Nuclear Physics B220[FS8] (1983) 383–400

GLUEBALL MASS CALCULATIONS ON AN ARRAY OF COMPUTERS*

Eugene BROOKS III, Geoffrey FOX, Steve OTTO and Mohit RANDERIA

Caltech High Energy Physics Group, Pasadena, California, USA

Bill ATHAS, Erik DeBENEDICTIS, Mike NEWTON and Charles SEITZ

Caltech Computer Science, California Institute of Technology, Pasadena, California, 91125, USA

Received 21 March 1983

We have calculated the mass of the 0^+ glueball in SU(2) pure gauge theory in 4 dimensions, with very high statistics. The computation was done on an array of microprocessors with nearest-neighbor connections which run concurrently. We discuss, in detail, the implementation of the pure gauge algorithm for SU(2) and SU(3) and also the algorithm for calculating arbitrarily shaped Wilson loops on the array. The extension of these algorithms to the inclusion of dynamical fermions is also discussed. Finally, we present the results of our variational calculation of glueball masses which are in agreement with published results.

1. Introduction

A numerical approach to the solution of quantum field theories via Monte Carlo simulations of the lattice versions of these theories has recently proven to be quite fruitful [1]. It has become increasingly clear, however, that computers with orders of magnitude more power are needed in order for this field to progress much further. The reasons for this include the statistical nature of the convergence of the numerical estimates, the need to calculate exponentially small quantities which are rapidly lost in this statistical noise (e.g. glueball mass calculations), and the need to go to much larger lattices to remove spurious finite-size effects [2].

Fundamental limits on computational speed and feature size in VLSI technologies suggest that significant increases in performance will come not from pushing current designs yet further, but instead from new computer architectures utilizing many

* Work supported in part by the US Department of Energy under contract no. DE-AC03-81-ER40050.

computers in parallel: concurrent processing [3,4]. A simple design for such a computer is a "homogeneous machine": a regular array of (independent) processors with a small number of interconnections per processor [5]. Such a machine is actually being built at Caltech and consists of a $4 \times 4 \times 4$ array of microprocessors wired as a 3-dimensional (periodic) cube and has a total CPU power of 10 VAX 11/780's. We have found that the homogeneous machine design is ideal for use on many computationally intensive problems in the physical sciences such as partial differential equations, matrix inversion, and fast fourier transforms [6].

In this paper we concentrate on the applications of this type of machine to lattice gauge theories. In sect. 2 we describe the hardware and programming environment of the machine. In sect. 3, we discuss how the local nature of the action leads to a straightforward implementation of pure gauge Monte Carlos on the homogeneous machine. The details of this algorithm are given in sect. 4. In sect. 5, we present various figures of merit for the SU(2) and SU(3) algorithms, such as the interprocessor communication overhead and the speed through phase space. In sect. 6, after a brief discussion of the variational method to calculate glueball masses, we discuss at length an algorithm to calculate Wilson loops of arbitrary shape on the HM, and we present our results for the 0^+ SU(2) glueball mass. Sect. 7 discusses an alternative algorithm for the pure gauge update and suggests how it may be useful for the incorporation of dynamical fermions. Summary and conclusions appear in sect. 8.

2. Hardware and programming environment

This application was programmed on the four-element prototype array which was built to test the hardware and software for the 64-element microprocessor array currently being constructed. The four-element prototype is configured as shown in fig. 2. The processors labeled (0–3) are the nodes of the array which have bidirectional communication paths shown by solid lines. Each of these processors is based on the Intel 8086/8087 microprocessor. This microcomputer* has about $\frac{1}{8}$th the power of the Host VAX 11/780 in typical scientific computing. Each microprocessor board has 128k bytes of memory, 6 bidirectional data channels which allow communication with other processors in the array and a simple monitor in ROM which allows down-loading of programs through any of the communication channels.

Each bidirectional channel allows the transmission of fixed length (64 bit) packets and is implemented with two independent data paths, one for each direction. Each data path is implemented with two 64-bit FIFO buffers, one on each micro-processor board, and the necessary wiring between them to provide for asynchronous transmission of a packet from one FIFO to the other. The hardware bandwidth of this

* For an 8086-8087 processor running at 5 MHz.

channel is about 2 megabits/sec. The actual transmission time for a 4-word message, the time between the call to send a packet and the return of a pending read on the receiving end, is about 150 micro-seconds. This yields an effective (including software overhead) bandwidth of about a half a megabit/sec.

The machine labeled IH is the intermediate host. It is also an Intel 8086/8087 based processor which functions as the computer which controls the array. This could have been done with the host VAX directly but a separate microprocessor was used for two reasons.

(i) The array would typically require instructions much more often than our VAX11/780 could provide them. The host VAX is fully loaded providing services to an average load of 20–40 users.

(ii) The floating point formats for the Intel 8087/8087 and the VAX11/780 differ leading to conversion problems on the communication line between them. This conversion overhead would further reduce throughput if the VAX was used as the controlling computer.

The two problems mentioned above were neatly side stepped by having an Intel based microcomputer serve as an "intermediate host".

The application and system programming for the homogeneous machine is all performed on a VAX with a cross compiler producing 8086 code which is then loaded into the array. Currently all our programs are written in the high-level language C although a PASCAL environment is in the process of being developed. The programming model for our machine is that of independent processes communicating via messages. This can be formulated in a rather general and powerful fashion; the language CSP [7] and more sophisticatedly the "actor" formalism of Hewitt [8] and the SIMULA based work of Lang [9] are possible implementations of this general idea. The reader is referred to sect. 4 where the usage of our current operating system is described for the lattice gauge theory problem. We note that for this class of application (which we will call crystalline) we only need to implement some of the simplest features of the general systems. Currently, only one process per processor is allowed and messages are passed only between processors which have a direct hard wired link. A more general system is being planned but the current implementation [10, 6] should handle differential equations, short- and long-range particle dynamics and fast fourier transform applications. As described in [6], these cover a wide range of important scientific problems.

We emphasize that we have not found the programming either hard or inefficient. The suspicion of many that concurrent processors are impossible to program does not seem well founded.

3. Pure gauge theories on the homogeneous machine

The locality of the actions used in Monte Carlo simulations of gauge theories implies that these computations can be concurrently processed in a straightforward

way. In the standard Metropolis procedure, for instance, the change in the action due to the move of a link variable, ΔS, involves only nearby link variables coupled to the one in question through plaquettes. This means that sets of decoupled links (links which nowhere appear in the same plaquette) can be simultaneously updated via the standard procedure, and the gauge field configurations will be generated according to the correct distribution, i.e.

$$P(\{U_i\}) = e^{-\beta S(U)}.$$

This is because for each update, the ΔS's computed will not feel the effects of the other links being updated at the same time (they are decoupled) and so they will be exactly the same as for the usual sequential update algorithm. The simultaneous update of N decoupled links is exactly equivalent to the sequential update, in arbitrary order, of these same N links.

The maximum number of decoupled links which can exist on the lattice at any given time is $\frac{1}{4}N_{\text{tot}}$, where N_{tot} is the total number of links of the lattice. The

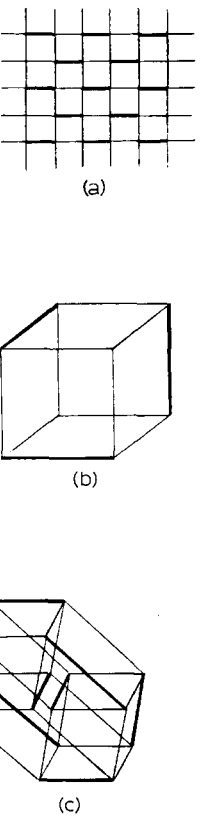

Fig. 1. Maximal set of decoupled links in (a) 2 dimensions, (b) 3 dimensions and (c) 4 dimensions.

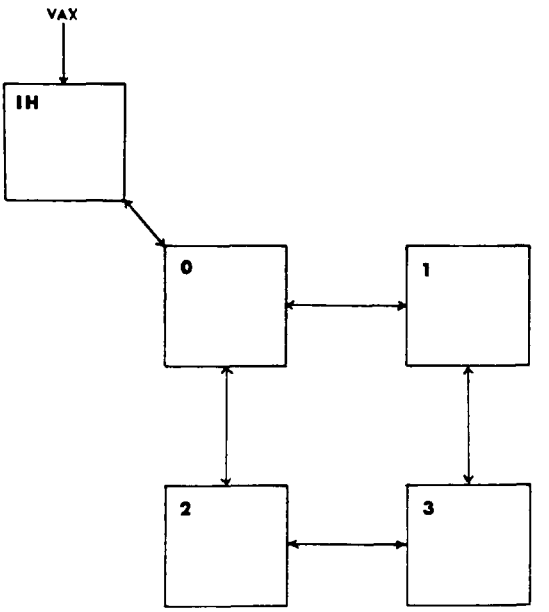

Fig. 2. Schematic diagram of the homogeneous machine.

structure of this maximal set of decoupled links is illustrated in fig. 1, for 2, 3 and 4 dimensions.

Beyond the simultaneous update of all the elements of a set of decoupled links, one could simultaneously update more than one link in a given plaquette. One must be careful, however, to use a procedure which generates the correct multi-link probability distribution, and not merely the product of the simple one-link distributions. As an example, suppose we wanted to simultaneously update 2 links, both of which are in the same plaquette. Using the Metropolis method, one could proceed as follows. Move each of the link matrices: $U_1 \to U_1 + \delta U_1$, $U_2 \to U_2 + \delta U_2$. Then compute the total change in the action,

$$\Delta S = S(U_1 + \delta U_1, U_2 + \delta U_2) - S(U_1, U_2)\,.$$

Now, if ΔS is < 0, or if $e^{-\beta \Delta S} > r$ with r a random number $\in [0, 1]$, accept *both* moves, δU_1 and δU_2. Otherwise, reject *both* moves, and return the matrices to U_1 and U_2. The above method satisfies the constraint of detailed balance and can of course be generalized to more than 2-link matrices. Currently, we are not running in this multi-link update mode: we are simultaneously updating sets of decoupled links.

4. Implementation on a 4-node machine

We will now discuss some of the details of 4-dimensional pure gauge algorithms on a 4-node concurrent processor. This machine illustrates all of the essential

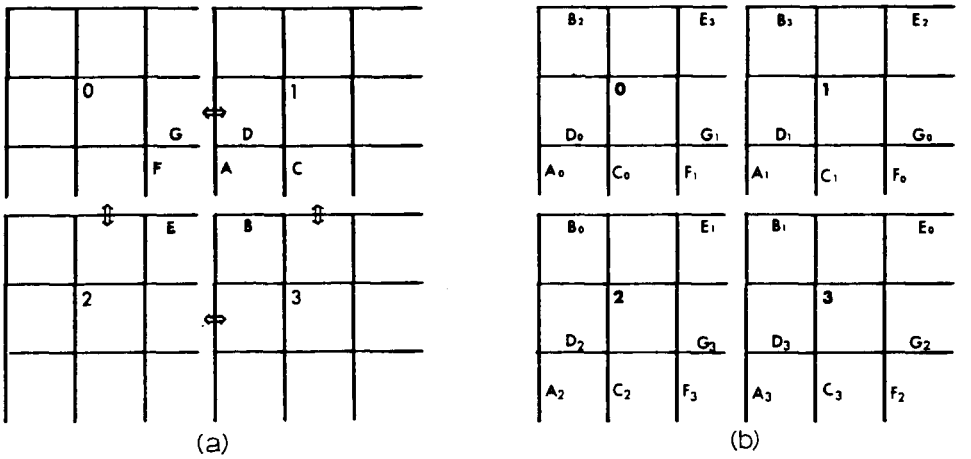

Fig. 3. Example of pure gauge update to illustrate inter-processor communication.

features of larger machines. In particular, the algorithms developed for this machine will run on the 64-node machine with a few minor modifications. The lattice is divided up among the computers so that neighboring variables of the lattice are either in the same node or in neighboring nodes. For the 2×2 square, the 4-dimensional lattice is divided up in 2 of its dimensions, the other 2 dimensions are "squashed" into the processors: if the total lattice is $4 \times 4 \times 4 \times 4$, each node stores a $2 \times 2 \times 4 \times 4$ subcell of the lattice.

To illustrate how the algorithm for gauge theories works on a homogeneous machine we will outline the steps required to update a link residing in one of the subcells of the lattice. To be definite, suppose we are updating the link labeled A in fig. 3a. This link is chosen to be on the surface of the subcell in order to illustrate the inter-processor communication required. For the sake of simplification only a two-dimensional example is discussed. To update link A, the matrices $BC^{-1}D^{-1}$ and $E^{-1}F^{-1}G$ must be constructed and passed to processor 1. As will become clear, the algorithm is written in such a way so as to keep the processors synchronized: that is, the situation is actually as shown in fig. 3b. At the same time that processor 1 is updating link A_1, processor 0 is updating link A_0, and so on. The corresponding matrices, $B_iC_i^{-1}D_i^{-1}$ and $E_i^{-1}F_i^{-1}G_i$, must be constructed and passed to processor i. The first step in the algorithm is for the B matrices to be exchanged between processors 0 and 2, 1 and 3. All such communications are done via a polled-mailbox scheme. Taking the 1, 3 exchange as an example, processor 3 sends the matrix B_1 to the "mailbox" (an internal buffer) of processor 1 across the bidirectional channel "vchan" (vertical channel). After putting the matrix in the mailbox, the "flag" of the mailbox is set, indicating to processor 1 that the mailbox is ready to be read. Processor 1 likewise sends matrix B_3 to the mailbox of processor 3 and sets the flag.

Each processor then polls its mailbox—checking the flag to see if there is something there to be read. If the flag indicates the mailbox is full, the processor empties the buffer, resets the flag to the "empty" position and proceeds to its next instruction. If the flag indicates that the mailbox is still empty, the processor effectively halts: it polls the flag indefinitely until the mailbox is filled. This is how the processors are kept synchronized; if processor 3 is lagging behind 1, when 1 gets to the instruction to read its mailbox, it will stop at that point in its instruction sequence until 3 writes to it.

Once this B exchange is finished, the matrices B_i reside in the same processors as C_i and D_i, so the products $B_i C_i^{-1} D_i^{-1}$ are formed and are ready to be used for the update. The products $E_i^{-1} F_i^{-1} G_i$ are a bit more difficult since the matrices E_i are in the next-nearest-neighbor processor to the one containing A_i. In order to minimize communication time, we adopt the strategy of passing E_1 to processor 0, forming the product $E_1^{-1} F_1^{-1} D_1$ in processor 0, and passing only the resultant product to processor 1. The same is, of course, simultaneously done for E_0, E_2 and E_3. The matrices are again passed via the polled-mailbox scheme as described above for the matrices B_i. Once all this is accomplished, the processors all update their A_i and then proceed to the next link, staying in step due to the nature of the communications software.

The above may seem complicated to implement but in reality it is not. Once the fundamental matrix exchange subroutines are written, all that is required beyond the usual Metropolis update algorithm is a few logical "if" statements which are needed to detect if the matrices B_i, etc. need to be communicated from a neighboring processor. In fact, the entire coding and debugging for SU(2) in 4 dimensions required only about 20 hours of (human) time: a modest investment for the computer power which is gained.

5. SU(2) on a 4-node machine

We have implemented pure gauge SU(2) on the 4-node machine, using the 120-element icosahedral subgroup method to speed up the computation [11]. Since we work with a finite subgroup, only integers labeling the members of the subgroup need to be communicated between the processors; actual matrices are not passed. Each node of the machine stores the entire group multiplication table of 14400 integers. As a partial check of the correctness of the algorithm, we have verified that the usual average plaquette results were obtained (see fig. 4).

Since the speed of each node is $\frac{1}{8}$th VAX, one would naively expect the performance of the 4-node machine to be that of $\frac{1}{2}$ of a VAX 11/780. This is, of course, degraded by the communications overhead present in a homogeneous machine, but not in a normal, sequential computer. This overhead was measured by timing the program, and then timing a version of the program in which all communications were done twice, doubling the communication overhead. The

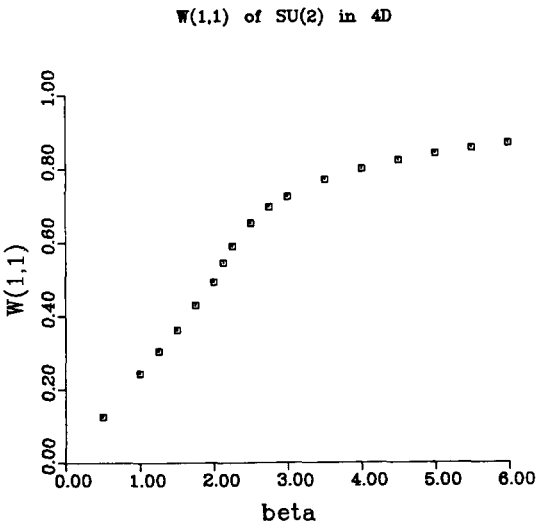

Fig. 4. Average plaquette as a function of coupling for SU(2) gauge theory in 4 dimensions.

difference in these timings then gave the time spent in interprocessor communication. The results are shown in fig. 5 for various lattice sizes, and are given in terms of percentage of total time spent in communications. Since this percentage is governed by the surface area to volume ratio of the subcells residing in each node, one expects the overhead to be worse for smaller subcells. This is apparent in fig. 5, where the worst case of a $2 \times 2 \times 4 \times 4$ subcell gave a communications percentage of 25%. The fact that the overhead grows slowly as the subcell is made smaller and that the "worst case" overhead is still a reasonably small fraction is important. We want to

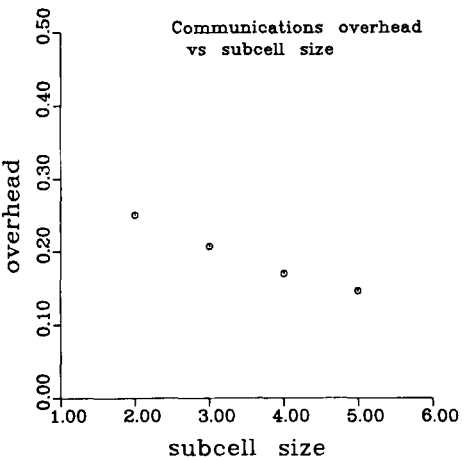

Fig. 5. Communication overhead as a function of subcell size for SU(2).

add more nodes (the 64-node machine) and run on lattices of $8^3 \times 16$, for example, so the subcells in each node will always be fairly small. The 25% figure means that the performance of these larger machines will not be severely degraded by communications overhead, at least for this icosahedral version of SU(2). The overheads given can actually be improved upon; we have not yet fully optimized our communications software.

One may worry that in a more realistic gauge theory, i.e. SU(3), one will have to pass full 3×3 complex matrices between the nodes so that the communication overhead may become quite large. This turns out not to be the case, however. The

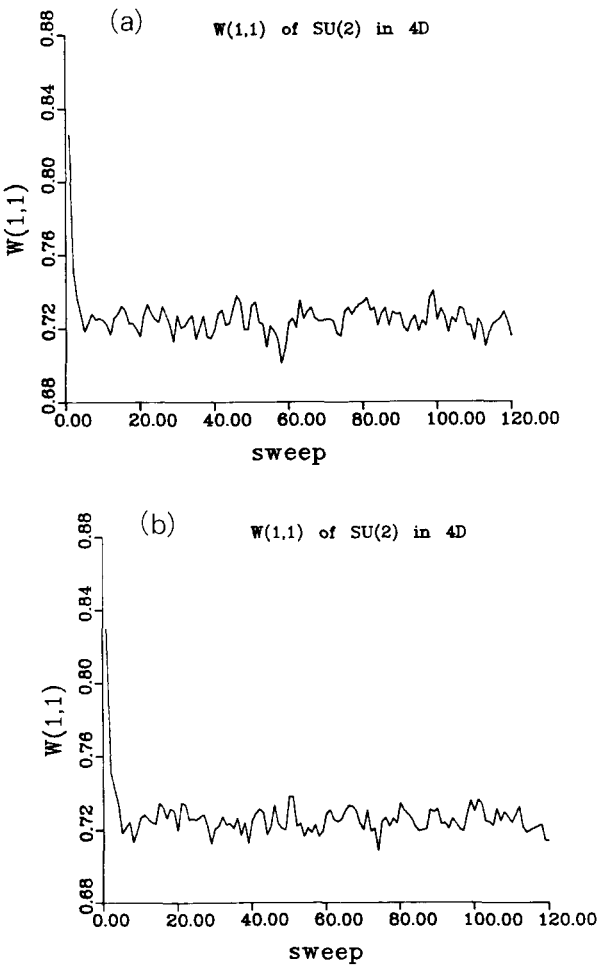

Fig. 6. Average plaquette (at $\beta = 3.0$) as a function of the sweep number for (a) regular update and (b) array update.

reason is that although the communication time does indeed grow, the computation time per matrix communication grows even faster. For example, if the size of the matrix is N, matrix multiplication grows as N^3, while the communication time goes as N^2. We have actually implemented SU(3) (to be discussed later) and find that the communications overhead is in fact smaller: for the worst case $2 \times 2 \times 4 \times 4$ subcell, we find an overhead of 18%.

Beyond the CPU time per sweep through the lattice, what ultimately counts is the speed through phase space of the observable which one is interested in. One may worry that our update procedure causes observables to go though phase space at a slower speed than the usual procedure adopted on a sequential computer. On a sequential computer, one starts a sweep by updating the links at a corner of the lattice and then looping in x, y, z, and t. At any given link update, some of the neighboring links have already been updated, causing the subsequent link choice to be less correlated with the previous sweep than if all the neighboring links were still at their "old" values. On the homogeneous machine, since we simultaneously update sets of decoupled links, it appears that more "old" links are used in the update and therefore our sweeps must be more correlated than in the usual procedure. This isn't quite correct, however. Towards the end of a sweep on the homogeneous machine less old links are used, so this must (at least partially) compensate for the larger correlation at the beginning of the sweep. To investigate this, we measured the speed through phase space of the average plaquette. Fig. 6 shows the sweep averages of this observable for the two update procedures. It is seen that both methods thermalize in approximately 4 sweeps and that the fluctuations are, at least roughly, the same. This is made more quantitative in table 1 where we show the sweep-sweep correlation. Table 1 shows that the sweep-to-sweep correlations of the two methods are similar; we conclude that the speed through phase space of the two update procedures are similar.

TABLE 1
Sweep-sweep correlations

	Average	Naive error	Error with correlations*
regular update	0.724328	0.000169	0.000209
HM update	0.724369	0.000168	0.000198

*"Error with correlations" means that sweep-to-sweep correlations among the data have been taken into account in the estimation of the error through the formula,

$$\sigma^2 = \frac{\langle A^2 \rangle - \langle A \rangle^2}{N} \left[1 + 2 \sum_{p=1}^{\infty} \frac{\langle A_i A_{i+p} \rangle - \langle A \rangle^2}{\langle A^2 \rangle - \langle A \rangle^2} \right],$$

where the subscript on the observable A labels sweep number.

6. Glueball masses in SU(2)

As a first attempt at a realistic calculation of an interesting physical observable, we have calculated the mass of the 0^+ glueball in SU(2) for a range of couplings. The method used is the combined variational 2-point correlation procedure used by various groups [12]. This is the usual 2-point method but with the operators chosen so as to maximize the signal from the particular state one is interested in. The connected correlation,

$$\Gamma(\tau) = \langle (O(\tau) - \langle O \rangle)(O(0) - \langle O \rangle) \rangle ,$$

is calculated, where O is some linear combination of gauge invariant operators:

$$O = \sum_i \alpha_i W_i .$$

The W_i are combinations of Wilson loops chosen so as to excite states of definite spin, parity and momentum. The mass is

$$m = \lim_{\tau \to \infty} -\ln \frac{\Gamma(\tau)}{\Gamma(\tau - 1)} .$$

The signal, $\Gamma(1)/\Gamma(0)$, is maximized as a function of the parameters α_i. This maximizes the overlap between O and the glueball wave function and is equivalent to minimizing m.

This method is perhaps the optimal way to calculate glueball masses. Even though m is gotten only by the asymptotic fall-off of $\Gamma(\tau)$ as $\tau \to \infty$, by including enough operators one can find a reasonable estimate of m from $\tau = 1, 2$. As a bonus, the coefficients α_i give one information about the glueball wave function.

At first sight, the calculation of Wilson loops of arbitrary shape and orientation on the homogeneous machine seems very difficult. A loop such as that shown in fig. 7d can intersect the subcells of *several* processors. Keeping track of all the necessary

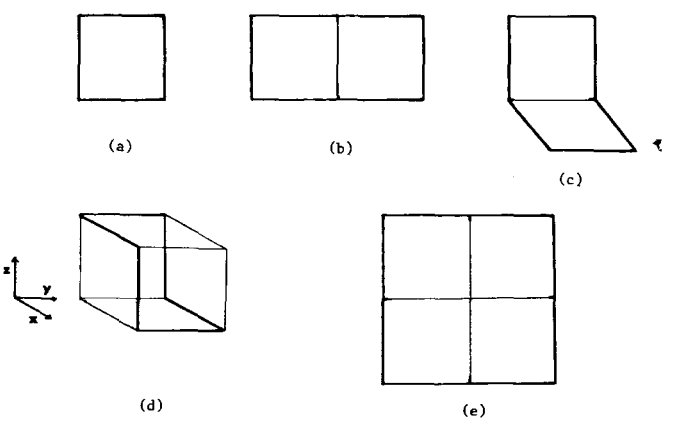

Fig. 7. Operators used in the variational calculation of the glueball wave function.

communications for increasingly complex loops (which we need for glueball calculations) is a difficult task for the programmer.

This turns out not to be the case, however, if one sets up the algorithm in the following way. Describe the shape of a loop by strings of integers, where each integer corresponds to a unit translation vector of the lattice: 1 corresponds to $+\hat{x}$, 2 corresponds to $+\hat{y}, \ldots,$ 5 corresponds to $-\hat{x}$, etc. For example, the loop of fig. 7d is described by the string "253617". A routine is then constructed which takes the starting location of the loop and a string of integers (of arbitrary length) as input, and produces the matrix product along that path. This is done by moving to the site of the lattice where the loop starts and then reading the input string one integer at a time. When an integer is read, the matrix in the corresponding direction is fetched, multiplied into the current matrix product and, finally, the current site location is incremented in the same direction. The routine is then ready to read the next integer. It continues this way, literally "walking around" the path described by the input string.

The only modification necessary for this algorithm to work on the homogeneous machine is a simple test which, at each step of the "walk", tests to see if one is stepping out of the current subcell. If this is true, the processor sends its current matrix product to the processor in the direction of the step. The string does not need to be passed. All the processors are calculating the same shape loop located at the same point in each subcell (the calculation is, again, synchronized by the communications procedure), so all processors will be at the same step of the same path.

The above algorithm for Wilson loops was quite easy to implement and once this was done, the code for glueball masses, including an arbitrary number of operators, required very little additional work. One slight complication is the fact that a 2-point correlation, being a global observable, cannot be easily calculated within the homogeneous machine. What we do is the following. First, all the Wilson loops on the lattice are calculated in the nodes of the machine. The zero-momentum operators are then gotten by adding the loops in the spatial directions—this is done by adding results and passing the results towards processor 0. The zero-momentum operators, for each time slice, now reside in processor 0 and all that needs to be done is for the 2-point correlations between the various slices to be calculated. Instead of doing this in processor 0, we pass the numbers to the intermediate host (IH) and have it calculate the correlations. This frees the array to continue on to the next sweep. Since the amount of work that the IH has to do is small in comparison to that required to sweep through a subcell, the IH has no trouble keeping up with the array.

We now present the results for the SU(2) 0^+ glueball mass. We have worked at four values of the coupling, $\beta = 2.0$, 2.1, 2.2 and 2.3 on a $4 \times 4 \times 4 \times 8$ lattice. We have collected data with very large statistics; for our most ambitious data point ($\beta = 2.3$) a total of 250 000 sweeps was generated. Roughly two-thirds of the time was spent in update and one-third in measurements. The entire computation, for all

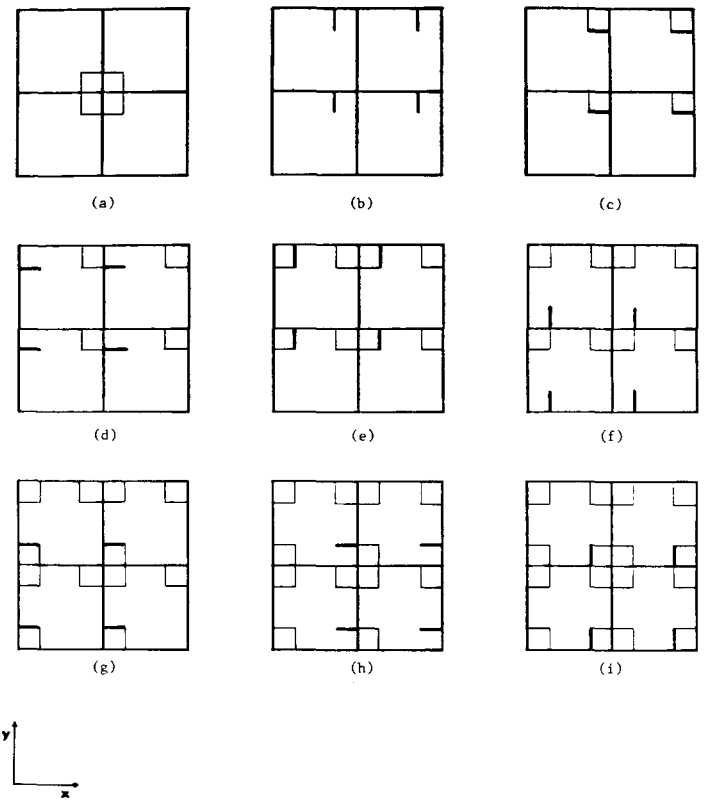

Fig. 8. Example of a Wilson-loop calculation on the 2×2 array. (a) The loop 61122556 to be calculated. (b) through (i) various stages of the computation with the dark links denoting those which are being used at the present step.

values of the coupling, took approximately 1000 hours on the 2×2 Homogeneous Machine, which is equivalent to 400 hours on the VAX.

The five operators used in the variational calculation are shown in fig. 7. These are the simple plaquette, all three operators of perimeter six and the two-by-two planar loop. Our results clearly show that in order to improve the overlap of our trial wave function with the true glueball wave function we need to use more operators. This would, however, require much more computer time than is presently available. Not only does the time spent in measurements go up with the number of operators but better statistics are also required to extract the mass reliably.

The glueball mass (in units of inverse lattice spacing) as a function of the coupling is plotted in fig. 9. The straight line drawn in the figure corresponds to the prediction of the renormalization group in two-loop perturbation theory:

$$\Lambda_{\mathrm{L}} = \frac{1}{\alpha}\left(\beta_0 g^2\right)^{-\beta_1/2\beta_0^2} \exp\left(-\frac{1}{2\beta_0 g^2}\right),$$

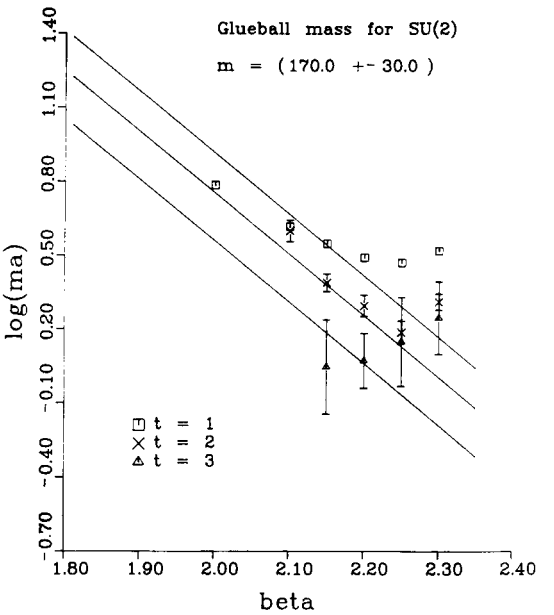

Fig. 9. Mass of the 0^+ glueball as a function of coupling for SU(2) gauge theory in 4 dimensions.

which is the lattice mass scale, related to the more conventional scale, Λ_{mom}, by

$$\Lambda_{\text{mom}} = 57.4\Lambda_{\text{L}}(\text{SU}(2)).$$

For SU(2) $\beta_0 = 11/24\pi^2$ and $\beta_1 = 17/96\pi^4$. We note that our results are almost identical to those of Berg et al. [13], though we have somewhat more statistics. If we interpret our results as being consistent with scaling behaviour we can give a value for the "physical" mass of the glueball. Using a value of 270 MeV for Λ_{mom} which is appropriate for a string tension of 400 MeV, we get

$$m(0^+) = 0.80 \pm 0.14 \text{ GeV}.$$

The meaning of this number is somewhat unclear since the string tension as obtained from phenomenology pertains to a theory with three colors and dynamical fermions.

However, we do not think that our results show unambiguously the scaling behaviour as predicted by the continuum renormalization group. The masses at $\beta = 2.15$, 2.2 are significantly lower than expected. This can, perhaps, be best understood in terms of the large peak in the specific heat which occurs at the same value of β [14]. This is purely a lattice artifact and has nothing to do with the physical continuum theory. In view of this we would be forced to work at $\beta \geqslant 2.3$ in order to investigate the scaling behaviour. We may interpret both the large specific

heat and the anomalously low mass (or large correlation length) as manifestations of a nearby phase transition in an extended coupling constant plane [15]. We find this point of view particularly illuminating, especially since it suggests a way of testing the scaling hypothesis without necessarily working at very large β's. This may be possible with the use of generalized actions. An analysis of renormalization group flows using the Migdal-Kadanoff recursion relations [16] suggests that the mixed SU(2)-SO(3) action

$$S_{\text{plaq}} = \beta_{\text{F}}\left[1 - \tfrac{1}{2}\text{tr}_{\text{F}}(U_{\text{plaq}})\right] + \beta_{\text{A}}\left[1 - \tfrac{1}{3}\text{tr}_{\text{A}}(U_{\text{plaq}})\right],$$

with the fundamental and adjoint couplings satisfying $\beta_{\text{A}} = -0.24\beta_{\text{F}}$, may approach perturbative scaling faster than the simple Wilson action. We are currently investigating this possibility.

It is difficult to go further into the weak-coupling (large β) region because of limitations on CPU time. Recall that the correlation length grows exponentially with β. This has several effects. First, one needs to use larger lattices. Second, as we approach the continuum limit the speed through phase space decreases due to critical slowing down and more sweeps have to be generated to obtain reliable statistics. Thirdly, the size of the glueball as measured in lattice units increases so that we need to include larger loops in the trial wave function. Finally, as the correlation length grows we have to go further into the correlation function to extract the mass reliably. Thus, in order to work at higher values of β one would need a more powerful computer.

7. SU(3) and an alternative algorithm

In addition to SU(2), we have also implemented pure gauge SU(3) on the 4-node machine, using the optimum Metropolis method of Okawa [17]. The character of this computation is very different from that of the SU(2) icosahedral computation. Here, floating point operations dominate, and full, 3×3 complex matrices are stored. The algorithm is identical in structure to the SU(2) algorithm; the only differences are the matrix exchange subroutines and the Metropolis update subroutine. Again, we have verified that the usual average plaquette results are obtained (compare fig. 10 and [18]).

As mentioned earlier, the communications time for this SU(3) algorithm, though larger than that for SU(2) in absolute terms is actually a smaller fraction of the total computation time (18% for SU(3) versus 25% for SU(2) for the "worst case" $2 \times 2 \times 4 \times 4$ subcell). As before, the smallness of the overhead shows that, for pure gauge SU(3), large lattices can be updated in parallel by a homogeneous machine with a large number of nodes (64 and greater) with high efficiency.

In the interest of simplifying our code, we wrote a version of the SU(3) algorithm which uses the Wilson loop routine described in sect. 7 not only to calculate

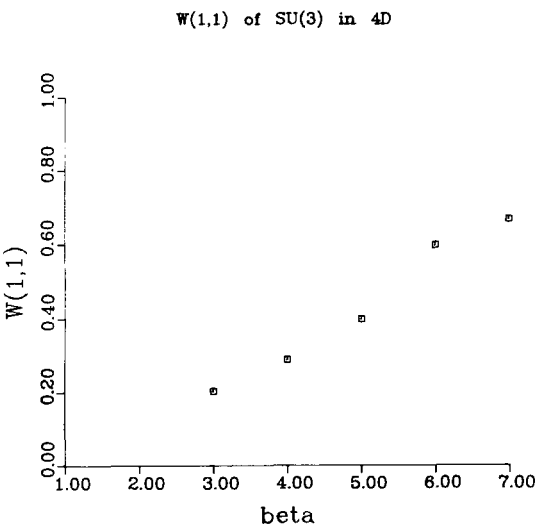

Fig. 10. Average plaquette as a function of coupling for SU(3) gauge theory in 4 dimensions.

observables, but also to do the update of the configuration. To update a link, we need the values of the 6 plaquettes which contain the link. This is found by merely calling the Wilson-loop routine with the appropriate input strings for the 6 plaquettes. This makes the code conceptually much simpler: all the parts of the code which are special to a homogeneous machine (versus a sequential computer) are localized to one small subroutine. This version of SU(3) runs somewhat slower than the original – about 10–15% slower due to additional overhead and non-optimal communications strategy coming from the use of the generalized Wilson-loop algorithm. The importance of this algorithm lies in the fact that it can now be easily extended so as to incorporate dynamical fermions using stochastic or hopping-parameter expansion methods [19]. These methods fundamentally consist of using a more complex action for the gauge links. Instead of interacting only via plaquettes, the action also contains all larger Wilson loops. These are trivial to incorporate in our algorithm: strings describing the various loops merely need to be input to the generalized Wilson-loop routine. Though, a priori, the incorporation of the hopping expansion on the homogeneous machine seems horrendous, we find that it can in fact be done with little extra work on the part of the programmer and that it will run with high efficiency. We intend to investigate this more thoroughly in the future.

8. Summary and conclusions

We have shown that pure gauge Monte Carlo calculations can be done on a homogeneous machine with high efficiency and without too much work for the programmer. We have calculated a non-trivial observable, the 0^+ glueball mass, to

high statistics and the result is consistent with previously published results. Continuing along this path, when our more powerful, 64-node machine is finished, we will find SU(3) glueball masses. We intend to find the masses of other spin, parity states besides the 0^+, and we will include many operators and determine them to very high statistics.

Leaving the flavor singlet sector of the theory (e.g. glueballs), another important problem is that of the inclusion of valence quarks ("quenched" approximation) and the calculation of flavor non-singlet masses [20]. These techniques involve inversion of matrices via Gauss-Seidel iteration. Since the matrix being inverted has a local, nearest-neighbor structure, these algorithms will be straightforward to implement on the homogeneous machine and we are currently working on this problem.

The ultimate problem is, of course, the inclusion of dynamical fermions. We have already mentioned the stochastic and hopping expansion methods; we think that these methods can be easily done on the homogeneous machine with high efficiency. Another promising technique for dynamical fermions is the pseudo-fermion method [21]. This will also work straightforwardly on the HM since, again, the action for the pseudo field is local.

It is probable that significant new progress can be made in the field of lattice gauge theories only if much more powerful computers become available – all current techniques are limited by CPU time. Since it also appears that the only way to achieve large factors in computer performance is through the use of concurrent processing, we feel that the only Monte Carlo algorithms worth considering are those which can run concurrently. We hope to continue to study these exciting problems in the future.

E.B. and S.O. acknowledge the support of a fellowship from the Shell Oil Company.

References

[1] K.G. Wilson, Cornell preprint CLNS/80/442 (1980);
M. Creutz, Phys. Rev. D21 (1980) 2308;
H. Hamber and G. Parisi, Phys. Rev. Lett. 47 (1981) 1792;
D. Weingarten, Phys. Lett. 109B (1982) 57
[2] P. Hasenfratz, Functional Integrals Workshop, ITP Santa Barbara (Aug. 1982);
C. Bernard et al., Phys. Rev. D27 (1983) 227; UCLA preprint, UCLA/82/TEP/22 (1982);
R. Gupta and A. Patel, CalTech preprint, CALT-68-966 (1982)
[3] C. Seitz, Proc. Conf. on Advanced research in VLSI, MIT, (1982)
[4] R. Levine, Scientific American (Jan. 1982) p. 118
[5] E. Brooks et al., CalTech preprint, CALT-68-867 (1982)
[6] G.C. Fox, CalTech preprint, CALT-68-986 (1982)
[7] C.A.R. Hoare, Communications of the ACM, vol. 21, no. 8 (1978) p. 666
[8] C. Hewitt, Conf. Proc. of the 1980 LISP conference, Stanford, (1980)
[9] C. Lang, Phd. Thesis, California Institute of Technology, 1982
[10] E. Brooks, G. Fox, Caltech preprints, CALT-68-920, 68-960, 68-959
[11] C. Rebbi, Phys. Rev. D21 (1980) 3350;
D. Petcher and D. Weingarten, Phys. Rev. D22 (1980) 2465;
G. Bhanot and C. Rebbi, Nucl. Phys. B180[FS2] (1981) 469

[12] K. Wilson, closing remarks at the Abingdon Lattice Meeting (March, 1981);
M. Falcioni et. al., Phys. Lett. 110B (1982) 295;
B. Berg, A. Billoire and C. Rebbi, Brookhaven preprint, BNL 30826 (1981);
K. Ishikawa, M. Teper and G. Schierholz, Phys. Lett. 110B, 399 (1982)
[13] B. Berg et al., Communications of the ACM, vol. 21, no. 8 (1978) p. 666; CERN preprint, TH.3327-CERN, (1982)
[14] B. Lautrup and M. Nauenberg, Phys. Rev. Lett. 45 (1980) 1755
[15] G. Bhanot and M. Creutz, Phys. Rev. D24 (1981) 3212
[16] K. Bitar, S. Gottlieb and C. Zachos, Phys. Rev. D26 (1982) 2853
[17] M. Okawa, Phys. Rev. Lett. 49 (1982) 353
[18] M. Creutz and K.J.M. Moriarty, Phys. Rev. D26 (1982) 2166
[19] A. Hasenfratz and P. Hasenfratz, Phys. Lett. 104B (1981) 489;
A. Hasenfratz, Z. Kunszt, P. Hasenfratz and C.B. Lang, Phys. Lett 110B (1982) 289; 117B (1982) 81;
C.B. Lang and H. Nicolai, Nucl. Phys. B200[FS4] (1982) 135;
J. Kuti, Phys. Rev. Lett. 49 (1982) 183
[20] H. Hamber and G. Parisi, Phys. Rev. Lett. 47 (1981) 1792; Phys. Rev. D27 (1983) 208;
D. Weingarten, Phys. Lett. 109B (1982) 57; Nucl. Phys. B215[FS7] (1983) 1
[21] F. Fucito, E. Marinari, G. Parisi and C. Rebbi, Nucl. Phys. B180[FS3] (1981) 369