A MODIFIED CONJUGATE GRADIENT SOLVER FOR VERY LARGE SYSTEMS^{*)}

D. Barkai

Control Data Corporation at the
Institute for Computational Studies at CSU
Fort Collins, CO 80522, USA

K.J.M. Moriarty⁺⁾

Statistical Mechanics Group
Lawrence Livermore National Laboratory
Livermore, CA 94550, USA

and

C. Rebbi^{")}

CERN - Geneva

A B S T R A C T

A modified conjugate gradient method is derived which requires only one pass through the coefficients and the temporary vectors. The method is applicable to problems which may be complex and non-symmetric. The method is implemented on a vector processor (the CDC CYBER 205) and applied to a high energy physics lattice gauge theory problem, though the implementation methodology is quite general.

^{*)} To be published in Computer Physics Communications. This work has also been presented at the International Supercomputer Applications Symposium, held at Purdue Center for Parallel and Vector Computing, West Lafayette, IN, October 31-November 1, 1984, and Stichting Academisch Rekencentrum, Amsterdam (SARA), November 7-9, 1984.

⁺⁾ Permanent address: Institute for Computational Studies, Department of Mathematics, Statistics and Computing Science, Dalhousie University, Halifax, Nova Scotia B3H 4H8, Canada.

^{")} Permanent address: Department of Physics, Brookhaven National Laboratory, Upton, NY, U.S.A.

1. Introduction

In the course of establishing a suite of programs for measuring physical observables predicted by the lattice formulation of Quantum Chromodynamics (QCD), it became necessary to develop an efficient solver for very large systems of linear equations. The outcome of this search turned out to be quite general and the detailed structure of the coefficients matrix to be inverted is incidental to the method. The only assumptions made here regarding this matrix are: (i) the size of the problem is large enough to warrant an iterative method, and (ii) there exist simple rules for applying the "operator" (i.e. the coefficients matrix) to a solution vector, without decompositions etc., so as to make a method like the Conjugate Gradient (CG) attractive.

The problem in the application of the conjugate gradient method to systems of very large extent is that all the relevant variables cannot be kept in fast memory; in the course of the iterations the data must then be repeatedly brought in and out of mass storage device. In the standard implementation of the algorithm two passes through the data are required at each iteration and this makes the whole computation strongly I/O bound. The main purpose of this paper is to illustrate a modification of the CG method which dramatically reduces the I/O requirements (by 50%), thus making the procedure substantially more efficient.

In the next section we derive a modified CG procedure based on a fairly simple-minded conventional CG algorithm, which also allows for a complex non-symmetric coefficients matrix, and such that only one pass through the data is needed for each iteration. This is the essence of this paper - a procedure that halves the "cost" of solving a very large system of linear

equations utilizing a CG method. Section 3 contains general considerations and details for implementing the modified algorithm on a powerful vector processor. The discussion there is also relevant to other parallel computer architectures. Finally, in section 4 we present a specific high-energy physics problem, to which the new method has been applied, its structure and performance figures on the CDC CYBER 205.

2. Derivation of the Modified Algorithm

The CG method proposed by Hestenes and Stiefel[1] may be regarded as a family of methods, each corresponding to an acceleration process for a particular basic iterative procedure. In the literature one can find slightly different presentations of the method, almost always restricted to the case of a symmetric-positive-definite (SPD) coefficients matrix [2,3,4]. It is, therefore, useful to establish our notation and formulation. We wish to find a vector f satisfying

$$Mf = b \quad (2.1)$$

where M is $N \times N$ matrix and f and b are vectors of length N ; M and b are given. Lower case variables, with and without a superscript, will denote vectors; lower case variables with a subscript are scalar quantities. Upper case letters are reserved for matrices. Let M be SPD and initialize the "residual" vector $u^{(0)} = b$, the "direction" vector $p^{(0)}$ to an arbitrary initial guess of the solution and the "solution" vector $f^{(0)} = 0$. Then iteration n of the CG method is given by

$$\begin{aligned}
 \alpha_n &= \frac{(u^{(n-1)}, u^{(n-1)})}{(p^{(n-1)}, M p^{(n-1)})} \quad \text{for } n > 1 ; \alpha_1 = 1 \\
 f^{(n)} &= f^{(n-1)} + \alpha_n p^{(n-1)} \\
 u^{(n)} &= u^{(n-1)} - \alpha_n M p^{(n-1)} \\
 \beta_n &= \frac{(u^{(n)}, u^{(n)})}{(u^{(n-1)}, u^{(n-1)})} \quad \text{for } n > 1 ; \beta_1 = 0 \\
 p^{(n)} &= u^{(n)} + \beta_n p^{(n-1)} .
 \end{aligned} \tag{2.2}$$

With a few renamings and simple manipulations one can establish the equivalence between this formulation and the one given e.g. in Ref. 2. We wish to consider very large systems, such that the vectors used and the non-zero elements of M cannot all fit into the real memory of the computer system. The problem has, then, to be "sliced", i.e. the variables (vector and matrix elements) must be subdivided in sets (slices), according to some reasonable partitioning of the matrix, and the computations must be carried out for one slice at a time. It is apparent, however, that it is not possible to execute a whole iteration with a single loop over slices in the algorithm above. For example, $u^{(n)}$ has to be known for all slices before β_n can be computed, and β_n itself is required in this iteration for computing each slice of $p^{(n)}$. The reader will notice that α_n may be computed at the end of the $(n-1)$ iterations, and, therefore, only two (rather than three) passes through all the slices are needed for the completion of each iteration.

We wish to derive a formulation where each iteration can be done with just one pass through the various vectors. Before getting into that let us note that the coefficients in M (representing, typically, most of the data to be accessed) are required only once - for computing the vector $M p^{(n-1)}$. This is true, however, only when M is SPD. The formulation for the case when M is not SPD is achieved by the following modification to Eqns. (2.2):

introduce an additional vector $r = Mp$; change the denominator of α_n to $(r^{(n-1)}, r^{(n-1)})$; change u in the expressions for α_n , β_n and $p^{(n)}$ to M^+u , and denote $|d|^2 = (d,d)$. We also allow M to be a complex matrix. With these changes the algorithm may be written as:

$$\begin{aligned}
 \text{initialization: } f^{(0)} &= 0, u^{(0)} = b, p^{(0)} = x \\
 r^{(n)} &= Mp^{(n-1)} \\
 \alpha_n &= \frac{|s^{(n-1)}|^2}{|r^{(n)}|^2} \quad \text{for } n > 1, \alpha_1 = 1 \\
 f^{(n)} &= f^{(n-1)} + \alpha_n p^{(n-1)} \\
 u^{(n)} &= u^{(n-1)} - \alpha_n r^{(n)} \\
 s^{(n)} &= M^+u^{(n)} \\
 \beta_n &= \frac{|s^{(n)}|^2}{|s^{(n-1)}|^2} \\
 p^{(n)} &= s^{(n)} + \beta_n p^{(n-1)} .
 \end{aligned} \tag{2.3}$$

One can easily verify that when M is real and symmetric Eqns. (2.3) can be converted back to Eqns. (2.2). The changes introduced here are equivalent to considering the problem

$$(M^+M)f = M^+b \tag{2.4}$$

without evaluating M^+M , which may, in general, reduce the sparsity of the coefficients matrix. In any event, the amount of data to be accessed is nearly double for this more general case than it is when M is SPD, since M is required in each of two passes - once for computing $r^{(n)}$, and again for $s^{(n)}$.

We will now proceed to derive the "one-pass" algorithm from Eqns. (2.3). The main idea is to foresee the inner-products needed for the next iteration so that all the computations can be done slice-by-slice without having to compute an inner-product in the course of the current iteration. As we shall see, an incomplete initial iteration (or, more extensive initialization) is added before getting into the modified one-pass iterations. The modified algorithm will be constructed in several stages. First, multiply the expression for $u^{(n)}$ from the left by $M^\dagger$, and use $s^{(n)} = M^\dagger u^{(n)}$. Also, define $q^{(n)} = M^\dagger r^{(n)}$. Eqns. (2.3) then become

$$\begin{aligned}
 f^{(0)} &= 0, \quad p^{(0)} = x, \quad s^{(0)} = M^\dagger b \quad \text{and} \\
 r^{(n)} &= M p^{(n-1)} \\
 \alpha_n &= \frac{|s^{(n-1)}|^2}{|r^{(n)}|^2} \quad \text{for } n > 1; \quad \alpha_1 = 1 \\
 f^{(n)} &= f^{(n-1)} + \alpha_n p^{(n-1)} \\
 q^{(n)} &= M^\dagger r^{(n)} \\
 s^{(n)} &= s^{(n-1)} - \alpha_n q^{(n)} \\
 \beta_n &= \frac{|s^{(n)}|^2}{|s^{(n-1)}|^2} \quad \text{for } n > 1; \quad \beta_1 = 0 \\
 p^{(n)} &= s^{(n)} + \beta_n p^{(n-1)}.
 \end{aligned} \tag{2.5}$$

Next, compute $s_n = |s^{(n)}|^2$ prior to computing the current slice of $s^{(n)}$ by noting that

$$\begin{aligned}
 s_n &= |s^{(n-1)} - \alpha_n q^{(n)}|^2 \\
 &= |s^{(n-1)}|^2 - 2\alpha_n \operatorname{Re} \bar{s}^{(n-1)} \cdot q^{(n)} + \alpha_n^2 |q^{(n)}|^2
 \end{aligned} \tag{2.6}$$

so that β_n can be computed before $s^{(n)}$ is known.

The $f^{(n)}$ computation can be moved down to the lower half of the procedure. What we have now is an equivalent representation of Eqns.

(2.5), namely:

$$\begin{aligned}
 \text{initialization: } f^{(0)} &= 0, p^{(0)} = x, s^{(0)} = M^{\dagger}_b \\
 r^{(n)} &= M p^{(n-1)} \\
 r_n &= |r^{(n)}|^2 \\
 \alpha_n &= s_{n-1}/r_n \quad \text{for } n > 1, \alpha_1 = 1 \\
 q^{(n)} &= M^{\dagger}_r r^{(n)} \\
 s_n &= |s^{(n-1)}|^2 - 2\alpha_n \operatorname{Re} \bar{s}^{(n-1)} \cdot q^{(n)} + \alpha_n^2 |q^{(n)}|^2 \\
 \beta_n &= s_n/s_{n-1} \quad \text{for } n > 1, \beta_1 = 0
 \end{aligned} \tag{2.7a}$$

$$\begin{aligned}
 f^{(n)} &= f^{(n-1)} + \alpha_n p^{(n-1)} \\
 s^{(n)} &= s^{(n-1)} - \alpha_n q^{(n)} \\
 p^{(n)} &= s^{(n)} + \beta_n p^{(n-1)}
 \end{aligned} \tag{2.7b}$$

The last step involves swapping Eqns. (2.7a) and (2.7b), so that Eqns. (2.7a) will serve to compute α_n and β_n for the next iteration. Of course, the initialization will need to be more extensive (or, one may refer to it as an "incomplete" first iteration). The final procedure is:

$$\begin{aligned}
 f^{(0)} &= 0 \\
 p^{(0)} &= x \\
 s^{(0)} &= M^{\dagger}_b \\
 r^{(0)} &= M p^{(0)} \\
 q^{(0)} &= M^{\dagger}_r r^{(0)} \\
 \alpha_0 &= 1 \\
 \beta_0 &= 0
 \end{aligned} \tag{2.8a}$$

begin loop over "regular" iterations

begin loop over slices of data

$$\begin{aligned}
 f^{(n)} &= f^{(n-1)} + \alpha_{n-1} p^{(n-1)} \\
 s^{(n)} &= s^{(n-1)} - \alpha_{n-1} q^{(n-1)} \\
 p^{(n)} &= s^{(n)} + \beta_{n-1} p^{(n-1)} \\
 s_n &= |s^{(n)}|^2 \\
 r^{(n)} &= M p^{(n)} \\
 r_n &= |r^{(n)}|^2 \\
 q^{(n)} &= M^+ r^{(n)} \\
 q_n &= |q^{(n)}|^2 \\
 sq_n &= \text{Re } \bar{s}^{(n)} q^{(n)}
 \end{aligned} \tag{2.8b}$$

end loop over slices,

$$\alpha_n = s_n / r_n$$

$$\beta_n = (s_n - 2\alpha_n sq_n + \alpha_n^2 q_n) / s_n \tag{2.8c}$$

end loop over iterations.

The procedure expressed in Eqns. (2.8a, b and c) is perfectly equivalent to Eqns. (2.3), as indeed was verified by implementing the two procedures as computer programs. However, since in Eqns. (2.8) the full moduli s_n , r_n , q_n and the scalar product sq_n are needed only at the end of the loop over the iterations, the manipulations on the vectors s , q , f , p and r (Eqns. (2.8b)) can be done in a "sliced" form with a single pass through M and the previous values of these vectors. Eqns. (2.8c) set α_n and β_n for the next iteration.

3. Implementation

In the previous section we have presented an alternative formulation to a conjugate gradient method. Similar strategies may be applied to different, traditional variants of the CG family of algorithms (e.g., various acceleration schemes for the basic CG method). All one has to do is manipulate the given expressions so that results of inner-products are needed only at the end of the iteration for quantities to be used at the next iteration. A "look-ahead" approach is adopted whereby one employs scaling factors computed at the previous iteration and accumulates values to be used at the next one. Our discussion so far has been quite general, in the sense that we have not assumed any special structure of the coefficients matrix M ; in fact, we have allowed for M not to be SPD. The result is a formulation which is not very interesting for small-size problems. The amount of computations remains the same (it is assumed, as is almost always the case, that most of the computational effort goes in the multiplications of the vectors by M or M^T), and the convergence rate has not changed. The advantage of our proposed formulation pertains to problems so large that the coefficients and the vectors to be manipulated cannot all be simultaneously kept in memory. When this is the case the problem has to be implemented on a computer system in a sliced fashion; for example, for a three-dimensional problem one might choose one definite coordinate, e.g. the one along the z -axis, as a slicing parameter, and each slice would contain all the x - y plane values for a fixed z value. During the computation of an iteration the slices are being rolled in and out of memory, as required. If we knew (as will be shown below) that the arithmetic can be done very fast compared to the time it takes to swap

slices in and out of memory, then an algorithm requiring only half the amount of data motion becomes very attractive. It may, as was the case in our implementation, cut the real cost of executing the program in half, since one is normally charged for the clock time of keeping real memory (even if only to wait for input/output operations to complete) and for the amount of data read in and written out.

Before discussing a specific application of this method, let us mention some computer characteristics which are relevant to tackling a large CG problem. When treating a large multi-dimensional problem, supercomputers come to mind; both due to their big memories and their ability to perform very fast arithmetic. The word "supercomputers" refers today to vector processors or other powerful computers with parallel architectures, equipped with large, fast memories. From Eqns. (2.8) one can easily deduce that the CG method is likely to be very suitable for vector or parallel processing. The element of uncertainty is due to the fact that the structure, sparsity and storage mode of the matrix M is specific to each application, but is crucial to the parallelism of the procedure. Multiplying a matrix by a vector is easily vectorizable, of course, but it is assumed that M is sparse and that a set of rules is given for the product $M \cdot (\text{vector})$. The storage mode and slicing mechanism for the non-zero elements of M will determine the vectorability of this operation. We shall return to this issue when describing a specific implementation. In the meantime let us make the following observations:

- (i) Using vector arithmetic will be efficient since one can expect long vectors, if one assumes a three-, or even four-, dimensional problem. The vectors in Eqns. (2.8) will then be of length m^2 or m^3 , where m is

the grid size, and having already accounted for slicing; i.e., there are m "sections" or slices of the length given above for each of the vectors f , p , s , r and q . Long vectors are useful for efficient processing, because in their manipulations the effect of the fixed-time vector operation start-up time becomes quite negligible.

(ii) The updating of the vectors f , s and p in Eqns. (2.8) is accomplished by using only values from the previous iteration or from other vectors. There is no dependence of elements in any vector on elements of the same vector which are computed in the current iteration; in other words, there are no recursive relationships for any individual vector in the lifetime of an iteration. In fact, the combination of multiplying by a scalar and adding to, or subtracting from, another vector can be executed at the rate of a single operation on current vector processors. The two operations may be "linked" in such a way that a result element from the first operation can be used as input operand as soon as it becomes available.

(iii) There are more floating-point operations needed for performing inner-products than for the vector updating (we do not consider the matrix multiplications yet). The four scalars s_n , r_n , q_n and sq_n in Eqns. (2.8) are the result of accumulating, slice by slice, the sum of products of two vectors. This operation, when considered as progressing in lexicographic order, is recursive by nature - a previous partial sum is needed before the next product can be added to it. It is possible, however, to accumulate partial sums and products in parallel from different segments of the vectors, and, after several stages, get the final result. Such a logic has been implemented on the CDC CYBER 205, for example, in a way which enables the inner product to be performed at full streaming rate. It may

not seem significant, but the reader should realize that when only, say, 5% of the operations perform 10 times slower than the other 95%, they will amount to about one-third of the final computation time.

The overall efficiency of the arithmetic really depends on how fast the two "matrix by vector" multiplies in Eqns. (2.8) can be performed. This will determine the CPU time for executing an iteration. Our experience has shown, however, that the true crucial factor is how fast one can restructure and move data to and from memory, and not how fast the computations are done. The arithmetic is fairly simple, and when the computations of one slice of data can be done concurrently with fetching the next slice, the data motion is more time-consuming. Input/output bandwidth seems to be the most relevant factor for the CG method on present supercomputers. The implementation effort should, therefore, be directed at minimizing the time spent in moving data in and out of fast memory. Some or all of the following may be done when implementing the algorithm:

(a) Hardware and software features supporting concurrency of input/output with computations should be used. In addition to the slices needed for updating the current slice, there should be some space allocated for swapping the slice which is not needed anymore with the one which will be needed for updating the next slice. Then, at least part of the data fetching operation will be overlapped with computations.

(b) There are two major data items which may be treated separately. One is the set of coefficients in the matrix M ; the second is the set of vectors being updated at each iteration. It is important, of course, that the slicing mechanism used for the two be consistent; e.g., that the coefficients in slice i of M will be those needed for slice i of the vectors f, p, s, r and q . If the vectors are sliced by planes or cubes, for

example, so should the non-zero elements in M be arranged.

(c) It is possible to reduce the data swapping time by utilizing more than one input-output stream (or I/O channel). The obvious thing to do is to create two files, residing on storage media associated with different channels or memory ports, - one for elements of M and one for the elements of the vectors above. In this way two I/O requests may be initialized at the same time, and the total elapsed time corresponds then to the larger of the two slices of data files. This will be, typically, the one containing the elements of M . Therefore, there is no point in further subdividing the file containing the updated vectors. Depending on the structure of M it may make sense to split it into sections within the same slice.

(d) There are five vectors being computed each iteration. However, only four of them (f, p, s and q) must be preserved for the next iteration. These should, then, be part of the file of slices. The vector r , on the other hand, needs to be computed only for the current iteration, and does not have to be stored at all. This is advantageous even if a couple of r slices may have to be computed twice in an iteration.

(e) The time spent performing I/O operations is a function of the number of bits transferred. On some computer systems one can choose different size words. Halving the size of words in the data being sliced cuts in half the time spent performing I/O. One can truncate the word size before writing out and extending it after reading in, or just work with the smaller size word. We have found, on the CDC CYBER 205, that the CG method is stable enough to permit the latter procedure. Using 64-bit or 32-bit arithmetic on this system resulted in values identical to machine accuracy and no deterioration of convergence rates. The cost of executing

the code, and its elapsed time, were halved, of course.

4. An Application

In this section we shall give some details regarding a specific problem to which the modified CG method described before has been applied. This is where the structure of the matrix M , or the rules for applying the "operator" to a given vector, have to be spelled out. The application considered is a stage in a suite of programs for evaluating physical observables in Quantum Chromodynamics (QCD) (the theory describing strong interactions among subnuclear particles), discretized on a four-dimensional lattice. Monte Carlo simulation techniques are used to create space-time configurations of the gauge dynamical variables on the lattice. These configurations amount to collections of 3×3 complex unitary-unimodular matrices (corresponding to the $SU(3)$ gauge symmetry group of QCD) each associated with an oriented link (i.e. site and direction) on the four-dimensional lattice, as described in Ref. 5. A configuration contains (in our case) $16^3 \times 32$ $SU(3)$ matrices in each of four directions, 16 being the spatial size, and 32- the size along the time axis; i.e. more than 9.4 million real variables. In this lattice we place a "source" representing a quark (the fundamental building block of hadronic particles in the QCD theory), and then compute at each lattice site the value of the Green's function (or propagator) describing the propagation of the quark. This requires the solution of the lattice equivalent of the Dirac equation with a point-like source, namely

$$\sum_{\mu} S_{\mathbf{x}}^{\mu} (\Delta^{\mu} \psi)_{\mathbf{x}} + m \psi_{\mathbf{x}} = \delta_{\mathbf{x}, \mathbf{x}_0} s_0 . \quad (4.1)$$

In Eqn. (4.1) $\mathbf{x}$ represents a generic lattice point, $\mathbf{x}_0$ the position of the source, μ one of the 4-directions, s_0 a vector describing the

orientation of the source in unitary space ($\delta_{x,x_0} s_0$ is the b vector of the CG algorithm), S_x^μ suitable spin factors, reproducing on the lattice the algebra of Dirac's γ -matrices, ψ_x the quark field propagating from x_0 (ψ_x is also a 3-dimensional complex vector, in unitary space, and all the components of ψ constitute the vector f in the notation previously used) and Δ^μ is a gauge-covariant central difference operator. Δ^μ is defined by

$$(\Delta^\mu \psi)_x = \frac{U_x^{\mu\dagger} \psi_{x+\hat{\mu}} - U_{x-\hat{\mu}}^\mu \psi_{x-\hat{\mu}}}{2a}, \quad (4.2)$$

where the U 's are the $SU(3)$ matrices mentioned above and a is the spacing of the lattice.

From Eqns. (4.1) and (4.2) we can deduce the magnitude of the computational problem. There are $16^3 \times 32$ unknown quantities $\psi(x_\mu)$, each with 3 complex values (i.e., 6 numbers), so that the number of unknowns is 786,432. Eqn. (4.2) tells us that $4 \times 16^3 \times 32 = 524,288$ $SU(3)$ matrices are contained in the coefficients matrix M . In fact, this is all that is needed, if, for the sake of economy in storage, one is willing to reorder data elements instead of duplicating them in a different order. Looking at Eqns. (4.1) and (4.2) it is clear that U_x is to be aligned with ψ_x , as well as $\psi(x+\hat{\mu})$ and $\psi(x-\hat{\mu})$, where μ ranges over the 4 possible directions. This is not simply an offset of one unit, since we are dealing with four-dimensional space with periodic boundary conditions, and is best handled by creating index-lists pointing to the nearest neighbours in each direction. These index-lists can be utilized with the CDC CYBER 205's "gather" instruction for a fast reordering of data elements.

The number of non-zero coefficients in M does not have to be quite as high as 9.4 million. QCD is a gauge theory, which means that one can

perform gauge transformations on a configuration, without changing its physical content. Since it is convenient to slice the problem by time-axis values, a natural gauge transformation will be the one that converts the configuration to the "temporal" gauge. After this transformation the $SU(3)$ matrices associated with the time direction (between time slices) become the identity matrix for all but the last time-slice (which is treated as a special case). The computations thus simplify and the number of non-trivial real coefficients in the matrix is reduced to about 0.75 of what it would have been otherwise, i.e. just a little over 7 million values.

Without writing out explicitly all the terms in the discretized operator, it is apparent from Eqns. (4.1) and (4.2) that three time-slices are needed for applying the operator on a vector. The values of slices "t-1" and "t+1" are used even though they are connected via the identity matrix (which requires no matrix multiplication). The same dependence on neighbours exists for the spatial coordinates, but all their values are contained in time-slice "t" anyway. In fact, we need in memory 4, and not 3 time-slices, at any one time. This we see from Eqn. (2.8): the computation of the vector $q^{(n)}$ requires slices t-1, t and t+1 of r; but to compute the t+1 slice of r one also needs slice t+2 of the vector p. This means that we need real memory sufficient for 5 time-slices, the space for the fifth one used to initiate swapping slice t-2 which is not needed anymore with slice t+3 to be used at the next iteration. (As a matter of fact, less than 5 time slices need to be kept in fast memory for some of the other variables. In the temporal gauge only 3 slices of M are required.)

The vectorization of the $r^{(n)} = M_p^{(n-1)}$ and $s^{(n)} = M_u^{\dagger}(n)$ equations is not achieved by vectorizing the product of the $SU(3)$ matrix by a 3-component vector, but rather by doing the component multiplications over many lattice sites simultaneously. Indeed, we manipulate all of the spatial coordinates of a given time-slice in parallel.

With the slicing as described above the code executes comfortably in less than 8 million bytes of memory (using 32-bit operands). The elapsed time for one iteration improved substantially in the conversion to the single pass algorithm. Timing data are as follows: about 50 secs elapsed time with the original two-pass procedure, with 32-bit $SU(3)$ matrices and 64-bit updated vectors; 24-25 secs when the one-pass method was adopted; 13-14 secs when the I/O channels were used concurrently for the $SU(3)$ matrices and the vectors, respectively; and, finally, 7-8 secs per iteration when 32-bit operands are used for the updated values. This last change also doubles the arithmetic rate on the CDC CYBER 205. The CPU time per iteration is still considerably smaller because the arithmetic is done very efficiently. Utilizing 32-bit arithmetic one iteration lasts 1.32 CPU-secs (1.91 secs for 64-bit arithmetic), with an execution rate of 131 Mflops (78 Mflops for 64-bit). This includes the time spent in the "gather" instruction (about 30% of the total, 17% for 64-bit precision) and corresponds to a rate of 193 Mflops for the 32-bit arithmetic alone. All these figures pertain to a two-pipe CDC CYBER 205.

There is enough idle time while waiting for the I/O to complete to test for convergence every iteration (if needed) without adding any elapsed time to the execution. The test amounts to applying the operator a third time to the "solution" vector and comparing it to the right-hand-side. We then measure the L_2 , L_1 and L_∞ norms of the residual vector. It turns

out that the convergence rate is, as expected, very sensitive to the size of the main-diagonal elements determined by the parameter m in Eqn. (4.1). The bigger m is the faster the convergence is. It changes from an order of magnitude reduction in error size every 3 iterations when $m=1$, to similar decrease every 100 iterations when $m = 0.025$. The evaluation of the physical observables requires that, for a given configuration, the code be executed with the point-like source positioned at various sites in the lattice with a fixed value for the parameter m . Then the process is to be repeated for different smaller values of m . When this is all done a new configuration is formed (using Monte Carlo simulations) and the whole procedure is repeated. A number of configurations are to be examined this way until the statistical error of the observables becomes sufficiently small. The analysis of the solution vector for the observables is not part of the CG solver described here. The necessity of using the solver so many times is what prompted us to search for an efficient algorithm for very large systems.

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 Dr. B.M. Alder and the Directorate of the Lawrence Livermore Laboratory for the award of a Visiting Fellowship which made his visit to Livermore possible.

References

- [1] M.R. Hestenes and E.L. Stiefel, Nat. Bur. Std. J. Res. 49, 409(1952).
- [2] L.A. Hageman and D.M. Young, Applied Iterative Methods (Academic Press, New York, 1981), p. 138.
- [3] T. Ginsburg, in Handbook for Automatic Computation, Vol. II, ed. J.H. Wilkinson and C. Reinsch (Springer-Verlag, Berlin, 1971), p. 57.
- [4] J. Stoer and R. Bulirsch, Introduction to Numerical Analysis (Springer-Verlag, Berlin, 1980), p. 572.
- [5] D. Barkai, K.J.M. Moriarty and C. Rebbi, Comp. Phys. Comm. 32, 1(1984).