CALT-68-867
DOE Research and
Development Report

# Nearest Neighbor Concurrent Processor

E. Brooks, G. Fox, R. Gupta, O. Martin, S. Otto
*Caltech High Energy Physics Group*

E. DeBenedictis
*Caltech Computer Science*

*ABSTRACT*

We propose to build a concurrent processor based on an 4 by 4 by 4 array of microprocessors. These are arranged in a three dimensional lattice with each processor connected to its nearest neighbors. This architecture is optimized for numerical solution of differential equations (as the finite difference form of a differential only involves nearest neighbors) and to the many statistical mechanics problems with short range forces. We intend to use it to solve two and three dimensional quantum field theories using the Feynman path integral formulation of the theories. We will also investigate the generality of the architecture for other problems because we will need huge arrays (up to one million concurrent processors) to solve realistic four dimensional field theories. The proposed array will have the computer power of about one half a Cray I or about 15 times that of a VAX11/780 for problems that can use this concurrent architecture.

September 1981

CALT-68-867
DOE Research and
Development Report

# Nearest Neighbor Concurrent Processor

E. Brooks, G. Fox, R. Gupta, O. Martin, S. Otto
*Caltech High Energy Physics Group*

E. DeBenedictis
*Caltech Computer Science*

*ABSTRACT*

We propose to build a concurrent processor based on an 4 by 4 by 4 array of microprocessors. These are arranged in a three dimensional lattice with each processor connected to its nearest neighbors. This architecture is optimized for numerical solution of differential equations (as the finite difference form of a differential only involves nearest neighbors) and to the many statistical mechanics problems with short range forces. We intend to use it to solve two and three dimensional quantum field theories using the Feynman path integral formulation of the theories. We will also investigate the generality of the architecture for other problems because we will need huge arrays (up to one million concurrent processors) to solve realistic four dimensional field theories. The proposed array will have the computer power of about one half a Cray I or about 15 times that of a VAX11/780 for problems that can use this concurrent architecture.

September 1981

## 1. Introduction

Recently there has been substantial progress in the understanding of quantum field theories using numerical calculations in regions where the coupling is large and so normal perturbation series methods are inapplicable [1-2]. Field theories involve an infinite number of degrees of freedom labeled by a spacetime vector $\vec{x}$ and some discrete index. For instance in QCD, $\vec{x}$ runs over four dimensional space while the discrete index (for the pure gauge theory with no quarks) labels the elements of a 3 by 3 unitary matrix. As we describe in detail in the next section, current results are encouraging but the present calculations are far too limited to allow significant conclusions.

It is our understanding that the proper solution of this problem does NOT involve giant (e.g. Cray) computers. These are not more cost effective than VAX class machines; rather they offer the possibility of doing in a week something that would take a year on the VAX. Further the speed of very fast computers is limited to something like a 100 times the VAX. In fact the correct method of solution of quantum field theories (and no doubt many other problems) lies in parallel (concurrent) processing. In the past parallel processing has not been successful because one has not been able to design suitable general purpose compilers to take advantage of the hardware architecture. However it is easy to see that numerical algorithms for field theories have just the structure necessary to allow straightforward application of parallel processing. The first step is to change the continuous label $\vec{x}$ to a discrete set by dividing space time into a lattice. If each dimension is divided into N parts, we get $N^d$ sites for a problem in d dimensions. For the real problem d = 4 but there is a lot of interesting physics even in one space and one time dimension: d = 2. It is worth noticing that essentially no physically observable quantities (e.g. masses) have been

calculated in any field theory even for d = 2. Now we have to calculate multidimensional integrals of the form

$$\int PHY(\varphi)\,WGT(\varphi)\,d(\varphi_1)d(\varphi_2)....d(\varphi_M)$$

Here M is the total number of variables = $N^d \times L$ where $L$ is number of independent degrees of freedom at each site.

WGT($\varphi$) is a positive weight function which contains the dynamics. In the statistical physics analogy,

$$WGT(\varphi_1,\varphi_2,\cdots\varphi_M)=exp(-H/kT)$$

while in the quantum field theory case

$$WGT(\varphi_1,\varphi_2,\cdots\varphi_M)=\exp(-S)$$

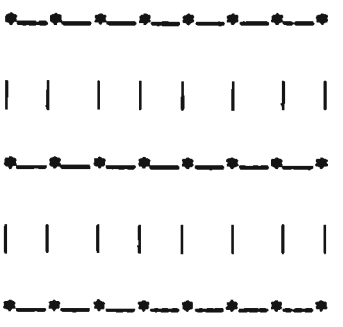
where $S$ is the action.

Finally, the choice of the function PHY($\varphi_1$,.. $\varphi_M$) depends on the physical observable of interest. For instance, for a theory with a mass gap, taking PHY to be the correlation between the fields at different sites, the integral falls off exponentially with distance between the sites. The mass gap of the theory is then given by the rate of fall off. There exists other choices of PHY to find the mass spectrum.[5]

This multidimensional integral is solved by monte carlo techniques. Each set of values of the integration variables is called a configuration. This involves specifying a value for each of the M $\varphi$'s. There are some clever but simple methods for choosing the $\varphi$'s distributed according to the weight function WGT (i.e. importance sampling). A sweep consists of updating all the variables on the lattice once. If the $\varphi$'s are chosen in this way, our integral becomes:

Average over configurations of PHY($\varphi_1 \cdots \varphi_M$)

To get good answers one needs many sweeps; the current studies are too limited to precisely estimate how many for realistic problems (see discussion in Section 2) but at least a million sweeps are probably necessary for interesting problems. One also needs $N \approx 100$ (about 10 sites inside a particle and a world whose total extent is also about 10 particles!), therefore the total number of sites needed is very large. For QCD (Quantum Chromodynamics, the gauge theory of quarks and gluons) in four dimensions we have, say, $10^9$ variables (for the product of $L \times N^d$) The generation of a configuration according to the importance sampling algorithm is an iterative technique which uses the previous configuration (i.e. the previous point in M dimensional $\varphi$-space). This technique calculates a new value of $\varphi$ at each site solely based on the values of the nearest neighbor fields. This is the critical feature of the algorithm but it is very general as its validity only depends on the interaction being local; something for which there is a lot of theoretical prejudice and experimental evidence. The calculation of the new field value at a site might involve 100 -> 1000 computer instructions and so total number of operations required to solve QCD is around $10^6(\#sweeps) \times 10^9(\#variables) \times 1000(\#operations/site)$. On a VAX an operation takes about $3\mu s$ and so we need about $10^6$ years to solve our problem. However, the fact that you only need to know about nearest neighbors implies that a very simple concurrent processing algorithm is possible. Taking the simplest case, we imagine a computer at each site to perform the updating. We have $N^d$ computers with nearest neighbor connections.



In the above, "-" and "|" denote connecting lines and "*" computers. The usual periodic boundary conditions imply that the computers at opposing ends are also connected to each other. It is the clear that the above architecture can be easily implemented; each computer needs just to be able to run programs accessing its own memory and at the end of each cycle, transfer its new site value to its neighbors. There is no problem in the concurrent algorithm and one can gain a factor of $N^d$ ! In practice the *'s above will represent computers each holding a block of B neighboring sites (e.g. 2×2×2×2 cube of values for B = 8 etc.). For QCD one can imagine $10^5$ ( $10^6$ ) computers each with 1000 (100) sites costing perhaps $10 million. As this is comparable in cost to a single experiment in high energy physics, it seems likely that the physics community would consider this a good investment if one could show there was a good chance of solving QCD. The above estimates suggest that the factor of $\approx 10^6$ improvement over a normal sequential computer provides a possibility of solving QCD. At the moment we do not know enough about simpler systems to know if this conclusion is optimistic. In the above diagram, one can choose * either to be a specialized chip or a general purpose microprocessor. To refine our techniques we need to investigate a variety of theories and different choices of PHY. Thus it is appropriate to use a general purpose microprocessor in the first system.

We propose an initial system of 64 computers arranged as a 4 by 4 by 4 three dimensional array. This can also be viewed as a 8 by 8 two dimensional array-in each case with connections appropriate for periodic boundary conditions. This will be interfaced to the unibus of a PDP-11 or VAX which will initiate our machine and accumulate results. Programs for the computers will be written in C, compiled on the High Energy Physics PDP-11/45 and down loaded into the microcomputers. With this system, we will have the power of about 15 VAXs for about one third the cost of a single VAX!

We can use this system not just to learn about the concurrent architecture but also to solve many physics problems. As described in section 2, we should be able to solve all two dimensional theories and make a start on the important three dimensional case. In particular we want to look at the SU(2) gauge theory in d = 3 dimensions.

If we are successful, the next step would be about a 1000 (say 8 by 8 by 16) component system which should be able to solve most three dimensional theories and make a stab at the four dimensional case. After this we may know enough to see if the grandiose million unit machine mentioned above is warranted.

Section 2 discusses the details of the application of our proposed machine to quantum field theories while section 3 describes the hardware and its cost. Section 4 discusses other possible applications of this parallel architecture. In the first appendix we evaluate our proposal from a computer science point of view; comparing the architecture with other possible designs and estimating the efficiency of our proposal. This turns out to be very high for problems related to differential equations. In appendices 2 and 3 we show that our design will in fact also be very useful for matrix problems and discuss a theoretical quantum mechanics application involving matrices which we intend to test on our system.

obeys the continuum renormalization group. Bhanot and Rebbi [4] have gone further and have estimated the mass of the glueball of this model via the fall off of plaquette-plaquette correlations.

Though the above results are encouraging in the sense that they show that lattice calculations are capable of giving quantitative results for physically interesting quantities, so far the results are crude. In the case of the glueball mass, for example, we believe the difficulty of the problem has been underestimated. The method used by Bhanot and Rebbi is unreliable and a correct estimate of the glueball mass requires much more statistics (i.e. sweeps) than they took in their calculation [5].

### 2.3. Physics Accessible with the NNCP

#### 2.3.1. 2-D Models

To see how difficult an estimate of something like a glueball mass is, we have first studied simpler 2-D field theories which can be done on a VAX via Monte Carlo methods. The O(n) models are of particular interest. The O(2) model (n=2, i.e. planar spins interacting with nearest neighbors), has a non zero mass gap which we have measured via correlation falloff and variational methods [5]. We are also interested in the continuum limit of this model - that gotten by approaching a critical point of the model. Kosterlitz and Thouless [6] predict that the mass gap goes as

$$m \approx C\,exp(-b/(T-T_c)^{\nu})$$

where $T_c$ is the critical temperature. b and $\nu$ are predicted by continuum theory. We determine the mass gap for a range of $T$ near $T_c$, $T>T_c$, and see if the lattice mass gap obeys the continuum form. If so, we can then extrapolate

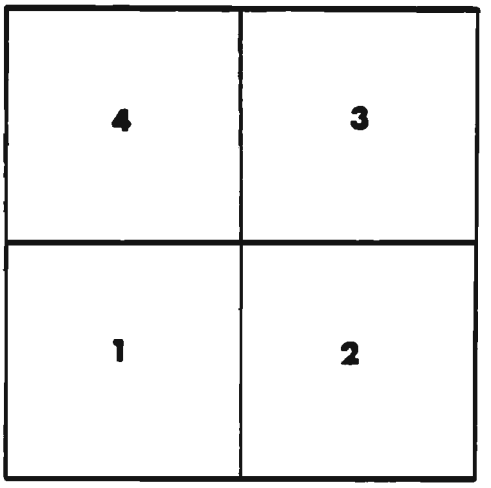
to the continuum limit using the analytical expression above to get the physical mass. This is the analogue of renormalization group behavior in QCD. In determining the exact value of $C$, the main source of error will be due to finite Monte Carlo statistics.

For the O(2) model, $T_c$ turns out to be ≈.9. Let us give an estimate of the run time needed to determine the mass gap at T = ≈1.0, where the correlation length of the lattice is approximately 10 lattice spacings. For correlation length of 10, we need to work on a lattice of size 50x50 or more. Smaller lattices give rise to systematic errors which are too large if good accuracy is needed. The Monte Carlo algorithm generates field configurations which are very correlated from one sweep to the next. This means that the effective number of field configurations our algorithm generates is much less than the naive number. From our studies of this model on our VAX we have measured this "speed of travel through phase space." At this value of the correlation length we estimate that one has 1 "useful" sweep per 300 sweeps generated. To determine the mass gap $m$ to an accuracy of $\delta m/m \approx .05$ we need to generate ≈100,00 sweeps. On the VAX this would take $2\times10^5$ sec or 60 cpu hours. For the 8×8 array of processors this determination of the mass gap near the continuum region to 5% accuracy would take 4 hours. This needs to be done for several values of T and one would probably need 1% accuracy for good estimates of the T dependence.

In conclusion, for 2 dimensional field theories of complexity similar to O(2), we estimate that it will indeed be possible to study the mass spectrum with the NNCP. A preliminary (5%) study will take 2 days; 1% accuracy a few months!

### 2.3.2. 3D models : Glueball Mass and Spin for SU(2)

As an example of an interesting model in higher dimensions which we can study with the NNCP we will discuss pure gauge SU(2) in 3D. This theory is thought to possess the same qualitative features as SU(3) in 4D, that is, confinement and the existence of a mass gap. Although it has the same qualitative features as the real theory, it is much more tractable for Monte Carlo studies and much can be learned from it. We would like to calculate the mass of the glueball i.e. the lowest bound state of glue for the pure gauge theory. It is also possible to determine the spin of this lowest excitation. This is done in the following way. Instead of just finding the 2-pt correlation function of the simplest operator - the average plaquette, one can find the 2-pt correlation for operators which transform non-trivially under the discrete rotational symmetry of the lattice. An example of such an operator is shown below.




The operator is gotten by adding the 4 plaquettes with relative phases: $plaq(1) + e^{i\pi/2}plaq(2) + e^{2i\pi/2}plaq(3) + e^{3i\pi/2}plaq(4)$. This operator only couples to states of specific spins. For the above case, the spins selected are: 1,5,9,... Since the mass of the spin 1 excitation is almost certainly much lower than that of the spin 5 excitation, finding a mass using this operator will give the mass of the lowest spin 1 excitation. Doing this for various operators gives us the masses of the lowest bound states of given spin.

We now estimate the cpu time needed to make an initial stab at this calculation. Suppose that a correlation length of 5 lattice sites is a reasonable approximation to continuum physics, and further that a 16×16×16 lattice (≈3 correlation lengths) is large enough. To calculate the mass gap to an accuracy of ≈10%, we need at least $10^6$ sweeps. This number comes from some

preliminary studies on our VAX. The time per gauge link update is 2.4 msec for our algorithm. $10^6$ sweeps on a $15^3$ lattice therefore implies 700 cpu hours = 1 month on a VAX! The NNCP would reduce this to the reasonable run time of 2 days for each value of the coupling. Again we would need to find the dependence of the mass gap on the coupling constant.

The lattice size mentioned above is rather small;one needs to store 48 bytes per site(see appendix 1) and the proposed machine(see section 3) allows ≈1000 sites to be stored in each processor.So we can investigate finite size effects with a 48×48×48 lattice although the running time would be prohibitive for a complete study with this lattice. However we believe that we can make an important initial study of these physics problems with the first NNCP proposed here. The next step (~1000 concurrent processors) should enable a complete study of this theory.

### 2.3.3. Fermions

One is ultimately interested in theories involving dynamical fermions. Several algorithms exist for treating fermions on the lattice [7-10]. These algorithms necessitate much more cpu time than for the case of bosonic fields but in several cases parallel processing is possible. We plan to implement such a fermion algorithm for 2 dimensional theories which allows one to gain a factor equal to the number of processors. Two particularly interesting models are the Schwinger model (fermions coupled to photons) and the Gross-Neveu model (massless fermions coupled via quartic interactions). The Schwinger model with massless fermions is exactly soluble; this gives us an opportunity to study different lattice techniques and see how well they reproduce exact results. The Gross-Neveu model is known to contain dynamical mass generation. It would be of interest to study the bound states.

## 3. Technical Features of the Computing Array

The first prototype of the computing array will consist of 64 processing units arranged in a 4×4×4 array.. Each processor will have the capability of asynchronous communication with both the host computer and its nearest neighbors. Communication between individual processors is kept asynchronous to prevent timing problems that would occur as the physical size of the system becomes large in future computing arrays. The processors on the edge of the array are connected to processors on the opposite side. (periodic boundary conditions)

The connections between processors are implemented in the form of two one-way mail boxes as opposed to a bidirectional data port. When a processor wishes to send a value to its neighbor it puts the value in the mail box and sets the flag. The receiving processor, which is running a program that requires the value removes it from the mailbox and resets the flag so that the sending processor may send a new value. Calculations that need communication of values between the processors run in two (possibly overlapped) phases. The first is when the processors are sending values to (and receiving values from) their neighbors. The second phase is when an individual processor has all of the information it needs to update its internal variables. The overlapping of these two phases of the computation is possible if the I/O through the ports connecting the processors is handled with interrupts as opposed to polling the flags on the ports.

Now that the general features of the architecture of the computing array have been discussed a description of the detailed characteristics of the individual processing unit is in order. The microprocessor that we intend to use in our first 4×4×4 prototype is Intel's iAPX86/20 two chip microcomputer. This microprocessor consists of the 8086-CPU, a general purpose 16 bit

microprocessor and the 8087 NPU coprocessor. The 8087-NPU adds 32, 64 and 80 bit floating point data types to the data types supported by the 8086-CPU. It also adds SIN, COS, EXP, LOG and SQRT along with +, -, * and / to the 8086 instruction set. The iAPX86/20 operated at 10 mhz is estimated to have approximately 1/4 -> 1/5 the power of the VAX in normal operations that are memory intensive. In applications that could take advantage of the 8 register stack in the 8087 the machine would be faster yet. A 64 element array of these units will have the computing power of approximately 15 VAX'es. The basic processor will have 128k bytes of programmable memory [1] (dynamic ram implemented in 64k or 16k by 1 bit chips) and 4k bytes of read only memory (EPROM). The read only memory will be used to house the monitor for the processor. See Fig. 1 of a block diagram for the individual processing unit. The arrows over the ports indicate the six nearest neighbor connections.

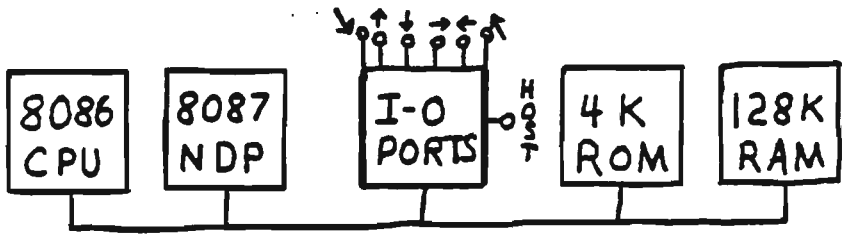



Fig. 1

The following table gives an approximate breakdown of the parts required to build the individual processing unit along with cost estimates . These costs are

[1] As shown in section 2 and appendix 2,this size memory is well matched to the 8086 CPU power to allow a good initial study of three dimensional gauge theories.

reasonably firm except for that of 8087-NPU which is not yet available in quantity. The construction costs are based on use of the High Energy Physics Shop. However we may well use an outside firm to massproduce the boards once the prototype has been built and tested.

| #req. | | item | cost each | extension |
|---|---|---|---|---|
| | | CPU | | |
| 1 | 8284 | clock generator | 5.00 | 5.00 |
| 1 | 8086 | cpu | 60.00 | 60.00 |
| 1 | 8087 | npu | 200.00 | 200.00 |
| 1 | 8288 | bus controller | 20.00 | 20.00 |
| 3 | 8283 | address latch | 6.00 | 18.00 |
| 2 | 8287 | data bus driver | 6.00 | 12.00 |
| | | MEMORY | | |
| 1 | 8286 | transceiver | 6.00 | 6.00 |
| 1 | 8205 | decoder | 3.00 | 3.00 |
| 4 | 8287 | transceiver | 6.00 | 24.00 |
| 2 | 8283 | data latch | 6.00 | 12.00 |
| 64 | 2118 | 16k ram chip | 2.00 | 128.00 |
| 2 | 2716 | 2k*8 eprom | 6.00 | 12.00 |
| | | I/O | | |
| 16 | 8212 | i/o port + decoding | 2.00 | 32.00 |
| | | printed circuit board, connectors | 100.00 | 100.00 |
| | | construction | 240.00 | 240.00 |
| | | TOTAL for one processor | | 872.00 |
| | | TOTAL for 64 units | | $56,000 |

The software interface to the host computer will be accomplished by cross compiling in C and down loading to the processing array. Bell Labs has a C compiler for the 8086 that will be our starting point. This cross compiler is expected to be available for $2,500. One must also consider the cost of the mother board and the controlling computer. We estimate that the mother board, power supplies and interconnecting cables should cost $2,500, the intermediary controlling computer (another 8086 with perhaps more memory) another $2,000. Finally the cost of the interface to the VAX unibus is $2,000. These

considerations bring the total hardware (+C compiler) cost of our processing array to $65,000.

As the 8087-NPU will not be available for perhaps a few months we intend to initially implement the floating instructions with software emulation that runs at about 1/100 the speed of the 8086-8087 combination. During this time we will be generating the software interface to the host which will either be the VAX 11/780 or the 11/45. By the time the 8087-NPU is available at a low price (perhaps at a price considerably better than the $200 estimate) we will be able to plug it in and remove the software emulator. This will have us up and running at a speed comparable to 15 VAX'es. With 128k bytes of ram on each processor we will have 8 megabytes of memory at our disposal in the system. Currently we are optimising the design described above. It seems best to implement the communication between processors with FIFO's rather than the 8212 of the first design. However this makes no no significant difference to the cost and as our problems are not I/O dominated the performance of machine is not materially affected by changes in this area.

**4. Other uses of the NNCP**

Although the current system can be justified for its effectiveness in lattice field theory computations alone, in order to justify building of larger machines in the future the array processor must have other applications. We would like to point out the fact that the proposed architecture is ideal for many problems in physics and engineering. The nearest neighbor connection arrangement can be used to advantage in solving any differential equation that can be reduced to a finite difference system. As discussed in appendices 2 and 3, one of us (E. Brooks) also intends to investigate using the architecture

to attack the eigenvalue problem for large sparse matrices. Matrices of this

sort occur in the matrix approach to Quantum Field Theory as well as many other branches of physics and engineering. The 4×4×4 computing array will be useful to any operating system which supports piped processes even after its use in physics calculations has become outmoded due to its "small" size. A pipe of N processes can be set up in the array by using the I/O ports connecting N processors in the array. In this way many CPU consuming tasks can be off loaded into the array by the host. This will reduce the overall system loading in the host reserving it to perform I/O functions to the disks and doing compiling. As the computing array can be used in this general purpose manner its useful lifetime will greatly exceed the relatively short lifetime (a few years) that it will have in theoretical physics calculations.

It should be emphasized that the importance of designing concurrent algorithms is not generally recognized in the physics community. We expect to try to see if there are better fermion algorithms optimized for concurrent processing. More generally the success of our machine may encourage scientists in other fields to reformulate their problems for concurrent machines. Problems requiring the solution of systems of coupled partial differential equations (fluid dynamics,weather prediction,geological survey gravity, plasmas,...) are amenable to parallel processing since the equations are local. The processors are placed on a spatial grid and as in our application, they each hold a block of neighboring variables. Then each operation of the NNCP will evolve the system forward in time. Taking gravity as an example, Larry Smarr's calculations of such things as gravitational waves emitted from the collision of two black holes are presently being limited by the huge amounts of cpu time needed to step the non-linear equations of gravity forward in time. Since these calculations are already taxing the Cray-1 at Livermore, we feel that further progress in this area may also lie in concurrent processing.

## 5. References

[1] K. Wilson, Cargese Lecture Notes (1979)

[2] M. Creutz, Phys. Rev. D 21, 2308 (1980)

[3] M. Creutz, Phys. Rev. Let. 45, 313 (1980)

[4] G. Bhanot, C. Rebbi, Nucl. Phys. B180 [FS2] 469 (1981)

[5] G. Fox, R. Gupta, O. Martin, S. Otto, "Monte Carlo Estimates of the Mass Gap of the O(2) and O(3) Spin Models in 1+1 Dimensions", Caltech preprint CALT-68-866(1981).

[6] J. Kosterlitz, D. Thouless, J. Phys. C.:Solid State Phys., 6, 1181 (1973)

[7] F. Fucito, E. Marinari, G. Parisi, C. Rebbi, Nucl. Phys. B180[FS2] 369 (1981)

[8] D. Weingarten, D. Petcher, Indiana University Preprint

[9] A. Duncan, M. Furman, Columbia University Preprint

[10] H. Hamber, Phys. Rev. D 24, 951 (1981)

[11] E.Brooks and S.Frautschi,"Scalars Coupled To Fermions in 0+1 Dimensions", Caltech Preprint (1982).

## Appendix 1: Computer Science Issues Relevant to the Nearest Neighbor Concurrent Processor Proposal

### A History of Similar Projects

A generalization can be drawn from a historical analysis of concurrent multiprocessor networks: the potential performance of the machine is proportional to the number of processors, and the fraction of that performance realized is related to the clarity of the algorithm programmed on the machine. The impressive projects are those with many processors with an extremely well understood algorithm. Unfortunately, many of these projects were less than successful, but most succeeded in implementing algorithms like those proposed here.

Two highly publicized multiprocessor projects have been undertaken at CMU, C.mmp and CM*. C.mmp was a 16 processor network consisting of PDP-11s, and CM* is an open-ended star connection, currently with 50 processors. Both projects involved programming various algorithms on the respective multiprocessors. In both cases **numerical analysis** problems were successfully implemented on the networks, but more ambitious problems such as distributed operating systems were less successful.

Alain Martin (Philips, Eindhoven, the Netherlands) implemented a 36-processor grid folded back in two dimensions to form an interconnection called a twisted torus. His work addresses very well the problem of distributing computations across an ensemble in such a way that each processor may have many concurrent processes eligible for execution, and the load is kept reasonably well balanced.

Browning [2] described programming a tree connected ensemble. The

[2] Browning, Sally, "The Tree Machine: a Highly Concurrent Computing Environment", PhD thesis, Caltech Computer Science, January, 1980.

variety of problems that were addressed included solution of np-complete graph problems, sorting, and vector and matrix operations. The number of concurrent processors was extremely large - a million processors was typical in her thesis.

H. T. Kung [3] at CMU has studied algorithms and corresponding communication structures for pipelined computational arrays that he refers to as systolic arrays. Similar work in this style, generalized to asynchronous data flow and using more stable and advanced numerical methods has been done by S. Y. Kung [4] and Lennart Johnsson [5]

Illiac IV was the most famous machine of this genera, but is distinguished from those described above by only having one instruction stream. The single instruction stream introduced so many problems in programming and communication that is it no longer technologically appropriate.

This proposal is for the construction of a machine similar in many ways to those described above, and will use well understood algorithms. The present proposal, while being for a modest 64 processors, can be viewed as a feasibility study for a machine with a million processors. The algorithms that are expected to be implemented on these processors are all of the **numerical** type: PDEs and matrix operations.

---

[3] Section entitled "Algorithms for VLSI Processor Arrays", in Mead and Conwey, "Introduction to VLSI Systems", Addison-Wesley, 1980.
[4] University of Southern California.
[5] Caltech Computer Science Department.

**The Performance of an Array Connected Network for Nearest Neighbor Problems**

The performance of concurrent processors can be evaluated in a variety of ways, such as fractional utilization of floating point capacity, or simply as a cost/performance ratio. The present proposal is for 64 concurrent processors, each with a significant amount of CPU capacity. It is easily seen that the potential floating point capacity is very great, and the cost is very small. It remains to be shown that this floating point capacity can be efficiently used.

The major use of this concurrent processor is expected to be solving differential equations on a uniform, rectangular lattice. The lattice will be represented by an array of values, with each processor representing many points of the lattice. There should be no argument that the lattice values in each processor should all be adjacent, to minimize communication between processors. [6)]

Within each processor there is only one impediment to continuous floating point operation, the necessity to communicate nearest neighbor lattice values between processors. We must develop a quantitative idea of the fraction of time that will be spent on this communication at the exclusion of numerical calculation.

Consider a two-dimensional lattice problem being solved on the proposed 64 processor machine. The entire lattice will consist of more than 64 points, say m points. In this case each processor will contain the lattice values for m/64 points. The iteration cycle for each processor will consist of communicating values for all boundry points to and from other processors and evaluating the iteration function m/64 times. If the processor represents a square array of

[6)] For example, if the entire problem were a 2-dimensional square, the processors should divide the entire problem into smaller squares.

m/64 points then the number of points along the edge is [7] $\sqrt{(m/64)}$. This is exactly the number of lattice values that must be transmitted along each communication pathway to an adjacent processor. Since the number of communicated values is related to the square-root [8] of the number of processing steps. As the size of each processor increases the ratio of processing steps to communications steps increases.

**A Performance Example**

Consider the proposed machine solving the physically relevent problem of SU(2) lattice gauge theory in three dimensions. Each lattice site contains three 2×2 unitary matrices which in turn can be represented as 4 real numbers each - we therefore have 12 real numbers per lattice site. The iterative step consists of replacing each matrix by an updated matrix, where this new matrix depends on 12 neighboring matrices (i.e. they are interacting through plaquettes). Since this is a three dimensional problem being implemented on a two dimensional array some mismatch will necessarily occur. We choose to flatten the space along one dimension into a two dimensional problem.

The proposed processors will each have 128k bytes of memory. Assume that about 100k of this is available for storing lattice points. Since we will represent each real number as 4 bytes and there are 12 numbers per point, each lattice point will require 48 bytes. Each processor will have a storage capacity of a little more than 2000 lattice points.

The entire problem can then be a 48×48×48 array of lattice points. Each processor would contain a 6×6×48 slice of the total solid. Of the 128k bytes in

[7] Ignoring the four corner points.
[8] In general, for k-dimensions the number of communicated values varies as the (k-1)/k power.

each processor, 82944 will be used.

Assume that the time to transfer one matrix to an adjacent processor is 160 $\mu s$ . [9] Now consider points on the surface of the 6×6×48 slice of the total solid. To update the matrices associated with these points requires, on the average, the communication of 6 matrices per lattice point. The number of points on the surface is 4×6×48 and so the time to transfer these values will be about 1.1 sec .

The computation performed on each lattice point consists of 528 multiplications and 430 additions. Assume 20 $\mu$S average for operand setup and a multiply or addition and the computation time per lattice point is 19 msec. This time is about three times slower than that on a VAX (coming from actual timings of this program on a VAX), so the 19 msec estimate is at least roughly correct. Total update time for the entire array is 34 seconds.

Summarizing, the machine will be able to preform an iteration on a 48×48×48 array of 2×2 matrices in approximately 34 seconds. The 34 seconds will be all computation except for 1.1 sec of interprocessor communication. In this example the fractional floating point utilization is about 97%.

**Laplace's Equation**

The previous example yielded attractive results due to the large amount of time required to manipulate matricies with complex entries, and the number of lattice points in each processor. We will consider another nearest neighbor problem where the efficiency will not be aided in this way: solving Laplace's equation on a small square lattice of 8×8 points. Each lattice point consists of a single real number, or 4 bytes.

---

[9] 10 $\mu s$ to service a port for one byte times 16 bytes per matrix.

The time to transfer one lattice point to an adjacent processor is 160$\mu$S [10] The number of lattice points to transfer is 1, or the total time will be 160 $\mu$S per iteration.

Each iteration consists of replacing each lattice point with a simply weighted average of its four neighbors. The weights are 1,1,1,1, and 4, and hence do not require any real multiplications. The updating can be done with 6 floating additions. Again assuming 20 $\mu$S for an operation we find an iteration will take 120 $\mu$S. CPU utilization is now under 50%.

Fifty percent utilization of a processor, although somewhat wasteful is better than average for multiprocessor programs. Of course in any practical problem(whose size would warrant use of the NNCP) one would have several sites per processor and much greater efficiency.

**Other Potential Architectures**

Why was an array network chosen over other networks such as a tree, a bus, a hypercube [11] , or a simple VNM [12] ? We will briefly discuss each of these architectures.

A tree network is less expensive and well suited to this sort of computation, but is not as good as an array. A tree is less expensive because each processor has, on the average, two connections to other processors [13] The organization of the tree causes a bottleneck at the root, however.

This effect of this bottleneck can be analyzed by calculating the number of values that must be transmitted through the root node. Recall that an array

[10] 10 uS to service each port for one byte, 4 ports, 4 bytes per lattice point..
[11] See appendix 5 for a discussion of the close relationship between the Hybercube and NNCP architectures.
[12] Von Neuman machine, or conventional computer.
[13] Leaf processors have one connection, all others have three.

processor must transmit lattice values for all lattice points on the boundary of its area. This number was the square root of the number of points in that particular array element. The root processor has similar behavior: the root node must transfer all boundary points of its left subtree to its right subtree, and vice versa. Again, the number of lattice points is the square root of the number of lattice points in the subtree. Unlike the array, however the size of a subtree is half the size of the *entire problem* , not 1/64th or an amount determined by the number of nodes in the tree.

The number of values transmitted through the root node of an equivalantly sized tree network would be about 8 times as large as between elements of an array. This would increase the communications overhead to an intolerably large amount in some cases.

A bus connected network is even worse. In a bus connected architecture all the values transmitted on the array network are transmitted, but on the same bus. The resulting traffic on that bus would be 6×64 times as large as on any of the array connections.

Other networks are known, but less well understood. Hypercubes have the advantage of offering a maximum interconnection distance between processors of log n, n the number of processors. While this is attractive for some problems, it is not well understood for nearest neighbor problems. (The maximal interconnection distance in the array processor, for nearest neighbor problems, is 1.)

An array network appears to be near optimal for these sorts of problems. Each element of the array has a processing unit that is used at close to 100% efficiency, as is the processing unit of a VNM. The total amount of memory in the entire array is about the same as that in a VNM solving the same sized problem.

**Appendix 2: Application of the NNCP to Quantum Mechanics**

Virtually every field of physical science has applications of quantum mechanics. Among these fields are atomic and nuclear physics, chemistry, quantum electronics, and recently even experimental gravitation. Of course we can't forget the theory that is considered (at least by field theorists) to be the root of all of this, quantum field theory.

The basic program in solving problems in quantum mechanics can be reduced to finding the eigenvalues and eigenvectors of a hermitian operator (the hamiltonian) that is dictated by the physics.[11] The energies of bound states are given by eigenvalues of the "discrete" part of the eigenspectrum of $\hat{H}$. Scattering amplitudes are calculated from the eigenvalues and eigenvectors of the "continuous" part of the eigenspectrum of $\hat{H}$. In many cases a particular problem can be adequately solved by the combination of analytical solution and perturbation methods but it seems that some quantum systems are resistant to solution by any method but numerical calculation. Any doubts of the utility of numerical solution of quantum systems can be immediately dispelled by the success of the worlds first digital computers in Hartree Fock calculations of the spectrum of many electron atoms.

The basic problem blocking the progress of digital calculations is that the size of the matrices that one wants to diagonalize in quantum systems of current interest have become much too large for our digital computers to tackle. This is particularly exemplified by the size of matrices that occur in investigations of quantum field theory on the lattice. The matrices grow like $Q^P$ where $Q$ is the number of possible configurations of quantum numbers on a particular lattice point and $P$ is the total number of lattice points. As one can see the size of the matrices involved grow exponentially with the accuracy of the approximation. It is quite obvious that current digital computer technology will

never be able to proceed too far with matrix sizes that grow in this manner and that the only possibility for success is to have computers that solve the problems in a way that can use parallel processing. In this way one can build a large computer that can solve the large problem in a finite amount of time. We will show that the NNCP structure will provide very high performance for attacking problems of this type.

Numerical solution of a quantum mechanical system consists of three steps. These are: finding a finite set of basis vectors that will adequately span the eigenvectors of the hamiltonian that one wants to diagonalize, calculation of the matrix elements in the chosen basis, and then diagonalization of the resulting finite dimensional matrix. The step of finding a finite set of basis vectors and calculation of matrix elements of the hamiltonian operator is trivially done on array of independent processors. The question of solving the matrix eigenvalue problem on the NNCP must be examined in detail as this task will require communication between processors. This examination will be carried out in the following appendix and will lead to the conclusion that the NNCP architecture will be very efficient for solving quantum systems numerically.

**Appendix 3: The eigenvalue problem on the NNCP**

The previous appendix has indicated the importance of the matrix eigenvalue problem. In this section we will show that the NNCP structure is well suited for the solution of the eigenvalue problem by the power method. Solving the matrix eigenvalue problem by the power method is achieved by repeatedly multiplying a trial vector by the matrix that one wants to diagonalize. If the vector is a mixture of the eigenvectors of the matrix the component of the eigenvector with largest absolute eigenvalue will grow with a rate that is controlled by the ratio of the largest absolute eigenvalue and the next to largest absolute

eigenvalue. The eigenvectors belonging to eigenvalues of lower absolute magnitude can be obtained by removing the component of the eigenvector already found (this is called purification) and iterating this process of multiplication and purification. Using this procedure one can find all of the eigenvectors and eigenvalues of a matrix to any desired accuracy. The computer operations needed are multiplication of a matrix and a vector, the vector dot product and vector arithmetic.

Below is a table of the number of time units required for the serial computer to do these tasks. The dimension of the vector space is denoted by $M$, and a "time unit" is defined as the basic combination of multiplies, adds, fetches and stores that must be iterated sequentially in time in order to accomplish the calculation. This basic time unit will be different for the various vector operations we wish to perform but what we wish to use as a source of comparison is the rate at which the number of time units required for an operation grows as we change $M$ the dimension of the vector space. Indeed certain array processors pipeline these operations in such a way as that a time unit is always one machine clock cycle. We will show in this section that the NNCP is well suited for matrix and vector operations that would be used to solve the matrix eigenvalue problem.

| MATRIX OPERATION | TIME UNITS FOR COMPLETION |
|---|---|
| $A_{ij} = B_{ik} C_{kj}$ | $M^3$ |
| $Y_i = A_{ij} X_j$ | $M^2$ |
| $Y_i = A_{ij} X_j$ sparse matrix | $Mm$ |
| $A_{ij} = cB_{ij}$ | $M^2$ |
| $Y_i = cX_i$ | $M$ |
| $c = Y_i X_i$ | $M$ |
| $Z_i = X_i + Y_i$ | $M$ |

($m$ is the average number of nonzero entries per row)

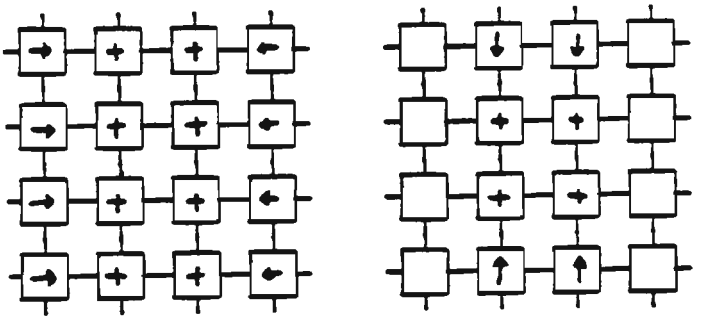
As can be seen from the above table all of the operations require some power of M to complete. Since serial computers are not likely to achieve speeds that are much greater than the Cray, time limits put a definite limit on the dimensions of vector spaces that we can work in. These limits are currently of order $10^3$ to $10^5$ depending on matrix sparseness. The ability to extend the size of the matrices that we can diagonalize on a computer will be necessary to progress in solving quantum systems using the matrix approach. It is quite evident that serial computers are doomed to failure as far as applications of the matrix formalism to quantum field theory is concerned. This is due to the fact that the dimensions of the matrices involved grow exponentially with the accuracy of the approximation. The NNCP structure is well suited to those matrix operations that are used in solving the eigenvalue problem by the power method, and in fact is very effective on the type of matrices that occur in field theory.

The feature of the eigenvalue problem that makes the NNCP useful in the solution of the eigenvalue problem is the fact that the vector operations that are performed are repeated iteratively in the algorithm using the same matrix and the same set of vectors. Because of this the data can be kept distributed throughout the memory of the NNCP and the algorithms used allow all of the processors in the array to be operating on various parts of the data at the same time. The NNCP will be efficient in any problem that can keep the data distributed through its memory, and any problem that would need the host to transmit a huge amount of data to and from the NNCP would find that the bottleneck between the NNCP and the host would dominate the time required to do the computation.

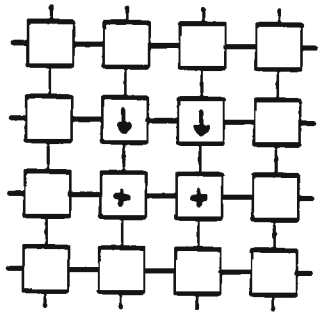
The previous table indicates the performance of a serial computer at these tasks, and we now consider the possible algorithms that can be used on the NNCP to complete these computations and their relative performance. Consider first the case of vector arithmetic, i.e. adding vectors to vectors and multiplying vectors by scalars. Suppose we have a computing array with the number of interconnected processors $N$ equal to the dimension of our vector space $M$. Then the efficient method of storage is to store the i'th element of each vector we wish to have in the machine in the memory of the i'th processor. The scalars would be stored in the host that can communicate with all of the processors in the NNCP at the same time. Since each processor in the array may add two vector elements independent of the of the other processors the number of time units for a vector arithmetic operation is always 1 if the array size is kept the same as the dimension of the space. If the number of processors in the array $N$ is smaller than the dimension of the space $M$ the number of time units for a vector arithmetic operation is $\frac{M}{N}$ (one stores $\frac{M}{N}$ elements of each vector in each processor). The improvement over the serial computer in this case is equal to

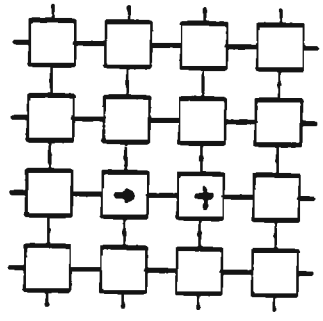
the number of processors in the NNCP. The efficiency of processor use in the NNCP is high.

Consider now the performance of the NNCP with a low cost addition in wiring for doing the vector dot product. We already have the restriction that each vector is distributed in the memory of the NNCP in order to do vector arithmetic. Therefore in order to do a dot product the host commands each processor in the array to multiply their element of the two vectors and then the sum over the array of the products must be evaluated. Consider the algorithm for evaluating the sum on a 4×4×4 subcube of processors depicted in the diagrams below. An arrow indicates passing a value to a neighboring processor, a plus sign indicates the processor adding the passed value to its internal one.



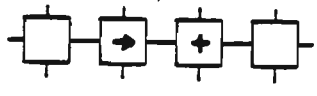
T=1 T=2

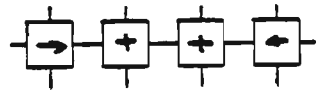




T=3 T=4

And finally along the third dimension.





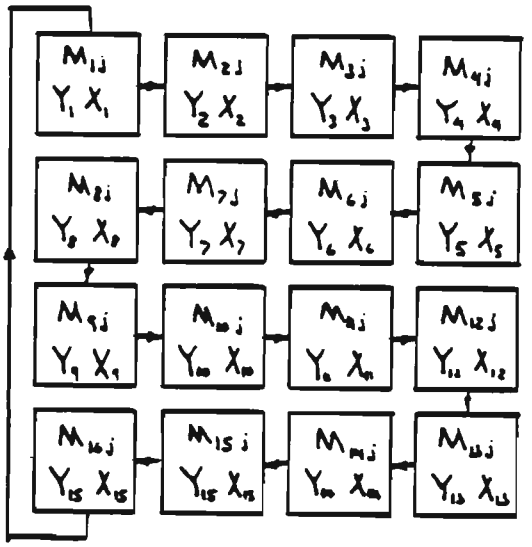
T=5 T=6

As one can easily see we have implemented the binary tree sum in the 4×4×4 [14] computing array. In order to get this binary tree sum to continue we only need to add interconnections to the nearest neighbor configuration in such a way that we can build a 4×4×4 block of 4×4×4 blocks and so on. This can be done

[14] Note that in appendix 5 we point out that this subcube is isomorphic to a K=6 hypercube which is well known to allow such logarithmic algorithms.

by adding an extra set of ports to one out of 64 processors in the NNCP(and then adding and extra set of ports to one out of every 64 of the processors that we just added an extra set to etc.). This incurs an extra cost that amounts to less than one percent of the total cost of the NNCP. It's essentially free. With the binary tree sum option added to the NNCP the number of time units needed to do a sum over the array is $\log_2(N)$. If again we have $N<M$ processors in the (NNCP+binary tree sum option) the number of time units to do a dot product is $\frac{M}{N}+\log_2(N)$. A performance increase that is proportional to the number of processors is achieved. The only time that processor efficiency falls significantly below 100% is when one is on the tail of the binary tree sum. This loss of efficiency is a low price to pay for the $\log_2(N)$ elapsed time for doing dot products. The only operation left to consider is the matrix vector product.

We consider first the matrix vector product $Y_i=H_{ij}X_j$ where M is a full matrix (that is all the matrix elements are nonzero). The dimension of the space is M and assume for the moment that we have $N=M$ processors in our NNCP. As we are already forced to dedicate one processor to each element of any vectors, a reasonable way to store the matrix is to put the i'th row of a matrix in the processor that holds the i'th vector element. The storage scheme is depicted for the case of $M=N=16$ in the square array in the diagram below. The arrows on the connections show the direction of data flow in the "Bucket Brigade" algorithm that will be described below.



The operation $Y_i = H_{ij} X_j$ is accomplished by circulating the elements of X around the NNCP array in the directions shown by the arrows on the connections in the above diagram. At each shift each processor multiplies the newly arrived element of X by the appropriate element of H and sums to it's stationary element of Y that it had initialized to zero at the start of the computation. In this way the vector Y is computed in M operations (16 for this case) and the vector X is left in its original storage position at the end of the calculation. For a case in

which the dimension of the vector space M is larger than $N$ the number of processors in the NNCP, $M/N$ rows of the matrix are stored in each processor. The i'th row of the matrix is stored in the same processor as the i'th vector element. The number of time units required to do the computation in this case is $\frac{M^2}{N}$ with high processor utilization.

It is important to consider how well one can do with the product $Y_i = H_{ij} X_j$ when H is a sparse matrix. It is usually the case that when the matrices one wants to diagonalize get large they also get very sparse. An example is the matrices that occur in field theoretic problems. Remembering that the serial computer will do the sparse matrix-vector multiply in time units that grow like $Mm$ where $M$ is the dimension of the space and $m$ is the average number of nonzero elements on a row of the matrix we already know that the Bucket Brigade algorithm can do the multiply $m$ times faster that a serial machine. But this is for a $M$ processor system and in a M processor system each processor will spend most of its time passing vector elements without multiplying and summing. For very sparse matrices the efficiency of this algorithm will be very poor indeed.

The loss of efficiency for sparse matrices does not happen unless the number of processors $N$ is larger than the mean number $m$ of nonzero elements per row. Thus each processor stores $\frac{M}{N}$ rows or a grand total of $\frac{Mm}{N}$ nonzero matrix elements. The bucket brigade algorithm will be efficient as long as each new vector element $X_i$ passed to a processor is multiplied by at least one matrix element. On average each processor has $\frac{m}{N}$ elements to be multiplied by each new vector element arriving. Therefore we need $\frac{m}{N} \geq 1$ for the bucket brigade algorithm to be satisfactory without any modification.

In the case where $m << M$ the loss of efficiency is caused by the processors having to shuttle along all $N$ vector elements even though a particular processor might not need a particular vector element due to the fact that the matching matrix element is zero. In this case it is better to send the vector elements along the shortest path to the processors that need them. This is accomplished by sending the processor address along with the vector element so that each processor on the path can check the address and push the vector element along the shortest path to the destination processor. An estimate for the time required to complete the transfer of the vector elements can be made as follows. Let the average number of processors that a vector element must travel through be $D$. The total number of transactions that must occur is $mMD$. The total number of processors available to do the transactions is $N$. If no bottle necks occur in the data paths the time required to complete the data transfer will grow like $\frac{mMD}{N}$ For our 3 dimensional array the average distance traveled $D$ will be bounded by $N^{1/3}$. The time for a matrix-vector multiplication on our NNCP to be $\frac{mMN^{1/3}}{N}$ for a 3 dimensional array and $\frac{mMN^{1/2}}{N}$ for a 2 dimensional array. If the hypercube architecture were used $D$ could be decreased to $\log_2(N)$ which would be a substantial increase in performance for large $N$. The cost of the copper wire that connects the processors and physically organizing the processors in the hypercube architecture would be prohibitive however. This process of mailing the vector elements directly is very useful when the matrix is hermitian as the processor that needs to send its vector element can find out where to send it by examining its own matrix elements and does not have to wait for a request from the receiving processor. For small $m$ this method will greatly improve the efficiency of the processors in comparison to the bucket brigade algorithm, but is a long way off from the time of order $m$ that is possible if all of the processors have global connections (an option too expensive to consider for

large machines).

There is one kind of matrix that can use the direct mailing method of communicating the vector elements very efficiently. This is a matrix that has a very small bandwidth. Let the band width be $q$, then the distance over which the vector elements must be transmitted shrinks to $q^{1/3}$ and the time required to do the multiplication is $\frac{mMq^{1/3}}{N}$. When the band width gets close to the number of nonzero matrix elements per row the bucket brigade algorithm becomes more efficient. In fact the matrices that occur in the analysis of field theoretic problems have a very small bandwidth and the NNCP will be effective in studying field theory using the matrix approach. In the table below the number of time units required to do the various matrix operations on the NNCP is summarized. The notation is that which we have adhered to in this section (M is the dimension of the space, N is the number of processors of the array, m is the number in nonzero elements in a row, q is the matrix bandwidth) .

| MATRIX OPERATION | TIME UNITS FOR COMPLETION |
|---|---|
| $Y_i = H_{ij} X_j$ (full matrix) | $\frac{M^2}{N}$ |
| $Y_i = H_{ij} X_j$ (sparse matrix) | $\frac{mM(min(q,N))^{1/3}}{N}$ |
| $Y_i = cX_i$ | $\frac{M}{N}$ |
| $Z_i = X_i + Y_i$ | $\frac{M}{N}$ |
| $c = X_i Y_i$ | $\frac{M}{N} + \log_2(N)$ |

By comparing this table with the table for the serial machine it is very evident that the NNCP is optimized for vector arithmetic and dot products, It also

gives very good performance for the full matrix-vector product considering the fact that the best one could do is a time of order $\log_2(N)$ if one had a machine that used the binary tree algorithm to evaluate all $N$ dot products at the same time. For the sparse matrices of small bandwidth the NNCP is optimum as the time required to do the matrix vector product gets very close to $m$ the number of nonzero elements on a row.

**Appendix 4: Considerations in developing larger machines**

Although the 4×4×4 NNCP array that we are proposing will have 8 megabytes of memory and the computational performance of 15 VAXs in problems that can take advantage of the nearest neighbor architecture, the 4×4×4 NNCP array can be regarded as a feasibility study for much larger machines that can be built using current technology. Since we are proposing the current project with the possibility of building much larger machines in the future in mind, some comments on possible technical limits to which this architecture can be pushed are due. This section considers the technical problems that one might expect to encounter as one generalizes the architecture to large arrays.

The first problem that one expects to encounter is actually designing, building and servicing arrays that might be 100 processors on a side. The only possible way that such a machine could be built is to have the physical layout of the machine exactly follow the 3 dimensional architecture. If this is done the wiring will remain simple and one will avoid the basket of snakes that would be inevitable if the conventional backplane and ribbon cable wiring that is used for current computer systems were used. In order to implement the 3 dimensional physical architecture in a serviceable way one would have to build the machine in modular subunits of perhaps 4×4×4 arrays and connect these subunits with cables of about 4 feet long so that there would be room for maintenance

personnel to get in to the array and remove defective modules quickly. The bad module could then be taken to a shop that could replace the individual processor that had developed a failure. Each module would have a connection to the host, power connections, coolant connections and connections to the 6 nearest neighbor modules.

One must also consider the effect that the physical size of the processor will have on the operating speed of the array. Since the lines connecting the neighboring modules will be short the speed of the array will not be hampered by communications delays as the size of the array is increased. Even the lines that connect the ends of the array can be made short for large arrays by arranging the array in a layered doughnut configuration. However this won't be necessary until one builds arrays of around 1 million processors (100×100×100) and the physical dimensions of the array exceed a few hundred feet. This fact that the operating speed of the array will not suffer for large array sizes is a feature that only the nearest neighbor connection has. Communication with the host will take longer time as the size of the machine grows but since communication with the host is done rarely a delay of a few tens of microseconds is allowable.

Another consideration is reliability. If one builds an array of $10^6$ processors and each processor has a mean time between failures of 10 years , the NNCP will fail on the average every 5 minutes. This is unacceptable and one will require hardware redundancy in the basic unit so as to automatically find errors and allow the system to continue operating once an error has been detected.

**Appendix 5: The Relationship between Hybercube and NNCP architectures in Two and Three Dimensions**

For the benefit of physicists, we will first define the K'th Hypercube architecture as an array of $N=2^K$ processors where it is convenient to label each processor by a K digit binary number. Then the Hypercube architecture is realised when every processor is connected directly to all other processors whose binary representation differs in one and only one digit. In this architecture,each processor has K connections and the mean (maximum) communication time between any two processors is K/2(K). As usual,we call NNCP the architecture with nearest neighbour connections in a specified number of dimensions d.Each processor has $2d$ connections in an NNCP.

One can easily show that:

(a) The K=4 Hypercube is isomorphic to the 4×4 d=2 NNCP. Each processor has four connections.

(b) The K=6 Hypercube is isomorphic to the 4×4×4 d=3 NNCP. Each processor has six connections.

(c) This result can easily be extended to higher dimensions (relating the K'th hypercube and the 4×4 ... ×4 NNCP in K/2 dimensions) but this does not seem interesting as the important features of the NNCP are seen in 2 and 3 dimensions. Only in these cases is it relatively easy to build with short physical communication paths and of course the (currently known) problems only need two or three dimensions.

In fact it is quite feasible and probably most flexible to build arrays with the hypercube architecture when the number of processors is ≤1000. However in this proposal we have emphasized the NNCP architecture as one can realistically plan on building extremely large arrays of this type (appendix 4). The NNCP is quite adequate for the local problems which are of greatest interest to us

although in matrix problems sketched in appendix 3, we see the advantage of the hypercube.Even in the latter case the NNCP is suprisingly powerful!

For solving two dimensional field theories, we will view our array as a d=2 NNCP. For this purpose, the following maps are of interest:

The d=3 4×4×4 NNCP contains the d=2 8×8 or 4×16 NNCP.

The d=3 8×4×4 NNCP contains the d=2 8×16 or 4×32 NNCP.

The d=3 16×8×8 NNCP contains the d=2 32×32,16×64,or 8×128 NNCP.

The useful relationships between NNCP and Hybercube architectures for 64-->1028 processors are:

The K=6 Hypercube is isomorphic to the d=3 4×4×4 NNCP.

The K=7 Hypercube contains the d=3 8×4×4 NNCP.

The K=8 Hypercube contains the d=3 8×8×4 NNCP.

The K=9 Hypercube contains the d=3 8×8×8 NNCP.

The K=10 Hypercube contains the d=3 16×8×8 NNCP.

In the above "contains" means that the first architecture reduces to the second when some of its communication channels are ignored.