Programming the FPS T Series

John Gustafson

Introduction

The T Series is perhaps the most revolutionary product ever introduced by Floating Point Systems. When the AP-120B was introduced in 1975, a new level of high-speed computing became generally available, and certain problems became solvable in a practical way for the first time. The FPS T Series continues this tradition by making accessible a profound leap in both performance and cost efficiency. Like the AP-120B, effective use of the T Series depends on a departure from traditional methods of programming.

The FPS T Series combines two ideas central to high-speed computing: parallelism and vector arithmetic. Because each processor has its own memory rather than a switched path to a global memory, very large configurations are possible. This paper contrasts the T Series with other FPS products, and describes the particular style of programming its architectural features encourage, giving examples and performance estimates.

Comparison with Other FPS Products

Until now, virtually all FPS processors have been architectural descendants of the AP-120B. The FPS 5000 Series added coprocessors and a larger shared memory; the FPS-164 and FPS-264 increased the word size, memory size, and overall robustness to that of a full scientific computer, and the MAX accelerators added specialized arithmetic pipelines to the 64 Series memory bus, building on what has proved to be a highly successful design. The T Series is quite different from any other FPS product in a number of important respects.

Arithmetic can be efficiently done in either 32- or 64-bit precision, so the same computer serves both signal processing and scientific simulation applications. The product is highly parallel even in its smallest configuration. There is no “uniprocessor” version. That parallelism is always accessed explicitly by the user, not by automatic analysis of software.

The T Series is best regarded as a stand-alone machine, although it will initially depend on other computers for program development. It will use a “front end” computer as a gateway to existing networks of terminals and computers and as a resource allocator, but not as the principal store for programs and data.

The architecture of the T Series is highly specific in its requirements for maximum efficiency. Vectors must be fairly long, aligned on certain boundaries, and stored in contiguous memory locations. Such restrictions are made acceptable by the extremely high cost efficiency to which they give rise.

Perhaps most significantly, the performance of the T Series covers a range from one to three *orders of magnitude* greater than any other product offered by FPS—or by any other computer manufacturer, for that matter.

System Description

The T Series consists of a number of computers connected as a binary *n*-cube. This means that there are 2^n processors, numbered from 0 to $2^n - 1$, with a point-to-point connection between processors whose numbers differ in only *one binary digit*. This interconnection contains meshes of all dimensions up to *n*, and, for example, easily represents two- and three- dimensional lattices. It is also precisely the interconnect required for the Fast Fourier Transform (FFT) of radix 2 (see Figure 1). The maximum number of connections between any two processors is n ; therefore, communication cost grows only as the logarithm of the number of processors.

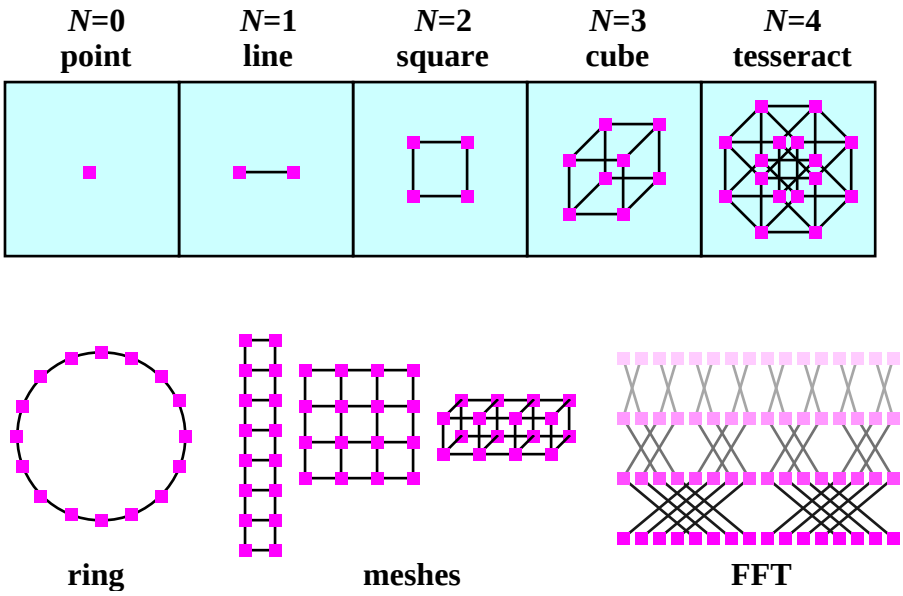


Figure 1. Binary N-Cubes

A single computer is called a *node*. Eight nodes are combined with disk storage and a *system board* to form a *module*. The system board provides input/output and management functions, and is connected to other system boards by a ring network that is independent of the binary *n*-cube network. A module is a "3-cube" and forms the basic unit that is repeatedly doubled to make larger T Series configurations.

Two modules form a *cabinet*, or 4-cube. There are enough links per node to permit a 14-cube to be constructed as the largest T Series configuration. A 14-cube would consist of 16384 nodes, grouped in 1024 cabinets.

Because the system is *homogeneous*, in that each module is identical and contains identical connections to other modules, programming is greatly simplified. The homogeneity also ensures that the balance between computing speed, main storage, mass storage, and external I/O can be preserved as configurations become large. Given this system interconnect, the specifications of any sized T Series can easily be derived from the properties of the individual processors. The following sections describe the *user-visible* features of each node.

Processor Description

An individual processor, or node, contains hardware for control, arithmetic, memory, and communication to other nodes.

Control

The control unit is a 32-bit microprocessor with the following features relevant to the application programmer:

- 7.5 MIPS computing rate (66.7 nanosecond clock cycle);
- Byte addressability (4 Gbyte address space);
- 2048 bytes of on-chip RAM with one-cycle access
- Four inbound and four outbound serial communications links;
- Stack-based instruction set with variable operand sizes;
- A single interrupt;
- Three-cycle minimum access time for off-chip memory.

Serving mainly to arrange vector operands to be sent to the arithmetic hardware, the control unit is also where most integer arithmetic operations should be done. All features of the microprocessor are directly accessed through a high-level language called Occam®, which differs from languages such as Pascal in that it is static, embodies point-to-point communication commands between procedures, and groups the procedures into sets that are declared *sequential* or *concurrent* in operation. Occam can thus easily describe the flow of data in a parallel-type algorithm. A detailed discussion of the Occam language and the T Series operating system is beyond the scope of this article; the approach to programming the T Series will be described in general terms, not specific Occam listings.

Arithmetic

The arithmetic hardware consists of a floating-point adder, a floating-point multiplier, and an interconnect microprogrammed to perform vector operations. The adder and multiplier can each produce a result every 125 nanoseconds, yielding a peak rate of

$$(2 \text{ operations}) / (0.125\ \mu\text{sec}) = 16 \text{ MFLOPS per node.}$$

Both 32-bit and 64-bit operations are supported. In 32-bit mode, the multiplier operates twice as fast, producing two results every 125 nanoseconds; hence, certain 32-bit operations that perform two multiplications for every addition can approach 24 MFLOPS per node.

All floating-point arithmetic conforms to the format of the IEEE proposed floating-point standard, which means that in 64-bit mode, the mantissa has approximately 15 decimal digits of precision and the dynamic range is roughly 10^{-309} to 10^{+307}.

The arithmetic units operate in a pipelined mode. The adder has a six-stage pipeline. It can perform floating-point addition and subtraction in 32-bit or 64-bit mode, comparison operations, conversions between 32-bit and 64-bit floating-point data, and conversions between integer and floating-point formats. The multiplier is five-stage in 32-bit mode, and seven-stage in 64-bit mode. The parts are managed by a microcoded control so that the programmer need only specify the vector operation, the input and output vectors, and the length of the vector. Scalars can be held in registers as inputs to the adder or multiplier, and the output of the adder or multiplier can be fed directly back to the input in order to perform reduction operations such as inner products, polynomial evaluation, and sums of terms.

Memory

The main memory of each node is 1048576 bytes of multi-ported dynamic RAM. The control processor and communication links access memory through a *random-access port*, and the arithmetic accesses memory through *serial ports*. The memory is organized in four 256 KByte blocks that are 32 bits wide. There is one parity bit for every byte in memory, and parity errors are reported directly to the system by an interrupt line.

The control processor sees the memory as a single bank of RAM with 262072 (32-bit) words. The arithmetic unit sees the memory as two banks of *vectors*, with 256 vectors in the small bank and 768 vectors in the large bank. For 32-bit operations, the vectors are 256 elements long; for 64-bit operations, they are 128 elements long. The division into two banks permits two inputs to the arithmetic unit every arithmetic cycle. To use a vector in an arithmetic operation, it is first loaded into a *shift register* which is an integral part of every block of memory. The output of the arithmetic unit shifts into the vector registers in *all four blocks* of memory at once (see Figure 2). The programmer then selects any or all of the shift registers to be deposited back into the memory proper. Thus, one can always arrange for a vector to be in the appropriate large or small bank of memory as needed by the next vector operation.

The control processor can access one 32-bit word of memory in 400 nanoseconds. The effective bandwidth is therefore

$$(4 \text{ bytes}) / (0.4\ \mu\text{second}) = 10 \text{ Mbytes/second}$$

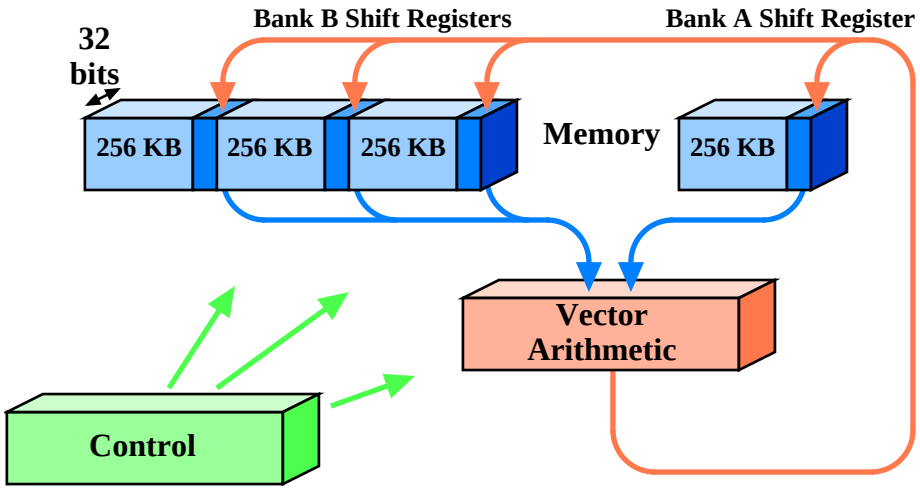


Figure 2. Memory Organization

One of the main functions of the control processor is to "gather" operands into a contiguous vector of storage, and "scatter" results back to random locations in memory. To move a 64-bit operand from one memory location to another requires two 32-bit reads and two 32-bit writes. This takes 1.6 µseconds, the "gather-scatter" time for a node. For 32-bit operands, the gather-scatter time is 8.0 µseconds per element.

An entire row of data can be moved to or from a vector register in only 400 nanoseconds; this means that the effective bandwidth between the memory and the vector register is

(1024 bytes) / (0.4 µsecond) = 2560 Mbytes/second

An application might make use of this extraordinary speed in a number of ways; for example, the pivoting of rows of a matrix or the sorting of records can be accomplished by moving *all* 1024 bytes of data rather than using linked lists and updating pointers.

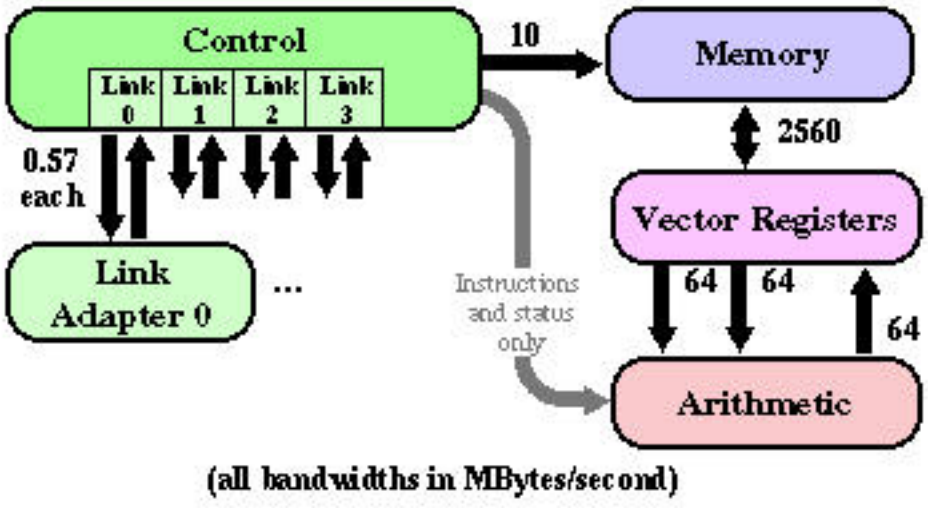
The vector registers each feed the arithmetic at a maximum rate of one 32-bit word every 66.7 nanoseconds. That is, the memory bandwidth supports two vector inputs and one vector output every 125 nanoseconds in 64-bit mode:

(3 words) x (8 bytes/word) / (0.125 µsecond) = 192 Mbytes/second

Communication

The communications links on the control processor have a nominal rate of 0.9375 Mbytes/second. Every 8-bit byte is sent with two synchronization bits and one stop bit, and requires two acknowledgment bits from the receiving processor. The acknowledgment bits can be sent over a separate link, giving an effective bandwidth of $^{8}/_{11}$ the nominal rate, or 0.68 Mbytes/second. If both the input and output links are busy, the effective bandwidth is reduced to $^{8}/_{13}$ the nominal rate, or 0.58 Mbytes/ second per link. The total bandwidth of the four input and four output links is thus about 4.6 Mbytes/second. When all communication links are operating, the control processor performance is degraded only slightly.

The links perform DMA-type transfers, with a startup time of about five µseconds. To transfer a block of 64-bit words, therefore, requires about (14*n* + 5) mseconds, where *n* is the number of words.



Each link is further divided into four sublinks on the T Series to provide 16 input and 16 output channels per node. Using software support, these sublinks subdivide the available bandwidth. Because two links are reserved for communication with the system board ring network, 14 links are left for *n*-cube connections, each representing a dimension of the cube. A typical system would dedicate one of these dimensions to a disk subsystem and perhaps another to a general-purpose high-speed I/O adapter for connections to graphics systems, other supercomputers, A/D or D/A devices, and so forth.

Figure 3 summarizes the bandwidths on a node that affect application programming.

A convenient way to think of the relative bandwidths is that

Arithmetic Time : Gather/Scatter Time : Link Transfer Time

are in the approximate ratios

1 : 13 : 110 (for 64-bit operations)
or
1 : 6 : 55 (for 32-bit operations).

This is the essential piece of information for optimal use of the T Series. For 64-bit applications, a vector should be used in about a dozen arithmetic operations before it is necessary to gather the next vector into place. If this is done, the control processor can *overlap* gather time and the arithmetic can approach peak speed. Similarly, roughly one hundred operations should be performed for every word that must be moved over a link. The following examples illustrate these issues.

Example: Wave Equation

Perhaps the most important equation of mathematical physics is the Wave Equation, which says, in physical terms, that *acceleration* of a medium is proportional to its *curvature*. For “nonlinear” waves, the proportionality is not constant but depends on time, space, and the state of the medium.

Mathematical Formulation

We consider here the simple case of the two-dimensional Wave Equation on a unit square domain. Mathematically, the partial differential equation is

$$\frac{\partial^2 F}{\partial x^2} + \frac{\partial^2 F}{\partial y^2} = \frac{1}{c^2}\frac{\partial^2 F}{\partial y^2}$$

for $0 < x < 1$, $0 < y < 1$, and $t > 0$. Loosely speaking, the left-hand side represents the curvature of the medium, and the right-hand side represents a proportionality constant times the acceleration. With c constant and the function fixed at zero on the boundary, this problem can be solved analytically; for purposes of illustration, we consider a finite-difference numerical method demonstrating techniques that apply equally well to problems with no hope of analytical solution.

Numerical Formulation

First, we discretize time into time steps t_0, t_1,… and the square domain into points (x_i, y_j) for $0 < i, j < n$. We consistently use the i subscript to denote the x dimension and the j subscript to denote the y dimension; that is, $x = i / n$ and $y = j / n$. Next, we approximate all second derivatives by the difference formula

$$\frac{\partial^2 F}{\partial u^2} \approx \frac{F(u - \Delta u) - 2F(u) + F(u + \Delta u)}{(\Delta y)^2}$$

The timestep can be chosen so that the constant c disappears from the difference form of the Wave Equation, yielding a simple explicit formula:

$$\text{New } F \leftarrow 0.5 \times \text{Present } F\left(_{\text{up+down+left+right}}\right) - \text{Old } F$$

where "$_{\text{up+down+left+right}}$" means that we add together the nearest-neighbor values.

Traditional Algorithm

"New F" can replace "Old F"; thus, all we need to store in a computer is two timesteps. Call these timesteps `F` and `G`; then a conventional serial algorithm might resemble the following:

```
For k = 1 to Number of Timesteps
 For i = 1 to n–1
  For j = 1 to n–1
   F[i,j] := 0.5*(G[i,j+1]+G[i,j-1]
      + G[i+1,j]+G[i-1,j]) - F[i,j]

 For i = 1 to n–1
  For j = 1 to n–1
   G[i,j] := 0.5*(F[i,j+1]+F[i,j-1]
      + F[i+1,j]+F[i-1,j]) - G[i,j]
```

By alternating between `F` and `G` updates, the numerical model imitates the progress of the wave through time where `F` represents odd timesteps and `G` represents even timesteps. The `F` and `G` values must be given initial values to start the alternation. For example, "poking" the medium by satisfying a unit displacement

somewhere in the F array (zeros elsewhere) results in a set of "ripples" that propagate away from the location of the original displacement. For *n* on the order of 100, the model is fairly accurate for the first several hundred timesteps. (More accurate numerical methods can be obtained by refining the approximations used for the second derivatives.) Notice that the boundaries are automatically taken care of by updating only points 1 to *n*–1 rather than 0 to *n*; this produces reflections as waves strike the boundary of the square domain.

Each timestep involves 3 additions, 1 multiplication, and 1 subtraction for every point being updated. The total computational work is therefore

$$W = 5(n-1)^2$$

floating-point operations per timestep. Later, we show that re-use of computations can reduce the constant 5 to a 4 in this formula.

T Series Uniprocessor Formulation

Assume that all operations are done with 64-bit numbers. If we choose $n = 127$, the problem can be run on T Series of any size from a single node all the way up to a 14th-order cube (16384 nodes). The memory required is

$$(2 \text{ timesteps}) \times (128^2 \text{ words/timestep}) \times (8 \text{ bytes/word}) = 131072 \text{ bytes}$$

plus space for scratch variables and the program itself; this easily fits in the memory available on one node, yet has 16384 mesh points computable in parallel on the largest T system.

Instead of attempting to "vectorize" the traditional algorithm, first consider the parallelism that exists in the *physics*. Because information can travel only at the speed of wave propagation in the medium, the spatial computations are concurrent except for points within a space-time "cone of influence." One would therefore expect that the algorithm *must* be serial in stepping through representations of timesteps, but could be completely parallel in computing changes at various points in space. For example, this parallelism could be across different processors, and, because the updating operations are repeated for the point data, could also be exploited by *vector arithmetic*.

The preceding type of reasoning generally forms the basis of a first attempt at putting an application up on the T Series: Consider the data dependency of the problem, preserving all concurrency implied by the physics; then subdivide spatially to allocate work evenly to all processors, and look for repeated operations that can make use of vector arithmetic within each subdivision. If possible, communication between subdivisions should be kept to the amount that can be overlapped with calculations.

Suppose there exist vector arithmetic routines such as

```
VecAdd(Input,Input,Result,Length)
```

and

```
VecAddScalarMult(Input,Input,
    Scalar,Result,Length)
```

which make efficient use of the T Series node arithmetic. A "vectorized" version of a timestep update might appear as follows:

```
For j = 1 to n-1
     VecAdd(F(1,i-1),F(2,j),Tmp1,n-1)
     VecAdd(F(0,j),F(1,j+1),Tmp2,n-1)
     VecAddScalarMult (Tmp1,Tmp2,0.5,Tmp3,n-1)
     VecSub (Tmp3,G(1,j),G(1,j),n-1)
```

(Repeat, interchanging F and G roles.)

The first `VecAdd` sums the "Down" and "Right" neighbors of a row of points `F(1,j-1)`,…, `F(n-1,j)`. The second `VecAdd` sums the "Left" and "Up" neighbors. The results of these two VecAdd operations are then summed and multiplied by 0.5 with the `VecAddScalarMult`. The `VecSub` operation then completes the updating of a row of the `G` array by subtracting that row from the result of the computation of $0.5 \times F_{\text{up+down+left+right}}$. This is repeated over all interior rows from $j = 1$ to $n-1$. Updating the `F` array is done by the same sequence with `F` and `G` interchanged.

A "trick" is possible in the above sequence, since it turns out that the `VecAdd` step to compute `Tmp2` on iteration `i` is almost identical to the computation of `Tmp1` on iteration `i+1`; both add diagonally adjacent pairs of points, as indicated by dark lines and light lines in Figure 4.

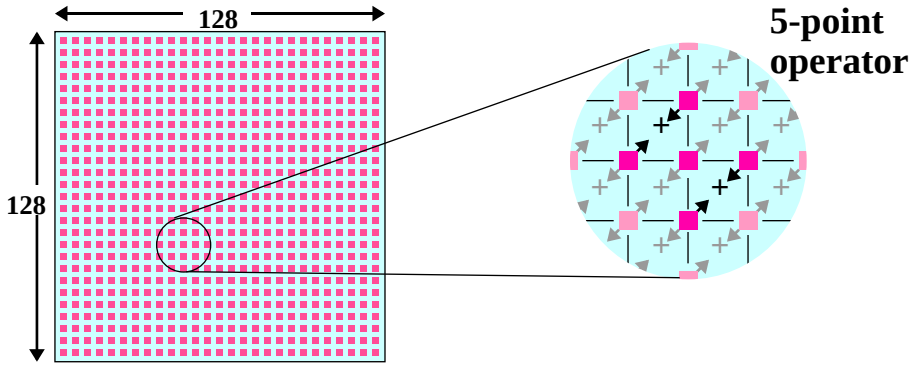


Figure 4. Vectorized 2D Wave Equation

Thus, the number of vector operations can be reduced from four to three in each iteration over i, by saving the `Tmp1` and `Tmp2` results:

```
VecAdd(F(1,0),F(2,1),Tmp1(1),n-1)
For j = 1,3,... to n-2
 VecAdd (F(0,j),F(1,j+1),Tmp2(0),n)
 VecAddScalarMult (Tmp1(1),Tmp2(0),0.5,Tmp1(1),n-1)
 VecSub (Tmp1(1),G(1,j),G(1,j),n-1)
 VecAdd (F(0,j+1),F(1,j+2),Tmp1(0),n)
 VecAddScalarMult (Tmp2(1),Tmp1(0),0.5,Tmp2(1),n-1)
 VecSub (Tmp2(1),G(1,j+1),G(1,j+1),n-1)
```

(Repeat, interchanging `F` and `G` roles.)

Note that n must be an odd number in the above. It is now possible to estimate the arithmetic cost of a timestep on a single-node version of the Wave Equation. If $n = 127$, then the domain is 128 by 128 points; each update involves 126 rows of length 127 each, where

```
1 VecAdd  (127 operations)
1 VecAddScalarMult  (254 operations)
1 VecSub  (127 operations)
```

must be performed on each row.

With 64-bit operands, a VecAdd operation costs

0.40	μsec to load the first vector,
0.40	μsec to load the second vector and issue the `VecAdd` instruction,
16.00	μsec to compute a vector of floating-point sums,
0.75	μsec to empty the 6-cycle adder pipeline, and
0.40	μsec to put the result vector back into memory.
17.95	μ*sec total.*

Thus, 17.95 μseconds are needed to compute 127 floating-point sums, for a compute rate of 7 MFLOPS. The `VecAddScalarMult` is similar:

0.40	μsec to load the first vector,
0.40	μsec to load the scalar,
0.40	μsec to load the second vector and issue the instruction,
16.00	μsec to compute a vector of floating-point sums,
1.63	μsec to empty the 13-cycle adder-multiplier pipeline, and
0.40	μsec to put the result vector back into memory.
19.23	μ*sec total.*

The `VecSub` operation is similar to the `VecAdd` operation in timing. The update of a row of the array therefore requires

$$17.95 + 19.23 + 17.95 = 55.12 \text{ μseconds}$$

and

$$127 + 254 + 127 = 508 \text{ floating-point operations.}$$

This formulation of the two-dimensional Wave Equation thus runs at approximately 9 MFLOPS on a single node, updating 140 complete timesteps per second. Many effects, such as procedure call overhead, memory refresh, interrupts, and outer loop counting, reduce this speed, but the net effect should be minor, and is best measured rather than estimated. This performance is similar to measured performance of the FPS-264 on a Fortran version of the vector algorithm.

T Series Multiprocessor Formulation

Given two nodes to run this 128 by 128 problem, the obvious approach is to divide the domain into two 128 by 64 problems, put one on each node, and let them communicate values on the boundary across the communications link. Each processor stores an additional row of neighboring data. Note that we are better off dividing along the natural “cleavage plane” of the memory, i.e., 64 vectors of length 128 rather than 128 vectors of length 64 (see Figure 5). Besides keeping the vectors long, the DMA transfers across links will be on contiguous data and will run about 40% faster.

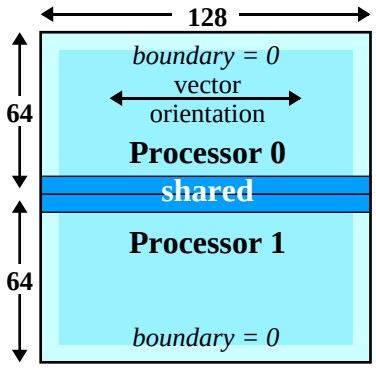


Figure 5. Two-Processor Subdivision

The DMA transfer involves a startup of about 5 useconds, and then about 14 μseconds per 64-bit word. The *communication cost* per timestep for a two-processor version is therefore

$$(14\ \mu\text{seconds/word}) \times (126\ \text{words}) + (5\ \mu\text{seconds}) \approx 1800\ \mu\text{seconds}$$

The *arithmetic cost* per timestep is

$$(63\ \text{vectors}) \times (35\ \mu\text{seconds/vector}) \approx 3500\ \mu\text{seconds}$$

The communication cost can therefore be *completely overlapped* in this case. While boundary data is being exchanged, the vectors distant from the boundary can be updated. By the time the vectors near the boundary need updating, the necessary information will have been sent to memory by the communication link.

In the general case, the 128 by 128 grid is easily divided among *P* processors into subdomains of size *M* by *N*, where *M* and *N* are powers of 2 between 1 and 128. Here, vectors represent rows rather than columns (Figure 6).

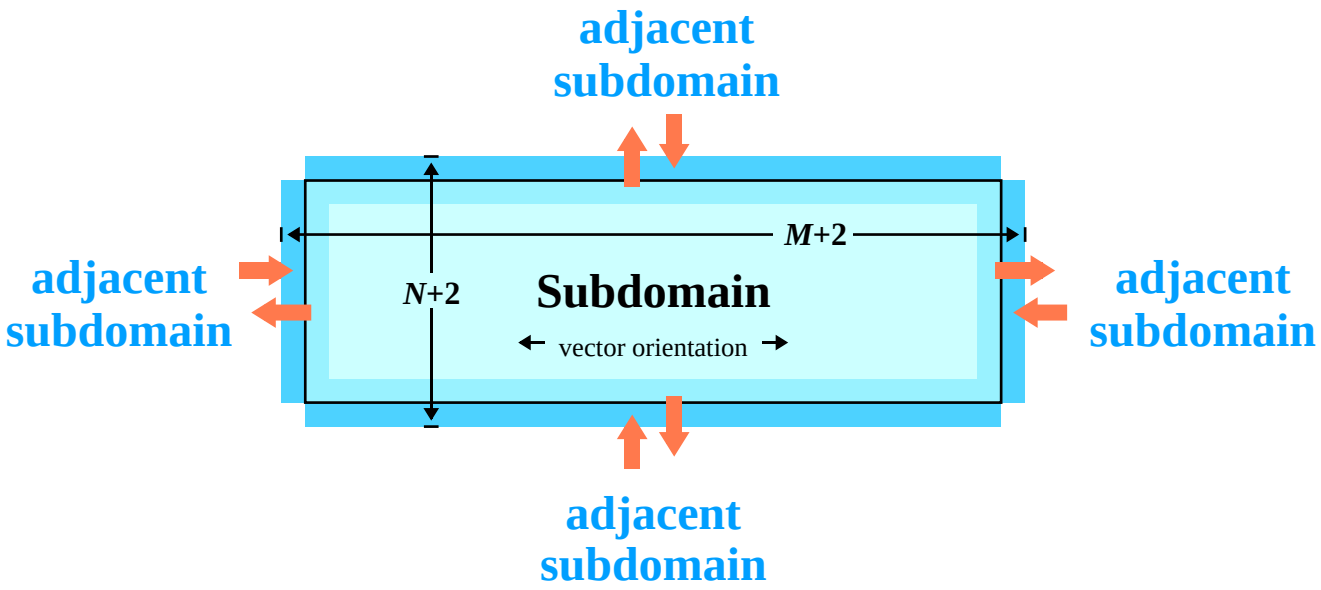


Figure 6. General 2D Subdomain

Let *VT(n)* denote the time to compute a vector sequence of length *n*, and *CT(n)* denote the time to communicate n 64-bit words across a link. In μseconds,

$$VT(n) \approx 0.125n + 3$$
$$CT(n) \approx 14n + 5$$

which indicates a strong need to minimize the "surface area" of each subdomain by making the proportions square. The best strategy is to increase the ratio of *M* to *N* (vector length to number of vectors) until communication time *just matches* the time spent computing updates to interior points. (Exterior points depend on the communicated values for updates, and hence cannot update concurrently with communications.) Efficiency increases monotonically with vector length, and communications are "free" to the extent that they occur during interior computations, so we will see that the best proportions for the rectangles are *M:N* equal to 2:1 or even 4:1. This is a major difference between the FPS T Series and other multiprocessors that make use of nearest-neighbor connections.

Because the rectangular subdomain can have one link for each side, the actual communication cost is the *maximum* of the time to communicate vertical and horizontal sides to an adjacent processor. For the *M* by *N* subdomain,

Communication Cost ≈ Max(14*M* + 5, 19*N*) μseconds

Arithmetic Cost ≈ 3VT(M) × N ≈ (0.375M + 9) μseconds.

A table of estimated performance can now be constructed for various subdomain proportions:

Estimated FLOPS for Two-Dimensional Wave Equation

	Domain size (M by N dimensions of discrete grid)									
Number of Processors	**128 ×128**	**128 ×256**	**256 ×256**	**256 ×512**	**512 ×512**	**512 ×1024**	**1024 ×1024**	**1024 ×2048**	**2048 ×2048**	**2048 ×4096**
1	9M	9M	—	—	—	—	—	—	—	—
2	18M	18M	18M	—	—	—	—	—	—	—
4	33M	36M	36M	36M	—	—	—	—	—	—
8	61M	67M	71M	72M	72M	—	—	—	—	—
16	95M	121M	135M	142M	143M	143M	—	—	—	—
32	128M	191M	244M	271M	285M	286M	286M	—	—	—
64	245M	259M	384M	491M	544M	572M	573M	573M	—	—
128	245M	494M	521M	771M	985M	1G	1G	1G	1G	—
256	450M	494M	996M	1G	2G	2G	2G	2G	2G	2G
512	450M	908M	996M	2G	2G	3G	4G	4G	5G	5G
1024	774M	909M	2G	2G	4G	4G	6G	8G	9G	9G
2048	774M	2G	2G	4G	4G	8G	8G	12G	16G	18G
4096	1G	2G	3G	4G	7G	8G	16G	17G	25G	32G
8192	1G	2G	3G	6G	7G	15G	16G	32G	34G	50G
16384	2G	3G	5G	6G	13G	15G	30G	32G	65G	68G

In this table,the "—" indicates insufficient memory per node, and the yellow entries indicate problem sizes for which computation runs near maximum efficiency through the overlap of communication between processors. This "band" of problem size - multiprocessor size match is characteristic of ensemble computer architectures such as the T Series.

Summary

The FPS T Series differs dramatically from other FPS products, and offers greatly increased performance and cost efficiency at the price of a complete break from established ways of doing computing. The user who wishes to exploit the power of the T Series should consider the following:

- Does the application have highly parallel content?
- Do the operations organize into repetitious patterns (vectors)?
- Is there a natural load balance among the parallel tasks?
- Do the speed and cost advantages outweigh program conversion effort?
- Is there a need for great expansion in future computing capability?

If there is a strong "no" to any of these questions, then other FPS products based on ideas closer to the scalar, von Neumann model might be appropriate. However, experience shows that "no" answers should be carefully considered, since there are few large-scale computing problems that do not ultimately satisfy the criteria listed above. If the answer is "yes" to all or most of these questions,the T Series computers are highly appropriate to the application and the desired computing environment.

Dr. John Gustafson is Senior Staff Scientist in the Engineering Department of Floating Point Systems, Inc., where he is working on the T Series and future computer architectures. His previous positions at FPS include Product Development Manager for Scientific/ Engineering Products, and Senior Applications Specialist. Before Joining FPS, Dr. Gustafson was a Software Engineer for the Jet Propulsion Laboratory in Pasadena, California. He has a BS degree from Caltech, and MS and PhD degrees from Iowa State University. His research interests include numerical analysis, algorithm theory, and special functions.