The Architecture of Tomorrow's Massively Parallel Computer

Transcribed from an after-dinner talk given by Dr. Ken Batcher Goodyear Aerospace Corporation on September 24, 1986

Goodyear Aerospace delivered the MPP to NASA/Goddard in May 1983, over three years ago. Ever since then we have tried to look in a forward direction. There is always some debate as to which way is forward when it comes to supercomputer architecture. In this talk, I will describe improvements to the MPP's massively parallel architecture in the areas of data I/O, memory capacity, connectivity, and indirect (or local) addressing.

I/O

Several years ago, Goodyear decided they should advertise the fact that they are something more than a tire company. They started a series of ads. A particular ad appeared in the *Wall Street Journal* a couple years ago saying that our computer can add and subtract 6 1/2 billion times a second (that's on eight bit additions). Someone at Goodyear thought up captions for the three men in the ad. The first man says "How long does it take to get the 6 1/2 billion pairs of numbers into the computer." The second man answers, "Oh, about a half an hour, then you add and subtract them in 1 second." So the third guy says, "Well, I hope they still make tires."

This points out the I/O problem of the MPP. Actually, the problem is shared by most supercomputers...they tend to be I/O bound. At the conference today, the speakers were saying that they were I/O bound. Figure 1 shows the rates at which data is transferred between various parts of the machine. The processing is going on between the PE registers and the ARU memory at 20,480 megabytes per second (assuming 16,384 wires pushing data at 100 nanoseconds per bit), so you can see the magnitude of the processing rate. Data slides in and out of the ARU memory from the side over 128, rather than 16,384 wires, so you get 160 megabytes per second, still a fairly respectable speed. In most computers

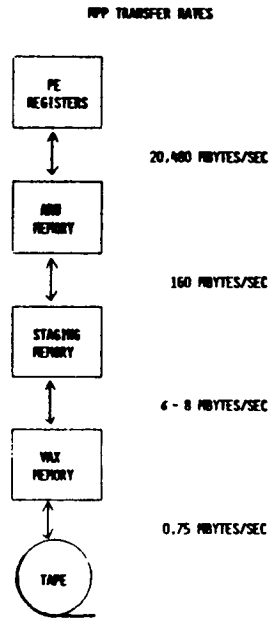

Figure 1

if you turn on all the I/O, the processing would grind to a halt. It would take all the memory cycles. On the MPP you can run the I/O full bore and slow down the processing rate by 1.6%. The processing rate doesn't see the I/O.

The I/O situation gets worse once we've put the data into the staging memory. Now we want to move it to the VAX computer through a DR780 channel, which is the fastest way you can get data in and out of a VAX but that's only at 6-8 megabytes per second. This is where the half hour figure comes from. You've got to put 6 1/2 billion pairs of numbers through this DR780 channel, which takes about half an hour, and the PEs would add them up in about 1 second. It's basically a limitation of the VAX. And it gets even worse when you look at what's the fastest way to get data in and out of the VAX, (unless you change the disk packs or something) you're limited to a tape drive that records at 6250 bits per inch---and 6250 bpi tape at 120 inches per second gives you .75 megabytes per

second. The significant difference between this .75 megabytes per second and the 160 megabytes per second rate out of the stager is basically your I/O problem.

You've got several ways of solving the problem, or at least making it less noticeable. We did design a disk farm for the MPP that would move data in and out of the staging memory at the staging memory rate of 160 megabytes per second. That's one possibility. The other possibility is some kind of high-speed network. If you had a device that was generating data at some fast rate, you could hook it directly into the staging memory at up to 160 megabytes per second. You want to bypass the VAX completely.

Another possibility is to increase the 160 megabyte per second rate to and from the array. Figure 2 shows the ARU as it exists right now. It has 128 columns, plus four spare columns. The data comes in from the left, goes out to the right, and is 128 wide at 160 megabytes per second input and output. If you aren't satisfied with that rate, you could divide the array up into four slices, put data into each slice simultaneously, and take it out of each slice simultaneously. This would give you four times the I/O rate, or 640 megabytes per second. If you want to preserve the redundancy feature of the array unit, you could add four spare columns to each slice of 32, so your array unit would have 144 columns in it, instead of 132.

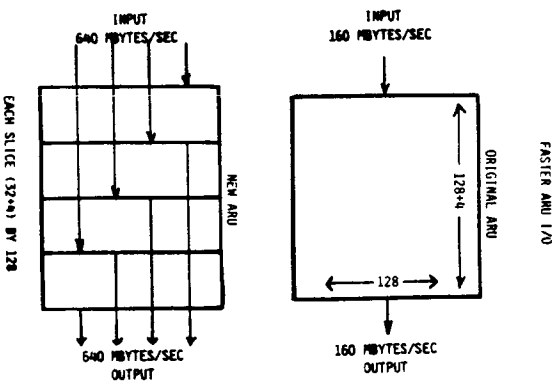

Figure 2

Memory Capacity

There seems to be a Parkinson's law of computer memory that says no matter how large you build the computer memory, there's always a computer problem that will overflow it. That's true of any computer, whether it's a personal microcomputer or a big supercomputer. The MPP is no different. Some of the speakers today talked about how they could use more memory...so there is a memory problem.

In the original specification back in 1979, NASA wanted 256 bits per PE for a total of half a megabyte of ARU memory (see table 3). We figured that was too small, so what we delivered was 1,024 bits, for a total of 2 megabytes of ARU memory. At that time we could get 4x1K static RAM chips with an access time of about 50 nanoseconds, and those are what we used to implement the MPP's ARU memory. We know that memory technology is always growing, so when we designed the machine we put in 16-bit addresses, so we could increase the ARU memory size later when the memory technology improved. Today, what we would do is build a board with memory sockets that could accept either 16K or 64K memory chips. Right now the 4x16K static RAM chips are readily available. Actually, the 4x64K RAMs are also available. They've got a high price tag but in a few years that should drop down. Today we could supply memory boards with either 16K or 64K bits per PE. This would increase the memory to either 32 or 128 megabytes. So you can get either 16 times or 64 times your present capacity---which is 2 megabytes. It's therefore real easy to expand ARU memory in the machine.

MPP ARU MEMORY

	BITS/PE	TOTAL (MBYTES)
ORIGINAL SPEC	256	0.5
DELIVERED IN 1983	1024	2
TODAY'S DESIGN	16384	33
	65536	134

Table 3

In 1983, we delivered four banks of staging memory using 64K dynamic RAM chips, so the staging memory had a capacity of 2 megabytes and an I/O rate of 20 megabytes

per second (see Table 4). We put 32 slots in the cabinet so that it could be expanded to 32 banks. This year we did expand it up to 16 banks. At the same time, we changed it to larger chips---256K chips. So right now the staging memory is 16 times larger than it was originally. The speed is 4 times greater. We still have half the slots available and at some time in the future we could expand it up to twice as big and twice as fast by populating all 32 slots. You can even go further than that; 1,024K (1 megabit) memory chips. These are starting to become available. So we could change the 32 boards and put in chips that are four times bigger and make the memory 256 megabytes. If you do that and you want the faster I/O in the ARU you've also got to feed it. You've got to take the staging memory and make it 128 banks...this will get the 640 megabytes per second. That will also give you a gigabyte of memory for the stager.

MPP STAGING MEMORY

	NUMBER OF BANKS	DRAM CHIPS	CAPACITY (MEGABYTES)	I/O RATES (MBYTES/SEC)
1983	4	64K	2	20
1986	16	256K	33	80
FUTURE	32	256K	67	160
LARGER CHIPS	32	1024K	268	160
FASTER I/O	128	1024K	1073	640

Table 4

Connectivity

It is real easy to change the architecture of the machine for faster I/O and more memory and still be compatible with the current MPP. Thus, the modifications I have described so far are still upwardly compatible with the current MPP. Programs wouldn't have to be changed to use the larger memory capacity and faster I/O. On the other hand, modifications to connectivity and addressing could reduce upward compatibility. Someone defined upward compatibility as meaning that you get to keep all the old mistakes. So if you want to forget about being compatible and look at other changes in the machine, then we can talk about connectivity and indirect addressing.

Figure 5 is a picture of the current ARU. It has 16,384 processors and they communicate with each other in the north, south, east, and west directions over a 2-dimensional mesh. This is good for those problems in which communication must be nearest neighbor over a 2-D mesh. With the topology on the outside it can be changed into a 1-D mesh. And for 3-D problems you can always run a third dimension down the random access memory, especially if you make that memory larger...to say 65,536 bits. So, you can treat the 1-D, 2-D, and 3-D problems without too much trouble. However, there are a lot of problems that require other connectivities. We did add the staging memory, which in some respects helps because if you don't like the mesh connectivity, you can always move the data out to the staging memory, rearrange it and bring it back into the ARU so that items that used to be far apart are now close together. So this staging memory does help somewhat in giving you more connectivity than the 2-D mesh.

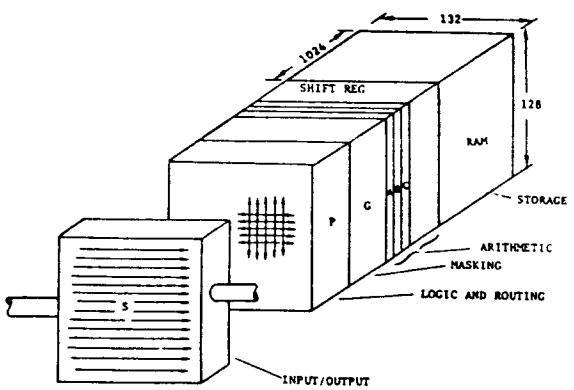

Figure 5

Looking back at the history of Goodyear Aerospace (see Table 6), some interesting trends in connectivity arise. Back in the mid 1960's we did a study contract for Griffis Air Force Base in Rome, New York, called *The Advanced Computer Organization Study*. We were looking at parallel processors that have 100 PEs in them. We figured that we wanted to hook anything to anything else and the only way we knew how to do this was with a sorting

network---which in some respects is kind of an ideal---it will do any permutation by sorting.

HISTORY AT GOODYEAR AEROSPACE

		NO. OF PE'S	CONNECTIVITY
MID 60'S	ADVANCED COMPUTER ORGANIZATION STUDY	~100	SORTING NETWORK
EARLY 70'S	STARAN	512-1024	256-WIDE FLIP
EARLY 80'S	ASPRO	> 1024	32-WIDE FLIP
EARLY 80'S	MPP	16384	2D-MESH

Table 6

In the early 1970's we built some STARANs that consist of about 1,000 PEs and we connected them together with a 256 wide flip network, which is basically the same as an omega network or a butterfly network...it has the same topology. So the STARAN had less connectivity and more PEs than we were looking at in the mid '60's. Later, we built the ASPRO computer with about 1,800 PEs and a narrower flip network at 32 wide, that had less connectivity than the STARAN. We then delivered the MPP with 16,384 PEs and a 2-D mesh. When you look at this over time, we have been increasing the number of PEs and reducing the connectivity.

One way to explain the trend is to say "Well, back in the '60's we were young and more visionary than we are now. But now we are more practical and have more practical connectivities." I think the real trend is that each of these projects was in response to an RFP, and basically, we gave the customer what he wanted. Back in the mid '60's Rome was being more visionary, and now NASA is being more practical and asked for a 2-D mesh. This is an illustration of the Golden rule, "Whoever has the gold makes the rules." Whatever the customer wants is what we give them. That's what I think is the real explanation of this.

Over the years, we have been looking at connectivity networks other than the mesh. If you had your druthers, you would like to hook the 16,384 processors with a full cross bar (see Figure 7) so that any processor could talk to any number of its neighbors and could broadcast to any number of its neighbors. Unfortunately, that requires something like a quarter of a billion cross points, so it's rather impractical.

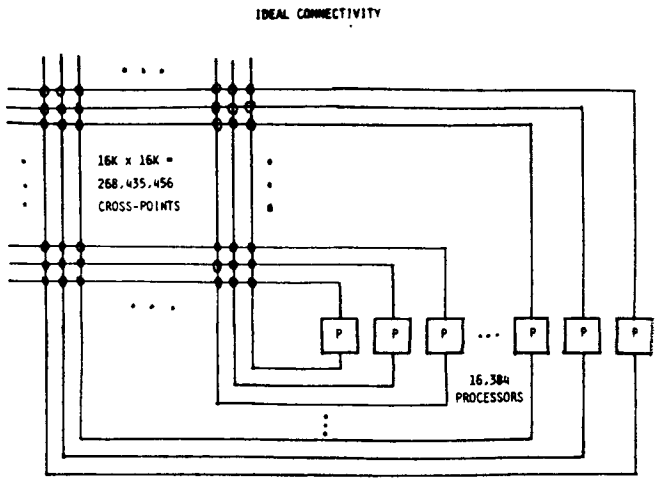

Figure 7

You can go various steps toward the ideal of full connectivity (see Table 8) using two approaches. You can take something like a flip network or an omega network...several people have talked about this network and they all give it different names. I call it a flip network, it has synonyms like omega network, butterfly network, delta network, etc.. They all have the same basic network topology, it's just different names. To connect n items together it takes $n/2(\log_2 n)$

INTERCONNECTION NETWORKS

- FLIP NETWORK

  N/2 LOG$_2$N SWITCHES
  114,688 SWITCHES FOR 16,384 ITEMS

- BITONIC SORT NETWORK

  SELF-CONTROLLING
  N/4 (LOG$_2$N)(1+LOG$_2$N) SWITCHES
  860,160 SWITCHES FOR 16,384 ITEMS

Table 8

switches. If you look at 16,384 processors, then it takes 114,688 switches, which is considerably less than the quarter billion needed for ideal connectivity. This does most of the useful permutations. For most problems this is what you want. If you want to be able to do any permutation of 16,384 items, then you would use a network that is about twice as big, requiring 221,184 switches. Unfortunately, it takes

awhile to compute how to set up such a network. If you want to go one step further you use the bitonic sort method, which in some respects is self controlling. It will compute the setting while the data is being fed into it. And it takes $n/4(\log_2 n)(1+\log_2 n)$ switches, which is 860,160 switches. So these are three ways of connecting the PEs, depending on what you want communicated between the processors.

The hypercube is basically somewhat like the flip network (or omega or butterfly), it has the same kind of complexity. These methods are the basic ways of improving the connectivity of the MPP, but then you lose something in compatibility. Currently your programs communicate north, east, west, and south, so you would have to do some work to use these other connection schemes.

Indirect (Local) Addressing

Figure 9 shows what one MPP PE looks like. The random access memory has a global address that comes from the control unit. In fact, if you look at how we implemented the PE, all the logic (except the memory) is on one chip, the PE chip, which doesn't even see the memory address. The

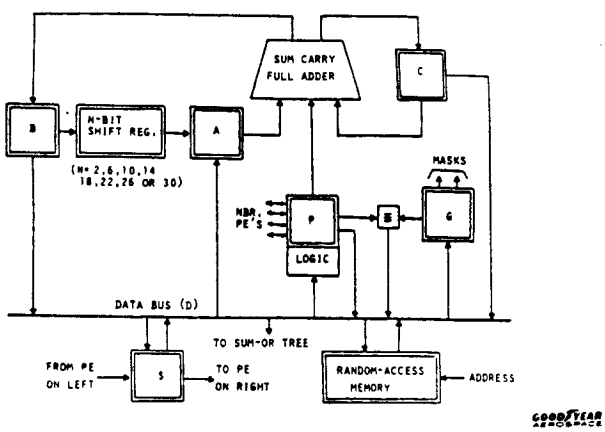

Figure 9

only connections between the memory chip and the PE chip are the data paths. The global address goes to the memory chips directly and doesn't know that there is a PE chip at all. So the memory address is a global address. This means that if one PE wants to look at bit 43 in its memory, then all PEs must look at bit 43 in their memories. This means that when we program the machine, we look at it as a bunch of memory planes and processing planes. For example, you could take memory plane number 43 and move it from the memory into one of the processing planes or store a processing plane into a memory plane. All the data in one plane moves en masse in one cycle. So you look at the machine as a bunch of memory planes and processing planes. If you look at one bit of a plane, then you're looking at all bits of the plane.

In all the textbooks the MPP and the Connection Machine are called SIMD machines---I question whether they are really SIMD machines. In the classic SIMD machine all the opcodes and addresses come from a common control unit (see Table 10)

CLASSICAL
SIMD - OPCODES & ADDRESSES FROM COMMON CONTROL UNIT (REALLY SISD).
MIMD - OPCODES & ADDRESSES FROM LOCAL PE MEMORIES AND REGISTERS.
TRUE
SIMD - OPCODES FROM CONTROL UNIT. ADDRESSES FROM LOCAL MEMORIES AND REGISTERS.

Table 10

and this is true of the Connection Machine, the MPP, and any bit-serial parallel processor. Though all the textbooks call this SIMD, I can argue that it is really SISD, Single Instruction Single Data path. If you look at a conventional computer, you have a processor and memory and you take memory words from the memory into the processor and you store memory words. The only difference is that our memory words are 16,384 bits each. So you look at this MPP (see Figure 1), or the Connection Machine, as a conventional computer that happens to have a very large memory word, in this case 16,384 bits per memory word, and all we are really doing is moving memory words back and forth between processors and memory. So you can argue that the MPP and these other bit-serial parallel machines are really SISD machines. If you look at what a MIMD machine is; the opcodes and addresses come from local PE memories and registers. Each processor generates its own addresses and opcodes from programs stored in its own memory,

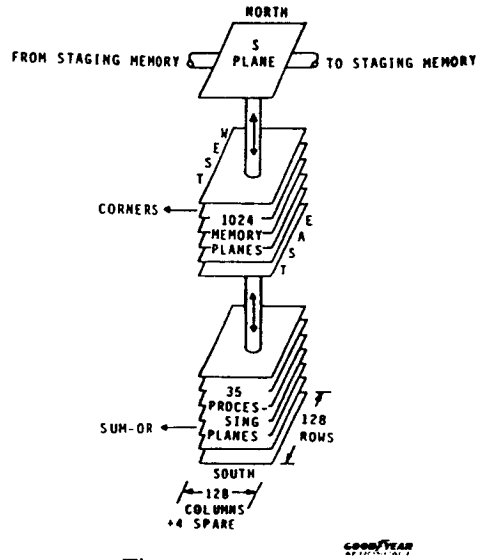

Figure 11

as opposed to the other case where they all come from a common control unit. So I can argue that a true SIMD machine would have the opcodes come from the control unit and still have a single instruction stream, however, if you really want multiple data, you should really have the addresses come from local memories and registers. This is somewhat in between the classic SIMD and MIMD machines. So, I think the true SIMD machine is one in which the addresses are generated locally, and the opcodes are generated in the control unit.

We found many problems where we really would like to address the memory via a local independent address rather than just through a global address. One such case comes up when trying to factor large numbers on the MPP. If we are trying to factor a particular large number, then what we do is look at factors of 1,000 to 4,000 quadratic residues. Table 12 shows an example of 4,000 quadratic residues. What we do is find the prime factors of each of these. We have maybe 4,000 primes in the horizontal dimension, and first we find what primes factor these residues, i.e., we try to divide them all by 2, then by 3, then by 5 and so on. We set up a flag matrix that shows where the division is exact. However, we are not done then. We want to take out, say, all the factors of 2, so if the number is 128, then we have to take out seven factors of 2 from that number. If it is 243, then we

4000 PRIMES →

4000 QUADRATIC RESIDUES ↓	2	3	5	7	11	13	17	19	23	29	31
128	1	0	0	0	0	0	0	0	0	0	0
690	1	1	1	0	0	0	0	0	1	0	0
243	0	1	0	0	0	0	0	0	0	0	0
26071	0	0	0	0	0	0	0	0	0	1	1
40841	0	0	0	0	0	1	1	0	0	0	0
343	0	0	0	1	0	0	0	0	0	0	0
8987	0	0	0	0	1	0	0	1	0	0	0
32	1	0	0	0	0	0	0	0	0	0	0

Table 12

have to take out five factors of three. One way of doing this is to keep dividing by two until they are all odd, then divide by three as much as possible, and it can be seen that this is a sparse operation, i.e., most of the PEs are not participating. Each one of these divisions is in a separate PE, and most of them are not participating. We keep dividing out the higher powers of these primes. What we would like to do to improve the situation is to pack the data together so that we get the kind of arrangement shown in Table 13. We now start with let's say 15 columns for the prime factors. And then we divide these numbers by these numbers, do that until all the powers have been reduced and then we go on to the next column. We would rather do something 15 times than 4,000 times, and we would like to rearrange the data like that, pack it together where it is sparse. This improves the utilization of the machine.

15 PRIME FACTORS →

4000 QUADRATIC RESIDUES ↓				
128	2			
690	2	3	5	23
243	3			
26071	29	31		
40841	13	17		
343	7			
8987	11	19	43	
32	2			

Table 13

When we were looking for a way to do this, we found that we could use the shift register (see Figure 9). The shift register was originally in there to improve the machine's times for multiplication, division, and floating point operations. However, it turns out that the shift register can be used for several other purposes. The shifting is maskable, so I can shift some shift registers and not shift others. So in effect, it turns out to be a locally addressable memory because I can turn the shifting of it on and off. Unfortunately, it only has 30 bits in it, I wish it were larger. So there is a small quantity of locally addressable memory in the MPP, and we have found out how to use it to help some of these problems.

To improve the MPP, I would probably replace the shift register with a RAM. You can simulate a shift register within a RAM and then you can do other things with the RAM. You would really like to make the whole memory locally addressable, but then you have some problems. If this is off on a separate chip, then you have to worry about how you transmit the address from the PE into the memory and back again. It either takes a lot of pins or some other circuitry. The compromise is to put a RAM on the chip and put another RAM off the chip. The on-chip RAM is implemented with VLSI techniques. Memory manufacturers make all their memory chips with their own rules to get a lot denser memory chips than what you can build just using the standard VLSI design rules. If you still want a lot of RAM, then you still need a standard memory chip from a manufacturer. Anyway, you could build your own locally addressable RAM. The places where you would like to see this is in Artificial Intelligence (AI).

I was looking at AI problems on the MPP. I found that this local addressability could be used to help out. For example, if several concepts are stored in each PE, while one PE is looking at its third concept, another PE may be looking at its fourth concept. With global addressing that is hard to do. With local addressability, it turns out to be a lot easier.

When I got Danny Hillis' book, the first thing I looked for was to see how they got local addressability...I couldn't find it anywhere. I know there are some Connection Machine people here, and I think that local addressability is a problem and I would like to see a newer machine have locally addressable RAM.

Conclusion

In conclusion, I talked about four topics (see Table 14). I/O---we can get transfer rates up to 640 megabytes per second. There are devices around that can supply the data and accept it at this rate. We can increase the memory capacity up to 128 megabytes in the ARU and over a gigabyte in the staging memory. For connectivity, there are several different kinds of multi-stage networks that we can consider. And we also should do something about local or indirect addressing.

SUMMARY

INPUT-OUTPUT	- 640 MBYTE/SEC
MEMORY CAPACITY	- 134 MBYTES -- ARU 1073 MBYTES -- STAGING MEMORY
CONNECTIVITY	- MULTI-STAGE NETWORK
LOCAL (INDIRECT) ADDRESSING	- ON-CHIP LOCAL ADDRESSABLE RAM

Table 14