CONF-9001172--1

Fermi National Accelerator Laboratory

59
7/7/92 (2)

FNAL/C--92/125

DE92 016006

Balance in Machine Architecture
Bandwidth on board and off board, integer/control speed and flops versus memory

M. Fischler

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

April 1992

Presented at the Terraflop Workshop, Tallahassee, Florida, January 1990

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CHO3000 with the United States Department of Energy

Disclaimer

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Balance in Machine Architecture

bandwidth on board and off board, integer/control speed, & flops versus memory

by Mark Fischler, Fermilab

Introduction — Issues Addressed, Assumptions and Algorithms Examined

The issues to be addressed here are those of 'balance' in machine architecture. By this, we mean how much emphasis must be placed on various aspects of the system to maximize its usefulness for physics. There are three components that contribute to the utility of a system: How the machine can be used, how big a problem can be attacked, and what the effective capabilities (power) of the hardware are like.

The effective power issue is a matter of evaluating the impact of design decisions trading off architectural features such as memory bandwidth and interprocessor communication capabilities. What is studied is the effect these machine parameters have on how quickly the system can solve desired problems. There is a reasonable method for studying this: One selects a few representative algorithms and computes the impact of changing memory bandwidths, and so forth. The only room for controversy here is in the selection of representative problems.

The issue of how big a problem can be attacked boils down to a balance of memory size versus power. Although this is a balance issue, it is very different than the effective power situation, because no firm answer can be given at this time. The power to memory ratio is highly problem dependant, and optimizing it requires several pieces of physics input, including: how big a lattice is needed for interesting results; what sort of algorithms are best to use; and how many sweeps are needed to get valid results. We seem to be at the threshold of learning things about these issues, but for now, the memory size issue will necessarily be addressed in terms of best guesses, rules of thumb, and researchers' opinions.

The important issue of how the machine can be used (MIMD/SIMD; nature of communications network, scalability, system sharing, ease of programming, etc.) will **not** be covered in this presentation. These issues are as critical as system size and power in determining how useful a machine is for doing physics. Another topic which will be covered in a different talk is the issue of bandwidth to mass storage devices.

In examining the machine parameter issues, we will concern ourselves with what values are needed, rather than with how to achieve those values. So, we will produce information like "if communication is slow in such-and-thus way, it will cost x percent in effective power". Our concession to realistic values comes when we fix all the parameters but one, to study the effect of the last value. The study is defined by the choice of parameters varied, the variety of machine architectures considered, the assumptions made about how coding will be done, and the selection of representative problems.

The parameters we will look at are:

- Bandwidth from local (or shared) memory to the processing unit.
- Interprocessor communications bandwidth and overhead.
- The effect of multiple floating point units per processor.
- How important integer capability is.
- The effect of caches, external registers, and register set size.
- The effect of double precision speed.

Details of how these parameters are described appear in appendix A.

To pin down estimated timings, it is necessary to have in mind a model of how the memory access and communication works (the machine "architecture"). We consider several models: Shared Memory, Lockstep, Lockstep Cluster, and MIMD (these are defined and illustrated in appendix B). Presenting all combinations of architecture and parameters would be a daunting task. Fortunately, with certain obvious exceptions, the effects of various parameter values are largely insensitive to the architecture chosen.

The coding assumptions made are conservative. If operations can be overlapped only at the cost of extreme custom programming efforts, and by harming the modularity and re-usability of code, we assume these optimizations will not be made. While we do not count on heroic efforts, we do assume that the structured kernels of algorithms are optimized, to yield the best local performance realistically possible. In the long run, for important programs this should be true, since this sort of optimization can be done in a modular fashion, with reasonable expectation of correctness.

Finally, the choice of problems. Benchmark (or, as in this case, *gedanken* benchmark) problems must be representative of the actual use they are modeling. Two ways to keep these models faithful are to avoid bias by using actual front-line production algorithms, and to avoid the pitfall of simplification, which tends to magnify any deviation from reality. Thus, one should study actual algorithms, including all the messy nitty-gritty that is always overlooked in cursory evaluations. This can be time-consuming; nonetheless, it is important to sample more than one algorithm, if only to get an idea of the statistical spread of the results.

Obviously, the algorithms used will evolve as physicists learn more about how various methods behave. The analysis presented here provides guidance for designing a system based on today's knowledge, and a quantitative framework within which requirements for running new algorithms can be discussed.

The machines we are interested in are targeted at problems in lattice gauge theory. Much of the work being done today can be categorized as follows:

- Gauge Field
  - Environment calculation
  - Heatbath computations
  - Langevin or molecular dynamical stepping
- Quark Field
  - Propagator calculation (quenched)

D-slash inversion (dynamic)
[Either Wilson or Susskind]

- Operator Analysis
  Local operators
  Smeared operators

It is important to be able to do the analysis phase on the same powerful system as the rest of the calculation. No existing computer is appropriate for handling the physics analysis of configurations which would be produced by a hundred Gigaflop machine. For these computations, the issues of memory size and how the machine can be used are vital, and the power needed is beyond that available on ordinary computers. If the "primary" machine is not suitable for doing this analysis, then another "analysis system" will need to be designed. This phase is, however, not CPU time-critical, since it takes more than an order of magnitude fewer cycles than even quenched field and propagator computation. Thus the effect of machine parameters on power is unimportant — the operator analysis is not a suitable problem for studying the appropriate balance in parameters (although it may be appropriate for examining how much I/O bandwidth is needed).

For physics without fermion loops, the time taken for generation of gauge configurations is (after recent advances in propagator inversion technique) comparable to the time required for finding propagators for a few mass values. This gauge configuration time is divided into two roughly equal parts — environment calculation (staple sums) and heatbath computation.

For dynamic fermion physics, a propagator inversion must be done for each step; this leads to those problems being dominated (at the 80 - 99% level) by Dslash inversion. Here, there is much greater uncertainty in the ultimate choice of algorithms, since techniques seem to be improving by an order of magnitude every few years. However, except for issues of MIMD capability, the balance between bandwidths and power required seems not to be changing radically any more.

We choose as our "benchmarks" two algorithms. The first is a minimum-residual method of inverting quark propagators, employing the 'Draper trick'; in terms of effects of various parameters, this is nearly identical to the minimum-residual LU method which currently seems best. Quark inversion will dominate dynamic calculations, and is more than half the time for pure gauge when many physically interesting quantities are to be extracted. The second problem is pure gauge environment and Cabibbo-Marinari heat-bath updating. This turns out to be a bit less bandwidth dependant than the propagator algorithm. Where there results for these problems differ significantly, that will be pointed out.

Descriptions of the Algorithms Studied

The two "benchmark" algorithms analyzed are described here. A more detailed analysis, along with quantitative results on how long each step will take for various values of machine parameters, is presented in appendix C.

The propagator inversion algorithm studied is a minimal residual method. This is very similar, in terms of requirements on integer power and various bandwidth requirements, to the entire broad class of conjugate gradient-like methods. A slight variant on this method (incomplete LU decomposition) seems to be the current best choice for inversion in the physically interesting region; the balance between power and bandwidths is nearly the same for this.

The algorithm can be separated into two parts — computing the $\not{D}$ operator on the sites, and the doing the various dot-product and linear local field operations to complete the minimum-residual step. The $\not{D}$ part is 80% of the problem in terms of raw flops, but ignoring the rest of the algorithm would introduce bias which would not be insignificant.

The $\not{D}$ calculation computes for each site x the quantity

$$\psi(x) - k \left\{ \sum_{\hat{\mu}} (1-\gamma_\mu)\, U_{\hat{\mu}}(x)\, \psi(x+\hat{\mu}) + \sum_{\hat{\mu}} (1+\gamma_\mu)\, U^\dagger_{\hat{\mu}}(x-\hat{\mu})\, \psi(x-\hat{\mu}) \right\}$$

(where k is the hopping parameter related to the bare quark mass). Thus for each of eight directions, one must accumulate an expression of the form $U\gamma_\mu\psi$.

The bulk of the flops appear in the multiplication of the quark field by the link U (but if only this step were analyzed, the results for power would be completely skewed). The naive computational burden can be halved by combining two rows of the quark field before multiplication — this takes advantage of the nature of the gamma matrices. So, the efficient computation of the $\not{D}$ operator can be broken into four phases: Locating the needed fields and getting any off-node data; combining the quark field into two color vectors; multiplying by the link field U, and accumulating the result.

The remainder of the minimum-residual algorithm consists of finding a pair of dot products of the form $\psi \bullet \psi$ and $\psi \bullet \phi$ where ψ and ϕ are quark fields, and doing a pair of linear accumulations of the forms $\psi = \psi + \alpha\ \phi$; $\phi = \phi - \alpha\ \omega$, where α is a complex scalar. The dot products can be done together, as can the linear accumulations, but α depends on the results of the dot products. So these miscellaneous operations must be broken into two phases.

The details of how long each of these six phases will take are presented in appendix C. Each is typically the maximum of two quantities, representing the fact that two concurrent sorts of operations are happening, with the time determined by the slower. For example, you may be doing memory operations to supply data for floating point computation. Here we present values for typical balances among the machine parameters and typical surface/volume ratio:

locating fields:	$122\,I + 2\,(1.5\,O_c + 33C)$
combining quark field:	$192\,M$

multiplying by link U:	576 F
accumulating result:	384 M
dot products:	26 I + 72 D
linear accumulations:	20 I + 192 F
Total time:	168 I + 576 M + 768 F + 72 D + 3 O_c + 66 C

The gauge configuration generation algorithm studied is the Cabibbo-Marinari heat bath method. People have spent quite a bit of computer time running this or similar methods to study quenched QCD. The algorithm can be separated into two parts — computing the "environment" in which the link is to be updated, and doing the heat-bath updating (using the Kennedy-Pendleton or Creutz technique on each of the SU(2) subspaces). The environment computation is 70% of the problem in terms of raw flops, but ignoring the rest of the algorithm would introduce bias which would not be insignificant.

The environment calculation computes for each link the "staple sum", that is, the sum of six three-link products, with each three-link "staple" forming three sides of a plaquette (square) which would be completed by the link being udpdated.

The bulk of the flops appear in the multiplications of the link fields. The naive computational burden can be cut by 25% by calculating only two rows of some SU(3) products, using the unitarity property to reconstruct a third row at the end of an entire staple. The efficient computation of the environment consists of three phases: Locating the needed fields and getting any off-node data; multiplying the three link fields to form a staple, and accumulating the result.

The heat-bath part of the problem repeats a computation three times. The calculation involves three phases: Using the Creutz (or another) algorithm to get the diagonal part of an SU(2) matrix, with distribution based on some SU(2) subset of the environment; constructing a full SU(2) element from that value; and multiplying the link and environment by that SU(2) matrix.

The details of how long each of these six phases will take are presented in appendix C. Here we present values for typical balances among the machine parameters and typical surface/volume ratio. (In some instances, although we show floating point speed as the determining factor, memory bandwidth might be close, so this floating point to memory ratio may be deceptive. In particular, although floating point seems 15 times as critical as memory bandwidth, a memory bandwidth of one word every 15 cycles would emphatically NOT be reasonable).

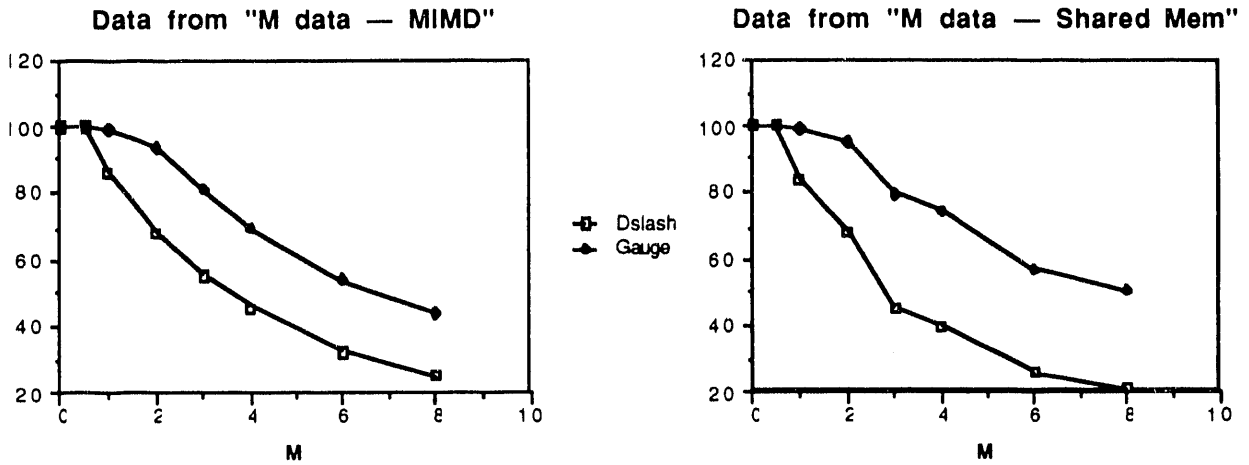
locating fields:	190 I + 2 (2 O_c + 36C)
multiplying links to get staple:	1008 F
accumulating staples:	108 M
Creutz algorithm:	186 I + 225 F
forming SU(2) matrix:	90 I + 203 F
updating link, environment:	240 F
Total time:	168 I + 108 M + 1676 F + 4 O_c + 72 C

Effects (on Power) of Changing Machine Parameters

Memory Bandwidth; Cache; Registers

The bandwidth between main local memory (or shared memory if there are no local banks) is the most important machine parameter, after floating point cycle time. This bandwidth is measured by how many cycles (M) it takes to load or store one 32-bit word of data. (M is, in a sense, the inverse bandwidth). An architecture which can deliver two words per floating point multiply/add cycle (M = .5) loses very little time to memory access. However, a good deal of effort can go into increasing memory bandwidth; high bandwidth may severely impact system memory size or cost by requiring fast static RAM.

The effect of increased M (decreased bandwidth) is dependant on the problem being done: The quark propagator computation is particularly memory intensive. The results for fixed, reasonable values of other parameters (I=3, C=2) are shown:



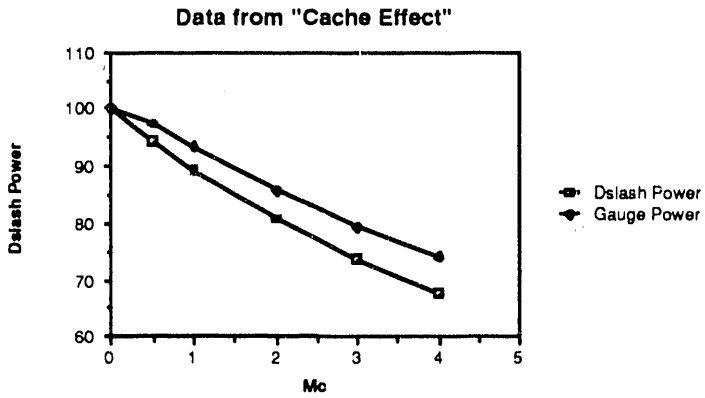
Three points are clear here: (1) Beyond M=1 the performance on quark inversion is quite sensitive to memory bandwidth, and in fact is memory bandwidth dominated by the time M gets to 4. (2) The memory bandwidth requirements are about half as severe for the pure gauge computation. (3) The impact of memory bandwidth limitations is not very sensitive to architecture (the two extremes in memory architecture are shown; the difference is slight).

We have ignored memory latency (the delay between requesting data and getting it). This latency will impact integer performance (and can be reflected by increased I values), but most of the other accesses will be in situations where useful work can be done during the latency cycles.

The effect of cache on performance is limited by low cache hit rates inherent in lattice algorithms. When sweeping through a grid and looking at data from neighboring sites, at most half of your accesses can be from sites "in your wake" (data recently accessed) — assuming that the entire lattice cannot fit into cache. The remainder of the data must be "fresh". The situation is worse for portions of the algorithm that do not require data from neighbors. This effect means that

cache hit rates are limited to 15 - 50%, depending on cache size. With realistic cache sizes, hit rates will be around 30%.

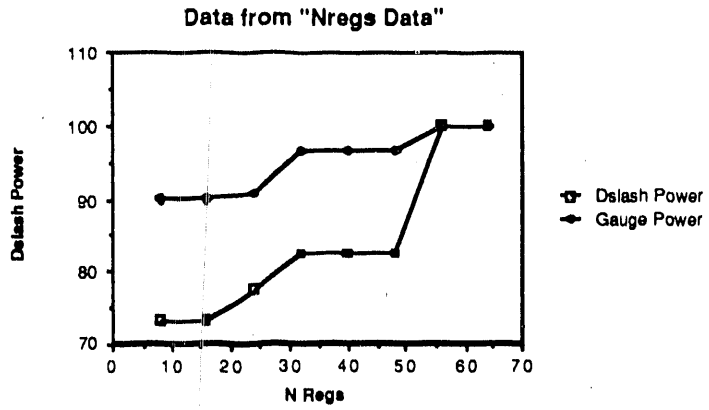
The apparent cache hit rate is actually higher than this, for two reasons. First, cache is typically filled several words at a time (lattice problems take good advantage of this). This is moot for our calculations, however — you still need the same number of words delivered from main memory. The second reason is re-used data: the second time a word is needed for doing something at a site, it will "always" hit cache. For the problems looked at, assuming there are enough registers, time is rarely wasted re-loading data. (Data is indeed re-loaded, but mostly during floating-point intensive parts of the problem.) This effect is not negligible, because "enough registers" may be more than 50, but with at least 32 registers, it is small.



The graph shown here explores the impact of cache for a system with main memory bandwidth of one word every four cycles (M=4). Mc is the number of cycles it takes to get a word from cache (thus if Mc were 4 or more, the cache would be useless). We see that for this case, a fast cache can impact performance at the 25% level for the important inversion problem. However, if M were somewhat better (say, M=2) then the impact of cache would be much less.

Cache can cost in one subtle and two obvious ways: It increases the cost of the board; it vastly increases complexity and debugging effort needed; and there is a strong tendency for the presence of cache to cause the bandwidth to main memory to be diminished. (For example, on two processor boards made at FNAL using similar technology and engineering effort, the one with cache has about 2/3 the bandwidth to main memory.) If putting in an extremely fast cache causes the main memory bandwidth to drop from M=3 to M=4, then the net effect is to <u>slow down</u> performance by 15%.

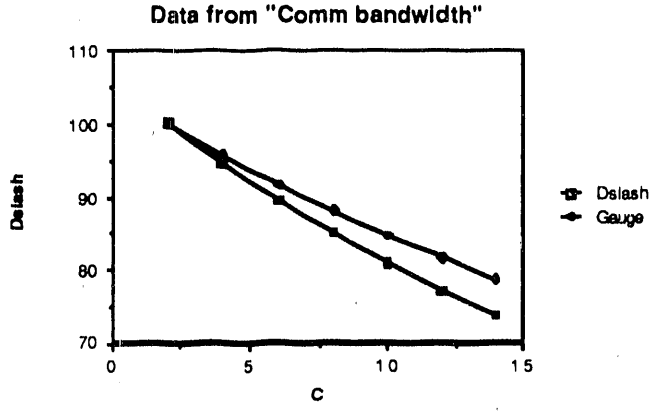
The number of registers has an effect on performance, especially if there are fewer than about 32 registers. In the specific case of propagator inversion, about 24 or more additional registers would have substantial effect — this comes from the "accumulate $\not{D}$" step, which is can be done without much cost if the accumulated $\not{D}$ can be kept in registers.



This shows the effect of the number of registers, using M=2 (the effect is half as much if M=1). External registers do not have much impact here (unless memory bandwidth is quite restricted or the number of registers is small); their primary utility may be to provide a way to buffer data from memory going to multiple floating point units.

Communications Bandwidth; Overhead; Shared Channels

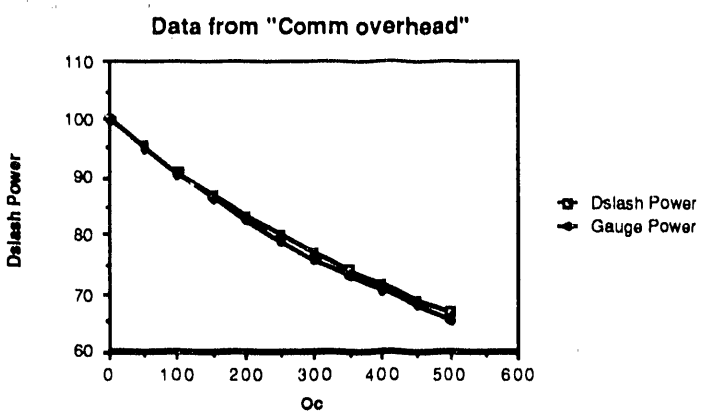
Another potentially important parameter is how rapidly communications can be done (how much time it takes to access data "belonging to" different processors). The nature of the communication (nearest neighbor only; barrel switch; reconfigurable switch; transparent global) will impact how the system can be used, but has not much effect on performance for the problems looked at, where most of the communication is between neighboring sites. The machine architecture matters in an obvious way — if there is shared memory, such that the memory bandwidth M is pegged to the communications bandwidth C, then communications requirements are much higher because M is very important. Beyond that, however, these results are insensitive to architecture.



This graph shows the effect of communications bandwidth limitations, assuming a 4**4 grid chunk in each node (S/V = 2 — appropriate for dynamic

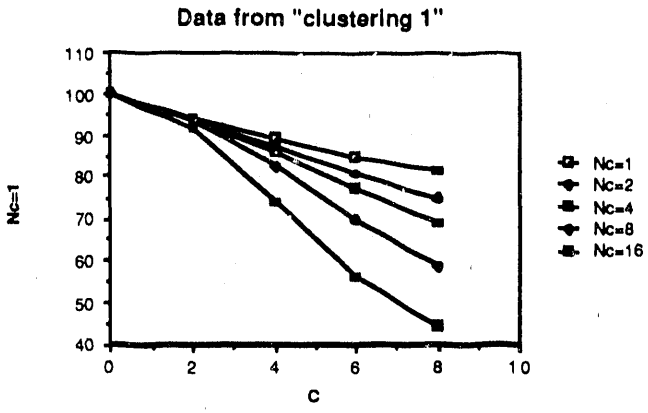
fermions). The inversion problem is slightly more communications intensive than link updating.

Communications overhead O_c (defined as the number of cycles required to send or get one word of data from another processor, minus the time taken per word transferred) tends to be small or zero for lockstep or shared memory architectures, and can be quite large for systems which require operating system assistance for communication. (For example, the current implementations of the iPSC/2 and CM2 machines have adequate communications bandwidth, but very large communications overhead.) Communications overhead can for certain algorithms be mitigated by gathering together the fields that need to be transfered, and doing one large transfer. This is a matter of how a machine can be used: the need to bunch communication restricts the class of algorithms available and can distort the way algorithms are coded.



This shows how effective power depends on communications overhead. O_c is influenced by hardware timings, but is typically dominated by the logic operations needed to decide that data is indeed off node, and set up the communications. Fortunately, for processors with reasonable integer power, with a bit of forethought (e.g. not requiring system calls or interrupts) the communications overhead can easily be kept small enough to have little impact on performance.

Clustering of communications is another parameter that can impact power. This involves several processors sharing one communications resource. The processors may be on one board with one communications channel, or in a crate with only one channel to each neighboring crate — there may be multiple levels of clustering. The effect of clustering is not the same as dividing the communications bandwidths by the number of nodes sharing a resource, for three reasons: Many communications are strictly intracluster; the chunk size for a cluster is larger (S/V is smaller) than for a single node; and the issue of queueing for the shared resource. The queueing issue is important, and is complicated in the intermediate range where there is significant contention but the problem is not completely intercluster communications dominated. The issues of surface/volume ratio and contention are examined in appendix D.



This graph shows how the number of nodes in a cluster affects performance. Obviously, this is critically dependant on the communications bandwidth (the intercluster bandwidth is assumed to be the same as the intracluster bandwidths here). We can see that for up to 4 clustered processors, this effect is small; for 8 or 16 processors, it becomes important to keep the intercluster bandwidth high.

Integer Power

The roles of integer operations in lattice algorithms include:

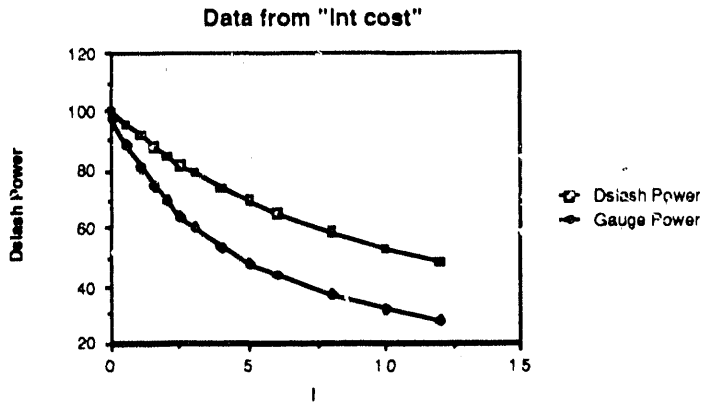
1 - Supplying addresses for data access. This functionality could in principle be accomplished by DMA devices, but they would have to be fairly sophisticated since many algorithms require non-trivial patters of data access to run efficiently.
2 - Calculation of locations of desired data elements. When done properly, this is largely a matter of tracking down pointers.
3 - "Bookkeeping" operations such as looping over sites.
4 - Integer arithmetic in support of such activities as random number generation, table lookups and interpolation, and computation of transcendental functions.
5 - Decision logic required by the algorithms.
6 - Support of communication protocol — establishing channels and perhaps moving the data.

Because the kernels of many algorithms, when run on conventional computers, are overwhelmingly dominated by floating point activities, there is a tendency to underestimate the importance of integer power. There are some techniques which compensate for lack of integer capability. One can vectorize the problem, ordering operations and placing data so as to allow special addressing hardware to handle the addressing calculation. One can avoid algorithms that require significant decision logic or integer support. For SIMD machines, a centralized fast integer unit can handle the requirements for some algorithms. In general, the price for inadequate integer capability is paid in restrictions on how the machine can be used.

There is some ambiguity about what is meant by an "integer operation". The useful work done per instruction varies greatly with machine architecture,

compiler efficiency, and other hard-to-quantify features. (This is why naive comparisons between two computers are often fuzzy at the order of magnitude level.) The yardstick we use to measure integer power is the Vax (780) equivalent: I = 5 would mean the equivalent of a Vax instruction executed every 5 cycles.

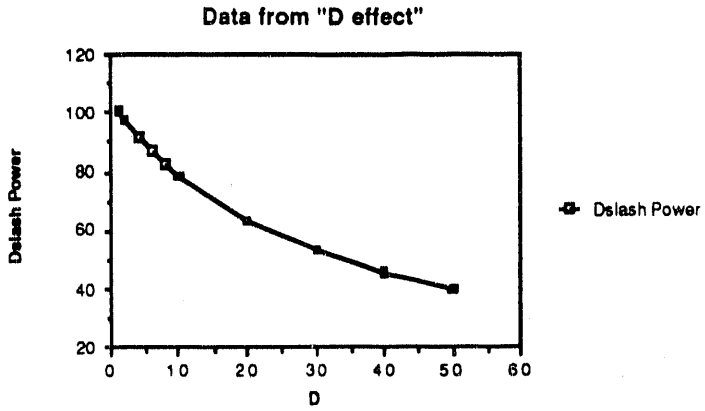
Aside from the power of the integer unit used, one parameter that can be controlled which influence integer power is memory bandwidth and latency. Another way to help the integer unit is to provide special hardware support for the common sequential address generation — in fact, we assume throughout that in simple cases address generation can keep up with the memory bandwidth. (If that is not so, then the effective value of M used must be increased.)



Integer power impacts the gauge configuration computation more heavily than propagator inversion. A value of I=4, which does not excessively degrade performance, corresponds to a 10-Vax integer unit on a 40 MHz processor, not at all a difficult achievement.

Double Precision Speed

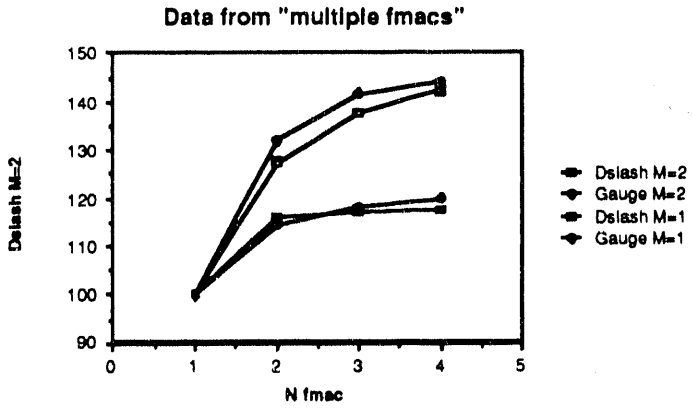
The issue of which portions of which algorithms require extended accuracy is still being explored. Because the newer floating point units tend to have 64-bit capabilities, at speeds such that the increased memory bandwidth limitations are more important than the double precision floating point speed restrictions, the issue of speed here may be moot. Of the two algorithms being studied, it seems that extended precision may be useful in one phase (the dot products) of the inversion problem.



What we see is that virtually any hardware double precision speed will be fine, but that doing things in software (or in a slower extended precision coprocessor) can degrade performance badly.

Multiple Floating Point Units

It is possible to connect more than one floating point unit in tandem, controlled by a single instruction stream and sharing memory and communications capabilities. The design complexity may increase, and one might require external registers so that both FPU's can share data fetched from memory; the software effort to utilize both processors may be large; but if the power boost is sufficient, all that might be worthwhile. We have examined the impact of multiple FPU's, keeping the bandwidth to memory fixed (adding an extra FPU will certainly not make it easier to have quicker memory access).



The qualitative features here are easy to comprehend: Extra floating point multiply/accumulate capability has a greater impact if memory bandwidth is high (M=1); the second FPU has a greater impact than subsequent units. Quantitatively, we see that a second FPU can improve performance by 25% in high-bandwidth systems. We also see that any floating point units beyond the level where each unit can access a word every four cycles, is completely wasted.

Summary of Effects of Machine Parameters

This section is a synopsis of the effects of various values of the parameters examined, on the performance of the system for a particular problem. The propagator inversion problem was chosen — computations of that sort are likely to occupy the bulk of machine time.

Percentages quoted here are comparisons to what power could be achieved for the best possible value of the parameter being varied, fixing other parameters at reasonable values. The boldfaced values are the "minimum acceptable" level of parameters, before the effects become overly large. Specific values in parentheses are examples pegged to a 40 MHz processor.

<u>Bandwidth to Memory (M)</u>

M = 1 — 87%;
M = 2 — 68%; (80 Mbytes/sec)
M = 3 — 55%;
M = 4 — 45%;

These results are not very architecture dependant; but are highly problem dependant. Pure gauge is less memory intensive: M=2 gives 94%, M=4 gives 70%. Because memory bandwidth is costly, systems will likely accept M=2; the perfect M=.5 is probably not feasible at all.

<u>Communications Bandwidth (C)</u>

C = 2 — 100%;
C = 4 — 94%;
C = 8 — 85%; (20 Mbytes/sec)
C = 12 — 77%;

These results are of course architecture dependant; for example, if memory access is pegged to communication speed (as in shared memory machines) M moves with C. The inversion problem is about twice as communications intensive as the pure gauge problem.

<u>Communications Clustering</u>

Nc = 2 — 93%;
Nc = 4 — 85%; (4 processors per board)
Nc = 8 — 73%;
Nc = 16 — 54%;

This makes several assumptions: The chunks handled by the clustered CPU's are formed into a brick (rather than a hyperplane), and the processors share communications resources in a reasonable way. The intercluster communications bandwidth is assumed to be slow (C=8); if C were 4, then even a 16 processor cluster can share communications with 82% efficiency.

Communications Overhead (Oc)	**Oc = 100 — 91% (2.5 μsec)** Oc = 200 — 83% Oc = 300 — 77% This is the time to send or read "zero words" of data from a neighbor; in some architectures, it is zero. In other cases, Oc may be heavily dependant on integer speed.
Multiple Floating Point Units	**Nf = 1 — 100%;** Nf = 2 — 126%; Nf = 3 — 136%; Nf = 4 — 140%; This assumes high memory bandwidth (1 word per cycle). If M = 2 instead, the potential improvement for multiple FPU's is halved. These numbers also assumes the software effort to take full advantage of the extra unit, without being affected by overheads.
Integer Power (I)	I = 2 — 92%; **I = 4 — 85%; (10 Vax power)** I = 6 — 79%; Integer power (decision logic and addressing capabilities) is more than twice as important for the pure gauge case. Substantially less reliance on integer power can be achieved at a high cost in program flexibility and coding effort.
Data Cache Access Bandwidth (Mc)	Mc = .5 — 94%; Mc = 1 — 89%; (160 Mbytes/sec) Mc = 2 — 74%; These numbers assume a low memory bandwidth (M=4). The savings due to cache go away as main memory bandwidth becomes reasonable. Also, the cache effect is much smaller for pure gauge. The cache size needed to achieve savings is roughly enough to hold a few "lines" of data.
Number of Registers (Nr)	Nr = 56 — 100%; **Nr = 32 — 83%;** Nr = 24 — 78%; Nr = 16 — 73%; The pure gauge problem is about 2.5 times less sensitive to the number of floating point registers available.

Double Precision Speed (D)

D = 2 — 97%;
D = 4 — 92%; (20 Mflops/sec 64-bit)
D = 8 — 82%;
D = 20 — 63%;

There is one part of the inversion for which it seems that double precision is advantageous. Note that almost any hardware double-precision will be fine (it can be 4 times as slow as 32-bit) but that software extended precision quickly becomes painful.

These impacts are not additive — when one factor forces a performance degradation, the requirements on other factors ease up a bit. We find that using these realistic minimal acceptable values for all the machine parameters only degrades speed by 35%, relative to the performance with high memory bandwidth, integer power, etc.

As an example, consider a board with a cluster of four 80 Mflop (peak) processors. If the memory bandwidth, communications speeds, integer power, etc. were all very high, each processor would achieve, on actual physics problems and without Herculean coding efforts, about 24 Mflops per processor. Choosing instead the "minimal acceptable" values (in boldface above) — 80 Mbytes/sec bandwidth to memory, 20 Mbytes/sec communications bandwidth, with 2.5 μsec overhead; 10 Vax equivalents of integer power, etc. — the effective power would decrease to 16 Mflops per processor.

These numbers may be disappointing, but it is a fact of life that the actual performance of a system is not the "machoflop rate" (the speed if the problem could be selected to maximally use the processor). It is not "God's megaflops (the speed arrived at if you count all the necessary operations in the actual problem, and assume that miraculously everything overlaps perfectly); nor is it "superman's megaflops" (the best possible speed, assuming infinite programming effort is available). Herculean efforts involving customizing every routine and interface will also be rare. The best one can expect is careful, modular optimization of all routines which run for significant times on the system.

Memory Size Requirements

It is harder to pin down quantitative requirements on the size of main memory in a system. Nonetheless, it is important to know what we can about this, because a large fraction of the cost of a system is the cost of memory. In cases where the users are completely unsure of how much memory and power will be needed ("the more the better"), the "cost rule" is reasonable — spend half your money on memory. We must try to do better than this, because memory requirements may influence design decisions such as the choice of SRAM vs DRAM. If the wrong selection is made here, and the cost rule is followed, then the system utility can be diminished by a large factor.

The first hope would be to set a natural physics scale for the memory. For example, if we could honestly say that 64**4 will be fine, and going to larger lattices going past 64**4 doesn't help any more, then the total memory size needed could be fixed as being no greater than 36 Gigaflops. Today's estimates for this "natural" physics scale range from 64**4 to 128**4 (and up to NEVER). Unfortunately, the feeling for this number always seems to be just one or two orders of magnitude beyond what we can study at the time. 128**4 *might* really be as big as you would ever want, but we won't know till we can get past there.

It is likely that for systems to be designed in the near future, neither CPU power nor memory will be totally adequate for all the physics one would like to do. Since, for a fixed system cost, speed and size can be traded off (to some extent), a study of memory needs is dominated by trying to determine the proper memory to power ratio. This can be expressed in Mbytes/Mflops, where a Mflops is one million peak floating point operations per second, although strictly speaking, the important power measure is **effective** Mflops.

The memory to power ratio is highly problem dependant, and optimizing it requires physics input. We must have an idea of what sort of algorithms are best to use (to see how much memory is needed per site). It would be nice to know how big a lattice is needed for interesting and correct physics results. Since we won't know this very well until the physics has been done on large lattices, we must make do with estimates of how many sweeps are needed to get valid results — then for a fixed amount of CPU power, one can see how large a lattice could be done in a reasonable time, and determine memory needs accordingly.

Based on experience with current computers, we have learned (with some degree of confidence) certain things about these requirements. Other numbers are much less firm, either because they depend critically on the choice of algorithm and progress is being made in that direction, or because they are sensitive to issues which can only be learned by doing the physics on larger lattices. We will first discuss what we know, then present best estimates for other quantities that are necessary to get a feel for memory to power ratios.

The number of bytes of data memory needed per site is easily calculated for any given algorithm; moreover, we have some idea of what this number is for algorithms which will likely be used, and the uncertainty in this number is fairly small. Almost independent of algorithm choice, there will either be some

pointers defining the connectivity of the lattice, or lists used to avoid excessive integer computation in computing locations of neighboring fields — this amounts to under 100 bytes per site. For the pure gauge field, there are four links per site — 288 bytes (one could save only two rows, but this would have a severe impact on the performance of many algorithms). The quark field is more subtle. At a minimum, one needs to be able to find one component of the propagator, while the gauge fields are still present (this result can then be sent to a distributed disk system for future use). Inversion methods like conjugate-gradient and minimum-residual typically require 3 - 5 copies of quark fields present — roughly up to 500 bytes. Thus, we need at about 900 bytes per site.

A class of methods which will require somewhat more storage is the set of molecular dynamics related algorithms. What these algorithms, which include the promising Hybrid Monte Carlo method, have in common is the need to remember values of "field momenta" or fields from previous steps. Typical first or second order methods require one or two extra copies of each field, amounting to up to 700 additional bytes per site. Another circumstance in which additional memory would be needed is if all the components of propagators were needed at one time. That would be the case if there was no usable disk system. The additional memory per site is roughly 1100 bytes (the "momentum fields" for molecular dynamics methods can share these extra bytes). At first glance it would seem that the physics analysis does need all the components of the propagator field at one time (worse yet, of two or three propagators with different mass values). However, the analysis can normally be done one time slice at a time, with only 1 - 3 time slices in memory at any instant. Under those circumstances, the memory needs for analysis are no greater than those for configuration generation. Thus the anticipated lattice size dependant memory needed is

$$\boxed{\text{900 - 1600 bytes/site}}$$

There is some amount of data memory needed independent of lattice size. This includes data structures describing the lattices in general and sets of sites and fields on the lattice; local variables and intermediate storage for use during the computations at each site; and memory in support of whatever operating system is running. (On machines with distinct integer units not coordinated with the floating point processors, this can be in a different, probably slower, memory.) We find for the ACPMAPS system that this "overhead" memory amounts to a few hundred kilobytes per processor, but it can probably be kept to under 100K bytes. The minimal extra overhead is then

$$\boxed{\text{100 Kbytes/processor}}$$

The instruction memory needed is obviously driven by the most complex algorithms that will be desired. It would be a disaster not to be able to run the codes you want because of instruction memory limitation; just as bad is the need to "overlay" problems or worry about breaking things up into multiple jobs. Fortunately, we can get a good idea of how large these codes can be once we have factored out all the grid/connectivity/bookkeeping work. This is what the libraries for CANOPY do — the CANOPY routines linked into most lattice codes amount to

over 200K bytes on ACPMAPS (but more like 100K on machines where half the instruction line is not occupied with floating point nops). To this can be added under 100 Kbytes for the operating system. The remainder of the problem code is what varies with algorithm complexity. This ranges from about 20 Kbytes up to 100 Kbytes for problems done to date. The coding complexity of algorithms is probably worst for large analysis codes, where several different types of analysis will be combined, to lessen the need for multiple passes through data kept on tape or disk.

It seems safe to assume that the library routines will at most double, and that the code complexities will become no worse than five times what they are today. In that case, if the instruction memory is shared with data memory, we need

$$\boxed{\text{800 Kbytes for instructions}}$$

Because running out of instruction room is so disastrous, if there is a distinct instruction memory, we would like to provide more than a factor of two cushioning:

$$\boxed{\text{2 Mbytes for separate instruction memory}}$$

We know some things about lattice size requirements. Attempts at doing interesting physics have shown that 16 (and probably 24) sites on a side is inadequate — either finite size or lattice spacing effects are significant. There is some hope that 32**4 sites (or 32**3 volume by a larger number of time slices) will be better; we would have to go a bit further to be somewhat convinced that size dependance is dying off. Thus we need to be able to work with at least two million sites. What we would really like to be able to study is 64**3 volumes and even longer in the time direction (24,000,000 sites) — perhaps only for quenched fermions. So, not including overheads described above

$$\boxed{\text{total memory}\left\{\begin{array}{c}\text{need at least 3.2 Gbytes}\\ \text{desire at least 21.2 Gbytes}\end{array}\right\}\text{plus overhead}}$$

These are absolute minimum requirements for advancing physics beyond the realm explored to date. Even the larger number is easy to attain for systems with many processors.

We also know something about the power needed for doing quenched calculations. This is made up of comparable times spent on gauge Monte-Carlo, and inversion to find propagators. (A limited amount of physics can be done without propagators.) The time needed to do about 1000 Monte-Carlo sweeps (to get a non-correlated configuration) is about 15 million cycles per site; the time for LU decomposition inversion for each of 3 mass values and each of 12 spin-color components (three times faster than without preconditioning) is about 20 million cycles. These times (especially the inversion time) may improve, but because they are balanced, both have to improve a lot to make a big difference in total cycles needed. If a physicist wanted to do a serious examination of quenched QCD on a big machine, this would require exploring about ten values of (beta, size). For each value, you would like at least 100 decorrelated points; and the study should take no longer than 6 months. This gives a value for the memory supportable by a given power for a high-statistics study of quenched QCD:

$$\frac{\text{1 cycle}}{\text{2 flops}} \bullet \frac{\text{1 decorellated point}}{\text{35M cycles/site}} \bullet \frac{\text{900 bytes}}{\text{site}} \bullet \frac{\text{1 value}}{\text{100 points}} \bullet \frac{\text{1 study}}{\text{10 values}} \bullet \frac{\text{15M sec}}{\text{study}} =$$

$$\boxed{\text{0.2 Mbytes/Mflops needed for high-statistics quenched QCD}}$$

This is an absolutely minimum appropriate memory size for quenched physics. At least five effects make it desirable to have more memory per unit power:

- It is often possible to trade memory for speed. For example, and alternative scheme of storing pointers to the link fields needed in environment calculations would save 10 - 20% in time, at the cost of 600 extra bytes per site.

- Frequently, one would like to explore either memory-intensive algorithms on reasonable lattices, or the behavior of proposed methods on really large lattices. Sometimes these low statistics studies are done to investigate algorithms; at other times, physics can be extracted. These studies require much more memory — one or two orders of magnitude. For example, just to double the number of points in all directions needs an order of magnitude increase in memory size.

- For a given machine size, the memory is a hard limitation; the power limitation can be worked around in important cases merely by tolerating longer running times.

- Although we have assumed the system scales, making memory/power ratio the important factor, this scaling does not go on forever. If there is a limit on how many processors can be in a system, you cannot indefinitely increase the size by adding memory and power together. For example, it may be possible to work with 4,000 32 Mbyte processors, but not 16,000 8 Mbyte processors.

- Some programs in the analysis stage may require more bytes per site than the corresponding program to generate the configurations and propagators. For example, if a particular smearing method requires the entire propagator (all time slices) for each mass value to be present at once, that could quintuple the needed memory.

A couple of interesting points also indicate that large memories are desirable. As long as a sufficient number of decorrelated lattices, very large lattices provide extra statistics compared to smaller volumes (the fields on one side may decorrelate with those on the other side). So it is not the case that power needed to do high-statistics physics scales with volume — it may level off at some point. Also, major breakthroughs (for example, cluster-like algorithms) are not ruled out; these tend to vastly increase the appropriate memory/power ratio. This is even more relevant to dynamic QCD.

Experience with the ACPMAPS system corroborates these observations: Physics can just be done comfortably with 0.4 Mbytes/Mflops, but low-statistics large-lattice or memory-intensive investigations are hampered.

Hard facts concerning memory needs for dynamic QCD are more difficult to come by. This is because of there is much greater uncertainty in the nature of algorithms that will be appropriate, and in the speed with which these algorithms produce decorrelated data points. However, certain obvious constraints can be stated.

Firstly, dynamic QCD will (probably) never be quicker than quenched calculations. (It is conceivable that the presence of quark loops damps some critical point effect in a big enough way to overcome the extra time taken, but that seems prohibitively unlikely.) Therefore, the memory to power ratio required for dynamic QCD tends to be smaller than that for quenched physics.

Secondly, the absolute size limitations still hold. On the low end, if you can't fit a large enough lattice to do interesting physics, decent statistics will do no good. On the high end, a machine large enough to do dynamic QCD on 64**3 by 96 lattices will probably be satisfactory for a lot of physics, although there may be some reason why bigger lattices are interesting.

at least 3.2 Gbytes needed even for dynamic QCD

Finally, because algorithm investigation is for dynamic QCD is an extremely important endeavor, quite a bit of work will be low statistics studies on lattices which are larger than would be feasible (for good statistics) based on power.

As to the estimated appropriate memory/power ratio for high-statistics dynamic QCD, we can proceed as follows: Today's best dynamic methods (e.g. hybrid Monte Carlo) take between a factor of 100 and 1000 more time than is used for quenched calculations on the same size of lattice. (The factor depends both on the algorithm and on the values of beta and quark masses used.) If no further algorithm improvements occur, this would give quite a small memory/power ratio; the memory needs are controlled by the desire to have at least as much site data memory available as overhead — this means memories of 250K (or 1-2 Megabytes if instruction memory shares the same space) would be appropriate. However, in the absence of algorithm improvements, the important work will be on algorithm development, which needs (as mentioned above) at least an order of magnitude more memory.

Recent improvements in propagator computation have increased speed by a factor of 3; these improvements have not yet been applied to dynamic QCD on reasonable lattices. We can suppose that the same sort of speedup will occur there — a factor of between 1 and 10. It is more speculative to try to estimate the remaining room for improvement; we will guess that it is the same amount: between no further progress, and another order of magnitude. However, we will not include this last factor in our estimates when we also include the increased memory to do algorithm exploration; you don't need both the last improvement factor and the exploration room. This will give a conservative estimate of the memory/power ratio appropriate for dynamic QCD.

Putting together the estimated numbers, we get that the memory required for dynamic QCD is that for quenched QCD, multiplied by the increase in bytes/site needed (a factor of about two) and by:

$$\frac{10^{.5 \pm .5} \text{ improvement in algorithm} \bullet 10^{1.5 \pm .5} \text{ for further algorithm exploration}}{10^{2.5 \pm .5} \text{ times more power needed for dynamic QCD}}$$

This means that the appropriate memory size for doing dynamic QCD is half of that for quenched physics — there is an uncertainty here of an order of magnitude.

$$\boxed{\text{0.1 Mbytes/Mflops needed for dynamic QCD}}$$

The memory to processing power ratio needed for high statistics dynamic QCD may be about ten times less than that (because the improvement factor is likely to be smaller than the extra memory needed for algorithm exploration); uncertainties are large. To allow for the possibility that the analysis phase will require more bytes per site, we will estimate that high statistics dynamic QCD needs three times less memory

$$\boxed{\text{.03 Mbytes/Mflops needed for high statistics dynamic QCD}}$$

To summarize appropriate memory/power ratios are:

0.2 — 6.0 Mbytes/Mflops for quenched QCD with algorithm exploration

0.2 — 0.6 Mbytes/Mflops for high statistics quenched QCD and analysis

0.01 - 1.0 Mbytes/Mflops for dynamic QCD with algorithm exploration

0.003 - 0.3 Mbytes/Mflops for high statistics dynamic QCD

To put this in perspective, it means that for a 100 Gflop machine, you might have 30 Gbytes of memory to be confident that high statistics physics can be done, and to provide for most dynamic algorithm exploration. For a true Teraflop machine, you might wish to stop at 100 Gbytes, which would comfortably allow for algorithm exploration and support high-statistics quenched physics on 100**4 lattices.

Appendix A — Parameters Studied

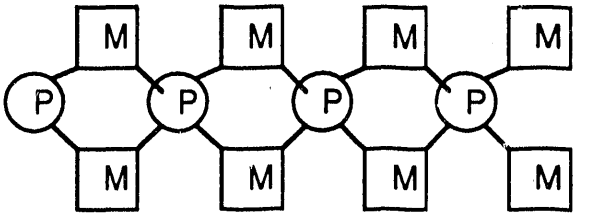
To parameterize the nature of a machine architecture (for the purpose of evaluating its effective power), we adapt the following conventions: Since for fixed numbers of cycles for every operation the power scales completely with cycle rate, we express everything in terms of number of cycles taken for a particular operation. (This need not be an integer, and can easily be less than one; and in some circumstances is a statistical average number.) Clock cycle rate is somewhat arbitrary; we ordinarily take our 'cycle' to be the time needed for one floating point multiply and accumulate operation, since many proposed architectures are oriented in that way. In the analysis of how long various operations will take, we come across several parameters — each is the time taken to do something (so that a lower value for a parameter means more power).

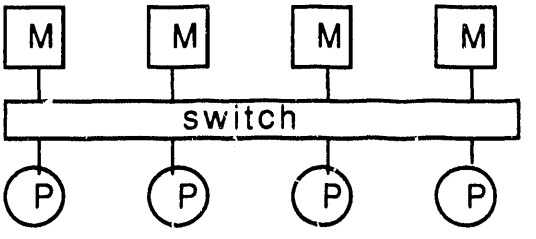
- F — A floating point multiply and accumulate (or sometimes a multiply and an add). Normally taken to be 1.
- f — A floating point multiply or add. Often the same as F; smaller the architecture is not multiply/accumulate.
- D — Double precision operation.
- M — Load a 32-bit word from memory (actually, a usual floating point word, 64-bits if 32-bit operations are not typical). Typically will range from .5 to around 4.
- M_c — Load a 32-bit word from a fairly large cache memory — same as M if there is non cache.
- M_e — Load a 32-bit word from an external register — same as M_c if there are none.
- S — Store a 32-bit word to memory.
- S_e — Store a 32-bit word to an external register.
- O_c — Overhead to establish communication to another (neighboring) processor. In some SIMD architectures, this can be zero cycles; it ranges up to hundreds or thousands.
- C — Communicate one 32-bit word to or from another processor (as part of a block; the overhead is covered by O_c).
- I — Integer operation. This is normalized to Vax Mips — a 1 Mhz processor with I = 1 would match one Vax (780) in integer power. Many factors go into I, including memory speed and compiler efficiency; typical RISC machines require 2-3 actual cycles to do one 'integer operation' by this definition.

Another parameter which comes into consideration is controlled not by the machine architecture but by how big a chunk of the lattice is handled by each processor. This is surface/volume ratio, S/V, and will typically be around 1 - 2. Details of S/V considerations are presented in appendix D.

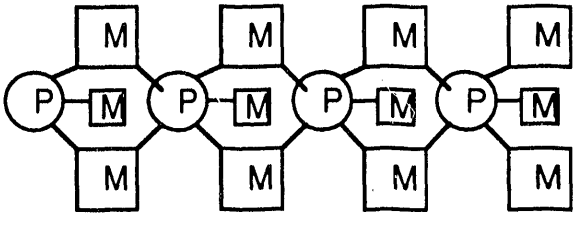
Appendix B — Architectures Examined

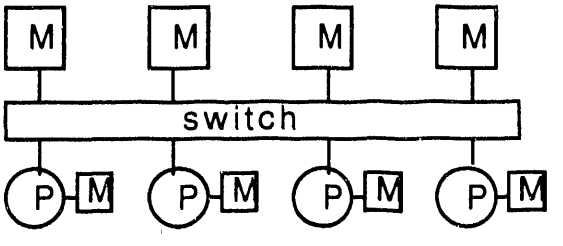
1 - Shared Memory — Communications tied to memory. Every memory access takes the same time, whether or not communications is involved. The architecture can look like a grid or a switch:





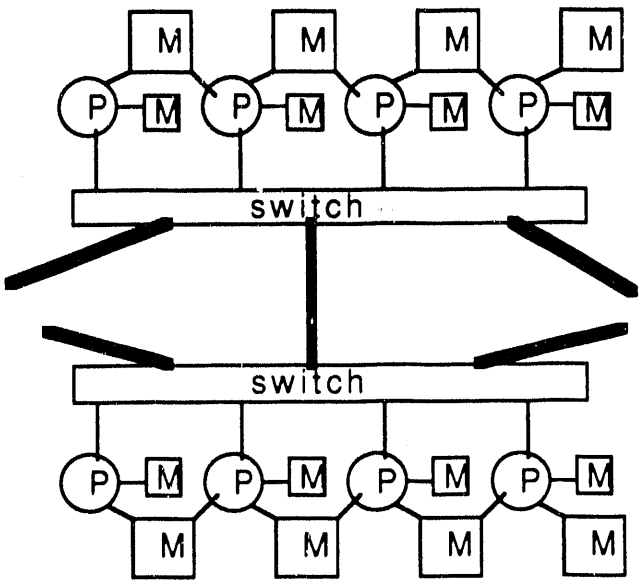
2 - Lockstep: Shared Memory with Explicit Local Caches — Communications tied to certain memory accesses. Accesses which are known to be available locally are quicker. The same architectures apply, with some additional small, fast memory attached to each processor:





3 - Lockstep Clusters — Lockstep communication, but with shared paths between nodes of groups of nodes. For example, there may be several nodes on a single board, which has a communications path (or paths) to other clusters.

Obviously, there may be multiple levels of clustering: nodes on a board, boards in a crate, etc. Assumedly, this shared path is not fast enough to match the intra-cluster bandwidths; it is a potential bottleneck. For these architectures, the details of the size and shape of the portion of the grid on each cluster is important. Possible architectures include having switches or fixed interconnections between clusters and within each cluster:



4 - MIMD — Shared communication paths, with or without attempts to minimize contention. There can, of course, be levels of clustering which will effect communications bottleneck computations.

Appendix C: Detailed Breakdown of Anticipated Timings

Propagators and Dynamic Quarks — Dslash

The object is the computation of the $\not{D}$ operator being inverted when quark propagator calculations are being done. This is done once per conjugate-gradient sweep for each site. Quark inversion represents about half the work for quenched calculations, and probably 80% - 99% for dynamic QCD. The quantity needed for each site x is

$$\psi(x) - k\left\{\sum_{\hat{\mu}} (1-\gamma_\mu)\, U_{\hat{\mu}}(x)\, \psi(x+\hat{\mu}) + \sum_{\hat{\mu}} (1+\gamma_\mu)\, U^\dagger_{\hat{\mu}}(x-\hat{\mu})\, \psi(x-\hat{\mu})\right\}$$

where k is the hopping parameter (related to the bare quark mass). Thus for each of eight directions, one must accumulate an expression of the form $U\gamma_\mu\psi$. The quark field ψ is a set of four color-vectors (complex 3-vectors), but the multiplication of ψ by γ_μ does not imply a multiplying a 4 x 3 and a 3 x 3 complex matrix, because γ_μ consists of 1's and i's, one element in each row. Naively, multiplying ψ by U does involve multiplying a 4 x 3 complex matrix by a complex matrix (144 multiply/add cycles). However, there is a technique (the "Draper trick") to reduce the requirement to multiplying 2 x 3 and a 3 x 3 complex matrices, saving half the arithmetic. The trick relies on the fact that $1 \pm \gamma_\mu$ either has two zero rows, or has rows 3 and 4 trivially dependant on rows 1 and 2. Thus, it suffices to combine the top and bottom halves of ψ, and multiply U by the resulting 2 x 3 matrix. For example, in a representation where γ_x is given by

$$\begin{matrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{matrix}$$

to compute $(1+\gamma_x)\psi$ one need only use $(\psi_1+\psi_4)$ and $(\psi_2+\psi_3)$.

So, the efficient computation of the $\not{D}$ operator can be broken into four phases:

- Phase 0 — Locating all the necessary ψ and U fields, and loading any off-node data needed.

- Phase 1 — Combining the four rows of into two color vectors, incorporating the appropriate i and signs (or in one case, multiplying two rows by 2, since the other two components of are $(1+\gamma_\mu)\psi$ zero).

- Phase 2 — Multiplying the combined color vectors by the gauge field U. This is the step that has been shortened by the trick, but is still the dominant step.

- Phase 3 — Accumulating the result of phase 2. At the end, this result can be multiplied by k and added to $\psi(x)$. Alternatively, one can multiply by k and accumulate immediately.

We will calculate the time $T[\not{D}]$ needed for accumulating all eight components (one entire site). Note that phases 2 and 3 may be combined, if there are enough registers or sufficient bandwidth to memory. Phases 1 and 2 could in principle be combined, but in practice, to gain from this would require extra memory bandwidth, substantial extra integer power (to generate the complicated addressing pattern needed), and a lot of extra coding work — we assume this won't happen. Phase 0 might be combined with phase 1, but this would involve inserting new memory operations into a very ubiquitous software tool (getting pointers to fields), and would probably not be attempted.

A total of 1260 floating point operations is required to compute $\not{D}$ in this way.

Phase 0: There is a computational part and a communications part. The computational part is mainly pointer chasing. Basically, you are offsetting the home site's pointer list with the direction (shifted appropriately) to get a pointer to the site desired, looking up and adding the field offset corresponding to the selected field, and checking that the node looked up is indeed the local node. Including the subroutine calling overhead, this amounts to about 8 integer instructions for each field; there are 9 ψ (including the home site) and only 5 U (the four gauge links on the home site are known to be bunched together) field pointers needed. We will also add in the overhead for moving from one site to the next in a task (10 instructions). The time taken for the local portion of phase 0 is thus 122 I. [One can picture some macho arrangement of sites and fields in memory such that this is cut by a bit, computing field locations by shifting and adding appropriately. Aside from the fact that this limits the physics you can do, and has to be reconsidered for each new problem, it is also observed that the pointer chasing method actually is faster in practice.] In this phase, it turns out that only about 15% of the data accesses are inherently uncachable — at any rate, these memory cycles are factored in to the effective integer speed I.

The communications part of phase 0 depends critically on two things: how many fields must be brought in from off node, and how long it takes to load a field. The latter is sensitive to communications overhead and bandwidth, and to just how the communications are synchronized. That is, the bandwidth requirements may be much less if nodes are not all attempting to communicate simultaneously. Several models will be considered here: (1) A MIMD system, with communication needs eventually being randomly distributed in time. (2) A system with a large configurable switch, employing simultaneous

communication over the switch. (3) A system with lockstep local communication, with a group of processors (perhaps on one board) sharing a channel to neighboring boards. (4) A system with lockstep local communication, with each processor's bandwidth unaffected by other activity.

The number of fields needed from other nodes depends on how the sites are distributed among the nodes. It ranges from 6 fields needed (4 quarks and 2 links) for the "pancake" case (e.g. a 64*64 plane in each node) to an average of two or less for hypercubic chunks (1 for 8**4 chunks in each node). This number is equal to 1.5 S/V, the surface to volume ratio for each processor (because even when a quark must be fetched, there is a 50% chance that the associated U for transporting that quark belongs to the home site). In cases (2) and (4), the communications time adds to the rest of the time – the time taken is S/V (1.5 O_c + 33C). In cases (1) and (3), the time taken is the same, but there is the potential for a bottleneck in communications — if N nodes share a bottleneck, N S/V (33C) cycles of communication must happen. This is discussed at length in the section on communications saturation (appendix D). In case (1), this communication is distributed (to a good approximation) along the entire computation time, but in case (3), it must all be part of phase 0. Because of this, for lockstep shared communication, one must multiply C by the number of nodes haring the bottleneck, while for MIMD shared communication, one multiplies C by a contention factor equal to 1/(1-saturation), where the saturation is the total N node communication cycle time divided by the time for the entire Dslash calculation.

Phase 0 thus takes $\boxed{122\,I + S/V\,(1.5\,O_c + 33C)}$ cycles.

<u>Phase 1:</u> This is pulling four color vectors in and adding them in pairs to form two color vectors. (In one case out of four, it instead pulls in two vectors and multiplies by 2; we will ignore this small deviation.) The integer overhead involved is pretty trivial (two primary addressing operations, then DMA). The memory requirements, which are likely to dominate, amount to 24 M. The floating point requirements are 12 adds (12 f). This is done for each of eight directions. (Note -- we assume that there are sufficient registers to hold the two resulting color vectors. If there are fewer than about 16 registers, this and phase 2 will change, requiring many more memory operations.)

The memory accesses here are largely uncachable. Obviously, the fields from the site before home in the most rapidly changing dimension (or two dimensions if the cache is very large) can hit cache; depending on details of the communication mechanism, so can any fields that had to be fetch from neighbors. Still, between 50% and 87% of these accesses are inherently "fresh" data. We will assume a 75% miss rate.

Phase 1 thus takes $\boxed{\mathrm{Max}(144\,M + 48\,M_c,\ 96\,f)}$ cycles.

<u>Phase 2:</u> This is the multiplication by U — 72 multiply/add pairs (72 F). During this time, U must be pulled in. Assuming the two color vectors remain in registers, each row of U need only be loaded once. This phase requires 24 - 30 registers to accomplish in this optimal way. If there were only 16 registers, then

you could do the loading in the same manner, but each component of the answer would have to be stored back, and the pipe overheads would be felt more strongly. We assume at least 32 registers. This phase is done eight times. Again, most of the memory accesses will miss cache.

Phase 2 thus takes $\boxed{\text{Max}(108\ M + 36\ M_c,\ 576\ F)}$ cycles.

Phase 3: Here we have two choices. If the registers are limited, we pull in the accumulated $\psi(x) + k(1+\gamma_\mu)\psi$ terms, multiply or two color vectors produced in phase two by k, add one of them to each of the four accumulation color vectors, and store the new accumulations. If we have sufficient registers to hold the accumulations, (a total of 56 - 64 registers) then the loads and stores go away, except for the first loads and last stores. In principle, the k multiplications can be done at the same time as the adds. In practice, the chip architecture may or may not allow this, but this step may be memory dominated anyway, so we will just assume multiply/add cycles. These are not the sort of "regular" F cycles found in complex multiply and accumulate operations, so architectures with F less than 1 (e.g. two or more fmac units in tandem) will use F=1 here. That is only important if there are lots of registers, so that this step is not memory dominated.

Phase 3 thus takes either $\boxed{\text{Max}(168\ M_e + 168\ S_e + 24\ M + 24\ S,\ 192\ F)}$ cycles,

or with at least 56 registers $\boxed{\text{Max}(24\ M,\ 24\ F) + \text{Max}(24\ S,\ 24\ F) + 144\ F}$ cycles,

but in any case at least 192 cycles.

The total time in cycles for the Dslash operation is then (assuming we are in the range where a memory operation takes at least half as long as a floating point multiply and add, and no more than four times as long)

$$\boxed{576\ M + 576\ F + 122\ I + S/V\ (1.5\ O_c + 33C)}$$

Of the 576 memory cycles in memory-dominated phases, one third are inherently uncacheable loads; one third are stores. 336 cycles (including most of the stores) could be to extended registers or not involve memory at all if there are at least 56 registers — but these do not overlap the other memory operations. Therefore, for this operation, an 80 Mflop unit might be reasonably balanced with 160 Mbytes/sec bandwidth to memory, 16 Mips of integer power, and 20 Mbytes/second of communications bandwidth (giving 15% communication saturation, if S/V is 1). Under those circumstances, the $\not{D}$ operation will take about 45 μsec; this represents 35% of the peak floating point speed, or about 45% of the maximum power in principle possible by overlapping all the integer, communications, memory and floating point operations.

Communications bandwidths between groups of nodes are particularly stressed here, because the number of sites handled by each node would not be large small for such a computing-intensive problem. Assuming decent chunking, a card with 8 nodes might well contain 8*8*8*8 sites; S/V is 1. Making the queueing assumptions discussed previously, a 40 Mbytes/second inter-group

bandwidth would increase the execution time by 18% (the communications bandwidth being 30% saturated); 28 Mbytes/second would cost 42%, and 20 would cost 80%.

Minimal Residual Incomplete LU Decomposition

The operations other than $\not{D}$ necessary for propagator calculation vary widely with what particular algorithm is selected (even the $\not{D}$ computation can vary, but those changes don't alter the mix of operations much, except as mentioned in the next paragraph). Rather than try to study all possible algorithms, we will examine a sample algorithm, choosing one which is currently "state of the art" in the sense that for interesting values of β and k on fairly large lattices it performs much better than most methods, and as well as any. This algorithm is the method of minimal residuals, preconditioned using incomplete LU conditioning.

The steps involved are an ordinary $\not{D}$, followed by special Dslash-like operations acting with the L and then the U operators (these backsolve the conditioning), followed by the miscellaneous additional operations we will consider here. The L and U operations involve the same sort of quark transport, multiply by gamma matrix, and accumulate steps as $\not{D}$, but the accumulations average only half the number of quarks. In addition, needs for synchronization of data affect the performance on those operations; a full analysis would depend heavily on the particulars of the architecture. We will approximate the LU portion as being equivalent to one $\not{D}$, for the purposes of estimating how important the miscellaneous additional operations are.

These additional operations consist of finding a pair of dot products of the form $\psi \bullet \psi$ and $\psi \bullet \phi$ where ψ and ϕ are quark fields, and doing a pair of linear accumulations of the forms $\psi = \psi + \alpha\ \phi$; $\phi = \phi - \alpha\ \omega$, where α is a complex scalar. The dot products can be done together, as can the linear accumulations, but α depends on the results of the dot products. So these miscellaneous operations must be broken into two phases — phases 4 and 5 for the overall propagator inversion.

Phase 4: The dot products require, per site, about 10 integer operations for task overhead, roughly 8 integer operations to locate two fields (which can be clustered together) at the home site, and about 8 integer operations to guide the summing of results across sites, for a total of 26 integer operations. The actual dot product takes 24 multiply/accumulate cycles for $\psi \bullet \psi$ and 48 multiply/accumulates for $\psi \bullet \phi$, and requires 48 non-cacheable memory cycles to load in ψ and ϕ.

There is a potential complication here involving double precision. It seems likely that small errors in these dot products can radically effect the speed of

convergence for the inversion. Since these dot products can be sums of millions of terms, systematic loss of precision is a major concern. An obvious solution would be to do the accumulation in double precision. This does not affect the loading needed (assuming the quarks themselves are still single precision) but it means that the multiply/accumulate (or at least the accumulate part) is done in double precision. It is prudent to assume that this is the case.

Phase 4 thus takes $\boxed{26\ I + \mathrm{Max}\,(48\ M,\ 72\ D)}$ cycles.

Phase 5: The linear accumulations require, per site, about 10 integer operations for task overhead, and 10 operations to locate three fields clustered together at the home site. The actual calculations then require 72 uncachable memory loads to bring in the three quarks (there will surely be sufficient registers to not need to bring ϕ in twice) and 48 stores, as well as 192 multiply/add cycles. (For architectures like the XL-3132, where multiply/accumulate must accumulate with the result of a prior operation, an extra 24 floating point cycles are needed.)

Phase 5 thus takes $\boxed{20\ I + \mathrm{Max}\,(\ 72\ M + 48\ S,\ 192\ F\ )}$ cycles.

The total time in cycles for these miscellaneous operations (which amount to 528 flops) is

$$\boxed{46\ I + \mathrm{Max}\,(48\ M,\ 72\ D\ ) + \mathrm{Max}\,(\ 72\ M + 48\ S,\ 192\ F\ )}$$

Three things to notice are:

- The ratio of memory needs to floating point cycles is, surprisingly, smaller for these calculations than for the $\not{D}$ computation. But when at least 56 registers are available, $\not{D}$ becomes less memory intensive.
- Integer operations are about twice as important in these miscellaneous calculations.
- The Dslash and Dslash-like computations together take roughly 10 times longer than the rest of the calculation.

To summarize, for the entire propagator inversion (representing 1788 flops) the number of cycles required per site is:

field location:	$122\ I + S/V\ (1.5\ O_c + 33C)$
combine color vectors:	$\mathrm{Max}(144\ M + 48\ M_c,\ 96\ f)$
multiply by links:	$\mathrm{Max}(108\ M + 36\ M_c,\ 576\ F)$
accumulate $\not{D}$:	$\mathrm{Max}(168\ M_e + 168\ S_e + 24\ M + 24\ S,\ 192\ F)$
dot products:	$26\ I + \mathrm{Max}\,(48\ M,\ 72\ D\ )$
linear accumulations:	$20\ I + \mathrm{Max}\,(\ 72\ M + 48\ S,\ 192\ F\ )$

Quenched Gauge Configurations — Environment

When evolving the gauge field configuration in the absence of fermion loops, the bulk of the floating point operations are in the calculation of the "environment" with respect to a link being updated. This environment is the sum of six 3-link "staples", so this procedure can be referred to as "staple-sum". While the computation of a staple naively involves two SU(3) multiplications, in fact, it is sufficient to compute only the first two rows of the initial multiplication, and reconstruct the necessary third row of the final product by cross-multiplying the first two rows. This means that the entire process requires 1932 flops, rather than the naive value of 2592.

The efficient computation of the environment can be broken into three phases:

- Phase 0 — Locating all the necessary U fields, and loading any off-node data needed.
- Phase 1 — The SU(3) multiplication of pairs of fields.
- Phase 2 — Accumulating the result of phase 1.

We will calculate the time T[E] needed for accumulating all six staples to get the environment for one link. Note that phases 1 and 2 may be combined, if there are enough registers or sufficient bandwidth to memory. Phase 0 might be combined with phase 1, but this would involve inserting new memory operations into a very ubiquitous software tool (getting pointers to fields), and would probably not be attempted.

Phase 0: There are two components to the accumulation of the field data. One is the computation of pointers to the data on the node — this takes 10 integer operations per element, or 190 I (including the 10 I site task overhead). The other is communication — each boundary link will require one (if it is at the top of a region) or three (if it is on the bottom) fields from other nodes. Thus the communications needs are S/V (2 O_c + 36C).

The total time for phase 0 is $\boxed{190\ I + S/V\ (\ 2\ O_c + 36C\ )}$ cycles.

Phase 1: Each element of the SU(3) products requires 12 multiply/accumulate cycles to compute by finding the dot product of a row of one matrix with a column of another. Computing a staple involves doing a "half" multiply of the first two links (giving only two rows of the product) followed by a "full" multiply by the third link matrix. To reconstruct the third row of the answer, without needing the third row of the intermediate product, one takes advantage of the SU(3) nature of the matrices, and uses the cross product of rows one and two. (This is what allows the first multiply to be incomplete, and also saves time on its own.) The reconstruction takes 8 multiply/add cycles per element. This leads to a total of 168 floating point cycles per staple.

The memory operations needed are heavily dependant on the number of registers available. The obvious minimum is pulling in two rows of the first link, and all three rows of the other two links — 48 M. (These operations are roughly half cachable, depending on whether all four links associated with a site are updated sequentially.) To achieve this minimal memory usage would require sufficient registers. It would seem that 36 registers are needed — at one instant, there is a row from the first link, the entire second link, and two answer rows for the intermediate product. With about 28 registers, one can accomplish the staple computation using 6 extra loads (any of these extra loads will, of course, hit cache) — in the first multiplication, one column of the second link gets overwritten by answers, and is later pulled in over the no longer needed first row of the first link. With as few as 16 registers, the number of memory operations could go up to 78 loads and 12 stores (the intermediate answers would need to be stored somewhere as they are computed). The memory operations associated with storing the answer for each staple (18 S for each except the last) are included here, although if there are sufficient registers to do the phase 2 summing without putting the staple results into memory, these stores can be avoided.

The total time for phase 1 (for all six staples), assuming 32 registers, is $\boxed{\text{Max}(144M + 180M_c + 90\,S, 1008\,F)}$ cycles; one could adjust the memory requirements down by $36M_c$ if there were at least 36 registers, or up by as much as $144M_e + 72S_e$ if there were as few as 16 registers.

<u>Phase 2:</u> Adding up the six staples can be fairly memory intensive unless there are enough registers to keep the accumulations as answers are found. The number of floating point cycles involved is 18 f for each of five staples. (In principle, the additions for the third row can be absorbed under the cross product operations to find elements of that row; this small savings is not worth worrying about). The simplest case is if there are at least about 54 registers — then the accumulation is done without any extra memory involvement. With fewer registers, it becomes tempting to consider overlapping the memory-intensive accumulation with the floating-point dominated phase 1. However, this requires the accumulated answer to be stored and re-loaded each time, further complicating matters and putting a lot of stress on register and memory usage. Instead, we can assume that the first five staples will have been stored, and after the last staple is found, the addition will be done.

The total time for phase 2 is $\boxed{\text{Max}(90\,M_c + 18\,S, 90\,f)}$ cycles, with the memory load needs completely eliminated if there are at least 54 registers, and the memory needs increasing by $18\,M_c$ (the answer for the last staple is not there for free) if there are fewer than about 24 registers.

The total time in cycles for the environment calculation is then (assuming we are in the range where a memory operation takes at least as long as a floating point multiply and add, and no more than 2.5 times as long)

$$\boxed{90\,M_c + 18\,S + 1008\,F + 190\,I + S/V\,(2\,O_c + 36C)}$$

The 90 memory load cycles in the memory-dominated phase 2 are inherently uncacheable loads. For the environment computation, operation, an 80 Mflop unit might be reasonably balanced with 80 Mbytes/sec bandwidth to memory, 16 Mips of integer power, and 20 Mbytes/second of communications bandwidth. Under those circumstances, the environment calculation will take about 50 μsec; this represents 50% of the maximum power in principle possible by overlapping all the integer, communications, memory and floating point operations.

Communications bandwidths between groups of nodes are not particularly stressed here, because the number of sites handled by each node would not be large for any problem dominated by these pure gauge environment computations. A card with 8 nodes might well contain 16*16*16*16 sites; S/V is 1/2. Making the queueing assumptions discussed previously, a 20 Mbytes/second inter-group bandwidth would increase the execution time by 8%; 20 Mbytes/second costs 23%.

Quenched Gauge Configurations — Cabibbo-Marinari

When evolving the gauge field configuration in the absence of fermion loops, updating a link involves the environment calculation detailed above, followed by a heat-bath in that environment to determine the new link. For SU(3), the method of Cabibbo and Marinari is commonly employed: Choose a matrix in the [1,2] SU(2) subspace (with the correct heat-bath distribution); multiply by a matrix in the [1,3] subspace, and then by one in the [2,3] subspace. This gives a result which obeys detailed balance and is very nearly distributed as a heat-bath in the full SU(3) space. (One could choose to do only two SU(2) subspaces, since the product does cover all of SU(3); the feeling is that the better uniformity associated with the extra step leads to quicker decorrelation, more than offsetting the additional work.)

The process then involves three SU(2) heat-bath computations; after each one, the link and the environment must be updated by multiplying a 3 by 3 matrix times a 'promoted' 2 by 2 matrix. (Actually, after the third link update, the environment is no longer needed, so we have a total of only 5 of these "mul23" updates.)

The SU(2) heat-bath calculation involves finding a random magnitude **a** obeying some probability distribution which depends on beta times the "magnitude" of the SU(2) environment bmag, and then choosing a point uniformly distributed inside a circle and forming the SU(2) matrix based on **a** and that point. The only tricky part is constructing **a**. In principle, a table lookup and interpolation is conceivable, but this would have to be a two-dimensional table (indexed by a random number R and the value of bmag — that a single table based on some function of R and bmag is inadequate is non-obvious). The construction and use of the lookup table is not straightforward, and the table would have to be rather large to allow for simple interpolation, or, in the alternative, costly higher-order interpolation would be necessary. In practice, users are likely to employ a more mathematical approach, such as that of Creutz or Kennedy-Pendleton, which constructs **a** from R and bmag and then rejects or accepts according to a further random number. The Creutz algorithm is a bit quicker, but in the

physically interesting regions rejects more tries (and thus has to be repeated more often) — roughly 66% acceptance versus 98% for Kennedy-Pendleton, which makes the choice between them pretty much a wash for MIMD machines. (In the SIMD case, Kennedy-Pendleton has the advantage that the 'tail' of the rejection distribution is shorter — for 256 nodes, it takes an average of two KP steps, but six Creutz steps, for every node to have its value.)

So, we can break this link updating into three sorts of computation, forming phases 3, 4 and 5 of the overall gauge configuration algorithm:

- Phase 3 — Using the Creutz (or another) algorithm to get **a.**
- Phase 4 — Constructing SU(2) elements from a values, which involves picking points in a unit circle.
- Phase 5 — Multiplying 3 by 3 times promoted 2 by 2 matrices to update the link and environment.

Phase 3: The Creutz method involves and exponential, followed by a loop in which two random numbers, a sqrt and a logarithm are needed — the loop repeats until a value is accepted, an average of 1.5 times for physically interesting cases. Including the floating point operations to relate these quantities, the number of cycles taken is 4R + 3T + 15 f, where R is the number of cycles for a random number, and T the number of cycles to compute a transcendental function.

The time taken per random number is, of course, sensitive to the pseudorandom generator employed — there may even be random number hardware. We have found that using a combination of fairly sophisticated random number generators (to insure independent streams) and generating several numbers at once for efficiency is a reasonably quick way of getting good random numbers. This method uses mostly integer operations (including pulling the desired number off the queue of pre-computed randoms) and seems to take about 8 I cycles.

The transcendentals are done by a table lookup and perturbation expansion. The business of getting started (by finding the table index, checking ranges, truncating and subtracting, etc.) takes longer than the actual floating point involved. In principle, one can save a bit of time by relying on the fact that we know in advance what the range of possible inputs can be in this case, and that we care only about absolute (not relative) accuracy in the log case. In practice, one writes the best and most efficient "gold plated" transcendental functions possible, and uses those. The typical transcendental takes 10 I + 20 f (the overlap is negligible) cycles.

Thus, the total time for three instances of phase 3 is $\boxed{12\,R + 9\,T + 45\,f}$ or $\boxed{186\,I + 225\,f}$ cycles.

Phase 4: Here, it is clearly right to simply choose two random numbers, and reject if the sum of squares is greater than 1. This amounts to an average of 2.5 random numbers per SU(2) matrix generated, plus a bit of floating point to put the

randoms into the (-1,1) range and check the magnitude. (However, for SIMD systems, this has a long tail, such that an average of four pairs of randoms are needed before 256 nodes all have accepted a pair. In that case, it may pay to do a table lookup and interpolation on the first random number to get a magnitude, followed by taking the cos and sin of a second random number.) For all three SU(3) multiplies, this amounts to 7.5 R + 26 f.

Once the point on the circle has been determined, it takes about 39 floating point operations and a square root to combine it with the SU(2) environment fragment, to form an SU(2) multiplier. For three such operations, we have 117 f + 3 T cycles. The memory operations involved are negligible.

So, the time taken for phase 4 is $\boxed{7.5\,R + 3\,T + 143\,f}$ or $\boxed{90\,I + 203\,f}$ cycles.

<u>Phase 5:</u> A promoted SU(2) matrix is formed from an SU(2) matrix by inserting a 1 on the diagonal and zeros off diagonal for elements with the third index value, for example,

$$\begin{matrix} a & 0 & b \\ 0 & 1 & 0 \\ c & 0 & d \end{matrix}$$

To multiply an SU(3) matrix by this requires computation of six answer elements (one row remains unchanged). Each answer element requires 8 multiplies and 6 adds — 8 F cycles. So a single "mul23" involves 48 F cycles for the floating point operations.

The memory required is 20 loads (the SU(2) matrix and two rows of the SU(3) matrix) and 12 stores. These operations will hit cache if there is one, and could utilize extended registers. The memory burden could be reduced by keeping the link being updated and the environment matrix in registers, but they would need to be there across the Creutz and SU(2)-forming phases. We assume this savings would not be realized.

This operation must be done to update both the link and the environment, except the final time, when only the link needs to be updated. This means five mul23 operations (420 flops).

So, the time for phase 5 is $\boxed{\mathrm{Max}\,(\,240\,F\,,\,100\,M_c + 60\,S\,)}$ cycles.

The entire heat-bath procedure then involves about 848 flops, and takes

$$\boxed{19.5\,R + 12\,T + 188\,f + \mathrm{Max}\,(\,240\,F,\ 100\,M_c + 60\,S\,)}$$

or, using our values for R and T,

$$\boxed{276\,I + 428\,f + \mathrm{Max}\,(\,240\,F,\ 100\,M_c + 60\,S\,)}$$

cycles.

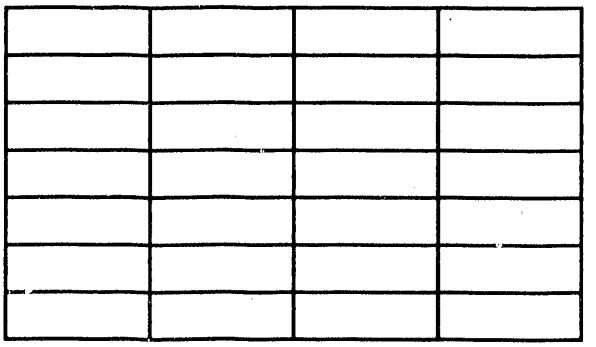
To summarize, for the link updating (representing 2800 flops) the number of cycles required per link is:

field location:	$190\ I + S/V\ (\ 2\ O_c + 36C\ )$
staple products of links:	$\mathrm{Max}(\ 144M + 180M_c + 90\ S,\ 1008\ F)$
accumulating staples:	$\mathrm{Max}(90\ M_c + 18\ S,\ 90\ f)$
Creutz algorithm:	$186\ I + 225\ f$
forming SU(2) matrices:	$90\ I + 203\ f$
updating links, etc:	$\mathrm{Max}\ (\ 240\ F\ ,\ 100\ M_c + 60\ S\ )$

Appendix D — Site Distribution and Communications Issues

When computing communications requirements, it is critical to have a good estimate for the "surface to volume" ratio in a processor — this determines what fraction of accesses will be off node. Three models can apply to this:

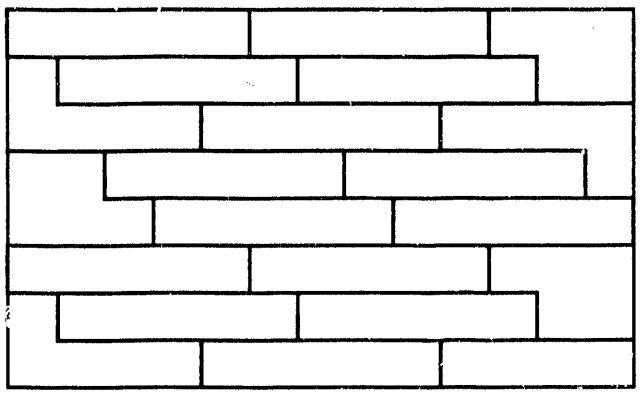
- Dimension Filling (DF) — the portion of the grid in each node is of size L by L by M by 1, where L is the length of the entire lattice (in the appropriate dimension) and M is some fraction of the length in the third dimension. Suppressing the first two dimensions, this looks as follows:



  This method is fairly flexible and easy to implement, and can lead to trivial computation of locations of site data structures in special cases. For example, a 32**3 by 64 grid fits into 512 nodes using 32 * 32 * 4 * 1 slabs in each node, giving a surface/volume ratio of 5/2.

- Chunks of Sites (CS) — each processor handles a hyper-rectangle of sites which is as close to a hypercube as possible. This minimizes the surface to volume ratio, at the cost of some flexibility: Only certain grids can fit onto a fixed number of processes in this way. For example, a 32**3 by 64 grid could fit onto 512 nodes using 8**4 chunks in each, giving a surface/volume ratio of 1/1.

- Broken Dimension Filling (BDF) — each processor handles it share of sites, allocated in some such that contiguous sites in that order are physical neighbors. In two dimensions, this looks as follows:



The big advantage of broken dimension filling is that any shape grid can be mapped onto any number of sites. The surface/volume ratio is not much worse than for dimension filling distributions. One caveat is that if computations are done in lockstep, although there is a reasonable surface/volume ratio, it is much less frequent that all of a set of accesses will be local.

A fourth method would be to attempt to fit irregular almost-rectilinear, almost hypercubic chunks onto the grid. The problem here is that the assignment of sites is not easy to generate, and that the irregularity in 4 dimensional volumes contributes greatly to the surface area. Assuming flexible hardware, the "right" method is probably to use chunks if the number of nodes is such that there is a convenient fit, and BDF (or DF) otherwise. The importance of minimizing surface area is increased for dynamic fermion problems, which tend to be on smaller lattices and be more communications dominated.

For a given site distribution, the order in which sites are processed within each node can still be important. If communications are not done in lockstep, the idea is to spread the interprocessor communication needs over the entire processing time, so as to avoid bottlenecks. This can be less important in MIMD architectures, for which an initial bottleneck has the effect of getting the processes out of sync, thus ameliorating later delays.

The effects of saturation of communication paths can be estimated in the following manner: Let us assume that in the absence of contention, a processor does a process in some time which can be broken into non-communication time T_0 and some communication time τ. The communication time is assumed to be multiplied by some factor F due to saturation effects: $T = T_0 + \tau F$. Queueing theory estimates F as being $\frac{1}{1-P}$, where P is the represents how saturated the channel is. If n nodes are each attempting to use communication time in T total

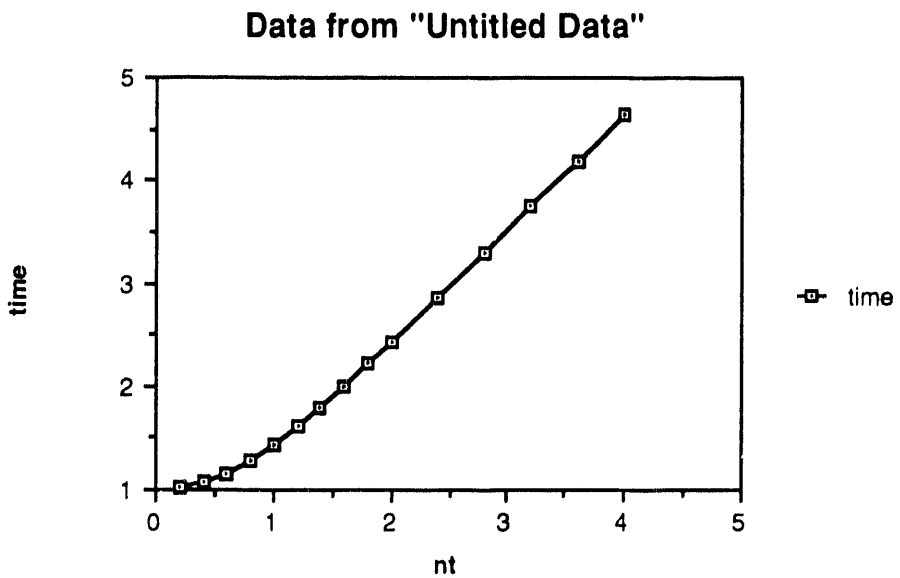
execution time, $P = \frac{n\tau}{T}$. (Note that no matter how large n gets, P never exceeds one; T simply increases, asymptotically being dominated by the communications time.) Combining these, we get an expression for the total time taken:

$$\boxed{T = T_0 + \frac{\tau}{T - n\tau}}$$

This is solved, giving

$$\boxed{T = \frac{1}{2}\left[T_0 + (n+1)\,\tau + \sqrt{\left[T_0 + (n+1)\,\tau\right]^2 - 4\,n\,\tau\,T_0}\,\right]}$$

which behaves in the following way (taking n = 8):



This behavior is such that if the channel would naively be half saturated, the time take is only 10% longer than T_0 ; if it would be fully saturated, the time cost is 42% (and the channel is actually only 70% saturated); and past there, the time quickly becomes communications dominated (and the channel almost fully saturated).

Appendix E: Rules of Thumb

» Cost Rule: In a system with two major costly components with linear improvements per unit cost, if you have no idea as to what the right balance is, then spend half the money on each. This insures that you are within a factor of two (and normally much better) of optimal. This rule requires a scalable system.

» Components Rule: A corollary to the cost rule is that it is much more difficult to design a system with more than two major components, the balances among which are uncertain.

» Design Rule: If you know the proper balance between components for your problems, and this balance grossly violates the cost rule, then re-examine your system design. For example, if you are using DRAM, and your memory costs are negligible for the size of memory needed, then see if using SRAM will improve your cost effectiveness by leveraging your critical FPU power. On the other hand, if almost all your costs are in SRAM, it may pay to design more numerous, less powerful processors using DRAM.

» Software Technology Principle: Don't count on major software technology advances. While hardware technology can be counted on to improve with time, it is very risky to say "I will somehow write a compiler to optimize for this architecture". This rule avoids painting yourself into the "flaky software" corner.

» Keep Good Software: This is a corollary of the Software Technology Principle. If you have some software which is solid and liked by the users, build on that rather than discarding it for something which may someday be much better.

» Software Quagmire Principle: If the Software Technology Principle and the Keep Good Software principle are followed religiously, you will eventually be possessed of hopelessly outmoded, impossible to maintain bodies of software which are relied on by many users. The time to develop major new software packages is before the hardware design for a machine is set.

» Slack Rule: It does not pay do sweat the last tiny balance details. The last 20% of balancing one parameter to all the others is often worth much less than 20% in machine improvement. This is because now if any of the other parameters is stressed, it begins costing full value — there are many things to go wrong, and no slack. However, if there is only one other parameter at the balance point, then your are likely to get nearly the full improvement.

Appendix F: Issues Affecting How the System Can be Used

Probably more important than the balance issues discussed in this presentation, are the factors that will impact how a system can be used to do physics. These include properties of the hardware and system software. The following features might be wanted in a system. They are listed in no particular order, as being essential, important, or only desirable — this list is not intended to be exhaustive. The bottom line is always how much physics can be done on the machine. That may imply tradeoffs between machine size and power, and usability issues.

Multiple Simultaneous Users — Important

Without this, development machines of various sizes will be needed. They would relieve some (but not all) of the development pressure from the main machine.

Sophisticated Scheduling — Desirable

(Only if Multiple Users.) Avoids the tendency to set aside some portion of the system at certain times for immediate development availability.

Checkpointing Capability — Essential

Means that jobs comparable to the mean time between failures (or longer) can be run. Can use either intermediate or long-term storage.

Intermediate (Disk) Storage — Important

Allows user-controlled use to implement larger lattices on memory-intensive problems. Permits quick checkpointing and scheduling. Makes efficient staging to long-term storage possible.

Long Term Storage — Important

Users can keep configurations and propagators for future analysis. Avoids having to re-generate results. Either this or disk storage is needed for checkpointing and for analysis (which requires 12 components of propagators).

Error Checking — Desirable

The possibility of uncaught errors would lead to some fraction of the programs being rerun to check. This depends, of course, on how reliable the hardware is.

Interjob Robustness — Desirable

The need to keep a cache of spare modules, and to keep the machine down when a module has died until it is replaced, is avoided in systems which can run with several modules excluded.

Good Programmability — Essential

Even excluding the issues of not being able to do complicated analysis or of not being able to program the algorithms desired, very difficult programming implies that ambitious optimization is impossible. Once a program is working there is reluctance to change things if programming is difficult.

MIMD — Important

There are some algorithms which appear to require MIMD and which improve speeds by factors of more than 2. Even if these can be made SIMD, MIMD makes it easier to provide programming tools. It also provides a natural way of breaking communications bottlenecks. MIMD systems are also more likely to be able to survive the loss of one module. At least some minimal form of MIMD is required for multiple users.

Global Communication — Desirable

Helps provide clean programming tools, robustness, and flexible resource allocation (disks and tapes). Some algorithms rely on non-local communication; this can be accomplished by passing data along, but that is costly. It is not clear whether global communication algorithms will need to be run for production.

Adequate Memory — Essential

Allows one to let physics dictate decisions on how large a lattice one studies (at the cost of more time). Enables exploration of alternative algorithms on large lattices.

Standardization of Stored Data — Desirable

It should be possible to take configurations and propagators created on one machine over to other systems for later analysis.

Acknowledgements

Much of the information here was amalgamated from discussions with key researchers in the lattice gauge field, without which the uncertainties in these numbers would grow from large to enormous. I would particularly like to thank G. Hockney, D. Toussaint and T. Kennedy for insights into algorithms and software; E. Eichten, A. Kronfeld and P. Mackenzie for discussions of what is really needed to do physics, and D. Husby and M. Gao for hardware insights. Also invaluable were discussions about machines, algorithms and systems with N. Christ, D. Weingarten and J. Sexton.

DATE
FILMED
8/04/92