Computer Physics Communications 30 (1983) 13–19
North-Holland Publishing Company

WILSON LOOP CALCULATIONS IN FOUR-DIMENSIONAL LATTICE GAUGE THEORY ON THE CDC CYBER 205 *

D. BARKAI

Center for Applied Vector Technology, Control Data Corporation at the Institute for Computational Studies, Colorado State University, Fort Collins, CO 80523, USA

M. CREUTZ

Department of Physics, Brookhaven National Laboratory, Upton, NY 11973, USA

and

K.J.M. MORIARTY **

Department of Physics, University of Bielefeld, D-4800 Bielefeld 1, Fed. Rep. Germany

Received 26 February 1983; in revised form 30 March 1983

Pure SU(4) gauge theory is simulated by Monte Carlo methods on an 8^4 lattice. The method of Metropolis et al. is used to equilibriate the space–time lattice. All Wilson loops up to size 4×4 are calculated. Because of memory requirements we work on the 2 Mword CDC CYBER 205 at Colorado State University and take full advantage of the parallel processing capabilities of this vector machine.

1. Introduction

It is generally believed that the theory of strong interactions is Quantum Chromodynamics (QCD) which is a quantum field theory. However, quantum field theory is plagued by infinities which require a regularization technique. By expressing quantum field theory in the Feynman path integral approach, the equivalence of quantum field theory and statistical mechanics can be established. Thus, formulating quantum field theory on a discrete space–time lattice [1] which acts as a regularizer, and taking all the established techniques of statistical mechanics, allows us to solve QCD and obtain physically meaningful results [2]. However, in order to eliminate finite size effects and measure interesting physical observables [3] we need to work on large lattices which requires large computer memory for simulations. In order to obtain reasonable statistics this computer memory must be fast memory. The best source of large fast core is the current supercomputer, e.g., the CDC CYBER 205. The CPU times required for some recent Monte Carlo lattice gauge theory calculations are given in table 1.

The Monte Carlo lattice gauge theory technique for SU(N) gauge theory on a scalar machine (the CDC 7600) was described in great detail recently [3]. The vectorization of this algorithm for the average plaquette calculation on the CDC CYBER 205 has been described for an odd lattice [12] and for an even lattice [13]. However, the recent calculations of the $U(4)$ and SU(4) string tension on a

* Talk presented at the Three Day In-Depth Review on the Impact of Specialized Processors in Elementary Particle Physics, held at Padova, 23–25 March 1983.

** Permanent address: Department of Mathematics, Royal Holloway College, Englefield Green, Surrey TW20 0EX, UK

0010-4655/83/0000–0000/$03.00

Table 1
Some typical recent lattice gauge theory calculations carried out on the supercomputers CDC CYBER 205 and CRAY-1S

Calculation	Computer	CPU time time (h)
$U(N)$, $N = 2, 3, 4, 5$ and 6 [4]	CRAY-1S	70
$SU(N)/Z_N$, $N = 2, 3, 4, 5$ and 6 [5]	CRAY-1S	75
$U(2)$ string tension [6]	CRAY-1S	80
$U(3)$ string tension [7]	CRAY-1S	88
$U(4)$ and SU(4) string tension [8]	CDC CYBER 205 and CRAY-1S	6 153
SU(3) string tension on 6^4 lattice [9]	CRAY-1S	79
SU(3) renormalization study on 8^4 lattice [10]	CRAY-1S	192
SU(4) renormalization study on 8^4 lattice [11]	CDC CYBER 205	9.45

6^4 lattice [8] and the SU(4) renormalization study on an 8^4 lattice [11] require the calculation of Wilson loops. Thus, in the present paper we wish to describe the vectorization of the Wilson loop calculation.

2. Field theory and Monte Carlo methods

In lattice gauge theory, the Feynman path integral is mathematically equivalent to the partition function for a discrete set of statistical variables located on the lattice bonds. The field theoretical bare coupling constant corresponds directly to the statistical temperature. A high temperature expansion then provides information on large couplings. In this way Wilson [1] derived confinement in the strong coupling domain, where the theory reduces to a model of quarks on the ends of strings of gluonic flux. These strings have a finite energy per unit length and thus the quarks experience a linear confinement potential.

Although the Monte Carlo approach to simulating statistical systems [14] is quite old, only recently have particle physicists applied the technique to gauge theories. One stores the numerical values for the variables in the computer memory. Pseudo-random changes, weighted by the Boltzmann factor, then mimic thermal evolution and fluctuation. Thus we do "experiments" on a system with a predetermined dynamics. Such studies have given the strongest evidence that the phenomenon of quark confinement persists in the continuum limit of a non-Abelian gauge theory.

The Monte Carlo method has a few inherent limitations. Statistical errors only drop with the quare root of the computer time. Consequently, the extraction of some parameters can be severely statistics limited. Also, for the four-dimensional systems of interest to the particle physicist, the linear dimensions of the lattice are necessarily limited. Finally, although fermionic fields are a major area of current research, the techniques for computing with Grassmann variables are as yet quite awkward and voracious of computer time.

The detailed techniques for simulating statistical systems are standard and need not be explicitly discussed here. One difference between gauge simulations and more traditional Monte Carlo applications is the large amount of arithmetic that must be done to calculate the interaction of a single link. This means that it is quite important to do as careful a job as possible to calculate the trial changes on a variable. In particular, it is usually economical to try several changes on a single link before going on to the next.

3. Vectorization and performance issues

The programming considerations and techniques, needed for vectorization for part of the application were described in some detail in previous publications [12,13]. In this paper the main issues and conclusions discussed there will be revisited, but more attention will be paid to the vectorization process of an additional routine, namely, the Wilson loops routine. In addition, some figures will be given regarding the relative importance of various computational procedures in the code, the effect of vectorization and the dependence of performance upon the parameters determining the size of the problem.

For the sake of clarity let us establish some common terms for describing the problem. The physics is derived by considering a four-dimen-

sional lattice with L link values along each axis. (L values treated here will be 8 and 6.) With each link value a complex matrix of $N \times N$ is associated which corresponds to the SU(N) symmetry group ($N = 4$ throughout this section). Thus, the numerical values of interest are contained in a $4L^4 \times 2N^2$ size array. It will be referred to as the "links array". Two main routines will be analyzed: MONTE, which employs the Metropolis et al. [15] variant of the Monte Carlo technique for evaluating the average action per plaquette, and RECTS which computes the Wilson loops. MONTE is the part of the code described in refs. [12,13].

Before proceeding let us reiterate the method and tools for overcoming a couple of obstacles in the way of effectively vectorizing a Monte Carlo application for lattice gauge theories. The first obstacle is a conflict between random access to data and the requirement of the CDC CYBER 205 that vectors are sets made of elements stored in consecutive locations. The random access of data originates from the Metropolis [15] selection process which assures the convergence of the Monte Carlo process. For SU(N), $N > 2$, a look-up table of unitary-unimodular matrices is generated, using a random-number generator, for each MONTE iteration. The selection procedure then retrieves values at random from this table. For parallel processing it is required that many such values be retrieved before the pipe-lined arithmetic can be performed. If this retrieval stage had to be executed in a scalar (serial) manner then, no matter how fast the arithmetic operations are, a considerable amount of time would be spent not utilizing the vector pipelines, and thus degrading the computer's performance accordingly.

Another reason for the need to "collect" data stored at irregular intervals into a contiguous area is that to update any link value, its neighbouring link values, in every direction, must be available. So that if a number of links are to be processed in parallel, their neighbours must constitute vectors too. The usage of periodic boundary conditions in a hypercube makes it impossible to pre-order the "links-array" in a way which assures the presence of vectors in every direction. This constraint regarding the neighbours' dependence applies both for MONTE and RECTS.

The conclusion of the above arguments is that unless one can "gather" (and "scatter" back into place) data elements at rates comparable to computation rates, an "effective" vectorization cannot be achieved. The CDC CYBER 205 is a system where such random (or, indirect) "load" or "store", or the ordering of a random collection can be achieved through a pair of vector instructions, commonly known as GATHER and SCATTER. They have two input streams – one is a data array, the second is an index list. The GATHER instruction steps through the index list, picks up the data element pointed to by the current index and puts it into the next location of the output stream. For example, consider the following:

data array: 20, 19, 18,..., 2, 1, 0,
index-list: 3, 1, 10, 10, 9, 5, 6, 6, 6.

Then, executing the GATHER instruction will produce the array:

18, 20, 11, 11, 12, 16, 15, 15, 15.

The SCATTER instruction uses the index list to point to the location in the output array where the next data element is to go.

The results rate for these two instructions is one result-element every 1.25 clock periods, that is every 25 ns since the CDC CYBER 205 has a 20 ns clock. This is to be compared with a result-rate of one element per clock-period per pipe for a vector add or multiply, or, one result every 10 ns for a 2-pipe CDC CYBER 205.

Other aspects of vectorization will be discussed below; suffice it to say here that the code is fully vectorized. The significance of the GATHER and SCATTER instructions may be appreciated by observing that, for the vectorized code, these instructions amount to 12.5% of the time spent in MONTE and 27% of RECTS processing time. One can easily appreciate the disastrous effect on the execution time if these operations performed an order of magnitude slower, as they would in scalar, serial mode. Incidentally, as will be demonstrated below, the larger L (the size of the lattice) is, the bigger is the time spent in RECTS comparred to that of MONTE.

Another hurdle in the path of vectorization may be summed up as follows: The convergence of the

Monte Carlo iterations depends upon the usage of updated link values as soon as they are available. Vectorization means taking a set of consecutive links and updating them in parallel. However, since computing each link value requires its neighbour values, new ones, if available, the convergence of the vectorized code cannot be achieved as long as the links are processed in the usual lexicographic order. One needs to identify groups of links which are independent of each other and, therefore, may be computed in parallel without affecting the convergence rate or correctness. The solution for this is to discard the lexicographic order of processing and use what is known as "red–black" or "checker-board" ordering [13] where each "colour" can be processed in parallel, resulting in vector lengths of $L^4/2$. Due to the periodic boundary conditions this two-colour scheme works for even lattice size (L) only, for odd lattice size a multi-colour scheme is required as described in refs. [12,13]. Since we consider L values of 6 and 8 this will not concern us now. Here, again, as was alluded to above, the GATHER instruction plays a crucial role in retrieving the group of links and the appropriate groups of "neighbours" (of the other "colour") for pipe-lined (parallel) processing.

It may be useful, at this point, to pause and examine the importance of being able to use "long" vectors. On a vector processor a vector operation amounts to issuing a single instruction, such as add or multiply, which will return a whole array (vector) of results. The timing formula for completing such an instruction contains two components. One is fixed, i.e. independent of the number of elements to be computed, and is called "start-up" time. In fact, it amounts to start-up and shut-down; it involves fetching the pointers to the input and output streams, aligning the arrays so as to eliminate bank conflicts (the vector instructions on the CDC CYBER 205 operate from memory to memory without the use of registers), and the time to get the first pair of operands to the functional unit (the pipe-line) and the last one back to memory. Typical time for the "start-up" component is 1 μs, or about 50 cycles (clock periods). The other component of the timing formula is the "stream-time" which is proportional to the number of elements in the vector. As mentioned earlier the result rate for a 2-pipe CDC CYBER 205 for an add or multiply is 2 results per cycle. It is apparent now that in order to offset the "wasted" cycles of start-up times it is beneficial to work with longer vectors. The system is better utilised if a single operation is performed on a long vector, rather than several operations to compute the same number of results. Given a vector length, M, one can evaluate the efficiency of the computation as the ratio between the number of cycles used to produce useful results and the total number of cycles the instruction has taken, i.e. $(M/2)/((M/2)+50)$. The maximum vector length the CDC CYBER 205 hardware allows is 65 535 elements. The start-up time becomes quite negligible long before that.

The discussion above may help the reader, who is familiar with the application, to guess the vectorization strategy which was adopted for the arithmetic portion of the code. For $L = 8$, 76% of the time spent in MONTE is used for computing complex matrix multiplies, the matrices being the 4×4 complex SU(4) matrices associated with each link. In RECTS 72% of the time is dedicated to complex matrix multiplies and summing up the results of such products. These figures apply to the vectorized code on the CDC CYBER 205, but for any computer it is obviously essential to perform these computations fast. It is well known how to vectorize a product of two matrices, but, of course, in our case we will be dealing with vectors of length 4 (coming from the 4 in SU(4)). Knowing the "cost" of vector instruction start-up, one does not really want to be restricted to vectors that short. We know, however, that each of the $4L^4$ ($= 16\,384$ for $L = 8$) link values has such a 4×4 complex matrix associated with it. We also know how to separate the "links-array" into 8 independent sets of $L^4/2$ links each, using the "red–black" ordering. So here is the answer: perform the matrix-product as in a serial processor, only do each of the operations required for $L^4/2$ such matrices in parallel. This is the vector length used in MONTE; 2048 elements for $L = 8$, 648 for $L = 6$.

Another method for performing this complex matrix-product was described in refs. [12,13]. It was termed the "long outer product". It is suitable for small, odd values of L, e.g. $L = 3$, and, at the

cost of some extra copy operations, vector lengths of L^3N^2 (N being that of SU(N)) are achieved.

The considerations and solutions described so far make the explanation for the vectorization of RECTS fairly straightforward. In this routine Wilson loops are evaluated by summing up complex matrix-product (SU(N) matrices) values of all subdivisions of the lattice up to the size of $L/2$ along each axis. Since link values are only summed up but not updated, all the values in the "links array" can, in principle, be processed in parallel. Moreover, it turns out that the sequence of moving along the boundaries of the Wilson loops needs to be preserved. Therefore, as with the matrix-product in MONTE, the strategy here was to compute the contribution from as many links as possible to the appropriate loops. In FORTRAN terminology this amounts to "loop-inversion", i.e. take the external loops and make them the innermost. The subroutine RECTS requires many temporary quantities to be computed, which, while vectorizing, turn into temporary arrays of a size equal to the vector length. Thus, in order not to indulge in excessive memory usage, and still have long enough vectors, L^3 links are processed in parallel in this routine (i.e. for $L = 8$ vector length of 512 is used in RECTS). After the contribution of L^3 links is processed by vector instructions it needs to be summed up. Summing up values contained in an array seems to be a serial-recursive process. However, on the CDC CYBER 205 there is a vector macro instruction, the SUM instruction, which performs the summation at the rate at which arithmetic is done (and, by the way, there is an instruction which performs dot-product at the same rate).

The GATHER instruction is put to good use in RECTS in collecting together all the "neighbour" values needed for summing up the subdivisions, since, as was mentioned earlier, even when one steps through the "links array" in order, the periodic boundary conditions and the hypercube structure cause irregular intervals between successive values which are to constitute a vector.

Another performance improvement for the vectorized code over that of the original CDC 7600 code was achieved through the use of the large memory of the CDC CYBER 205. Various indices used for indirect-addressing to the "neighbours" were computed each iteration, both for MONTE and RECTS. This is done now only once, and the "index-lists" computed are saved (on a disk file in the case of RECTS) and used for the GATHER instructions on each iteration. The evaluation of these "index-lists" has to be done using scalar instructions, but since it is done only once the processing time needed amounts to well below 0.5% of the total time even for as little as 10 MONTE or RECTS iterations.

Throughout this section frequent references were made to "complex" arithmetic. The reader who is familiar with the FORTRAN storage conventions may realize by now that this storage by pairs of real and imaginary parts is not advantageous for vectorization on the CDC CYBER 205. Indeed, the first modification to the original code was to arrange all the complex arrays so that they contain all the real values followed by all the imaginary values (or split complex arrays into two arrays). The FORTRAN complex arithmetic statements were coded explicitly as real arithmetic involving the two parts of the original complex array. As an aside, this change actually benefits most serial processors from the point of view of improving the resultant object code produced by the compiler.

At this stage, after having covered the main issues involved in the vectorization process, we can address some overall performance and timings details. Some other minor details will be mentioned as we go along. First, let us be reminded of the general flow when executing the application. The main program calls subroutine MONTE to perform a number of iterations so as to achieve convergence for a given inverse-temperature value. When this is done the subroutine RECTS is called to executive another number of iterations. The calculations performed in RECTS measure Wilson loops. For each RECTS iteration MONTE is called to perform two iterations and the loop values are accumulated and standard deviations calculated. There is another routine, which has not been mentioned so far, for renormalizing the SU(N) matrices to assure that they stay unitary-unimodular. This routine is called once every 20 to 50 MONTE iterations and therefore never amounts to more than 1% of the total run time even when it executes by scalar instructions, as it does now (it is

a vectorizable process, though).

The number of iterations required for meaningful results depends on several parameters: the symmetry group SU(N), the inverse-temperature and how near it is to a phase transition value, and the lattice size. Let us consider SU(4); then, for lattice size L equal to 6 we would typically need 200 MONTE iterations and 25 RECTS iterations when not near a phase transition (when near a phase-transition both numbers should be about doubled). With this in mind it makes sense now to talk about execution times per iteration for MONTE and RECTS. For the purpose of comparison to other similar codes two more parameters should be highlighted. The first one pertains to MONTE; for the Metropolis selection process the random-number look-up table is sampled N^2 times for each link value and each iteration for the SU(N) symmetry group. This selection process, which speeds up the convergence, amounts, for $L = 8$, to 66% of the total time in MONTE (this section of the code contains some GATHER's and matrix products). The second parameter which may change between variants of the code is the maximum size of a subdivision of the lattice in RECTS. We sum all subdivisions up to the size of $L/2$, where L is the lattice size. With this understanding we can now consider table 2, which gives the execution times per iteration for MONTE and RECTS, for SU(4) and $L = 6$ and 8. The reader may notice the effect of increased vector length when considering the timings for MONTE, where the amount of work increases like the fourth power of L. Thus, while the amount of work increased by a factor of 3.16, by changing from $L = 6$ to 8, the execution time increased only by a factor of 2.88.

Table 2
Execution times in seconds per iteration for MONTE and RECTS for lattice sizes of 6 and 8

Routine	Lattice size		
	6^4	8^4	Ratio: $\frac{8^4\text{-time}}{6^4\text{-time}}$
MONTE	1.70	4.9	2.88
RECTS	4.27	21.2	4.96
Ratio: $\frac{\text{RECTS time}}{\text{MONTE time}}$	2.51	4.33	

One also observes the fast growth of the contribution to the execution-time arising from RECTS (where the time quoted does not include the two MONTE iterations associated with each RECTS iteration).

For the purpose of comparing these timings to other computer systems, the following may be adequate: for $L = 6$ MONTE performs 12 times faster on the CDC CYBER 205 than it does on the CDC 7600. For RECTS the ratio between these two systems is 28 in favour of the CDC CYBER 205. The same ratios for $L = 8$ are bigger, however, the size of the problem makes it impracticable to attempt such runs on the CDC 7600. The dependence of the execution time on the symmetry-group parameter, N, was measured previously, and was found to be approximately proportional to $N^{2.5}$.

There are two more operations in MONTE which have not been explicitly mentioned yet. The Metropolis selection process involves the generation of many random numbers used for picking-up values from the look-up table; the selection criterion itself involves computing an exponential. Being part of the section which takes two thirds of the time spent in MONTE it is obviously crucial to perform these operations efficiently. Luckily, the CDC CYBER 200 FORTRAN provides us with appropriate tools for achieving just that. A library subroutine exists which returns a whole array of random numbers with one call. The FORTRAN library also contains "vector functions" for various mathematical and trigonometric functions. One of these is a vector exponential function. As a result, the random-number generation takes only 3.8% and the exponential computation takes only 2.4% of the time spent in MONTE.

In conclusion, it will be stated that all the computations referred to in this paper were performed using 64-bit arithmetic. The CDC CYBER 205 hardware provides for working with 32-bit word size. When this feature is employed scalar operations are performed at the same speed as in 64-bit mode, as do the GATHER and SCATTER instructions (due to their dependence on memory access); but, vector floating point operations have a result-rate which is twice as fast as 64-bit operations and the same start-up time. The advantage of

migrating to the 32-bit mode is not just for achieving better performance, it also means halving the memory required for the application, and therefore being able to tackle larger size problems quite easily. Experience has shown that adequate accuracy may be maintained in 32-bit mode when the SU(N) matrices are renormalized more frequently. We are planning to migrate our existing code to the 32-bit mode in the near future.

Acknowledgements

We would like to thank the Control Data Corporation for the award of a research grant (Grant 82CSU13) to buy time on the CDC CYBER 205 at the Institute for Computational Studies at Colorado State University. The calculations of the present paper took 9.45 h of CDC CYBER 205 time to perform. One of the authors (K.J.M.M.) would like to thank the University of Bielefeld for the award of a Visiting Fellowship which made his visit to Bielefeld possible. This research was also carried out in part under the auspices of the US Department of Energy under contract No. DE-AC02-76CH00016.

References

[1] K.G. Wilson, Phys. Rev. D10 (1974) 2455.
[2] See, e.g. M. Creutz, Quarks, Gluons and Lattices (Cambridge Univ. Press, Cambridge, England, 1983), for an introduction to this subject.
[3] R.W.B. Ardill, M. Creutz and K.J.M. Moriarty, Comput. Phys. Commun. 29 (1983) 97.
[4] M. Creutz and K.J.M. Moriarty, Phys. Rev. D25 (1982) 610.
[5] M. Creutz and K.J.M. Moriarty, Nucl. Phys. B210 (FS6) (1982) 50.
[6] M. Creutz and K.J.M. Moriarty, Nucl. Phys. B210 (FS6) (1982) 59.
[7] M. Creutz and K.J.M. Moriarty, Nucl. Phys. B210 (FS6) (1982) 377.
[8] D. Barkai, M. Creutz and K.J.M. Moriarty, Nucl. Phys. B (FS) (1983) in press.
[9] M. Creutz and K.J.M. Moriarty, Phys. Rev. D26 (1982) 2166.
[10] R.W.B. Ardill, M. Creutz and K.J.M. Moriarty, Phys. Rev. D27 (1983) 1956.
[11] D. Barkai, M. Creutz and K.J.M. Moriarty, Phys. Rev. D submitted.
[12] D. Barkai and K.J.M. Moriarty, Comput. Phys. Commun. 25 (1982) 57, 26 (1982) 477.
[13] D. Barkai and K.J.M. Moriarty, Comput. Phys. Commun. 27 (1982) 105.
[14] Monte Carlo Methods in Statistical Physics, ed. K. Binder (Springer-Verlag, Berlin, 1979).
[15] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller and E. Teller, J. Chem. Phys. 21 (1953) 1087.