Journal of Statistical Physics, Vol. 43, Nos. 5/6, 1986

GF11

J. Beetem,[1] M. Denneau,[1] and D. Weingarten[1]

GF11 is a parallel processor currently under construction at the IBM Yorktown Research Center. The machine incorporates 576 floating-point boards. Each board has space for 2×10^6 bytes of memory and is capable of 2×10^7 floating point operations per second, given the total machine a peak of 1.15×10^9 bytes of memory and 1.15×10^{10} floating point operations per second. The floating-point processors are interconnected by a dynamically reconfigurable switching network. At each machine cycle any of 1024 preselected permutations of data can be realized among the processors. The main intended application of GF11 is a class of calculations arising from quantum chormodynamics.

KEY WORDS: Parallel processing; super computer; quantum chromodynamics; single instruction multiple data (SIMD); Benes network.

1. INTRODUCTION

GF11 is a large parallel processor presently under construction at IBM Yorktown Heights. GF11 is intended primarily for numerical work with lattice QCD. We would like, for example, to do a fairly realistic calculation of hadron masses including the full effect of quark vacuum polarization. At this point we still don't know how much computation that would require. A possible guess is around 3×10^{17} arithmetic operations. On one of the present generation of vector machines running, say, at 100 million floating point operations per second (100 Mflops) this calculation would take 100 years. GF11 has a peak rate of 11.5 billion floating point operations per second (11.5 Gflops) and will sustain about 10 Gflops on QCD calculations. At this rate 3×10^{17} operations will take 1 year.

GF11 does not incorporate any special purpose hardware restricted to QCD. The architecture is actually fairly flexible. We can probably sustain more than two or so Gflops on a wide range of problems in science and engineering. For an earlier discussion of GF11 see Ref. 1.

[1] IBM, T. J. Watson Research Center, Yorktown Heights, New York 10598.

0022-4715/86/0600-1171$05.00/0

822/43/5-6-29

The remainder of this paper is organized as follows:

1. We begin with an overview of GF11. Then each of its component parts will be considered in greater detail.
2. The floating point processor boards are described.
3. We consider a switch through which they communicate.
4. We present the central controller which orchestrates the machine.
5. The method for generating programs to run on GF11 is described.
6. Finally, we will show how the machine can be applied to a typical problem in QCD.

2. ARCHITECTURE

The machine's overall structure is shown in Fig. 1. The heart of GF11 consists of 576 floating point processor boards labeled P_1,..., P_{576}. Each processor is capable of 20 Mflops. At any one time only some of these will be active computing. The remainder will stand by as spares, to be brought into action when an active processor fails. A typical division is 512 active processors and 64 spares. The processors communicate through a three-stage switching network, labeled Stage 1–3 in Fig. 1. The processors send data to the switch 1 byte at a time on each of 576 channels and receive data back 1 byte at a time on another 576 channels. Neither the processor

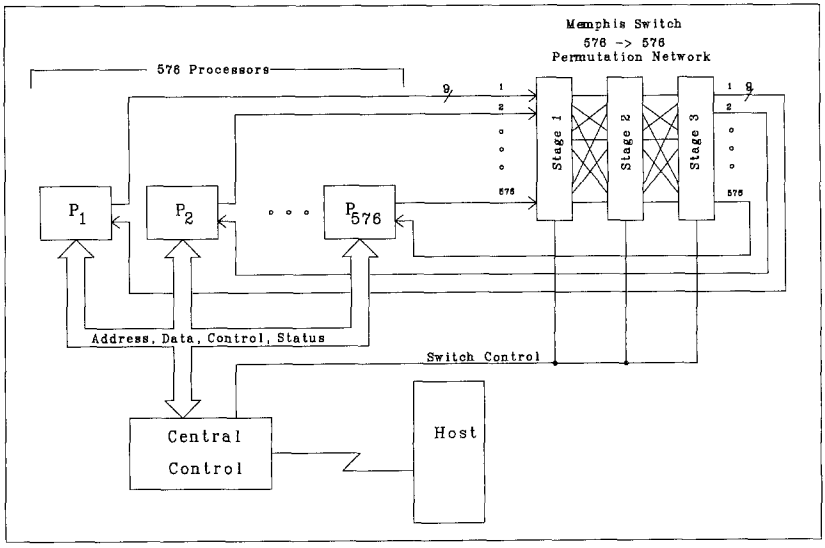

Fig. 1. The overall organization of GF11.

nor the switch carry on board any microprocessors or programs. The program is carried in a central control unit which dispatches control signals to the processors and switch.

Each processor receives an identical set of control signals from the central controller. Thus GF11 turns out to be a modified version of SIMD (Single Instruction Multiple Data) architecture. A single instruction stream in the central controller causes operations on multiple data sets, one in each of the active processors. An alternative and more popular structure for a parallel processor is MIMD (Multiple Instruction Multiple Data) architecture. In a MIMD machine each processor has its own program and controller. Each processor can then run its own independent and potentially different program. The Columbia and Caltech machines use MIMD architecture,[2] while the machine planned in Ref. 3 is SIMD.

SIMD architecture, however, has a variety of advantages over MIMD for our purposes:

1. The machine is simpler to design, debug, program, and understand.
2. Communication can be done more efficiently. All communication can be scheduled in advance with no conflicts. In a typical MIMD machine processors communicate, in effect, by making plone calls to each other. A phone call can result in a busy signal causing a processor to wait. This can turn out to be a major overhead for MIMD machines.
3. There is only one common instruction memory for the whole machine. We can therefore easily afford to make it gigantic. This is more than just a convenience, as will be shown in more detail later.
4. It is very difficult to build a general purpose processor which can keep up with the 20-Mflops nonvector floating point units in each of our processors. No microprocessors are capable of this. In GF11 only one such processor is needed in the central controller. A MIMD machine would require 576 such units.

So why does anyone ever design a MIMD parallel processor? For the simple reason that not all problems can be mapped onto a SIMD machine efficiently. As it turns out, however, a large class of scientific problems, and the calculations we want to do for QCD in particular, can be made to run efficiently on a SIMD machine. More on this later.

3. PROCESSORS

The GF11 processor is shown in Fig. 2. Each processor is built around a 20-Mflops 32-bit floating point unit and a 20-Mips 32-bit integer unit. The floating point unit consists of two Weitek 32-bit IEEE floating point

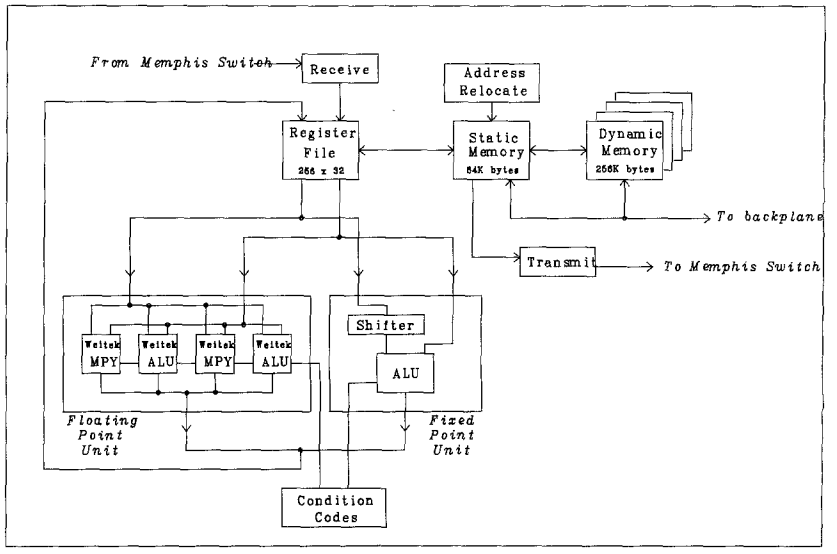

Fig. 2. The GF11 arithmetic processor.

multipliers and two Weitek 32-bit IEEE floating point arithmetic-logical units (ALUs). The ALUs can add, subtract, take absolute values, and convert between fixed and floating point formats. Each Weitek unit runs at five Mflops in pipelined mode. There is no hardware division. Division can be made a rare operation in the algorithms we are interested in. It can be done by a table look up followed by two adds and five multiplies.

The integer unit can add, subtract, and perform all logical operations. There is also a full 32-bit barrel shifter on one input. The integer unit is intended primarily for bit manipulation needed to prepare floating point numbers to be used as addresses in table look up. If we wanted, however, we could also use them just as fixed point calculators. In this case GF11 can run up to a peak of 11.5 billion instructions per second (11.5 GIPS).

Handling data for the functional units is a three-stage hierarchy of progressively larger and slower memory units. Communicating directly with the fixed and floating point units is the fastest and smallest stage, a 256-word register file. Every 50 ns it is capable of four I/O operators. It sends out two input words to the functional units, collects one result, and either receives an incoming word from the switch or sends or receives a word from the next stage of memory, 16,000 words of high-speed static random access memory. Since the floating point unit and the fixed point unit, running full tilt, both take two inputs and produce one output every 50 ns, the register file is capable of keeping either fully occupied, but not

both at once. On any cycle when one is used for input or output, the other can not perform the same operation.

The static random access memory (SRAM), behind the register file, can perform one 32-bit I/O operation every 50 ns. This may seem too slow, by a factor of 3, to keep the functional units satisfied. In scientific calculations, however, and especially for QCD, there is a large amount of chaining. Results of one operation are quickly reused as input to a new operation. The numbers to be reused are kept in the register file and not sent back to the static memory. For typical QCD calculations there are between seven and nine operations done for each word of I/O of the SRAM. The floating point units would be satisfied even if this factor were only 3.

Finally, behind the SRAM is the largest and slowest chunk of memory, 512 thousand words of dynamic random access memory (DRAM). The dynamic memory is divided into two banks of 256 thousand words which together can provide one I/O operation every 200 ns. The DRAM will be used, in effect, like a disk. The total memory capacity of the machine is 256 million words of data, equivalently 1 billion bytes (1 Gbyte).

There are two additional small chunks of memory on each board which help circumvent some of the limitations of SIMD architecture. A set of 256 registers carry bases added on to each incoming SRAM address. Therefore, different processors can work on different data at the same time. A set of eight condition code bits are set as the result of fixed or floating point operations. These bits can be used to modulate in a processor-dependent way the effect of incoming instructions. For example, every operation on a pair of words A and B can be caused, by a zero condition code, to return just A, and any data store to SRAM can be aborted by a zero condition code.

This is essentially all the internal function of the processor. There are also four channels of communication between each processor and the rest of the world. An incoming line carries 256 bits of control signal every 50 ns to run all the various units. Every 50 ns, 1 byte of data also arrives from the switch into a buffer register and is available to be grabbed by the register file. Every 50 ns another byte of data goes off to the switch from a register filled from the SRAM. Finally there is a single common bus running through the machine to which each processor can dump data and from which it can receive data at the rate of one 32-bit word every 50 ns. This is used to load the machine initially and collect up final results.

4. SWITCH

We now consider the switch. The switch is shown in Fig. 3. It consists, as we have already said, of three stages. Each stage consists of 24 nodes. Each node has 24 inputs 1 byte wide and 24 outputs 1 byte wide. Thus $24 \times 24 = 576$ inputs to the first stage and outputs from the third stage of the full switch. Every 50 ns, each of the 24 outputs of each node receive the data of one of the 24 inputs. It follows from a result of Benes,[4] that there is at least one setting of the nodes of the switch which will make the 576 outputs any chosen permutation of the 576 inputs.

Each word of data is sent through the switch in four sequential bytes and requires, therefore, 200 ns. Every 200 ns the switch setting can be reset by the central controller. As it turns out, however, the amount of data required to select a complete setting of all 72 nodes is rather large, 8640 bits, and could not be shipped down to the switch in time. Instead, 1024 different sets of switch setting bits can be stored in memory on board the switch. The controller chooses one of these every 200 ns by sending down only a 10-bit address.

So before a job is run, the switch is loaded with 1024 settings selected to suit the problem. These are kept in place through the course of the calculation. By choosing these permutations appropriately, we can get the machine to behave as though there were permanent data paths connecting each processor to a set of neighbors with a wide range of topologies. For

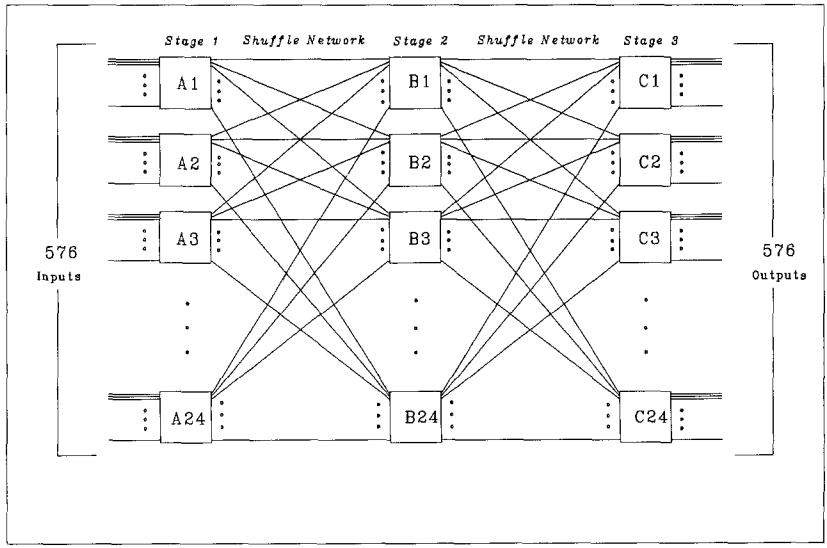

Fig. 3. The GF11 switch.

example, the machine can be configured as a hypercubic lattice with an arbitrarily chosen dimension d and arbitrarily chosen sizes in each direction. To do this requires $2d$ switch settings. Setting 1 passes data from each processor to its nearest neighbor in the positive 1-direction, setting 2 passes data to the neighbor in the negative 1-direction, setting 3 passes data in the positive 2-direction, and so on. The only limit is that the number of processor nodes must be no greater than 576. This in effect limits d to nine or less since $2^9 = 512$. All sorts of irregular coupling schemes can also be used.

The switch makes the machine's communication quite flexible and allows it to work effectively on problems defined on a wide range of different lattices. The switch also provides a convenient means of replacing failed processors by spares. All processors are always physically connected to the switch. The inactive ones have merely not been loaded with data and are never allowed, by a choice of switch setting, to send or received data from an active processor. To remove a failed processor and replace it with a spare, we simply modify the switch setting data in the switch's memory so that data which was originally sent to and from the failed processor is now sent to and from its replacement. To permit inexpensive recovery from failures, the machine's full configuration will be run onto disk every so often (every few hours or so). When a failure is found, the switch setting will be changed and the machine reloaded with the most recent correct checkpoint file and started again. Another way to replace failed processors, of course, is by physically reconnecting wires. This is more time-consuming, however, and less reliable. Every time cables are moved, you run the risk of breaking some and thereby introducing still more problems to correct.

5. CONTROLLER

As mentioned earlier, the control signals for the processors and the switch come from a single central control unit. The controller is shown in Fig. 4. The main requirement on the controller is that it must produce over 220 bits of control signal every 50 ns. This immediately rules out a microprocessor or moderate-size group of microprocessors. A special purpose high-speed controller could be built to compute the control signals on the fly, but this would be a major undertaking and might require compromises, such as a choice of instruction set, which would limit the machine's flexibiility. The procedure we adopted instead is to generate almost all of the required control signals in advance and place the result in a large central microcode store. The microcode store is divided into address storage and instruction storage, both of which are shown in Fig. 4. The

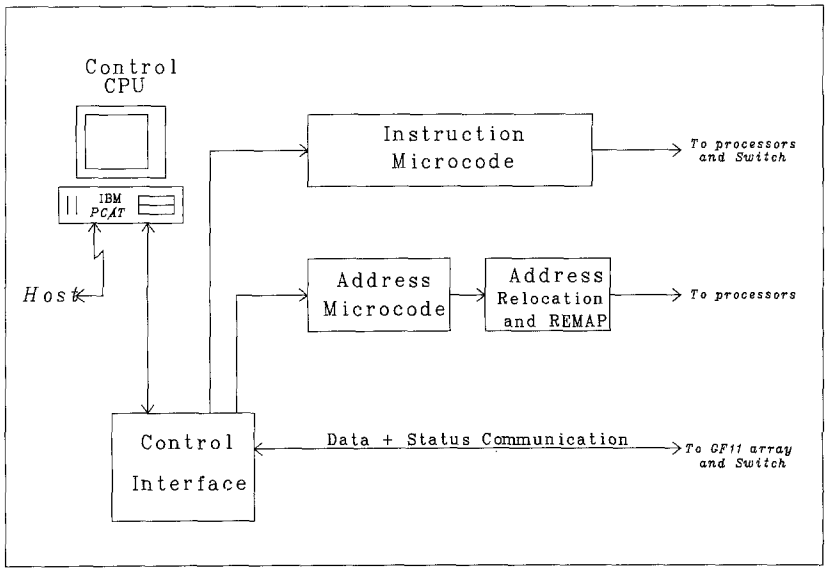

Fig. 4. The GF11 central controller.

maximum total capacity of the microcode store will be 4×10^6 256 bit control words.

Used only once, the total stored microcode will run the machine for only 200 ms. This limitation is circumvented by using the microcode as a collection of separate subroutines each of which can be called by itself and each of which has variables for addresses rather than absolute memory locations. The address variables in the address store are then converted into real addresses by the address relocation hardware shown in Fig. 4. In a typical scientific applications such as QCD calculations, the time-consuming work is contained in inner loops which are repeated a large number of times with only the loop index altered. For GF11 this can be done by cycling through a single microcode routine with the address relocation hardware set differently on each cycle.

The relocation hardware can affect two different types of modifications. The addresses for SRAM can be relocated by adding a base address from any one of 1024 relocation registers. The resulting address can then be remapped through a look-up table. This table can be used, for example, to convert base shifts from the first relocation stage into periodic shifts through arrays with periodic boundaries.

The selection of microcode subroutines, loading of relocation registers and control of the remap look-up table are all done through the control interface by the control CPU. An IBM PC/AT will be used as the control

CPU. If the microcode subroutines are long enough (several hundred instructions or more) the PC/AT has enough time while each subroutine is executing to set up the subroutine to follow.

6. SOFTWARE

Generating the microcode for GF11 by hand would be impossible. The control word is too large, over 220 bits, and the processor's function and pipeline skews are too complicated for anyone to manipulate directly. On the other hand, we did not want to invent a new high-level language and compiler. The alternative approach we have chosen uses Pascal on the host machine and on the control CPU as our high-level language.

GF11 software comes in two parts: microcode subroutines and a master calling program. The master program runs on the control CPU and is a conventional Pascal program augmented with calls to special purpose procedures. These procedures communicate with the control interface to set up address relocation, start and stop microcode execution, and transfer data and the load module between the control CPU and the microcode and processor memories. Microcode subroutines are created using Pascal on an IBM 3081 host. The code creation program looks almost like a program which causes the host to do the calculation intended for GF11. But in place of Pascal + or * arithmetic operations which would cause arithmetic on the 3081, we have procedures we have written with names like ADD and MULT. The result of these procedures is to create a description of the actions needed on GF11 to do an add or multiply, respectively. More precisely, a microcode generation program run on the 3081 yields a file containing the operation tree required on GF11. The operation tree is then passed to an optimizer which packs it into GF11 control words as tightly as possible to complete the required work in the smallest number of machine cycles. The corresponding optimal microcode is then ready to be loaded into the GF11 controller's microcode store. For typical pieces of QCD computation, which we have examined in detail, the optimized microcode achieves over 90% utilization of the GF11 arithmetic units.

Crucial to sustaining high utilization of the arithmetic units across a full program is our ability to divide any problem into fairly large units of microcode with of the order of hundreds of operations in each subroutine. Since much of the machine's operation has rather deep pipelines, as much as 20 or 30 stages deep in some places, shorter units of microcode loose efficiency to pipeline start-up and shut-down overhead. This is a problem encountered also in conventional vector machines. Here, however, we require no special vector structure for our algorithms to obtain efficient

microcode. We can obtain high utilization by packing the microcode with arithmetic done on an arbitrarily scattered set of data located anywhere in a processor's memory.

7. QCD

This completes the description of GF11. In this final section we briefly discuss how a typical QCD problem can be mapped onto GF11.

A typical QCD calculation is the evaluation of hardon masses and other parameters in the full theory including the vacuum polarization arising from virtual quarks. Some of the first algorithms suggested for this problem are given in Refs. 5 and 6. An improved version of the algorithm of Ref. 5 was given recently in Ref. 7, and a related method proposed independently in Ref. 8. Based in part on the results of Ref. 9, we suspect that the fastest methods presently available are the algorithms of Refs. 7, 8.

For all of these algorithms the basic setup for lattice QCD is essentially the same. The theory is defined on some N^4 hypercubic lattice with periodic boundary conditions. On each nearest-neighbor link (x, y) is defined a matrix $U(x, y) \in SU(3)$ representing the chromoelectric field. On each sight x is defined a 12-component complex vector $\phi_a(x)$ related to the quark field and an auxiliary 12-component complex vector $\psi_a(x)$. We will not give a full definition of lattice QCD or discuss how physical quantities can be extracted by the various algorithms of Refs. 5–8. It turns out, however, that for the algorithms of Refs. 5, 7, and 8, essentially all of the arithmetic is spent solving an equation of the form

$$\sum_{b,y} M_{ab}(x, y)\, \psi_b(y) = \phi_a(x) \tag{7.1}$$

for the field $\psi_a(x)$ with $\phi_a(x)$ given. The matrix $M_{ab}(x, y)$ is nonzero only for a pair of sites x and y which differ by at most one lattice spacing and is determined by $U(x, y)$. Equation (7.1) can be solved conveniently by either a Gauss–Seidel iteration or the conjugate gradient algorithm.

At present, unfortunately, we do not know how many times the algorithms of Refs. 7 and 8 will have to solve (7.1) to calculate hadron masses. Therefore, we do not know what size lattice GF11 will be capable of handling. As a result of Ref. 9 we do have a pretty good estimate, however, of the amount of work which would be required using the algorithms of Refs. 5 and 6. For either of these methods, GF11 would be capable of calculation on a lattice of size 6^4 or perhaps 8^4. Thus, for the methods of Refs. 7 and 8, we expect to be able to run on a lattice of at least

Table I. GF11 Configuration for Various QCD Lattices

QCD lattice	Processor lattice	Lattice segment in each processor	GF11 peak speed in Gflops
6^4	$6^3 \times 2$	3	8.64
8^4	8^3	8^3	10.12
10^4	$10^2 \times 5$	2×10	10.0
12^4	$6^3 \times 2$	$2^3 \times 6$	8.64
16^4	8^3	$2^3 \times 16$	10.12
18^4	$6^3 \times 2$	$3^3 \times 9$	8.64
20^4	$10^2 \times 5$	$2^2 \times 4 \times 20$	10.0
22^4	22^2	22^2	9.68
24^4	8^3	$3^3 \times 24$	10.12
28^4	28×4^2	$7^2 \times 28$	8.96
30^4	$10^2 \times 5$	$3^2 \times 6 \times 30$	10.0
32^4	8^3	$4^3 \times 32$	10.12

this size. The largest lattice for which we will have memory space on GF11 will be 32^4. Where in the range from 6^4 to 32^4 we will actually want to do calculations is not yet clear. With this uncertainty, the GF11 switch turns out to be a great advantage. By choosing the correct set of permutations for the switch memory, we can connect up GF11's processors in a wide range of different lattices and thereby solve (7.1) efficiently for many different choices of QCD lattice. Table I shows how GF11 can be configured for various different QCD lattices. The first column shows the QCD lattice dimensions and the second shows the GF11 processor lattice. In each case the GF11 switch will be used to realize nearest-neighbor connections with periodic boundary conditions among the GF11 processors. Each processor will then be assigned some contiguous segment of the QCD lattice to store in its memory. This is shown in the third column of Table I. Column 4 shows the peak arithmetic rate which can be obtained with the corresponding lattice of GF11 processors.

Let us now consider the solution of (7.1) on a typical lattice, say 24^4, in a bit more detail. For a 24^4 lattice, according to Table I, we use 512 processors indexed as an 8^3 array. Each processor is given the task of managing the data for a $3^3 \times 24$ contiguous segment of lattice. The data for $\phi_a(x)$, $\psi_a(x)$, $U(x, y)$, and data sets needed as working space for a conjugate gradient algorithm occupy too much space to fit into the processor's SRAM and are therefore placed in DRAM and brought down to the SRAM as needed. The conjugate gradient algorithm spends essentially all

its arithmetic on three routines. The first of these, MUL(A, B), takes a vector $A_a(x)$ as input and returns

$$B_a(x) = \sum_{b,y} M_{ab}(x, y) A_b(y) \tag{7.2}$$

The second routine, MULAD(A, B), does the same as MUL(A, B) but for the adjoint of the matrix $M_{ab}(x, y)$. The third routine, LIN(A, B, C, s), takes $A_a(x)$, $B_a(x)$, and s as inputs and returns the linear combination

$$C_a(x) = A_a(x) + sB_a(x) \tag{7.3}$$

To implement MUL(A, B), data in each processor is brought from the DRAM into SRAM in slices 3^3 each, with three adjoining slices present in SRAM at any time. To find $B_a(x)$ according to (7.2), data for $M_{ab}(x, y)$ and $A_b(y)$ for some of the nearest-neighbor y can be obtained from the base processor assigned the site x, and data for the remaining y can be gotten from nearest-neighbor processors through the switch. All processors will travel over their segments of the QCD lattice in lock step. In this case, the single instruction stream sent down from the central controller does not even need any processor-dependent modulation using condition codes from the eight-bit condition code memory. When all the sites on a particular 3^3 slice have been finished in the processors, each processor will fetch a new 3^3 slice from DRAM to SRAM, and the evaluation of (7.2) on a new slice will be started. A detailed examination of the arithmetic required for this process shows that for each word of I/O required of the DRAM, the arithmetic units will do 20 operations. Since 20 operations takes 1000 ns while one I/O takes only 200 ns, this process will not be obstructed by dynamic RAM speed. A simulation of the register allocation problem for MUL shows that we will be able to obtain close to 95% of the peak rate listed in Table I. Identical results hold for MULAD. A similar analysis can be done for LIN. Here we find we are limited by DRAM speed and will obtain only about 20% of the peak. Since almost all of the arithmetic for the solution of (7.1) by conjugate gradient is spent in MUL and MULAD, however, we find a sustained rate still in the neighborhood of 90% of the peak.

Since almost all the arithmetic for the algorithms of Ref. 7 and 8 is spent in the conjugate gradient, we expect an overall sustained rate again in the neighborhood of 90% of peak. For a 24^4 lattice we arrive at a sustained rate near 9 Gflops. This whole discussion could be repeated almost identically for the other lattices in Table I larger than 12^4. For lattices of size 12^4 and smaller we are not forced to work with DRAM to solve (7.2), and slightly higher fractions of the peak rates in Table I can be sustained.

REFERENCES

1. J. Beetem, M. Denneau, and D. Weingarten, *IEEE Proceedings of the 12th Annual International Symposium on Computer Architecture* (IEEE Computer Society, Washington, D.C., 1985).
2. N. Christ and A. Terrano, *IEEE Trans. Comput. C* **33**:344 (1984); C. Seitz, *J. VLSI Comput. Syst.* **1**, no. 3 (1984).
3. R. Brower, R. Giles, and G. Maturana, Boston University preprint (1985).
4. V. Benes, *Bell Syst. Tech. J.* **43**:1641 (1964).
5. D. Weingarten and D. Petcher, *Phys. Lett. B* **99**:333 (1981).
6. F. Fucito, E. Marinari, G. Parisi, and C. Rebbi, *Nucl. Phys. B* **180**:369 (1981).
7. A. Ukawa and M. Fukugita, *Phys. Rev. Lett.* **55**:1854 (1985).
8. G. Batrouni, G. Katz, A. Kronfeld, P. Lepage, and K. Wilson, *Phys. Rev. D* **32**:2736 (1985).
9. D. Weingarten, *Nucl. Phys. B* **257**:629 (1985).