Nearest Neighbour Concurrent Processor

E. Brooks, G. Fox, R. Gupta, O. Martin, S. Otto

Caltech High Energy Physics Group

E. DeBenedictis

Caltech Computer Science

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.Field theories involve an infinite number of degrees of freedom labelled by a vector <x> and some discrete index.For instance in QCD,<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 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 succesful becauce 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 our numerical algorithms for field theories have just the structure necessary to allow straightforward application of parallel processing.The first step is to replace the continuous label <x> by 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 (eg masses) have been calculated in any field theory even for d=2.Now we have to calculate integrals of the form

INT (PHY(fi) WGT(fi) d(fi1) d(fi2) .... d(fiM))

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

WGT(fi) is a positive weight function which contains the dynamics.In the statistical physics spin analogy,

WGT(fi1,fi2, ... fiM)= exp(-H/kT)

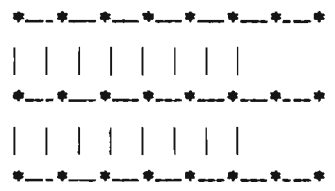
while in the quantum field theory case WGT(fi1,fi2, ... fiM)=exp (-S) where S is the action.

Finally the function PHY(fi1,.. fiM) contains the physics .For instance choosing PHY to correspond to correlation function between the fields at different sites,this function falls off exponentially wi'.h distance between the sites and one can find the mass spectrum of theory from the rate of fall off. Other choices of PHY give different observables and in fact there are better choices of PHY to find the mass spectrum.

This multidimensional integral is solved by monte carlo techniques.Each set of values of the integration variables is called a sweep.This involves specifying a value for each of the M fi's.THere are some clever but simple methods for choosing the fi's according the weight function WGT (called importance sampling).If the fi's are chosen in this way,our integrals just become

Average over sweeps of PHY(each sweep).

To get good answers one needs many sweeps;the current studies are too limited to estimate how many for realistic problems (see discussion in section 2) but at least a million sweeps are probably necessary for interesting problems. As one also needs N~100(about 10 sites inside a particle and a world whose total extent is also about 10 particles!),the number of sites needed is very large ;say 10^9(for the product of L*N^d) for QCD(Quantum Chromodynamics,the gauge theory of quarks and gluons) in four dimensions. The calculation of a sweep according to the importance sampling algorithm is an iterative technique starting at the previous sweep.(ie at previous point in M dimensional fi-space) This technique calculates a new value of fi at each site solely based on the values of the nearest neighbour fields. This is the critical feature of algorithm but it is very general as its validity only depends on locality of interaction;something for which there is a lot of theoretica predjudice 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 to solve QCD is around 10^6(#sweeps)*10^9(#variables)*1000(#operations/site) On a VAX an operation takes about 3mus and so one needs about 10^5 years to complete problem. However the independence of the field generating algorithm at each site and the fact that you only need to know about nearest neighbours 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 neighbour connections.



In the above, - and | denote connecting lines and * a computer. The normal periodic boundary conditions imply that the computers at each end 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 of each cycle,transfer its new site value to its neighbours. 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 neighbouring sites (eg 2*2*2*2 cube of values for B=8 etc.). For QCD one can imagine 10^6 computers each with 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 community would consider this a good investment if one can show there was a good chance of solving QCD. The above estimates suggest that the factor of ~10^6 improvement over a normal sequential computer should allow one good possibilities 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.Initially we need to investigate a bunch of

theories and many choices of PHY to refine our techniques.Thus it is appropiate to use a general purpose microprocessor.

We propose an initial system of 64 computers arranged either in a line or a small (8 by 8) square array.This will be interfaced to the unibus of a PDP11 or VAX which will initiate our machine and accumulate results.Programs for the computers will be written in C,compiled on the PDP11/45 and down loaded into microcomputers.With this system,we will have the power of about sixteen VAXs for about one fifth 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.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 succesful,the next step would be about a 1000(30 by 30 say) 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.

Technical features of the array

The first prototype of the computing array will consist of 64 processing units arranged in a square 8*8 array. Each processor has the capability of communicating asyncronously with the controlling computer and its nearest neighbors. The edges of the computing array are connected to give periodic boundary conditions (fig 1). The ports between the processors are operated asyncronously so that each processor can have its own internal clock to prevent timing problems as the physical size of the computing array becomes large.

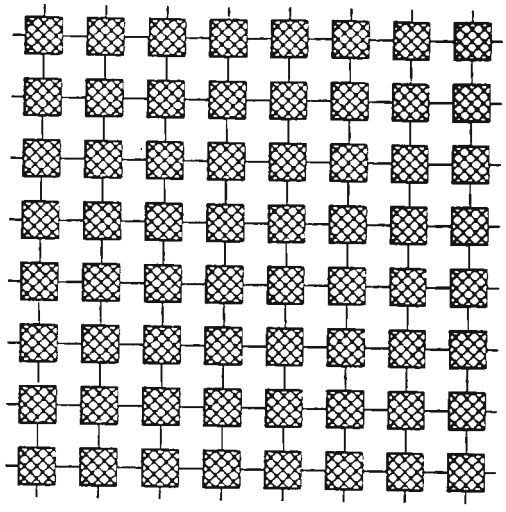


fig 1

The basic mode of operation is one in which each processor carries out its own program independently. Each independent process is loaded and initiated by the host computer. This type of operation can be used in situations where the host is very loaded and would benefit from offloading small jobs that can run independently. A good example of this case would be offloading of graphics pipes, document processing and small numerical jobs that are known to load our system down during normal working hours. Since the intention here is to create a general purpose machine, once the array processor has become superceeded by a larger unit for physics calculations it will still be very useful for the offloading of small jobs and therefore will have a long lifetime.

In large calculations such as lattice calculations in field theory and large matrix diagonalizations that occurs in all branches of physics and engineering, the entire computing array would be used at one time. In lattice calculations two basic operations need to be performed. The first is the generation of a new lattice configuration or "update". The second is the sum over the lattice of some value computed at each site. The cooperation between processors that is required by these tasks is effected by using handshaking on the ports that connect the processors. In the case of lattice updates each processor needs to know the values of certain variables on its nearest neighbors. Thus the first section of the program in each processor is the call to a subroutine that puts the apropiate values on the output ports and receives appropiate values on the input ports. As each processor completes this task, it becomes free to update the values of its internal variables. When is has completed the update a signal is sent to the host so that the host may initiate a new process on the computing array.

In the case of taking a sum over the lattice a different kind of cooperation between the elements of the computing array is needed. Consider the square 8*8 array (fig 1). The first column of processors must send their value to the right and second column waits for the variable to be passed. When the values are received the second column adds them to its internal variables and passes the partial sum to the right. This proceeds along the columns until the partial sums have appeared in the last column. Once this has happened the sum is completed along the last column and the total sum is delivered to the host by the corner processor. It must be noted that this is the most simple summing method and not necessarily the optimum one. One could pipe line the sum through the array if the next update did not depend on the value of some sum in the current lattice configuration. In cases where the sum is needed for any further computation to proceed on the array it is possible to add a very few extra interconnections and do the sum in a binary tree. This results in a sum occuring in a time that grows as the log of the total number of processors as opposed to the cube root of the number of processors (considering the three dimensional array). The increase in cost of adding the binary tree summing feature amounts to only a few percent of the total cost of the computing array.

We intend to base our computing array on Intel's 8086 microprocessor. This microcomputer has a 16-bit wide data path and can directly address 1 megabyte. We have chosen this unit as it operates at 10 mhz and has a companion numerical data processor (Intel's 8087) that will be introduced this year. When the companion NDP is added the instruction set is enhanced to include floating point instructions in 32, 64 and 80 bit precision. The floating instructions include sin, cos, exp, and log along with the four regulars. The 8087-NDP also adds an internal stack of eight individually addressable 80 bit registers. This stack increases throughput as it reduces the load on the 16 bit data buss that connects to memory.

The combination of the 8086-CPU and the 8087-NDP result in a computer with about one fifth of the power of the vax for numerical computation that is memory intensive. In computations that can take advantage of the eight register stack the processor will be faster yet. A 64 element array of these units will have the computing power of approximately 15 vaxes. Each individual processor will have I/O ports to connect it to both the host computer and the neighboring processors. The basic processor will have 128k bytes of programable memory (dynamic ram) and 4k bytes of read only memory (EPROM). The read only memory will be used to house the monitor for the processor. See (fig 2) for a block diagram for the individual processing unit.

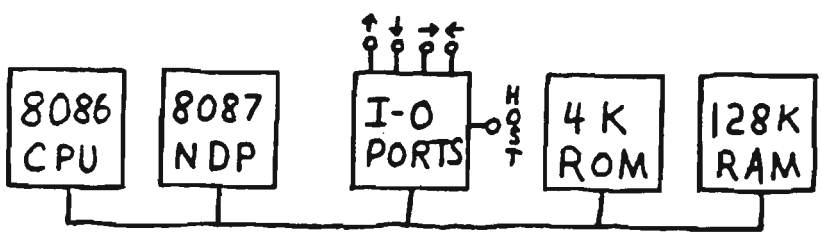


fig 2

As the 8087-NDP will not be available till later this year we intend to initially implement the floating instructions with software which runs at about 1/50 the speed of the 8086-8087 combination. During this time we will be generating the software interface to the host computer which will either be the HEP vax or the 11/45. By the time that the 8087-NDP will be available we will be able to plug it in and remove the software emulator and be running at a speed comparable to 15 vaxes. With 128k bytes of ram on each processor we will have a machine with 8 megabytes at our disposal and will be able to do some interesting physics computations. It is estimated that we can build these individual processors for about $500 each making the hardware cost of the computing array $32,000. The actual cost of building and bringing the machine into service (including labor costs) will probably be around $50,000.

9/17/81

DEBENEDICTIS

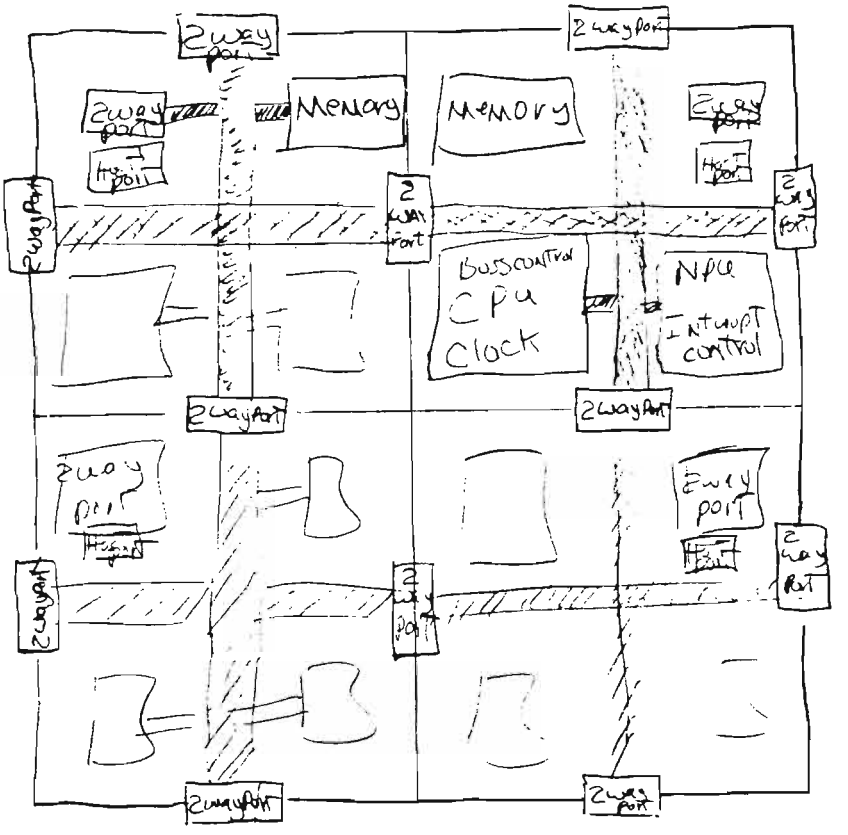
A note on the physical construction of the NNCP

E. Brooks

Current computers benefit well from the card cage-backplane type of construction as all cards need to be connected to the system bus and power. the backplane provides this function in an efficient manner. The NNCP on the other hand has a nearest neighbor connection system of ports that the processors use to communicate with each other along with the usual connection to the host that could be served well by a backplane. Since the communication between the processors is the factor which determines the speed of the machine the wiring rats nest which would occur if the NNCP were physically constructed in the usual card cage system must be avoided. Although wiring the 4*4*4 cube with the usual construction techniques is feasible (The wiring problem is getting very severe however) if one considers building larger machines a simple and elegant solution to the problem needs to be found. Fortunately there is such a solution to the problem. The wiring rats nest and long cables can be avoided if one builds a physical architecture that follows the the 3-dimensional logical architecture. This is indeed the reason for stopping at a 3-dimensional logical architecture. This means building a 3-dimensional machine and abandoning the currently used methods of constructing computers. I show here that such a physical architecture is possible and that use of it removes the objections that people have against using standard TTL signals on cables by removing or shortening most of them. The few long cables that are left can be handled by special drivers since the long cables represent a small percentage of the connections in the machine. Since the fraction of long cables is small the cost incurred by the use of special drivers for them is small.

Consider the 4*4*4 cube NNCP computer as being built of 4 planar levels each of which is a 4*4 array. Each level will require at most 3 inches of vertical space including space which is a generous overestimate. Even with this spacing the NNCP is only one foot thick. This space is thicker that the usual card cage spacing to allow for distributing cable connections over the surface of the board. Notice that this is the first break with traditional construction. Usually boards are slid into the card cage with all connections at the edge of the card. The reason for the connectors distributed over the board surface is to be able the make the connections to the boards above and below a given board with a very short (3 inches) cable. Cables that are this short can be wired directly to the buss on the board without noise problems.

As it is not feasible to etch large enough pc boards to contain the whole 4*4 for such a small project as the 4*4*4 cube I propose that the 4*4 surface be constructed from 2*2 processor array boards. In larger machines it would be possible to use 4*4 processor boards which would further reduce connector and cable cost. The logical (physical) layout of each board is shown in the following figure. The individual processors will require one square foot of board space at most so the 2*2 array can be constructed on a board that is 24*24 inches or less.



The desirable features of this type of construction are that all cables except those that make up the cube boundary connections are 3 inches or less in length. The cables that connect the top and bottom of the array are less than one foot in length. The cables that connect the opposite edges of the 4*4*4 array are about 4 feet in length. Only 8 of the 192 cables in the system are 4 foot long and thus will require special buffering. Another 16 of the cables are 1 foot long and thus "might" require special buffering. The other 168 cables will be 3 inches or less in length and will be able to be connected directly to computer busses without the need for special attention. Of these there 64 connections internal to the 2*2 array boards that will not need cables or connectors at all. Connections to the host can be made by busses that run through the array vertically. These busses will be short and one could also use these to distribute power and ground from a mother board below the array.

It is obvious that the 3 dimensional physical architecture will remove many

of the problems and associated with constructing the NNCP array. It is also obvious that servicing the array will be difficult unless modular construction is used. The 2*2 card module provides for this within the 4*4*4 array. In the case of a much larger machine the array could be constructed from 4*4*4 sub arrays.

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[1] described programming a tree connected ensemble. The 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[2] 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, however generalized to asynchronous data flow and using more stable and advanced numerical methods has been done by S. Y. Kung[3] and Lennart Johnsson[4].

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

---

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

[2] Section entitled Algorithms for VLSI Processor Arrays, in Mead and Conway, Introduction to VLSI Systems, Addison-Wesley, 1980.

[3] University of Southern California.

[4] Caltech Computer Science Department.

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.

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[5].

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 quantatative 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 m/64 points then the number of points along the edge is[6] $m/64^{0.5}$. 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[7] of the number of processing steps. As the size of each processor increases the ratio of processing steps to communications

---

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

[6] Ignoring the four corner points.

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

steps increases.

A Performance Example

Consider the proposed machine solving a three dimensional nearest neighbor problem where the elements of the lattice are 3x3 unitary matrices. The iterative step will consist of replacing each lattice element with a weighted sum of itself and its six nearest neighbors. 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 have 128k bytes of memory. Assume that about 100k of this is available for storing lattice points. Each lattice point is a matrix with 6 complex entries. Since we will represent each real number as 4 bytes, 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 48x48x48 array of lattice points. Each processor would contain a 6x6x48 slice of the total solid. Of the 128k bytes in each processor, 82944 will be used.

Assume that the time to transfer on lattice point to each adjacent processor is 2 mS[8]. The number of lattice points on the boundry to each adjacent processor is 6x48=288, and the time to transfer these values will be about 576 mS.

The computation performed on each lattice point will consist of seven matrix multiplications, and six additions. Each 3x3 matrix multiplication consists of 27 complex multiplications and 18 complex additions. A complex multiplication consists of 4 real multiplications and 2 real additions, and a complex addition consists of 2 real additions. The total number of real operations is 7x27x4 = 756 multiplications and 7x(27x2+18x2) = 630 additions. Assume 20 uS average for operand setup and a multiply or addition and the computation time per lattice point is 27.7mS. Total time for the entire array is 48 seconds.

Summarizing, the machine will be able to preform an iteration on a 48x48x48 array of 3x3 matrices in approximately 48 seconds. The 48 seconds will be all computation except for 576mS of interprocessor communication. In this example the fractional floating point utilization is about 99%.

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 8x8 points. Each lattice point consists of a single real number, or 4 bytes.

---

[8] 10 uS to service each port for one byte, 4 ports, 48 bytes per lattice point.

The time to transfer one lattice point to an adjacent processor is 160 uS[9]. The number of lattice points to transfer is 1, or the total time will be 160 uS 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 uS for an operation we find an iteration will take 120 uS. CPU utilization is now under 50%.

Fifty percent utilization of a processor, although somewhat wasteful is better than average for multiprocessor programs.

Other Potential Architectures

Why was an array network chosen over other networks such as a tree, a bus, a hypercube, or a simple VNM[10]? 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[11]. 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 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 256 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

---

[9] 10 uS to service each port for one byte, 4 ports, 4 bytes per lattice point.

[10] Von Neuman machine, or conventional computer.

[11] Leaf processors have one connection, all others have three.

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.