, the action difference is obtained by calculating, separately,

$$\mathrm{ReTr}(F_x^{\mu+} U_x^{\mu}) \quad \text{and} \quad \mathrm{ReTr}(F_x^{\mu+} U_x^{\mu'})$$

(notice that $\mathrm{ReTr}(A^{+}B)$ is the vector product of the arrays containing

the real and imaginary parts of A and B) , forming an array with exp(-ΔS) , comparing with an array of pseudorandom numbers and accepting or rejecting the change, via a masking operation, according to the outcome of the vectorized comparison between the random numbers and the exponentiated action differences. These steps are repeated for a prefixed number of hits before commencing the upgrade of the other set or the variables corresponding to different directions.

The conditional acceptance of elements in a vector, or, the masking operation referred to above, is handled through the usage of a "bit-vector" (the CDC CYBER 205 is bit addressable and the software allows the Fortran user to use this feature). It is exploited as a part of the vector instruction, and inhibits storing results where zeros are encountered in the bit-vector.

The reader should by now realize that many thousands of random numbers are required for each iteration. The conventional congruent method for generating random numbers is recursive, and may be described by

$$y_{i+1} = (a \cdot y_i) \bmod (b) \qquad (3.7)$$

where a is the "multiplier" and b is determined so as y_{i+1} will be approximately the lower half of the coefficient of the product $a \cdot y_i$ The nature of this calculation suggests that in order to produce N random numbers one has to repeat it serially N times. There is, however, a way to reproduce the same sequence of N numbers in parallel, using vector instructions [7]. Define a new multiplier by

$$A = (a^N)\mathrm{mod}(b)$$
$$= (\ldots(a*a)\mathrm{mod}(b)*a)\mathrm{mod}(b)\ldots*a)\mathrm{mod}(b) \qquad (3.8)$$

and let

$$\underline{Y}_1 = (y_1, y_2, \ldots, y_N) \qquad (3.9)$$

be the vector containing the first N random numbers.
Then

$$\underline{Y}_{i+1} = (A*\underline{Y}_i)\mathrm{mod}(b) \qquad (3.10)$$

reproduces the same sequence of random numbers one gets with a repeated application of Eqn. 3.7 (the computation of Eqn. 3.10 requires only 3 vector operations on the CDC CYBER 205).

To conclude this section, let us discuss the way matrix multiplication is done, being the most time-consuming aspect of the computation . First, the reader will remember that we do not vectorize the matrix multiplication as such, but, rather, perform the operations on many matrices in parallel, where for each matrix the "scalar" sequence of operations is followed.

When computing the products of two SU(3) matrices, one need not evaluate all the columns of the result, since the third column of the product matrix (which is again unitary-unimodular) is related to the first two by Eqn. 3.4. In the code we have exploited this fact whenever possible. It is particularly advantageous when several SU(3) matrices must be multiplied together, since one may limit the calculations to two columns out of three in all intermediate products and simply reconstruct the third column of the final result as shown in Eqn. 3.4.

Finally, all complex arithmetic has been done in terms of real variables, separating real and imaginary parts (which would also result in a more efficient code for a scalar machine), and we have used the identity

$$(A+iB)(C+i\,D) = (A+B)(C-D)-BC + AD + i(BC+AD) \qquad (3.11)$$

to perform the product of two complex matrices in terms of three real multiplications and five real matrix additions. Using complex arithmetic the product of two matrices would require four real multiplications and two additions. Due to the fact that matrix multiplication requires $2N^3$ operations, where N is the dimension of the matrix, and matrix addition requires only N^2 operations, our method pays off even for $N = 3$.

A schematic outline of the flow of the calculations is shown in Fig. 3.

4. Performance and Timings

The figures quoted here are based on runs executed on a two-pipe, 2m 64-bit words CDC CYBER 205. They apply to a 16^4 lattice ($n_s = 16$, $n_t = 16$), SU(3) gauge theory with 10 hits per link upgrade (unless stated explicitly otherwise). We present performance figures for both 64-bit and 32-bit arithmetic operations. In both modes the exponentiation and the generation of random numbers were carried out using 64-bit arithmetic. It should be noted here that due to our slicing mechanism the 32-bit version requires real memory of only 852,000 words (64-bit words, or 6.8m bytes), so it actually fits comfortably on a 1m words system. With these parameters

the code performs at 98% CPU utilization. The 64-bit version requires, of course, twice as much memory.

In Table 4.1 we give the percentage of the execution time for the two arithmetic modes spent in the force (F_x^μ) and the Metropolis updating calculations. It becomes clear from these figures why it is worth while using a single force computation for a number of attempts at updating (rather than the one attempt proposed by the original Metropolis method).

It should be added here the normalization procedure discussed in Sec. 3, performed every 5 iterations adds only 0.74% and 0.59% in 64-bit and 32-bit modes, respectively, to the total execution time.

Table 4.2 presents a percentage breakdown of the code by operation type. The reader will notice that the Gather, random number generation and the exponentiation operations are more heavily weighted in the 32-bit mode compared with that of the 64-bit mode. These three types of operations perform at the same rate in both modes. The last two execute in 64-bit mode in both versions of the code. The Gather instruction performs at the same rate regardless of whether the operands are 64-bit or 32-bit variables. This is because the performance of the Gather operation is driven by memory access (and not by computation complexity). The matrix multiplication, being made up of floating-point operations only, executes at near peak rate of 95 MFLOPS and 182 MFLOPS for the 64-bit and 32-bit modes,respectively. The effect of vectorizing the random number generator can be illustrated by noting that this operation amounted to 6% (64-bit) and 11% (32-bit) of the total time when it was not vectorized. The "action" involves taking the real part of the trace of products of SU(3) matrices (purely floating-point operations). The "acceptance" is the portion of the code where the conditional acceptance

of new U_x^μ matrices occurs under the control of a bit-vector created for that purpose.

The actual time for one iteration of the 16^4 lattice with 10 hits is 16.27 secs. (64-bit) and 10.72 secs. (32-bit). This amounts to a substained performance rate of 66.8 MFLOPS (64-bit) and 101.5 MFLOPS (32-bit). Another way, commonly used by physicists, to express the performance of Monte Carlo lattice gauge theories implemented on a computer system, is the link update time, i.e., the time needed to update one link of the lattice once. This measure is useful for comparisons since it is independent of the lattice size. The link update times (in μsecs.) for our implementation are given in Table 4.3. These figures may be compared to a link update time of about 1,100 μsecs on the CDC 7600 computer system with a highly optimized code.

Acknowledgements

We would like to thank Control Data Corporation for awarding time on the CDC CYBER 205 at the Institute for Computational Studies at Colorado State University where the code described in the text was developed. One of the authors (K.J.M.M.) would like to thank Dalhousie University for the award of a Visiting Fellowship which made his visit to Fort Collins, Colorado possible. This research was also carried out in part under the auspices of the US Department of Energy under contract No. DE-AC02-76CH00016.

Table 3.1. Stream rate proportionality factor (α) .

No. of pipes \ Arithmetic mode	64-bit	32-bit
2	1/2	1/4
4	1/4	1/8

Table 4.1. Breakdown by percentage of sections of code.

	64-bit	32-bit
force	43.49	42.46
update	56.40	57.40

Table 4.2. Breakdown by percentage of the main operation types.

operation type	64-bit	32-bit
matrix multiplication	58.33	47.05
Gather	20.78	29.27
random number generator	0.95	1.83
exponentiation	7.43	11.72
action	5.93	4.70
acceptance	3.62	3.01

Table 4.3. The upgrades times for a link (in μsecs).

number of hits	64-bit	32-bit
10	62.1	40.9
8	55.1	36.3

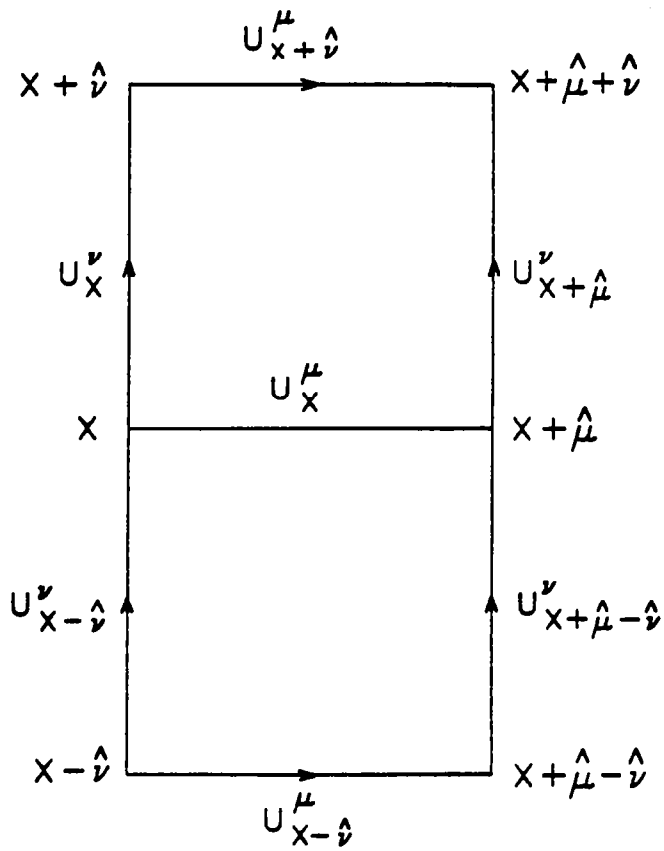

Figure 1. "Forward" (upper half) and "backward" (lower half) plaquettes in the μ-ν plane, where $x \equiv (x_1,x_2,x_3,x_4)$ is a point in our four-dimensional lattice. This is one out of three such planes which can be formed in a four-dimensional space.

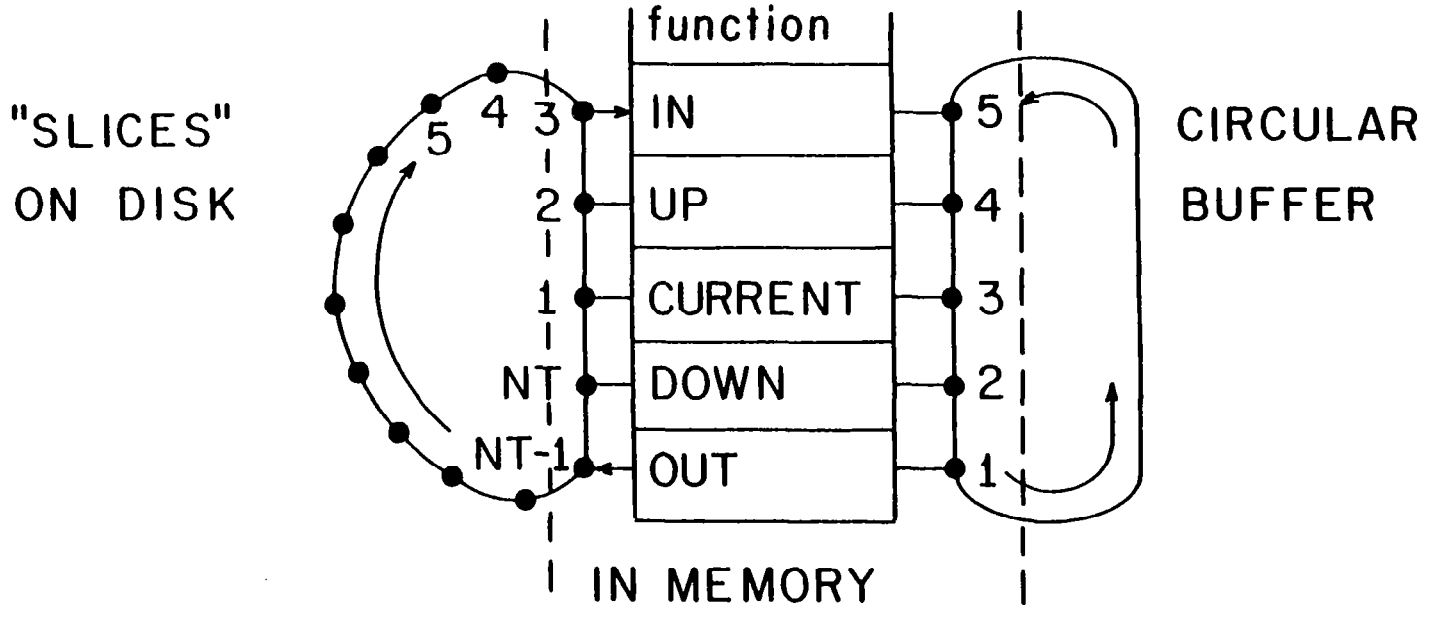

Figure 2. The I/O scheme. In our implementation the "in" and "out" boxes occupy the same physical memory.

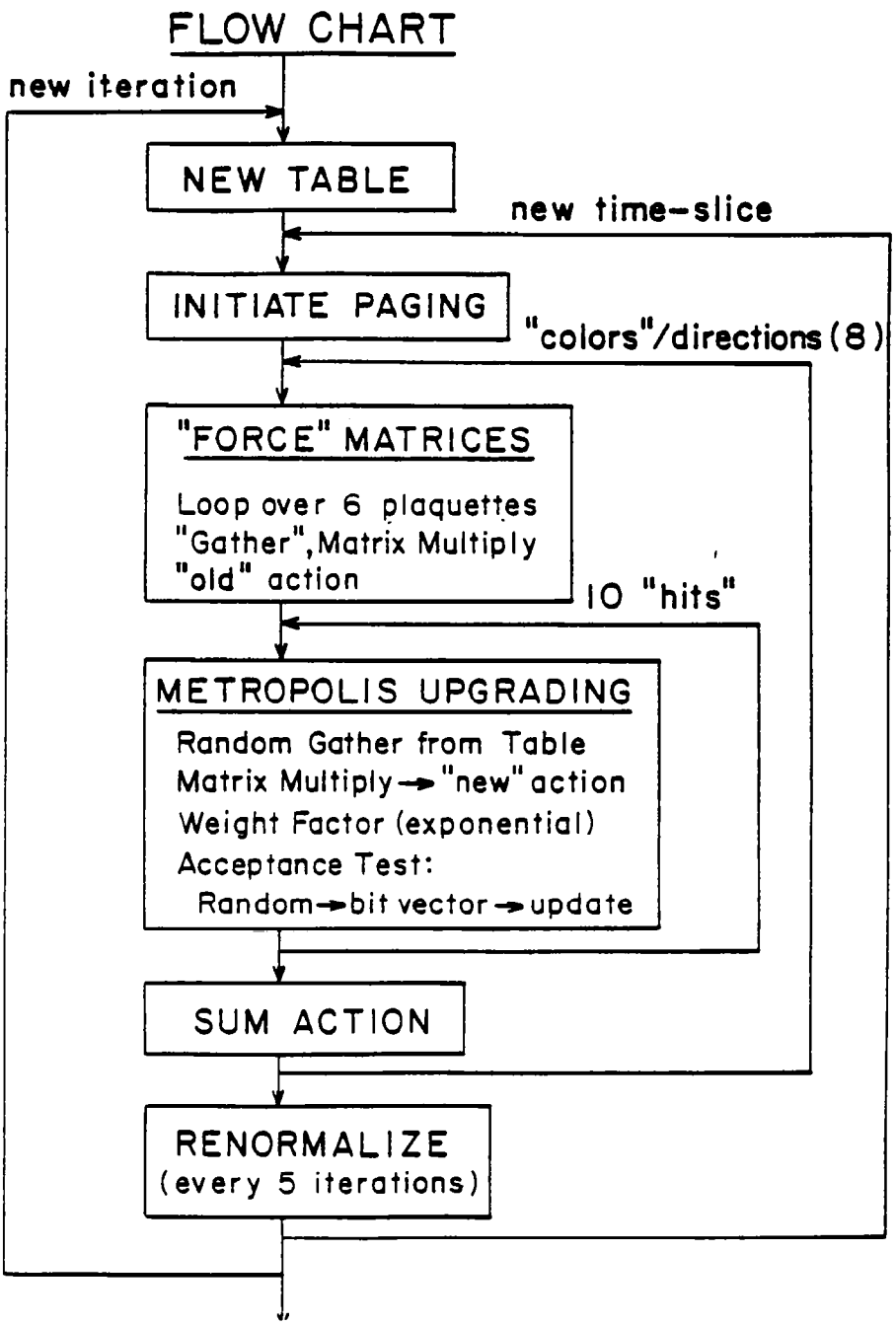

Figure 3. Schematic description of the computational process.

References

[1] K.G. Wilson, Phys. Rev. D10, 2445 (1974).

[2] J.-M. Drouffe and J.-B. Zuber, Phys. Reports (to be published).

[3] M. Creutz, L. Jacobs and C. Rebbi, Phys. Reports 95, 201(1983).

[4] D. Barkai and K.J.M. Moriarty, Comput. Phys. Commun. 25, 57(1982); 26, 477(1982); 27, 105(1982); D. Barkai, M. Creutz and K.J.M. Moriarty, Comput. Phys. Commun. 30, 13(1983).

[5] D. Barkai, K.J.M. Moriarty and C. Rebbi, (to be published).

[6] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller and E. Teller, J. Chem. Phys. 21, 1087(1953).

[7] Forrest Brown, private communication.