Hypercubes are General-Purpose Multiprocessors with High Speedup

Marina Chen, Yale
Erik DeBenedictis, Bell Labs
Geoffrey Fox, Caltech
Jingke Li, Yale
David Walker, Caltech

ABSTRACT

This is a submission for the Gordon Bell Award for multiprocessor speedup in the general purpose category. The entry is for the Hypercube architecture.

Program	*PEs*	*Speedup*
QCD Simulation	512	458
Circuit Simulation	128	39
LU Decomposition	128	98

1. Introduction

Reporting the speedup of a multiprocessor on three different programs is a remarkably objective method of measuring progress in parallel processing by a single number. To the extent that we are given the opportunity to write freely here, we would like to expand upon this concise metric with some subjective issues that confront the Hypercube industry.

It seems that the desire for cost-effective computing is the ultimate driving force behind this award. Advancing microelectronic technology is pushing progress toward this goal at a rapid clip -- but there are other approaches. To separate the other approaches from technology, we look for a machine architecture where performance is proportional to cost over a large range. Speedup measures this range. We have indeed seen reported speedup figures climb as parallel architectures have improved. Speedup is not the ultimate measure, however. There are now architectures where performance is sufficiently proportional to cost that speedup can be obtained either by a large monetary investment or by skillful work. To separate brute force from skillful design, we hope that the judges will temper their reliance on speedup in favor of consideration of the potential of the architecture for cost-effective computation.

The generality of multiprocessor systems is an issue we are very concerned about. It seems the real goal is to "fix" the Von Neumann Bottleneck while creating as little disruption as possible in the rest of the computer industry. There are, however, important milestones beyond the inventor himself writing three programs. A milestone is passed when people without a vested interest in the architecture become its users and advocates. Also, it is possible for a system that runs an infinite number of programs to only be able to run an infinitesimal fraction of the world's applications. Try writing a compiler with a relational database system, for example. The milestone here is to shift emphasis from increasing the number of applications that run on a system to reducing the number of applications that will not. Finally, the computer industry has a superstructure of mechanical tools, such as compilers and performance optimizers, which only work with the Von Neumann architecture. To avoid disrupting the industry, these tools must be adapted to parallel architectures.

Our approach is to submit three programs and their speedup figures that not only address different applications, but which were programmed in fundamentally different ways. One program is representative of the programming style supported by the Hypercube vendors, and which has generated hundreds of published papers on solutions of computationally intensive numerical applications. A second program is written for a different machine model which allows the programmer to use some of the same programming skills as are used for regular computers.

The third program is based on a compiler that takes definitive control over spatial and temporal concurrency.

The first and third programs achieve performance proportional to the cost of the multiprocessor, and we have run them on the biggest machines we can afford. All three programs require larger input data sets to achieve high speedup, but input for the second program is exorbitantly expensive for multiprocessor research purposes. The second program gets good speedup for those inputs we can provide.

1.1. Architecture

A Hypercube consists of processors and as little stuff as possible to connect them together. Where the intellectual competitors to Hypercubes might be described as "processors with shared memory" or "processors with fetch-and-add objects;" a Hypercube would be "processors with more processors." Figure 1 is a excellent illustration of Hypercube technology. The board contains 64 processing elements, each consisting of one microprocessor chip (purple) and six memory chips (black). The circuitry that interconnects the processors is within purple chips along with a microprocessor. We believe these are Hypercubes' strong points, while its message-based nature and hypercube topology are less important.

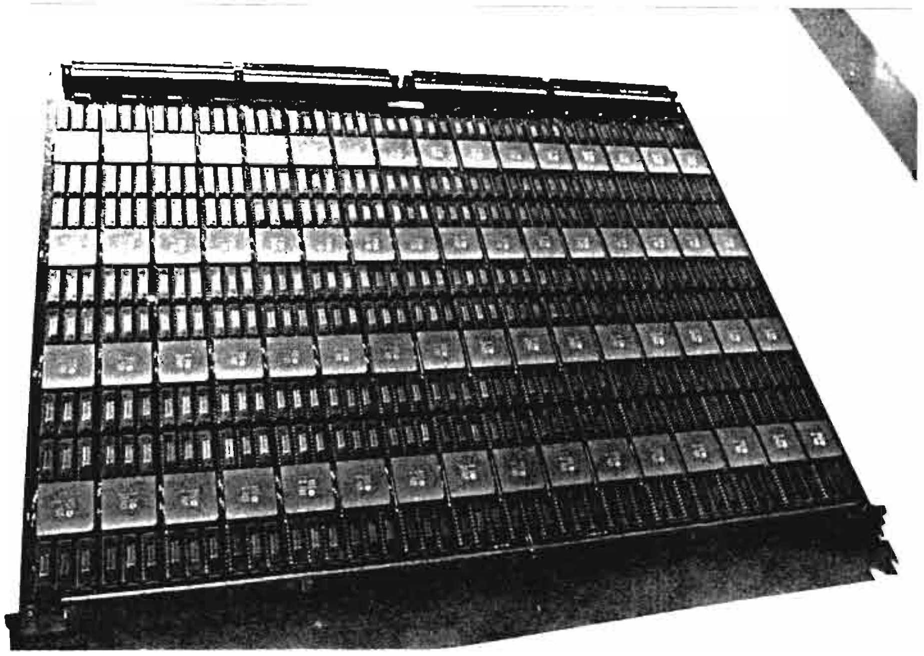


Figure 1: Hypercube Physical Design

The Hypercube is favored by technology even more than is apparent from speedup measures. Figure 1 illustrates 64 Hypercube processing elements (PEs), with each PE having seven chips and retailing for under $1500. Furthermore, one multi-layer PC board can form a backplane for up to 1024 PEs of this design. The chips in this Hypercube consist entirely of IC memories and VLSI microprocessors -- both flagships of modern technology. There are a minimal number of connectors and wire -- which are undesirable overhead. The future is bright also. While retaining a physical design with about a half dozen chips, more advanced microprocessor architectures and larger memories can be incorporated easily. Scalability is good; the limits are not understood now, but certainly lie in the 10K-100K PE range. In addition to having good speedup, the Hypercube gets this speedup with a cost-effective design that is well positioned to exploit cost/performance improvements made possible by advancing technology.

There is a significant subjective advantage of using the Hypercube architecture to study parallel programming: it is virtually impossible to cheat. Since there are no unscalable assumptions underlying the architecture, once a program works efficiently on a modest number of PEs, it is likely to work efficiently on any number. It is said that it only takes 32 PEs to develop Hypercube algorithms, indicating that this "modest number" is less than 32. This is not necessarily true for other architectures: degradation of network latency in shared-memory architectures may only appear when there are hundreds of PEs. There are examples where a medium-size shared-memory machine will solve a problem with a method that appears to be more efficient than a Hypercube can use. In many cases, such a method fails on larger shared-memory machines and a different method is required. Frequently, the new method is similar to the method used for Hypercubes. As a result, the best Hypercube work is often done on machines with 32-128 PEs.

1.2. Hypercubes' Application Domain

One method of demonstrating the generality of Hypercubes is to identify potential application areas and persuade practitioners in those areas to try the architecture. To be successful not only requires good technology and programmability, but the tools must be well developed and disseminated effectively. In the five years that an effort of this sort has been in progress at Caltech, it has been demonstrated that the vast majority of science and engineering problems can be run on Hypercubes. The rapidly growing Hypercube Conference represents the effective dissemination of this information.

A paradigm for Hypercube programming was pioneered by Caltech and is now accepted by most Hypercube users. In this paradigm the data associated with an algorithm resides in the memory of the PEs. The data is manipulated in a series of steps, which may involve local arithmetic, communication with other PEs, or both. Synchronization of the steps is either through a host program, or done autonomously with other PEs. Between synchronization points, the PEs run independently -- as is often required by irregularities and differences between the data domains in the PE.

This paradigm has been studied extensively and appears to apply to problems with the following two characteristics: First, the problem must be large. Larger problems have more intrinsic parallelism, which for this paradigm is called the edge-over-area effect. Second, the computation must be loosely synchronous. Loosely synchronous means that the algorithm has a natural synchronism built into it, such as an iterative sequence that creates time-synchronization. It is perhaps a dangerous oversimplification but one could summarize the lessons of the many years of the user oriented effort at Caltech by: "Large loosely synchronous problems run well on Hypercubes and this success will extrapolate to future much larger machines."

The three examples in this paper; high energy physics, circuit simulation, and LU decomposition are all effectively loosely synchronous with Monte Carlo sweep number, time, and "eliminated variable" respectively providing the necessary course grain synchronization. The above condition is presumably a sufficient and not a necessary condition; applications such as computer chess, transaction analysis, and event driven simulations do not fall into the loosely synchronous class. It is probably unknown at present if any parallel architecture will give scalable speedup to general cases of these non-loosely synchronous applications. We can usefully compare the Hypercube with machines like the Goodyear MPP, ICL DAP, and Connection Machine. These use the same "data parallelism" as in the above Hypercube paradigm but require fully synchronous problems instead of more general loosely synchronous ones. Synchronous problems have a fine grain synchronization allowing lockstep operation of their different components; loose synchronization only requires occasional rendezvous'.

Although this basic paradigm is well understood, the most appropriate or convenient programming environment has not been established. Most of the work at Caltech has used a rather brute force approach. This has lead to several ideas that could be part of future systems. For instance, CUBIX is a parallel generalization of conventional UNIX I/O. The language Coherent Parallel C (CPC), is a natural asynchronous generalization of the SIMD environments for the DAP and Connection Machine; explicit message passing is not necessary and is supplied by the operating system at the loosely synchronous rendezvous'. Neural network decomposition and load balancing tools promise automation of the breakup of the data domain. The sophisticated automatic compiler decomposition technology developed for shared memory parallelization and vectorizations appears to be portable to the Hypercube. Further, the object-oriented paradigm may well be a convenient implementation of the current methodology. Although these improved environments are still at an early stage, industrial use of the Hypercube, such as that at Bell Labs and Shell, has already begun and technology transfer is expected to continue and grow.

In section 2, we concentrate on the speedup shown with one example, a Monte Carlo calculation of a so-called lattice gauge theory. This application illustrates the essential points of the Caltech paradigm, including near-unity efficiency on 512 PEs -- for a speedup of 458. In the accompanying material, [Fox 87] and our book, "Solving Problems on Concurrent Processors," the general applicability of Hypercubes to loosely synchronous problems is fully reviewed.

1.3. Software Engineering for Hypercubes

While programmers have used the Caltech paradigm to code a large number of applications, there are applications for which it does not apply. The Bell Labs' contribution is to try to show that a programmer can, in principle, code any application for a Hypercube.

The field of software engineering studies how applications are coded for a uniprocessor by real people and with a moderate amount of human effort. Since uniprocessors are the prototypical general purpose computer, we can give some evidence that Hypercubes are general purpose by developing Hypercube analogies to the software engineering methods used for uniprocessors.

Programmers are believed to reason mentally with Plans, or mini-programming scenarios. An example for such a Plan might be to apply some function (another Plan) to every data item in a set (or array). Such a Plan would likely be coded with a do-loop in Fortran. On a Hypercube, where the data is resident on the PEs, the Plan could be coded as the host commanding the PEs to execute the embedded Plan via a broadcast protocol. Other information is associated with Plans: the execution time of a Fortran loop is the sum of the execution times of the embedded Plan, whereas on a Hypercube, the total execution time is the maximum over the embedded Plans. The important point is that programmers reason effectively with Plans and their abstract complexity measures, and Plans are not biased for or against multiprocessors.

The Bell Labs' approach has been to identify Hypercube Plans and to develop system software to allow them to be used efficiently. Plans are frequently coded as manipulations of distributed objects (or hybrid objects), which are communication protocols that do data distribution, combining, synchronization, or a similar function. To allow Plans to be combined in parallel or in a hierarchy with other Plans, it is necessary that some PEs of a Hypercube be able to operate independently of other PEs. As a result, the run time system supports distributed objects executing ephemerally and asynchronously.

The second entered program is an IC circuit simulator that is in actual use in a research laboratory at Bell Labs. In making the most effective simulator, several characteristics emerged that are at odds with demonstrating high speedup. We use an algorithm that avoids unnecessary transistor model calculations, reducing uniprocessor running time, but introducing load imbalance on the Hypercube. We also use real circuits which each have a limited amount of available parallelism. Finally, our circuit descriptions are up to 1.5 mByte of compressed binary data, which introduces significant load time. The simulator nevertheless has moderate efficiency for practical combinations of circuit and multiprocessor size. Several versions of the simulator were originally coded on a Hypercube called "BTL Hypercube" using an experimental operating system that supports "Protocols and Plans." It was later moved to an Ncube Hypercube. Simulations of the 5K transistor control portion of a fuzzy logic chip yield a speedup of 18 on the 64 PE BTL Hypercube, and 31 on a 128 PE Ncube. The entered speedup result is 39 on 127 PEs, (speedup of 43 derated by loading time) for simulating a 68K transistor circuit that constitutes a variable length delay line. The largest chip simulated to date implements a "neural net" and has 119K transistors. The application is described in more detail in section 3 and in [DeBenedictis 87], which accompanies this submission.

1.4. Compilers for Hypercubes

While the Caltech and Bell Labs' contributions have shown that programmers can code any application for a Hypercube, Yale's contribution is to suggest that they need not do it at all. The Crystal compiler takes control of the allocation of data and computations to PEs, and of communications. This is analogous to a regular compiler, which takes control of the assignment of memory addresses and the allocation of machine registers, thereby relieving the programmer of a burden. Crystal has variables, but they are not localized to a particular PE in source code. The compiler decides where a variable will be located and generates code to send its value to the places where it is used. The compiler also decides on the order of computations and generates synchronization events as necessary.

The Crystal compiler uses a symbolic representation of the data flow in a program. Heuristics are used to generate symbolic mappings of data items and computations to PEs and time sequence. The compiler then instantiates some of the symbolic representation to produce behavior peculiar to particular PEs, and generates code to evaluate the rest of the program within each PE at execution time.

The third entered program computes the LU decomposition of a matrix and was written in the Crystal language, compiled by the Crystal compiler, and run originally on a Intel iPSC Hypercube with 32 PEs. For this submission, the program was run on a 128 node Ncube multiprocessor. The application is described in more detail in section 4 and in the accompanying documents [Chen 86] and [Chen 87].

The compiler-generated program is presented to represent all programs that can be coded in the Crystal language -- rather than as an engineering application. We believe the ability of Hypercubes to run engineering applications has been established by the previous examples and that Crystal makes an equally strong statement on the generality of Hypercubes in its own way. As a uniprocessor language, Crystal has been demonstrated on a wide variety of algorithms, and has a flavor similar to APL. Compiling techniques for Crystal on parallel machines have been studied extensively and reported in the literature. The Crystal Hypercube compiler has been demonstrated on three algorithms to date.

Hypercube compilers -- as well as Hypercube software engineering and Hypercube conferences -- could be applied to other architectures, so why did we demonstrate these speedup figures on a Hypercube? There seems to be an obvious answer: we own Hypercubes with many PEs, and not other machines. This relates back to the technological argument presented in the beginning -- Hypercubes are unusually favored by technology. The large number of Intel iPSC machines with 32-128 PEs in universities show that a significant parallel processor can be obtained with modest funds. The Ncube/10 with 1024 PEs shows that, with a million dollars, you can build a more powerful Hypercube than any other architecture.

2. Quantum Chromodynamics by Domain Decomposition

2.1. Overview of Loosely Synchronous Problems on the Hypercube (The Hypercube Application in Breadth)

In this section, we briefly review the general issues while in Sec. 2.2 we focus on lattice gauge theory on the Hypercube and finally in Sec. 2.3, consider a particular code on a particular Hypercube and its performance analysis.

The breadth of applicability of the Hypercube and its acceptance by the science and engineering communities can be illustrated by the approximately 80 abstracts submitted to the relevant applications and algorithm sessions at the Third Hypercube Conference (HCCA3) to be held in Pasadena on January 19 and 20, 1988. Caltech has been a pioneer in the scientific use of this architecture and we will concentrate on the work of the Caltech Concurrent Computation Program (C^3P), in the following. We can quantify the application breadth by figure 2 which lists the projects currently under development by C^3P. Further information on these will be found in our annual report C^3P-487 and a qualitative analysis of these, as of January 1987, will be found in C^3P-391. In figure 2, all the implementations are loosely synchronous except those labeled 59 and 61 corresponding to computer chess and transaction analysis. Note that the loosely synchronous classification depends on the particular algorithm used to solve a problem. For instance, the branch and bound approach to optimization (such as the traveling salesman problem) is not loosely synchronous and it is difficult to get good speedup; however, simulated annealing or neural network methods for the same problem are loosely synchronous and speedups scale to large machines.

Our early papers, [C^3P-161, 255], show that the performance of loosely synchronous problems can be modeled by a communication overhead

$$f_C = \frac{constant}{n^{1/d}} \frac{t_{comm}}{t_{calc}} \qquad (2\text{-}1)$$

where n is the grain size and d is the system dimension. t_{comm} and t_{calc} specify the nodal communication and calculation performance. This quantifies the "edge over area" rule mentioned earlier. Using an appropriate definition of d allows the formula to apply to general systems when the natural geometric concept of "edge over area" is inapplicable. We have found the Hypercube quite suitable for nonlocal problems such as those involving medium or long-range particle interactions. The value of n in equation (2.1) allows one to quantify the "largeness" of problems already mentioned in Section 1.2. Typically, one would scale at approximately constant n, so that the problem size increases linearly with the size (in terms of number of nodes) of the machine.

The concept of loose synchronicity has only recently been introduced in [C^3P-451] and our

34 Applications from Caltech Computation Program as of November 1987. (Labelled 28-61) (I= In Progress, B= Begin, C= Complete).

Biology and CNS

28	I	Structural Simulations of Neural Networks Using a General-Purpose Neural Network Simulator and a Hypercube Concurrent Computer	(C^3P-404)
28A	B:	Periodic Orbits in the Piriform Cortex	
29	C:	Back Propagation Algorithms for Character Recognition and Computer Games	
30	I:	Pattern Recognition by Neural Networks on Hypercubes	(207)
31	I:	Collective Stereopsis	(16)
32	B:	Mapping the Human Genome	
32A	B:	Modeling Complex Neurons	

Chemistry and Chemical Engineering

33	I:	Integration of Coupled Sets of Ordinary Differential Equations on the Caltech Hypercubes: Matrix Inversion	(85)
34	B:	Polymer Simulations on the Hypercube	
35	B:	Concurrent Optimization and Dynamic Simulation in Chemical Engineering	
35A	B:	Quantum Lattice System for High T_c Superconductivity and Monte Carlo Simulation	

Engineering (See also Numerical Analysis)

36	C:	Ray Tracing on the Hypercube	(73)
37	I:	Plasma Simulations on the Mark III Hypercube Computer	(108)
38	I:	Vortex Dynamics	(131)
39	I:	Synthetic Aperture Radar (SAR) Analysis on the Hypercube	(468) (303D)
40	I:	Flux-Corrected Transport on the NCUBE	
41	I:	Parallel Free-Langrange Hydrodynamics with a Distributed-Memory Parallel Processor	(189)

Geophysics

42	I:	Finite-Difference Wave Propagation	
43	I:	Normal Modes of the Earth	
44	I:	Finite-Element Flow Modelling	(191)

Physics

45	I:	Lattice Gauge Theory with Fermions on the Hypercube	(184)
46	B:	Random Lattice Calculations	(183)
47	C:	Two Dimensional Melting	(95)
48	B:	Non Local Path Integral Monte Carlo for Helium	(177)
49	I:	N log N Algorithm for Astrophysical Particle Dynamics	(164)
50	I:	The Hypercube for Astronomical Data Analysis	(215)
51	I:	Statistical Gravitational Lensing	(184)
51A	I:	Multichannel Schrödinger Equation	

General Algorithms and Numerical Analysis

52	C:	LU Decomposition of Banded Matrices and the Solution of Linear Systems	(4)
53	C:	Optimal Matrix Algorithms and Communication Strategies for Homogeneous Hypercubes	(C^3P-314, 386)
54	I:	Adaptive Multigrid on the Mark III	(159)
55	I:	Finite Element Methods in Coherent Parallel C	(56)
56	C:	Communication Strategies for Network Simulations	(C^3P-405)
57	C:	A Concurrent Implementation of the Prime Factor Algorithm	(6)
58	C:	Concurrent Tracking Algorithms with Kalman Filters	(186)
59	I:	Chess on a Hypercube	383
60	C:	Shift Register Sequence Random Number Generators on the Hypercube	(182)
61	B:	Transaction Analysis on the NCUBE	

Figure 2: Caltech Applications

book; it ensures that communication and modest load imbalance f_L dominate the overheads. In the typical problem of figure 2, one finds that the speedup S is given in terms of the number of nodes by:

$$S = \frac{N}{1 + (f_L + f_C)} \tag{2-2}$$

or

$$S = \frac{N}{1 + f_L} Min(1, 1/f_C) \tag{2-3}$$

depending on the presence (Eq. (2-3)) of absence (Eq. (2-2)) of overlap between communication and calculation. Particular hardware and algorithms may lead to partial overlap with formulae intermediate between Eqs. (2-2) and (2-3).

f_C and f_L are -- up to logarithms -- independent of N and only depend on the grain size n. The success and simplicity of the model given in Eqs. (2-1, 2, 3) has lead us to cut back on our earlier detailed performance evaluation which was a reflex with each new application; this is "old hat" and we concentrate on other issues in our research.

2.2. Lattice Gauge Theory on the Hypercube (Hypercube Applications in Depth)

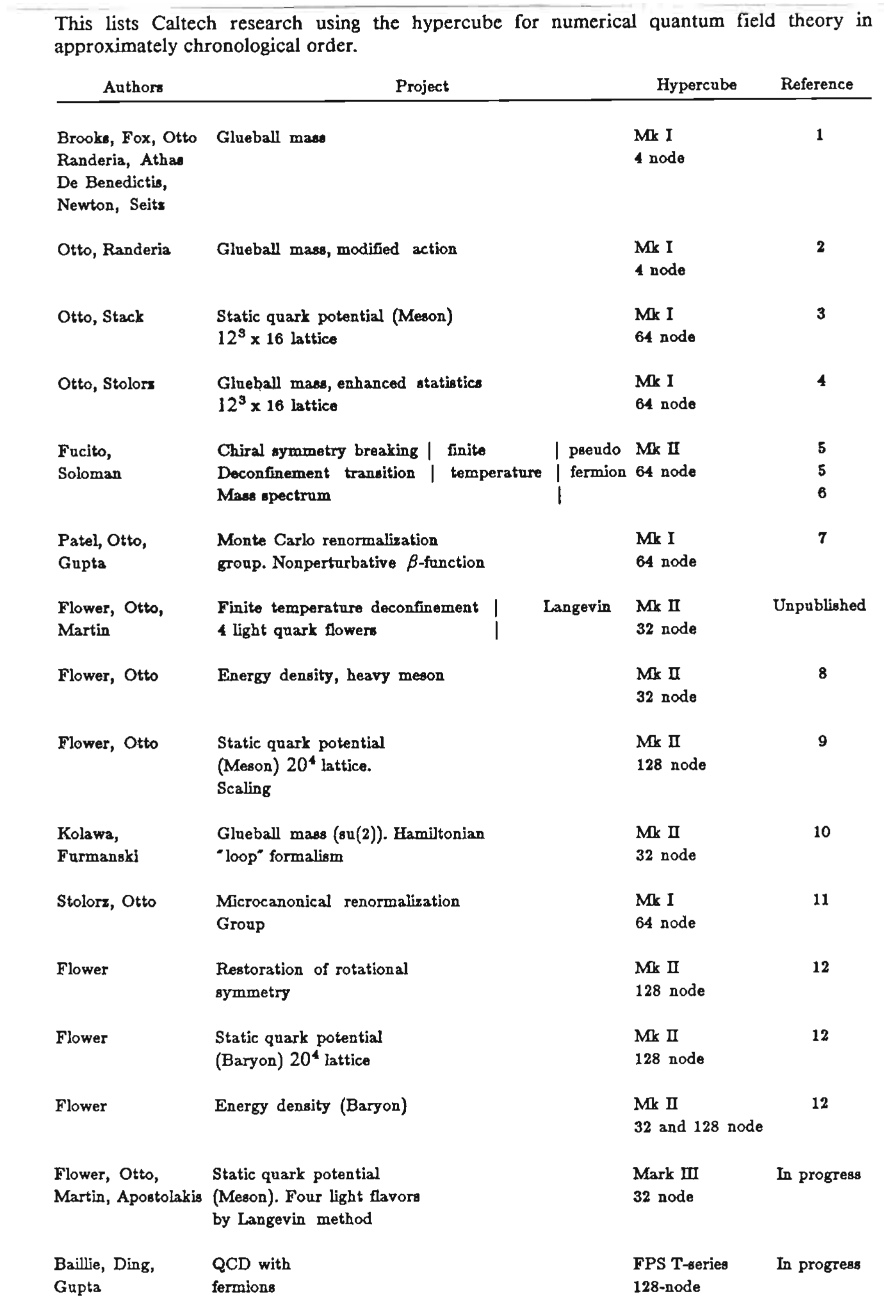
Computational high energy physics was one of the original motivating forces behind the original Cosmic Cube and has used much of the cycles on the Hypercubes at Caltech. Figure 3 lists the some sixteen calculations of this type that have used the Hypercube -- starting in 1982 with the original 4-node 8086-based prototype. The sixty-four node Cosmic Cube was used for 2500 hours on the computations reported in Ref. 3 of Table 2.2.

Gauge theories are ubiquitous in elementary particle physics: the electromagnetic interaction between electrons and photons is described by quantum electrodynamics (QED) based on the gauge group U(1), the strong force between quarks and gluons is believed to be explained by quantum chromodynamics (QCD) based on SU(3), and there is a unified description of the weak and electromagnetic interactions in terms of the gauge group SU(2) x U(1). The strength of these interactions is measured by a coupling constant. This coupling constant is small for QED so very accurate analytical calculations can be performed using perturbation theory. However, for QCD the coupling constant appears to increase with distance so that perturbative calculations are only possible at short distances. In order to solve QCD at longer distances, Wilson [Wilson 74] introduced lattice gauge theory in which the space-time continuum is discretized to provide a cut-off that regulates ultraviolet divergences non-perturbatively. This discretization onto a lattice, which is typically hypercubic, also makes the gauge theory amenable to numerical simulation by computer.

Most of the work on lattice gauge theory has been directed towards solving lattice QCD and thus deriving the hadron mass spectrum from first principles. This would confirm QCD as a theory of the strong force. Other calculations have also been performed, in particular, the properties of QCD at finite temperature and/or finite baryon density have been determined. Unfortunately, in order to simulate lattice QCD on a computer one must integrate out the quark variables (because they are fermions i. e., anticommuting elements of a Grassmann algebra rather than numbers) leaving a highly non-local fermion determinant for each flavor of quark. Physically, this determinant arises from closed quark loops. The simplest way to proceed is to ignore these quark loops and work in the so-called quenched approximation with no flavors of quark present. (This may be valid for heavy quarks.) Current state of the art quenched QCD hadron mass calculations are performed on lattices of size 16^3 x 32 (using ICL DAPs) [Bowler 86] and 24^3 x 48 (using CRAY) [de Forcrand 86]. Other quenched QCD calculations include determination of the scaling behavior of the deconfinement phase transition on a 19^3 x 14 lattice (using CYBER) [Gottlieb 85] and measurement of the heavy $q\overline{q}$ potential on a 20^4 lattice (using the Caltech 128-node Mark II Hypercube) [Flower 86].

However, to investigate the physically more realistic fully interacting QCD, with the inclusion of dynamical quarks, one must go beyond the quenched approximation and tackle the problem of the fermion determinant. One of the first methods invented for doing this was the approximate method of pseudofermions [Fucito 81] and this has been used extensively. There are also exact methods, for example, that of Weingarten and Petcher [Weingarten 81], and the block Lanczos algorithm [Barbour 85]. More recently, equation of motion methods -- using the stochastic Langevin equation [Parisi 81] or the deterministic microcanonical method [Callaway 82] or a mixture, hybrid, or both [Duane 85] -- have been applied to lattice gauge theory. All of these methods have been tested on small lattices. What one would like to do now is realistic simulations of QCD (with quarks) on larger lattices, calculating the hadron mass spectrum, the properties at finite temperature and/or finite baryon density, the $q\overline{q}$ potential, etc., to see the effect of the quark loops.

To illustrate the needs of these current computations consider the hadron mass calculation on a reasonable but probably still too small 24^3 x 48 lattice. This requires 1/4 gigabyte of



This lists Caltech research using the hypercube for numerical quantum field theory in approximately chronological order.

Authors	Project	Hypercube	Reference
Brooks, Fox, Otto Randeria, Athas De Benedictis, Newton, Seitz	Glueball mass	Mk I 4 node	1
Otto, Randeria	Glueball mass, modified action	Mk I 4 node	2
Otto, Stack	Static quark potential (Meson) $12^3 \times 16$ lattice	Mk I 64 node	3
Otto, Stolorz	Glueball mass, enhanced statistics $12^3 \times 16$ lattice	Mk I 64 node	4
Fucito, Soloman	Chiral symmetry breaking | finite temperature | pseudo fermion	Mk II 64 node	5
	Deconfinement transition		5
	Mass spectrum		6
Patel, Otto, Gupta	Monte Carlo renormalization group. Nonperturbative β-function	Mk I 64 node	7
Flower, Otto, Martin	Finite temperature deconfinement | 4 light quark flowers | Langevin	Mk II 32 node	Unpublished
Flower, Otto	Energy density, heavy meson	Mk II 32 node	8
Flower, Otto	Static quark potential (Meson) 20^4 lattice. Scaling	Mk II 128 node	9
Kolawa, Furmanski	Glueball mass (su(2)). Hamiltonian "loop" formalism	Mk II 32 node	10
Stolorz, Otto	Microcanonical renormalization Group	Mk I 64 node	11
Flower	Restoration of rotational symmetry	Mk II 128 node	12
Flower	Static quark potential (Baryon) 20^4 lattice	Mk II 128 node	12
Flower	Energy density (Baryon)	Mk II 32 and 128 node	12
Flower, Otto, Martin, Apostolakis	Static quark potential (Meson). Four light flavors by Langevin method	Mark III 32 node	In progress
Baillie, Ding, Gupta	QCD with fermions	FPS T-series 128-node	In progress

Figure 3: Quantum Physics Programs

memory and about a gigaflop-year of cycles. This would be possible on a dedicated Hypercube of the power just becoming available from JPL (Mark IIIfp) or INTEL (iPSC2/VX). Future machines of NCUBE would be applicable as we can estimate the current 1024 NCUBE/10 at about 200 mflops on this computation.

Lattice gauge theory can be used with the static domain decomposition although there are some subtleties with the order of the Monte Carlo updates and the calculations of so-called observables. These issues are reviewed in our book. Nevertheless, the ability to use high-level languages has enabled the Caltech group to use the very best algorithms and not as in many specialized machines, trapped into old or less efficient algorithms. This is again illustrated in figure 3 by the number of students and postdocs who have used the Hypercube for a rich variety of different computations within this single application.

2.3. The Performance of the QCD Code on the NCUBE Hypercube

The pure gauge, lattice QCD code has been run extensively on the 512-node NCUBE at Caltech. The performance of this code has been discussed in C^{3}P-490. This code spends most of its time updating the SU(3) matrices associated with each link of the lattice. The calculation time to update the lattice once is:

$$T_{calc}=20000N_{sites}(1+0.01AN_1(N,\beta))t_{calc} \tag{2-4}$$

where N_{sites} is the number of lattice sites per PE, t_{calc} is the time to perform a single floating-point operation, A is a constant close to unity, N is the number of PEs, and beta is the coupling constant. A Monte-Carlo technique is used in updating the lattice to generate random numbers with a particular distribution. The function N1(N, β) is the average number of trials necessary to generate each random number.

The time spent communicating when updating the lattice is given approximately by:

$$T_{comm}=\overset{216}{\cancel{864}}N_{sites}(\frac{1}{L_1}+\frac{1}{L_2}+\frac{1}{L_3}+\frac{1}{L_4}) \tag{2-5}$$

where L_i is the number of sites per nodes in dimension i.

The overhead, f = N/S - 1, is therefore

$$f=0.01A(N_1(N,\beta)-N_1(1,\beta))+\overset{.0108}{\cancel{0.0432}}(\frac{1}{L_1}+\frac{1}{L_2}+\frac{1}{L_3}+\frac{1}{L_4})\frac{t_{comm}}{t_{calc}} \tag{2-6}$$

where the first term is due to load imbalance and is small (for a 512-node Hypercube) compared with the second term which is due to communications. We give timings and speedups below for updating lattices of differing sizes on the 512-node NCUBE.

Total size	N_{sites}	T_{512}	T_1	*Speedup*	*Overhead*
16x8x8x8	16	5.202	2037.033	391.7	0.308
16x16x8x8	32	10.123	4073.586	402.4	0.272
16x16x16x8	64	19.598	8146.693	415.7	0.232
16x16x16x16	128	37.995	16292.905	429.1	0.194
32x16x16x16	256	73.386	32585.330	443.9	0.153
32x32x16x16	512	144.493	65170.179	451.1	0.135
32x32x32x16	1024	284.819	130339.879	457.7	0.119

The maximum speedup of 457.7 was obtained for a 32x32x32x16 lattice. The overheads given in the table above are plotted in figure 2.1, and the measured values agree with those predicted by Eq. 2-5. We can be confident, therefore, that equation 2-5 can be used to predict the speedup obtainable for different sized lattices and Hypercubes. For example, for a 32x32x32x16 lattice on the largest available NCUBE (1024 nodes) we predict a speedup of about 915.

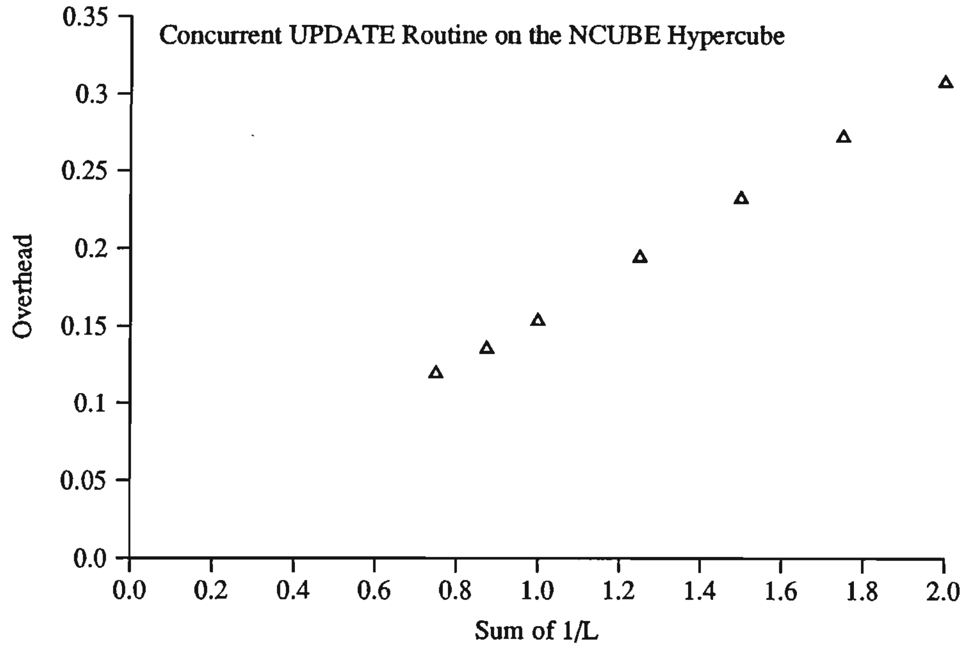


Figure 4: Update Phase Overhead on 512 Node Ncube

3. A Circuit Simulator Conceptualized By Programming Plans

Let us start by considering the way that people think about computer programming from a psychological rather than the usual mathematical viewpoint [Soloway 86, Waters 85].

```
   do 10 i = 1, k
10 f(data(i))
```

The code shown above represents programming knowledge, or a Plan, called the Loop-Iteration Plan. The Plan sequences through an array and applies function f to each data item.

The Loop-Iteration Plan is an example of something that an experienced programmer has used many times, but usually through variants and in combination with other activities. Here, data values are stored in an array, whereas in a variant they might come from a linked list.

Figure 5 is a nearly identical Plan that is relevant to multiprocessor programming. The Plan is called the Master-And-Slaves Plan (MS) and the action starts with the master, who picks a task and makes the slaves work on the task. When the slaves are all done, the master is notified and can do whatever action follows this Plan. Note that the activities performed by the slaves are generally asynchronous and different, thereby distinguishing this Plan from the way a SIMD computer operates.

An example of this Plan is a person running a multiprocessor program interactively. The person is the master and uses the program by repeatedly typing a command to the program and observing the output. The slaves are the PEs of the multiprocessor, and they repeatedly input commands from the master, compute something with the other PEs, and collectively report

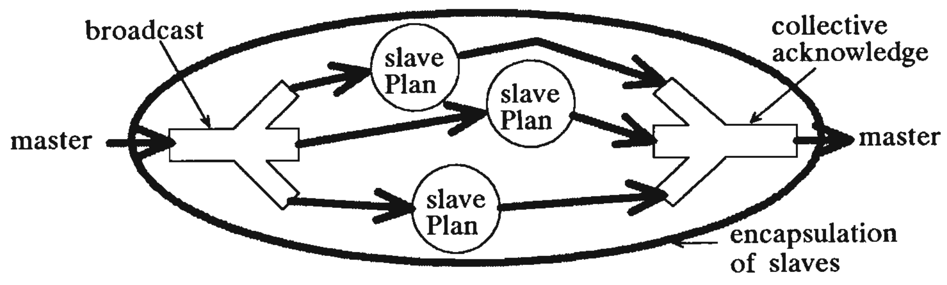


Figure 5: Master-And-Slaves Plan

completion to the master. Of course, a multiprocessor is not restricted to having one master; some parts can be master for other parts.

MS could be implemented by the master broadcasting commands to the slaves, and the slaves participating in some sort of collective acknowledgement protocol with other slaves and the master to indicate completion. We have a distributed object, called the MS primitive, which does the distribution and collective acknowledge for this Plan.

Programs are decomposed recursively into Plans and manipulations of distributed programming primitives. When only messages are used as distributed programming primitives, a programming style identical to "loosely synchronous" results. The internal operation of the distributed programming primitives is not, however, loosely synchronous. A little additional power is gained through use of primitives different from messages, and considerably more power is available if the programmer wishes to construct distributed programming primitives directly as communication protocols. The Bell Labs' approach addresses the difficulty of asynchronous programming by limiting it to the construction of the primitives that are used by the bulk of the program. While programmer productivity may be low for asynchronous programming, little time is actually used because the primitives are only a small part of the program.

The Bell Labs Hypercube operating system provides an efficient environment for common or custom protocols. Protocols such as shared memory [Rudolph 84], RPC [Birrell 84], and f&a-based [Gottlieb 83] are possible. Since Hypercube communication hardware is not specific to any particular programming primitive, the system designer is free to provide those that are most useful in conceptualizing Plans. The Hypercube operating system at Bell Labs simultaneously supports multiple independent instances of distributed programming primitives with different behaviors described by state-transition functions.

3.1. Programming Example - A Circuit Simulator

Integrated circuit simulation is an important computationally intensive application. Circuit simulations which use exact transistor models and accurately model the analog functional and timing behavior of integrated circuits are currently applied to portions of integrated circuits with around 100 transistors. It is important to industry, however, that whole integrated circuits, containing perhaps 1,000,000 transistors, be simulated. Whole integrated circuits can currently be simulated only by abstracting the analog and timing behavior of many small portions of the circuit and then functionally simulating the entire circuit with these abstractions. Functional simulation is inaccurate at modeling timing and analog properties. This section discusses a distributed algorithm for a simulator midway between circuit and functional simulators. By bringing more computational power to bear on a simulation task, this simulator [Ackland 86] permits more extensive simulation of chips during the design cycle, and might therefore speed progress in the IC industry.

3.2. Uniprocessor Circuit Simulation

The type of simulator discussed here divides the simulation into intervals (Δt) and repeatedly computes the voltage on each wire at time $t+\Delta t$ based on voltages at time t.

```
for each timestep
    for each element
        read $V(t)$ from inputs, simulate, and write $V(t+\Delta t)$ to outputs
```

The Plan shown above must be merged with the Simultaneous Update Plan. This Plan assures that the value computed for a wire at time $t+\Delta t$ is really based on voltages at the input of the circuit element at time t. Simply associating a variable with each wire to hold its voltage does not work. When a wire goes from the output of one element to the input of another, and the first element happens to be updated first, then the second element is updated using the new voltage value. A common uniprocessor version of the Simultaneous Update Plan, shown below, associates two variables with each wire, one for an old value and one for a new value. When each circuit element is updated values from the old variables are used to compute values for the new variables. A second phase iterates over each circuit element a second time moving the new variable to the old variable.

```
for each element
    new voltage = update(old voltage)
for each element
    old voltage = new voltage
```

3.3. Multiprocessor Circuit Simulation

Figure 6 illustrates a multiprocessor Plan for circuit simulation. The Simultaneous Update Plan is managed by queues which are written by circuit elements with outputs and read by circuit elements with inputs. During initialization, one voltage sample is put into each queue. In a one-step simulation, the number of voltages in some queues would follow the sequence 1-2-1, and some 1-0-1. In an asynchronous simulation, a quickly simulating element might encounter an empty input queue and have to wait.

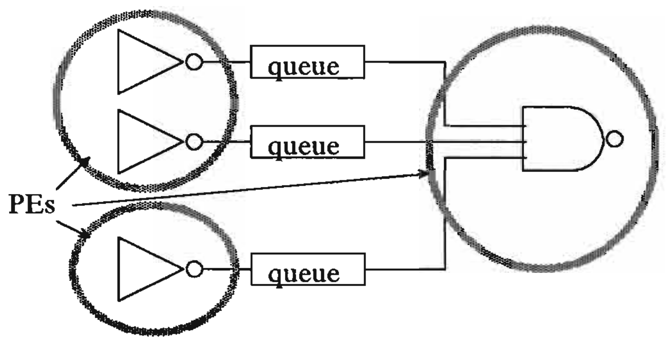


Figure 6: Multiprocessor Simulator with Queues

Figure 7 illustrates the MS Plan in the context of the circuit simulator. The definition of the circuit simulation problem requires that there be a person running the program issuing commands such as "simulate for 100 ns." Such a command must be delivered to every slave with circuit elements, which simulate until done, and then participate in a collective acknowledgement directed toward the master. The master then decides if more simulation is in order or if the answer is to be printed.

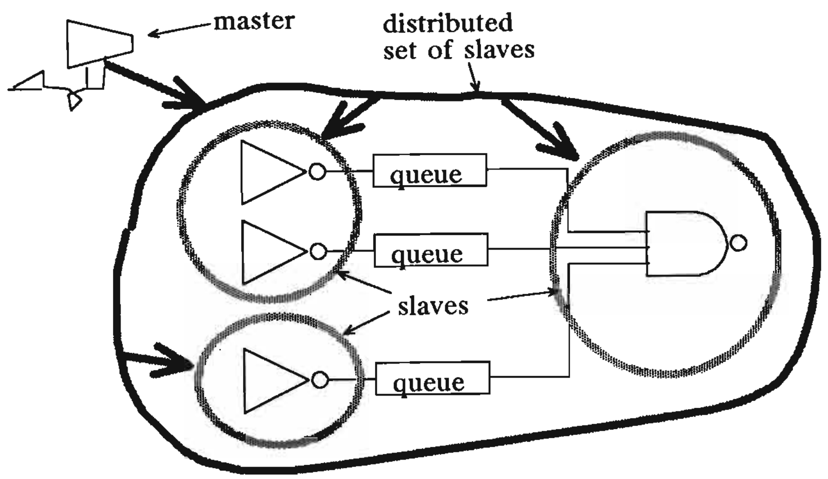


Figure 7: Simulator Control Plan

3.4. Variants of the Simulator

Several different versions of this circuit simulator have been studied at Bell Labs and are summarized below. The algorithm described earlier suggests that each PE synchronize after each simulation time step to avoid unbounded filling of the queues. The synchronization necessary to separate timesteps is less general that provided by MS; specifically, no data needs to flow for this synchronization. A special synchronization protocol was developed that has higher performance than MS, and this version is called synchronous. A version of the simulator was tried where each element asynchronously updates voltages when a new set of input voltages are ready at the input queues and all the output queues have at least one empty location for a new voltage. The asynchrony inherent in this version improves load balancing by allowing temporarily compute-bound elements to fall behind the rest of the simulation without incurring idle time on some PEs. This version is called asynchronous. A Linda tuple space [Lucco 87] was implemented as a distributed object and formed a third version. Speedup figures are illustrated as a guide only and are from the BTL Hypercube using different versions of the program on different circuits.

	synchronous	*asynchronous*	*linda*
MS's	1	1	none
queues	3700 w/flow control	3700 datagram	none
synchronizers	1	none	none
tuple spaces	none	none	1
error message paths	1	1	1
speedup on 64 PEs	18	6	n/a (17)

Figure 8 illustrates the efficiency degradation of the synchronous version of the simulator as a function of circuit size and the number of PEs. Overall efficiency is further degraded by communication cost, which for a Ncube is about 50%. The graph is based on a model which assumes that elements (gates) are randomly distributed and require non-trivial computation for simulation 20% of the time. If there are k elements on a PE, the number of element simulations in a timestep will equal k trials of a 20% random variable. As the number of elements in each

PE increases there are more random trials and the percentage variance decreases. Loss of efficiency due to load balance occurs because all PEs must wait until the PE with the greatest number of element simulations in the entire Hypercube completes. To maintain the same efficiency on a multiprocessor with 10 times as many PEs, the graph indicates a circuit with about 30 times as many transistors is required. While this is not perfect scalability, it shows 60% load balance efficiency (30% overall) for a 100K element IC (about 1M transistors) on a 1K PE Hypercube (the largest Ncube). We have not tried a 1M transistor chip because we do not have a circuit description, and a chip of that size will not be designed just to test the simulator!

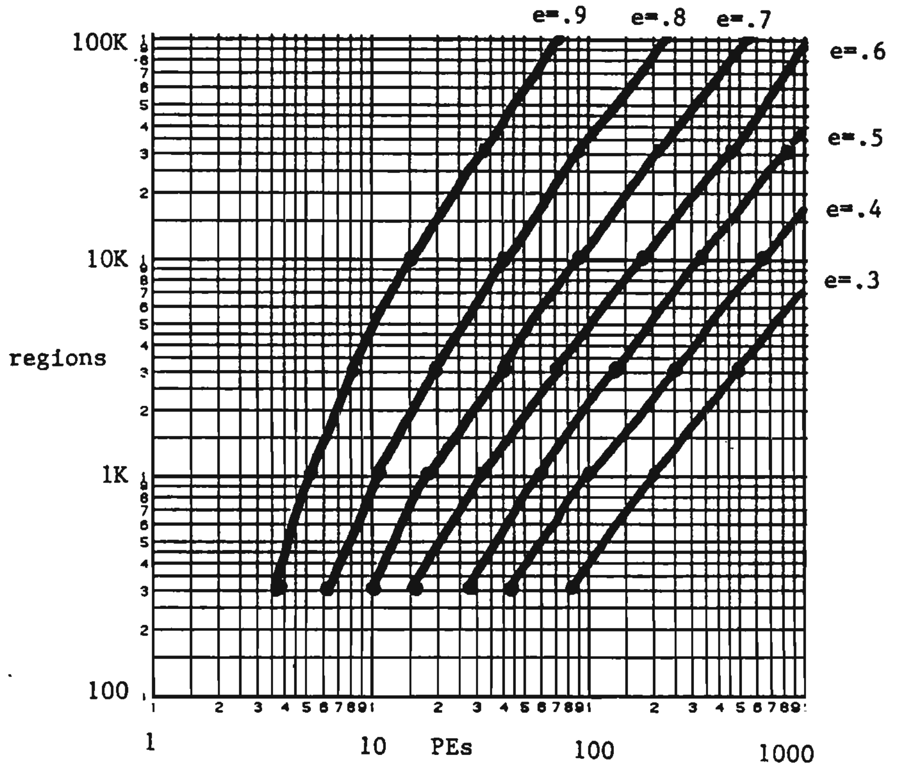


Figure 8: Load Balance Efficiency

The table below illustrates the measured speedups for the simulator running on a Ncube with 128 PEs. (One figure is included from a run on the 512 PE Ncube. Unstable hardware prevented other runs.) Speedup was calculated by the running program based on the percentage time executing the "inner loop" during actual simulation. A 100 Hz real time interrupt invokes code that examines the stacked program counter, identifies which subroutine is executing, and increments a counter for that subroutine. A particular subroutine which evaluates transistor models, but is unrelated to the distributed aspect of the program, is designated as doing useful work; everything else is designated as overhead. Stopwatch time for loading the program and circuit description is factored into the column labeled 127 PEs* only. Load time is amortized over the execution time of the simulation, and the simulation plot in figure 9 shows that excessively long simulations were not used.

circuit name	*transistors*	16 PEs	32 PEs	64 PEs	127 PEs	511 PEs	127* PEs
dly	68K	?	?	26	43	?	39
fuzzy	6K	10	14	22	31	39	30
full8	119K	?	18	32	39	?	35
hwaf1	46K	11	20	24	25	?	25

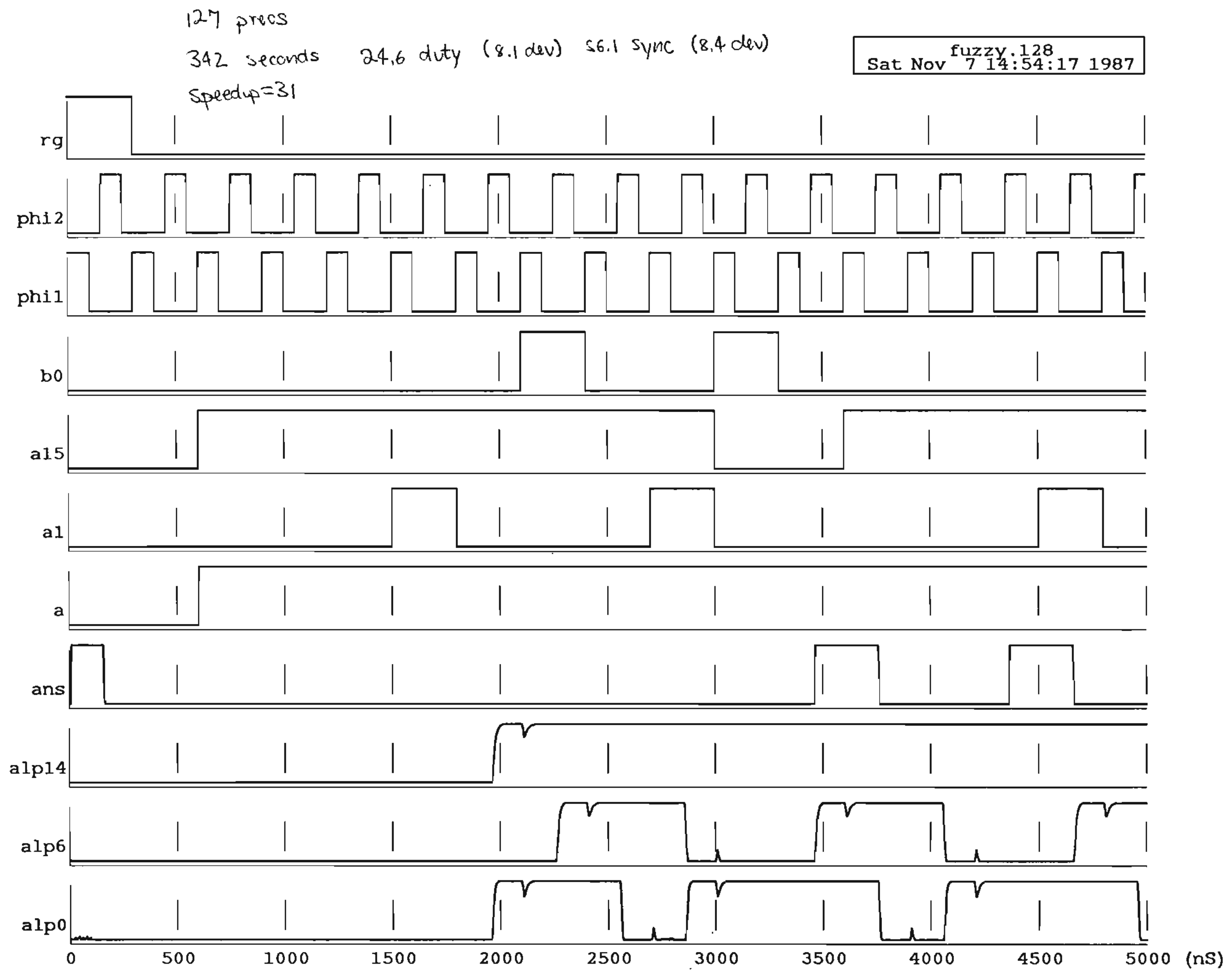


Figure 9: Simulation Plot

4. An LU-Decomposition Program Generated by the Crystal Hypercube Compiler

Crystal aims at providing an architecture-independent notation for programming: one or multiple processors; shared-memory or message passing architectures; SIMD or MIMD machines; and mesh or hypercube interconnections. The language is general purpose and its programs resemble mathematical descriptions.

Crystal's Hypercube compiler divides a program into a collection of code and data partitions automatically. Each partition is assigned to a single PE, and the computations are orchestrated in time and space by distributed control. This control is achieved via interprocessor communications on a message passing machine and synchronization on a shared-memory machine. The strategies in task decomposition and data distribution are geared towards minimizing communication overhead and maximizing efficiency.

Crystal's parallel programming space is multi-dimensional as opposed to the one-dimensional space of sequential programming. In the multi-dimensional programming space, PEs and their sequencing can be arranged in a variety of different ways. For instance, if a user program has indices i, j, and k to express data access and loop sequencing, the Crystal compiler tries to find an optimal time index among one of the original indices, or in a linear combination of these indices.

Hence the number of possible solutions to a given problem can be different not only in the algorithmic sense: also, the same algorithm can have many different realizations in multi-dimensional space and time. Each such realization has different control flow, data flow, granularity, and space-time trade-offs. Each space-time realization is determined systematically by an optimization procedure. In theory, integer programming is necessary to find the optimal space-time tradeoff although faster heuristics may be effective also. In the demonstrated program, the structure of the algorithm was used as a hint for a simple heuristic.

Among the many different possible (near) optimal realizations, the choice must be made on the basis of target machine characteristics such as computation to communication ratio and topology. Hence a program specified in a very-high-level notation allows room for such optimizations, and for maintaining the portability from one generation of the machine to the next.

4.1. A Programming Example

The following is the Crystal program for computing LU-decomposition of a matrix A0. The algorithm takes n (the dimension of the matrix) steps to complete. At each step k, a column of the L matrix, and a row of the U matrix are obtained. Indices i, j, and k correspond to subscripts (e.g. indices of matrix elements) and superscripts (e.g. indices for iterations) in the standard linear recurrence notation. In Crystal, the ranges of these indices are defined by the domains D, D1, D2, and D3 as shown in figure 10

4.2. Dependency Analysis and Granulization

A Crystal program consists of a set of equations. These equations define a data dependency graph where nodes correspond to index tuples as shown in figure 11.

A directed edge from node u to node v indicates that the computation of v depends on u; or u precedes v. A node with no incoming edge, or no predecessor, is a source node. This graph is a finite degree, directed acyclic graph (DAG), where any path must originate from some source node (well-foundedness). Thus the nodes and arcs of the graph represent computations and communications, respectively. The well-foundedness insures that the computation is physically realizable, in other words, always starts with some well-defined initial conditions.

Directed edges of a DAG constrain the order in which the nodes of the graph can be executed. Computation of a node cannot be started until its predecessor has finished. Thus, the maximum path length over all paths in the DAG gives the lower bound of the logical time steps it takes to complete the computation. For any given node, the maximum path length over all paths leading to it from some source node is called its wave number. It gives a lower bound of how soon the node can be computed. Obviously, those nodes with the same wave number are

```
LU_decomposition(A0) = [L_matrix, U_matrix]
 where (
 n = ||A0||,  ! dimension of the square matrix

 a(i, j, k) over D3 =
    << k = 0 -> A0[i-1, j-1],
      (0 < k) and (k <= n) -> a(i, j , k-1) - (L(i,  k) * ( U(k, j)))
    >>,

 L(i, k) over D1 =
    << (1 <= i) and (i < k) -> 0,
      i = k -> 1,
      k < i -> a(i, k, k-1) % U( k, k)
    >>,

 U(k, j) over D2 =
    << (1 <= j) and (j < k) -> 0,
      k <= j -> a(k, j, k-1)
    >>,

 ! define a 3-dimensional domain
 D = {(i,j,k) | 1 <= i < n+1, 1 <= j < n+1, 1 <= k < n+1},
 ! projection of D along the 1'st axis
 D1 = D proj 1,
 ! projection of D along the 0'th axis, then transpose the domain
 D2 = transpose(D proj 0),
 ! join of two domains into one
 D3 = {(i,j,0) | (i,j) in (D proj 2)} + D

 L_matrix = [ L(i,k) | (i,k) in D1],
 U_matrix = [ U(k,j) | (k,j) in D2]
 )
```

Figure 10: LU-Decomposition in Crystal

independent. Hence, they can be computed in parallel. We call the number of nodes that have the same wave number its width of parallelism. Thus the upper bound of the number of PEs required to achieve maximum parallelism is the maximum width over all wave numbers. For a so-called "embarrassingly parallel" problem, its width might be close to the total number of nodes in the graph.

With a symbolic representation of a dependency graph in hand, the computational task and data structure that correspond to each node of the DAG is determined. One can view the original computation as being decomposed into a parallel computation with the finest possible grain. Thus this process is called granulization.

4.3. Compiler Techniques

Compiler techniques described below are based on the analysis of the above-mentioned dependency graph. Four major techniques are discussed here. The overall structure of the compiler is illustrated in figure 12.

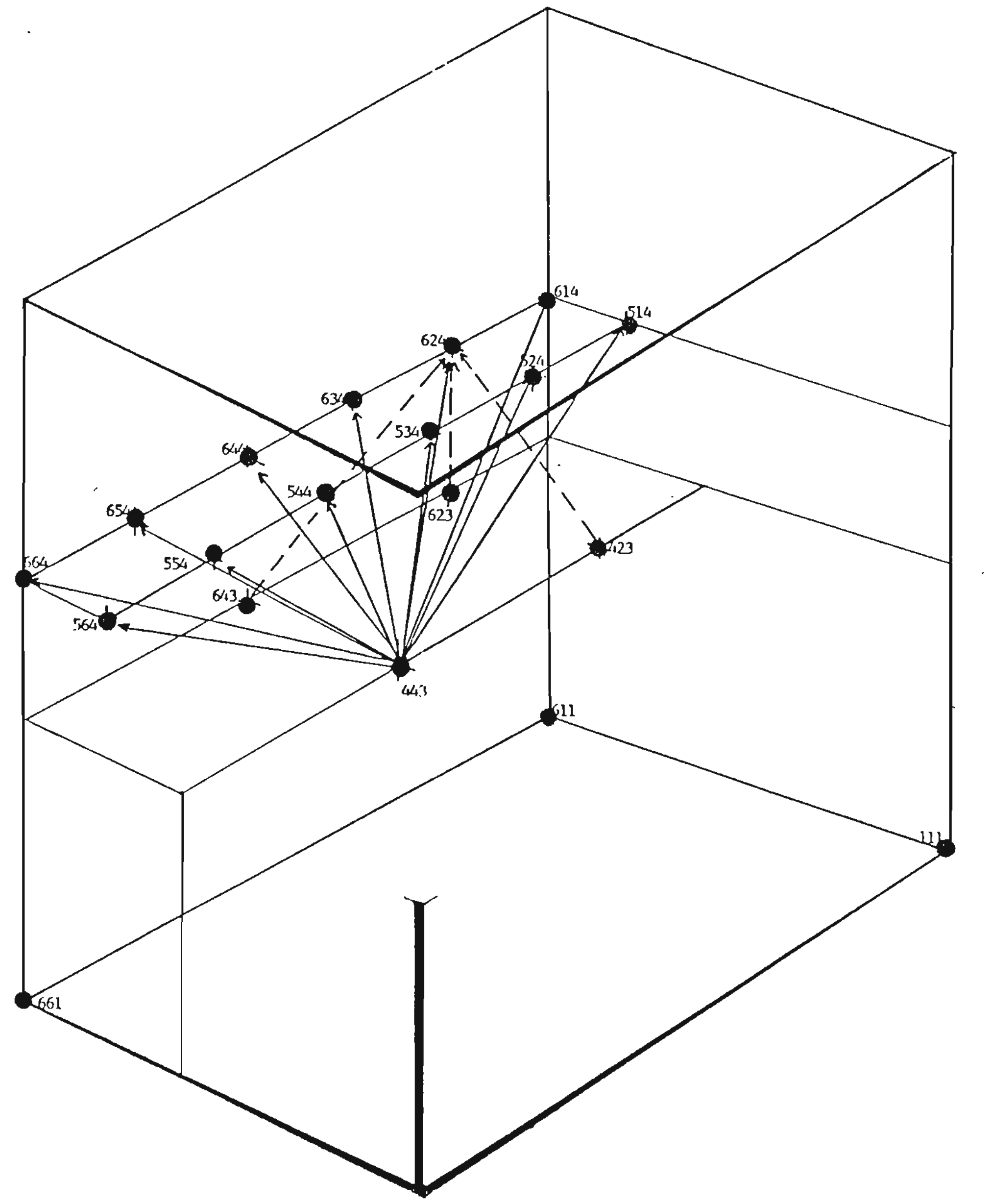


Figure 11: The Data-Dependency Graph of LU-Decomposition

4.3.1. Aggregation

The first technique is for the purpose of granularity control. A collection of DAG nodes are aggregated into an "execution quantum" with no communications within a given execution quantum. A new directed graph can be formed as follows: let each execution quantum be a node and let there be a directed edge from quantum p to quantum q if there exists an edge from a node u in quantum p to a node v in quantum q in the original DAG. The aggregation is chosen to have enough granularity and to satisfy the condition that the new directed graph is acyclic. Since no communication is allowed in a given quantum, a cyclic graph results in deadlock. Thus we call the new directed acyclic graph a QDAG.

For the target LU decomposition program, a single quantum contains a block of n_i rows, n_j columns, and n_k steps. Let the quanta be indexed by three indices i', j', and k'.

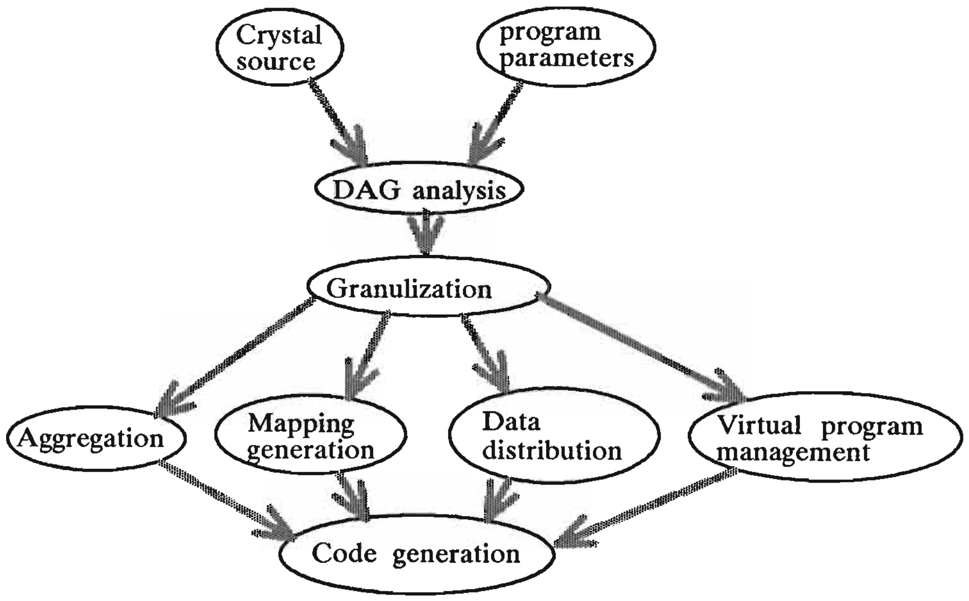


Figure 12: Block Diagram of Crystal Compiler

4.3.2. Mapping Generation

The second technique is space-time mapping. Two types of space-time mappings are used: one is a mapping from the set of quanta to a space of processors S and a linearly ordered logical time domain T. Then other mapping is from the set of nodes D_q within a given quantum q to nested levels of linearly ordered logical time domains (sequential loops). Optimal or near optimal mappings are rarely unique and the choices depend on particular objectives. A given target code may best use two different mappings mentioned above. The former is aimed at minimizing inter-processor communication overhead while the latter aims at minimizing the sequential computation time.

For the target LU decomposition program, two mappings are used. Within each quantum, the indices i, j, and k are mapped to three nested loop indices where i is the innermost level while k the outermost. In between the quanta, the indices i' and j' are mapped to those for a two dimensional array of processors. The time index for the target program, however, is i'+j'+k'.

4.3.3. Communication Methods

The third technique is the determination of communication methods for processors. How data should be distributed, by broadcasting or pipelining? What kind of network embedding shall be used? How can hot-spots be avoided? Data distribution affects performance and scalability of the target code in an essential manner.

For the target LU decomposition program, data produced by each virtual processor are pipelined in both directions on the two dimensional virtual network.

4.3.4. Virtual Processor Management

Finally, the virtual processors must be mapped to physical PEs. This technique is geared toward optimizing load balancing.

4.4. Performance Results

The Crystal compiler and LU decomposition program were written at Yale and using a 32 PE iPSC Hypercube and the adapted to a 128 PE Ncube Hypercube at Bell Labs. (The program also ran on the 512 PE Ncube, but much to our dismay, unstable hardware prevented the answer and timing statistics from being printed.) Performance figures are given below from the Ncube. A 50 Hz real time interrupt on the Ncube samples PC values and increments counts associated with various parts of the code. The compiler output was hand-edited to identify the computational part of the algorithm to this interrupt routine; all other code is designated as overhead. Stopwatch time of 6 seconds is added for program loading in the column labeled 128 PEs*.

Matrix Size	32 PEs	64 PEs	128 PEs	127* PEs
300x300	26	51	96	88
500x500	?	?	101	98

5. References

[Ackland 86] B. Ackland, S. Ahuja, E. DeBenedictis, T. London, S. Lucco, and D. Romero, *MOS Timing Simulation on a Message Based Multiprocessor.* In Proceedings of the IEEE International Conference on Computer Design, October 1986.

[Barbour 85] I. Barbour, N. Behilil, P. Gibbs, M. Rafiq, K. Moriarty, and G. Schierholz, *Updating Fermions with the Lanczos Method,* DESY 85/141 preprint, December 1985.

[Birrell 84] A. Birrell, B. Nelson, *Implementing Remote Procedure Calls.* In ACM Transactions on Computer Systems, February 1984. Pages 39-59.

[Bowler 86] K. Bowler, C. Chalmers, R. Kenway, G. Pawley, and D. Roweth, *Hadron Mass Calculations using Susskind Fermions at beta = 5.7 and 6.0,* Edinburgh-86/369 preprint, September 1986.

[Callaway 82] D. Callaway and A. Rahman, Phys. Rev. Lett. **49** (1982) 613.

[Chen 86] M. Chen, *Very-high-level Parallel Programming in Crystal,* Yale University Computer Science Department document YALEU/DCS/RR-506, December 1986. *Attached to this submission.*

[Chen 87] M. Chen, *Can Parallel Machines be Made Easy to Program? A Data-parallel Model for Functional Languages,* Yale University Computer Science Department document YALEU/DCS/RR-556, August 1987. *Attached to this submission.*

[DeBenedictis 87] E. DeBenedictis, *A Multiprocessor with Protocol-Based Distributed Programming Primitives,* International Journal of Parallel Programming, February 1987, pp 53-84. *Attached to this submission.*

[Duane 85] S. Duane, Nucl. Phys. **B257** [FS14] (1985) 652.

[de Forcrand 86] P. de Forcrand, A. Konig, K. Mutter, K. Schilling, R. Sommer, *Hadron Mass Calculation on a 24^3 x 48 Lattice,* WU B 86-3 preprint, January 86.

[Flower 86] J. Flower, S. Otto, Phys. Rev. **34** (1986) 1649.

[Fox 87] G. Fox and P. Messina, *Advanced Computer Architectures*, Scientific American, October 1987, pp. 66-74. *Attached to this submission.*

[Fucito 81] R. Fucito, E. Marinari, R. Parisi, and C. Rebbi, Nucl. Phys. **B180** [FS2] (1981) 369.

[Gottlieb 83] A. Gottlieb, R. Grishman, C. Kruskal, K. McAuliffe, L. Rudolph, M. Snir, *The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer.* In IEEE Transactions on Computers, February 1983. Pages 175-189.

[Gottlieb 85] S. Gottlieb *et al*, Phys. Rev. Lett. **55** (1985) 1958.

[Lucco 87] S. Lucco, *A Heuristic Linda Kernel for Hypercube Multiprocessors.* In M. Heath (ed.), Hypercube Multiprocessors 1987, SIAM, 1987. Pages 32-37.

[Parisi 81] G. Parisi and Y. Wu, Sci. Sin. **24** (1981) 483.

[Rudolph 84], L. Rudolph, Z. Segall, *Dynamic Decentralized Cache Schemes for MIMD Parallel Processors.* In Proceedings of the 11th Annual International Symposium on Computer Architecture, June 1984. Pages 340-347.

[Soloway 86] E. Soloway, *Learning to Program = Learning to Construct Mechanisms and Explanations.* In Communications of the ACM, September 1986, Pages 850-858.

[Waters 85] R. Waters, *The Programmer's Apprentice: A Session with KBEmacs.* In IEEE Transactions on Software Engineering, November 1985. Pages 1296-1320.

[Weingarten 81] D. Weingarten, D. Petcher, Phys. Lett. **99B** (1981) 333.

[Wilson 74] K. Wilson, Phys. Rev. **D10** (1974) 2445.