PRESENT AND FUTURE COMPUTER ARCHITECTURES

Richard P. Mount

California Institute of Technology, Pasadena, USA

ABSTRACT

I show how an increasing use of parallel computing is inevitable in experimental HEP, and in all other fields needing massive computing resources. I review vector, pipelined, SIMD and MIMD architectures, giving some detailed examples to illustrate the ideas. The somewhat abstract examination of the most interesting machines is balanced with a study of the less adventurous, but very practical, system to be installed by the L3 collaboration. Software for parallel architectures is an even greater challenge than the construction of the hardware. I examine the problems both for individual programs, and for overall resource management, and describe some particular solutions which may be appropriate for experimental HEP.

1. INTRODUCTION

These lectures are an expanded and updated version of the lectures I gave under the title 'Computer Architectures for High Energy Physics' at the 1986 CERN Computing School. Although the wonders of modern text processing would have made it easy for me to incorporate all of last year's material into this write-up, I decided not to do this so that I could have the freedom to describe the new topics in any necessary detail without producing undue editorial distress. Wherever it is appropriate I refer the reader to the write-up of my 1986 lectures.

In my introduction to last year's lectures I described my own involvement in HEP computing, and I compared the rapidly increasing computing needs of HEP experiments with the somewhat less rapidly decreasing costs of mainframe computers. I also stressed that 'man does not live by MIPS alone': in HEP computing most of the money goes to support interaction, disk and tape I/O, printing etc. rather than simple number crunching. All of the arguments and figures presented remain appropriate, and the latest estimates of LEP computing needs are at the upper end of the estimates I gave for the L3 experiment.

2. THE FUNDAMENTAL PROBLEM

In all the many scientific and commercial fields where computing needs seem certain to grow rapidly there is a fundamental problem. The way I like to express this is by pointing out that the CDC 7600 is still a fast machine.

When CERN installed a CDC 7600 fifteen years ago it was the fastest computer in existence. When it was switched off (and scrapped) in 1984 it was still one of the fastest scalar CPU's available. Today's fastest scalar CPU's beat the CDC 7600 by about a factor of 2.

This story illustrates the fact that, while construction and maintenance costs fall by a factor two every three or four years, the cycle times of the fastest CPU's fall much more slowly. Thus users of large amounts of CPU power cannot continue to work exactly as they do now, using one or two powerful CPU's to meet all their needs. As computing requirements continue to rise, more and more applications will have to use hardware working in parallel, regardless of financial constraints.

3. PARALLEL COMPUTING: THE CHALLENGE

The arguments above convince me that widespread use of parallel computing is inevitable, provided we find adequate answers to these questions:

- how can we make it cheap?
- how can we make it effective?
- how can we stay sane?

Parallel computing will be cheapest if we use technology close to the peak of the price-performance curve. If we manage to solve the logical problems associated with doing many things in parallel, we can then step back from the expensive leading edge of CPU technology and use a larger number of much cheaper building blocks.

Can we solve these logical problems? In other words, can we provide effective parallel computing without serious damage to our sanity? In the course of these lectures I will show that the answer is 'yes', provided that we use a parallel computer architecture and software which are well matched to a particular task.

3.1 Parallel Architectures

Parallel architectures will be examined in some detail later. Here I introduce a very simple classification scheme [1].

<u>SISD systems</u>

SISD, meaning 'single instruction single data', describes conventional computer architectures. For later comparison with more complex cases, a conceptual diagram of an SISD machine is shown in Fig. 1. Note the significance of the bi-directional arrow linking the Instruction Decoder and the Execution Unit. This shows that the results of instruction execution can affect program flow, usually via conditional branch instructions.

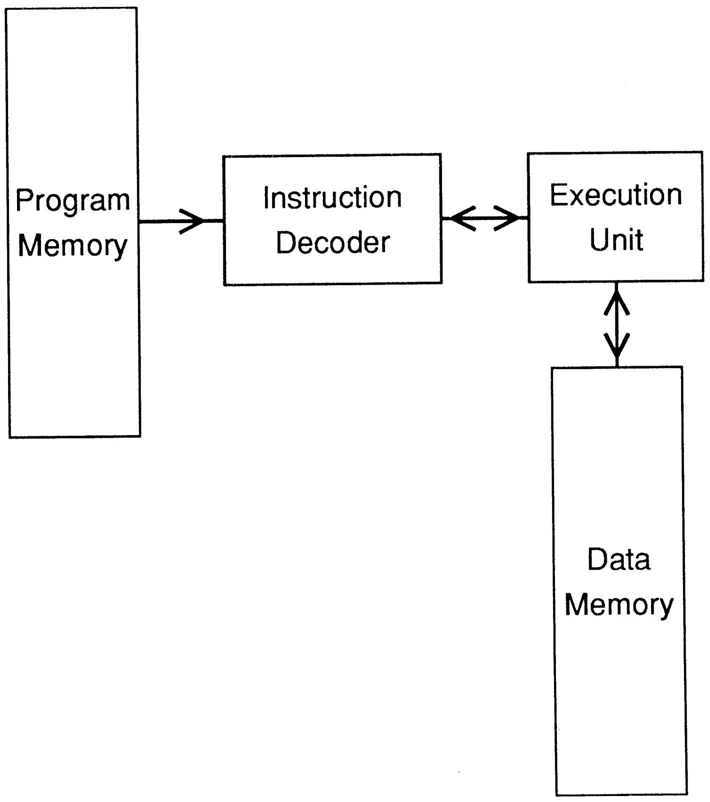

Figure 1 Architecture of an SISD Computer

In its simplest form an SISD machine involves no parallelism. It is now very common to extend this architecture with multiple or more complex execution units to support 'pipelining'. In a pipelined CPU, it is possible to initiate one instruction every machine cycle, even though most instructions take several cycles to complete. Vector computers are a particular example of this form of parallelism. More details of pipelined and vector architectures are given in sections 3.1 and 3.2 of last year's lectures.

SIMD Systems

SIMD means 'single instruction multiple data'. A conceptual diagram of an SIMD machine is shown in Fig. 2. An SIMD machine executes a single program and the Instruction Decoder broadcasts instructions to many Execution Units. Note the unidirectional arrow between the Instruction Decoder and the Execution Units; it is usually impossible for the Execution Units to have any direct effect on program flow.

SIMD systems can be very effective for a limited range of applications. Examples of SIMD systems will be described later.

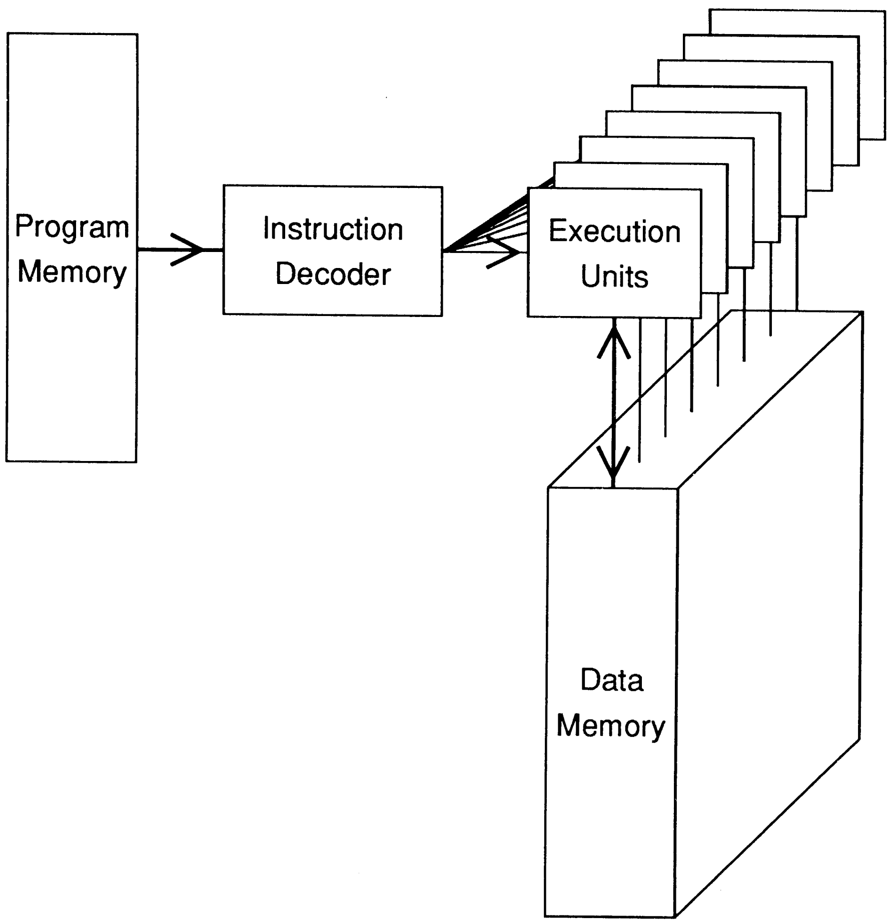

Figure 2 Architecture of an SIMD Computer

MIMD Systems

The 'Multiple Instruction Multiple Data' classification includes all the rest of parallel computing, and a single 'conceptual diagram' for this classification is beyond my skills. In an MIMD system several CPU's work together on a task. The speed and topology of the interconnections between the CPU's (and memories) determine the areas of effective application. In an MIMD system, network and memory performance can be at least as important as CPU cycle time, and the 'Digression on Networks and Memories' which follows is a necessary preparation for a detailed discussion of some MIMD systems.

Not (quite) forgetting ....

SISD, SIMD, MIMD (and MISD if it takes your fancy) are not a complete set of all possible computer architectures. More exotic ideas, such as Data Flow, Demand Flow, and machines optimised for Artificial Intelligence can involve departure from 'Control Flow' which is the central concept of all computers for which people currently pay money. Many of these architectures are intrinsically parallel and are promising for the long term future. Readers are encouraged to refer to Hertzberger's lectures at the 1986 school for a good introductory review.

4. DIGRESSION ON NETWORKS AND MEMORIES

A parallel computer is composed of a number of CPU's and memories linked together. The cost of high bandwidth links can easily dominate the cost of a many CPU system. In this section I show how network costs vary with size and topology, and describe a generalised memory hierarchy for a parallel system.

4.1 Networks

Crossbar Network

A typical crossbar network is shown in Fig. 3. The crossbar network can provide 'any-to-any' routing with very small delay, but becomes increasingly costly for large numbers of processors. The reason for the cost is that the switching elements, (the open and filled circles in the figure), may have to be complex devices, and there are N^2 of them.

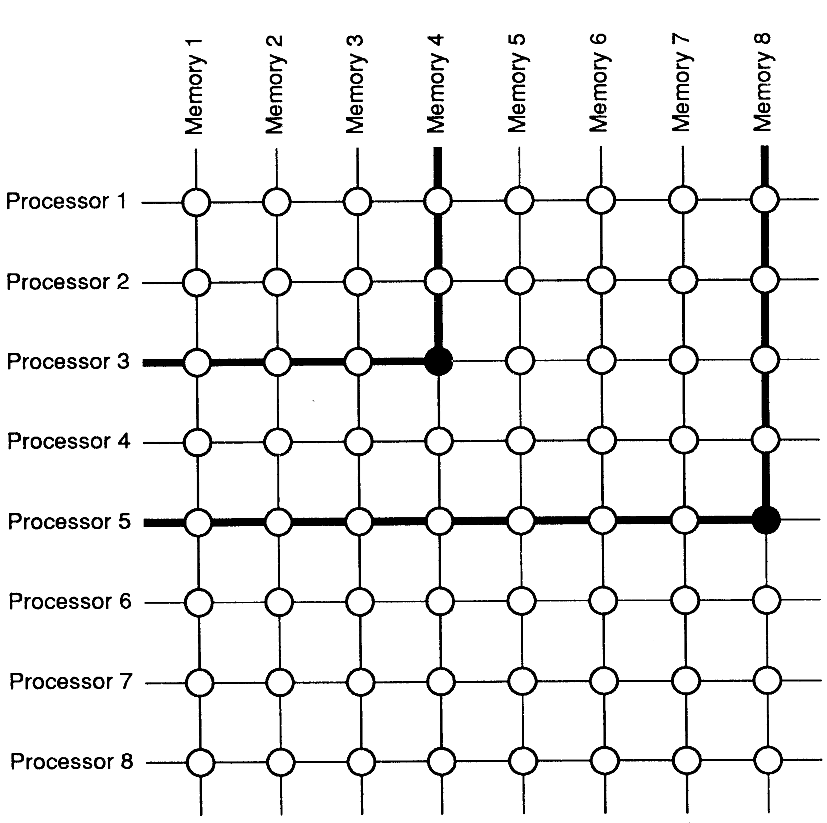

Figure 3 Example of a Crossbar Network

Routing through the network may be semi-static, (some supervising processor changes the paths occasionally), or 'packet-switched', (data are preceded by the address of their destination). The latter case is clearly far more flexible, but requires more complex switching elements, particularly if support is provided for features like queueing data bound for busy ports.

Shuffle or Ω Network

The shuffle network, of which examples are shown in Fig. 4, is a solution supporting any-to-any communication for large numbers of CPU's at significantly lower cost than the crossbar. The figure shows how shuffle networks are built up from elementary crossbar units. In the example shown, the elementary units are 2×2, but the shuffle network can move closer to crossbar performance and cost by incorporating elementary crossbar units with 4×4, 16×16, etc. ports.

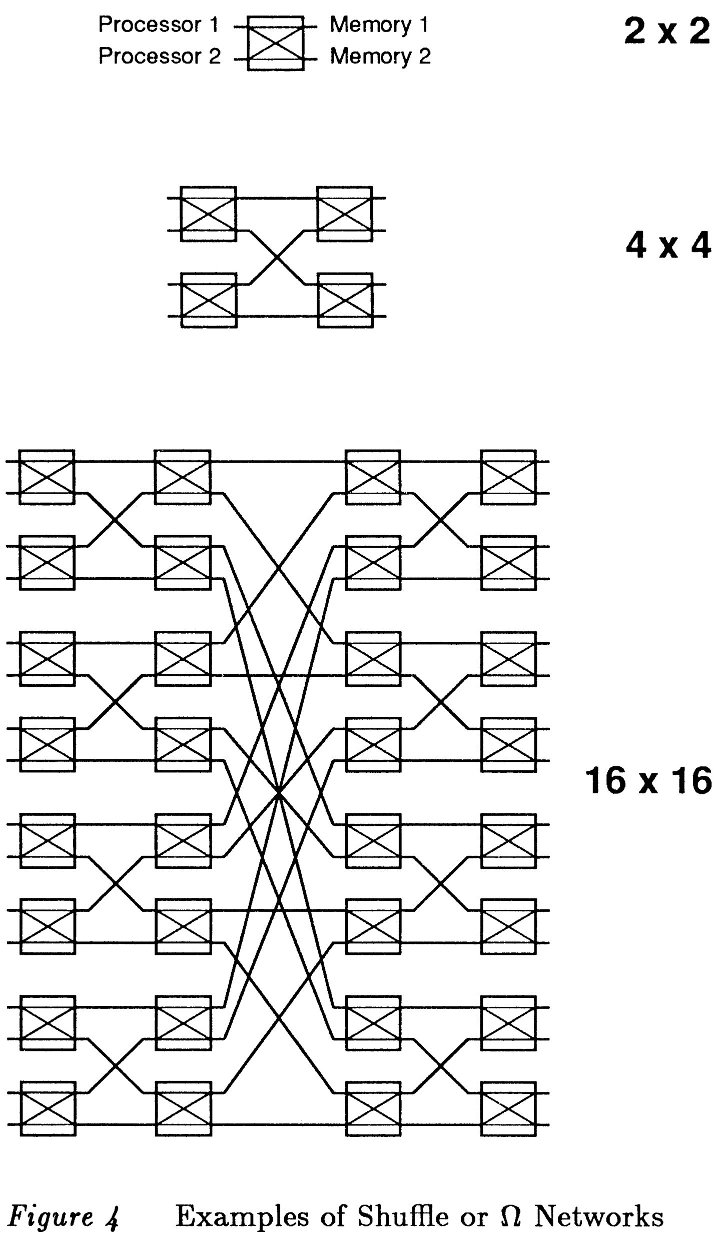

Figure 4 Examples of Shuffle or Ω Networks

The cost of the network is proportional to $N \log(N)$ and the transit delay to $\log(N)$. As for a crossbar network, routing can be semi-static or packet-switched, with similar trade-offs in versatility and cost.

Locally Connected Network

To remove any non-linear effects in the relation between cost and number of processors it is necessary to resort to a locally connected architecture such as that shown in Fig. 5. Each processor-memory unit is connected to a fixed numbers of neighbours. The transit delays across the local links are small. The any-to-any transit delay is highly dependent on the topology of the interconnections, and on how easy it is for processors to relay messages.

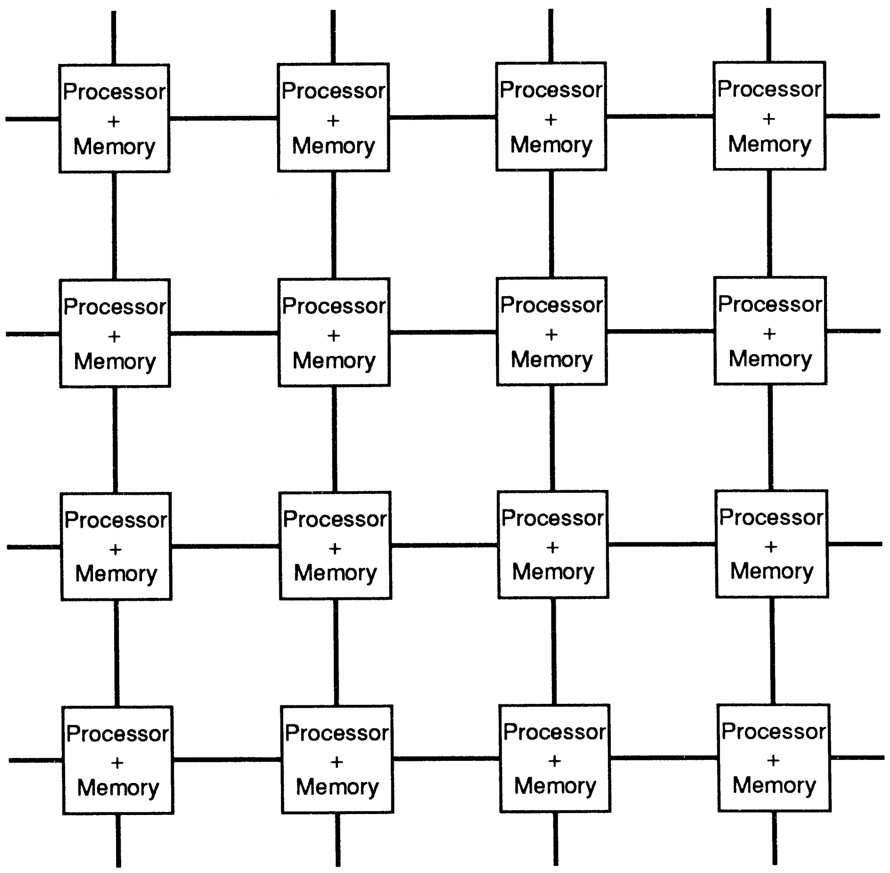

Figure 5 Example of a Locally Connected Network

Such networks with limited connectivity are well matched to certain problems. For example, a theoretical simulation using a lattice of space-time points maps well onto a lattice of locally connected processors. The dominance of short range interactions makes long range communication unimportant. Conversely, it is difficult to envisage making general purpose parallel computers based on large arrays of locally connected processors.

Bus System

The most common way to interconnect several CPU's and memories is to plug them all into a backplane or 'bus' as shown in Fig. 6. At first sight, one might imagine that this would be the ideal way to make a large parallel computer, since most busses are fast, and we just have to buy a lot of connectors. However, although the topology of a bus system might appear to be a radical departure from the networks I have just described, its economics show it to be just another worthy competitor.

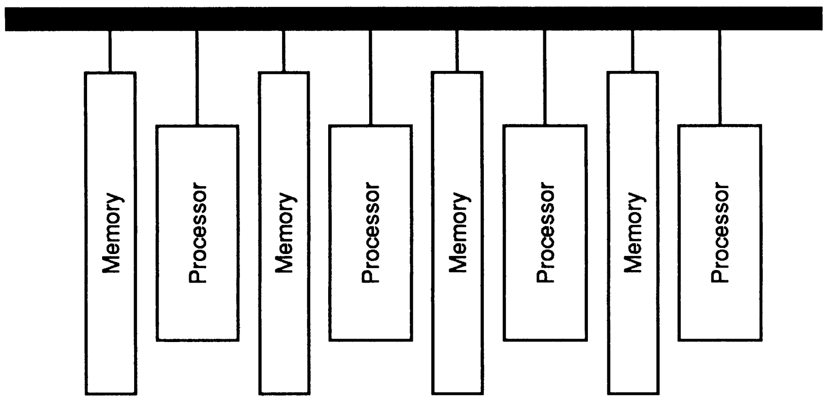

Figure 6 Example of a Bus System

Fast expensive bus systems rarely support more than 10 CPU's and their memories. The bus must be physically short to reduce propagation delays, and to plug into it you need not just a connector, but also a complex and expensive 'chip set' to drive the bus lines and handle the bus protocol at high speed.

Very large, loosely-coupled, parallel systems can be constructed using slow, often serial, busses. The slow communications and potential for congestion mean that only a few applications are suitable for such systems.

MIMD Coupling: Summary

Table 1 summarises what I have said about the cost and performance of interconnection schemes for MIMD computers. I also give some examples of machines which use the schemes. For each scheme I have assumed that the cost of a bare CPU and memory is α. The cost of a switching element within a crossbar is β. The cost of a 2×2 switch within a shuffle network is $\gamma/2$ if we assume my logarithms are taken to base two. For a locally connected network, the bare CPU and memory price is augmented by δ to pay for the communications interfaces, and typical processor-processor path lengths in an m dimensional hyper-cubic lattice rise like $N^{1/m}$. Finally for bus systems I offer the choice between paying Λ for an expensive high-speed bus, or λ for a cheap slow one, while augmenting the cost of CPU's by Θ for a high-speed bus interface or θ for a slow one.

Table 1

Cost and Performance of MIMD Interconnection Schemes

Coupling	System Cost	Any-any Delay	Examples
Crossbar	$N(\alpha + \beta N)$	ϵ	component of shuffle networks
Shuffle	$N(\alpha + \gamma \log(N))$	$\epsilon \log(N)$	NYU Ultra Computer RP3
Local	$N(\alpha + \delta)$	$\epsilon N^{1/m}$	Caltech/Intel Hypercube FPS T-Series
Bus Systems	$N(\alpha + \Theta) + \Lambda$	$\approx \epsilon$	ELXSI 6400 IBM 3090-600
	$N(\alpha + \theta) + \lambda$	large	Apollo Domain Ethernet VAX-Cluster

In all cases, the bus or network performance can be halved by halving the width of the bus or communications paths. Although it might seem obvious that a 32-bit machine should have 32-bit communication paths, 8-bit or even serial paths are common in large machines to keep network costs down.

Turning my greek alphabet into dollars or Swiss francs is left as an excercise for the reader.

Not all parallel computing systems fit neatly into one of these categories, although all are subject to similar economic constraints. For example, the systems comprising a 'host computer' and attached processors which are currently popular in High Energy Physics may be inelegantly categorised as bus systems, by declaring most host functions to be those of an active communications bus.

4.2 MIMD Memory Hierarchy

An MIMD architecture allows 'supercomputer' performance to be built up from cost-effective CPU's, at the expense of some discomforts. Modern memory hierarchies allow fast processors to work successfully with cheap, slow memories, again, at the expense of some discomfort.

Figure 7 shows a typical memory hierarchy in an MIMD machine. If we ignore the 'global memory', this is also the memory hierarchy of most conventional SISD machines, so I will first treat the SISD case before adding the complications of MIMD.

At the top of the pyramid is the CPU which normally performs operations on quantities held in 'registers'. The registers can exchange data with the 'cache' in one

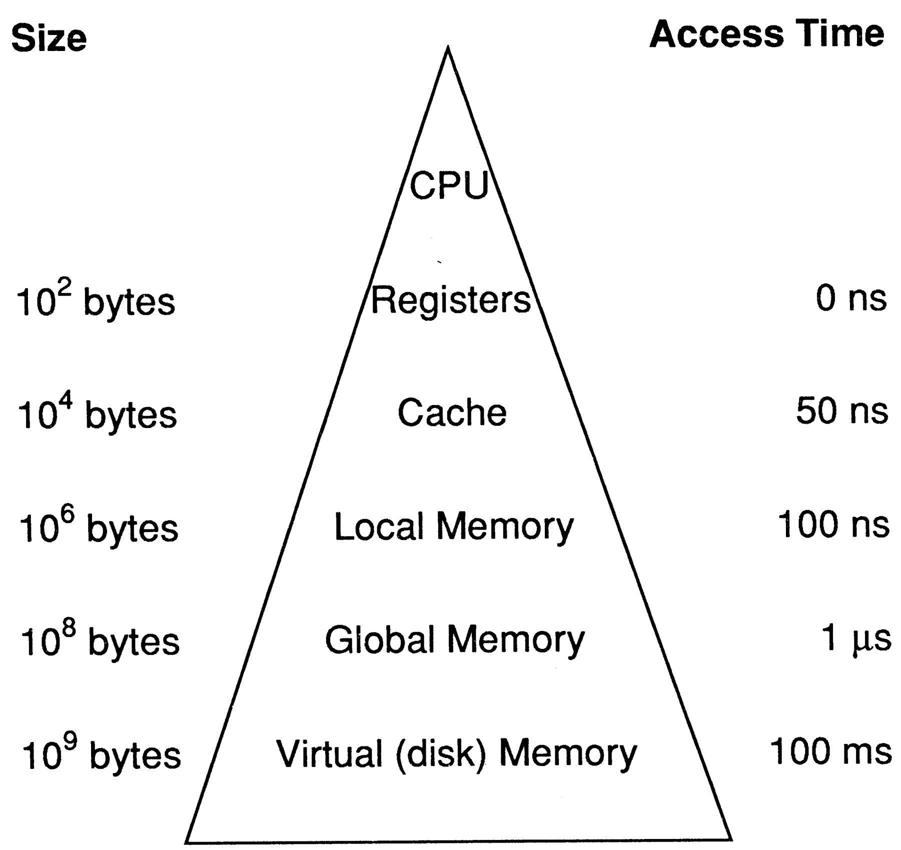

Figure 7 Typical MIMD Memory Hierarchy

CPU cycle. In a simple-minded world we would stop there, and equip the CPU with a gigabyte of water-cooled very fast memory. In the harder world where people who do this go bankrupt, we make use of the observation that most memory references are to locations close to a previous memory reference. A few kilobytes of expensive 'cache' memory is usually enough to keep the CPU busy most of the time, provided that sufficiently intelligent algorithms are used to exchange data between the cache and the much slower and cheaper 'local memory'.

In almost all modern computers, some megabytes of 'real' memory pretend to be gigabytes of 'virtual' memory by writing pages which are not currently in use to disk.

Moving back to MIMD, we re-insert the 'global memory' into the hierarchy. This doesn't appear to make things much more complicated, until we realise that there is only *one* global memory, whereas there are *many* CPU's and caches. If the algorithms needed when we had a single cache were supposed to be 'intelligent', then it is clear that 'omniscient' algorithms are needed to control multiple caches containing possibly conflicting copies of the same global memory. The result of this is that, whereas most users of SISD memory hierarchies can remain unaware that the hierarchy exists at all, efficient use of memory in MIMD machines normally requires the programmer to take some explicit control of the use of cache and local memory.

5. ARCHITECTURAL REVIEW

In this section I start to go a little deeper into the architecture, strengths, and weaknesses of various approaches. Where appropriate I discuss real examples, using especially the RP3 as a (nearly) living example of a range of techniques which may be important for future general purpose parallel computing. Even more down-to-earth information about my own experiences is reserved for the next section.

In section 3 of last year's lectures I gave outline reviews of vector, deep-pipeline, tightly-coupled and loosely-coupled computers, to which the reader is invited to refer.

5.1 SIMD Computers

I will review SIMD by giving three examples. The first, the AMT DAP 510, is a commercial product which attempts a wide, although by no means general, range of applications. The remaining two were conceived and optimised for specific tasks. It is my view that this is also typical of the future for SIMD machines: their range of applicability will be large enough to make commercial production profitable, but they are not the most effective parallel machines for most problems.

AMT DAP 510

Active Memory Technology is a new company which has taken over the production and development of the ICL 'Distributed Array Processor'. Figure 8 shows the structure of the DAP. An array of 32×32 single bit processors, each with 32 kilobits of memory receives instructions broadcast by a master control unit. In its turn, the DAP is controlled by a host computer, such as a VAX. For most applications, the potential power of the DAP is made effective through use of its significant internal and external communications possibilities. Each processor is linked to its four neighbours, and each row and column has its own bus with the possibility of external connection. These external busses can support a total of 70 megabytes/second of I/O. Although the hardware provides only single bit arithmetic, compilers provide the necessary instructions to perform 8, 16, 32 or 64-bit arithmetic (at decreasing speed of course).

Commercial success for this sort of machine may be assured by its applicability in military image and signal processing. Seismic data processing (i.e. oil exploration) is also likely to be efficiently performed by these machines. In large HEP experiments we often find ourselves building specialised arrays of fast 'trigger' electronics, which have certain superficial similarities to the DAP. The trigger for many current and future experiments requires parallel processing of large numbers of similar signals to produce results within a few or a few tens of microseconds. It seems likely that commercial

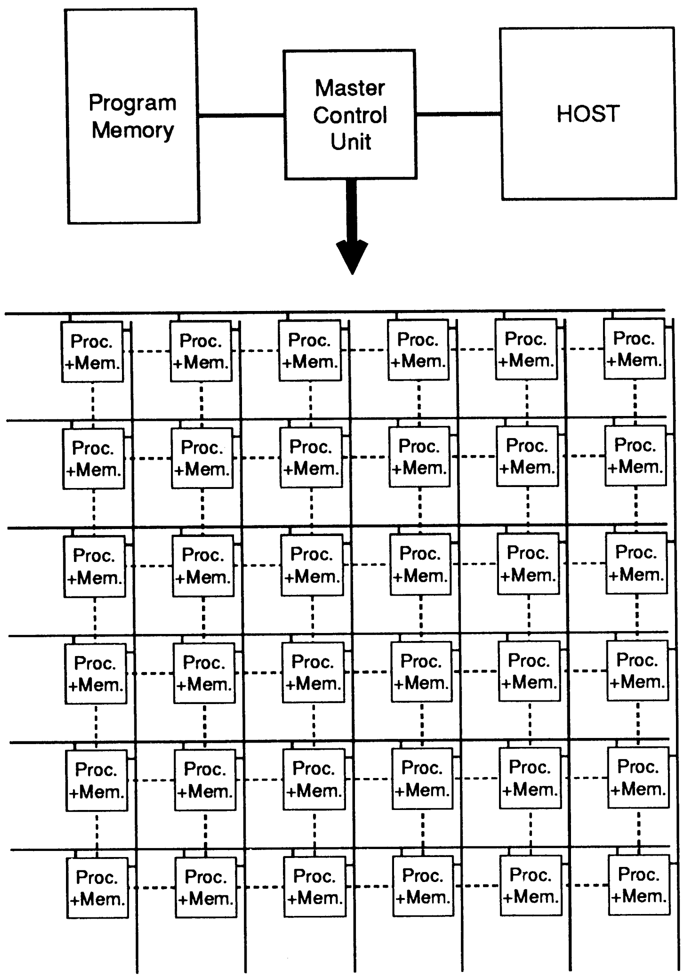

Figure 8 Structure of the AMT DAP 510

SIMD machines could be used effectively in future triggers, and I know of at least one serious study of this approach [2].

GF11

This machine, whose name derives from 'gigaflops 11', is under construction in IBM's Yorktown Heights research laboratory [3]. (A 'flop' is a 'floating point instruction per second'.) GF11 has been motivated by the computational needs of QCD lattice-gauge calculations. Quantum Chromodynamics is the apparently correct theory which describes the strong forces, which, for example, bind triplets of quarks together to make protons and neutrons. Although QCD has had a total qualitative success in describing strong interactions, it is exceedingly difficult to make quantitative calculations, except in cases where the effects of QCD are relatively small and perturbative calculations can be used.

A promising 'brute-force' approach to calculating non-perturbative quantities like masses and magnetic moments of nucleons, is to simulate a lattice of space-time points in which the value of a variable at one point depends directly only on its immediate neighbours. Existing lattice-gauge calculations, which have used a few hundred hours of CRAY-1 time (approximately 100 megaflops), have indicated that results precise enough to test QCD at the 1% level would require tens or hundreds of CRAY-years. Progress in QCD 'supercomputation' is reviewed in reference [4].

GF11, which is now close to completion, will comprise 576 processors, each of 20 megaflops for 32-bit calculations. Each processor has 64 kilobytes of fast 'static RAM' memory and 0.25 to 2 megabytes of slower 'dynamic RAM' memory. Communications between processors and memories is through a dynamically reconfigurable shuffle network. A single controller broadcasts instructions to all the processors, and reconfigures the network periodically.

The whole array is connected to an IBM 3084 host computer, which must oversee the medium and long term progress of the calculation since GF11 itself can only execute relatively short subroutines before returning control to the host.

Physically, GF11 has a striking resemblance to the 'counting room' of a large high energy physics experiment. Its racks dissipate some 200,000 watts of heat. Even in its design parameters, there is more of experimental physics than commercial computer manufacuture. The number of 576 processors was chosen so as to have a resonable chance that at least 512 would be working at any one time!

APE

The 'Array Processor with Emulator' [5] is another SIMD machine created with lattice-gauge QCD in mind. APE is a collaborative effort between several INFN supported groups in Italy, and a small group at CERN. Construction is now at an advanced stage.

APE occupies only two crates, but has an expected performance of 1 megaflop for lattice-gauge calculations. In order to keep the processor-memory network simple and cheap, APE has only 16 processors and 16 memories. Each processor, constructed from 32-bit Weitek chips, has a power of 64 megaflops, optimised for complex (i.e. real and imaginary) arithmetic. Each memory has 8 megabytes. Instructions are broadcast to the processors and the switching network by a 3081/E emulator. The 3081/E is a general purpose scalar CPU, and although it must be attached to some host for downloading the program and recovering the results, it is quite capable of managing the progress of the whole calculation.

5.2 MIMD computers: RP3

I have chosen to look in detail at the IBM's Research Parallel Processing Project, RP3, rather than to skim over the whole range of commercial and research machines which exist or are under development. This is not because I think that RP3 is the best possible machine, but because it embodies a wide range of concepts, of which some may be important in future massively parallel machines.

Like the GF11, RP3 is a research project and not a product. Unlike GF11, RP3 does not have a particular computational mission. RP3 is intended to be tool for a study of general purpose parallel computing. Intentionally, RP3 contains all hardware features which just might be important, so that studies of software techniques for parallel computing can proceed in as real an environment as possible.

Outline of the RP3 Architecture

The structure of RP3 [6]is shown in Fig. 9. The final machine will have 512 processor-memory-elements (PME's). The PME's contain 32-bit CPU's, supplemented by a vector/floating-point co-processor, a cache, and up to 4 megabytes of memory. The memory in each PME can be arbitrarily partitioned into local memory and global memory, so that part of the address space seen by each CPU will be local, and part (up to 2 gigabytes) will be global.

Figure 9 does not show the monitoring and diagnostic hardware. Each PME has seven monitoring busses which collect information about activity in all parts of the processor and the memory hierarchy.

If all processors can be kept busy, the RP3 will achieve about 1300 MIPS, or about 800 megaflops. It can be connected to the outside world by 64 IBM channels, each capable of 3 megabytes/second, and internally, its total processor to memory bandwidth is about 13 gigabytes/second.

Memory Interleaving

A further intriguing sophistication is embodied in the memory interleaving system. The memory interleaving hardware offers the possibility of treating some of the low order bits of a memory address requested by a CPU, as high order bits when the request is sent over the network. Thus, for example, sequential memory locations as seen by a CPU, might be arranged to refer to sequential PME's in the global memory. This seemingly bizarre approach can be effective in reducing network contention for access to very popular areas of global memory. Even more radical is the possibility to break up sequential memory references according to a 'hashing' or pseudo-randomising algorithm.

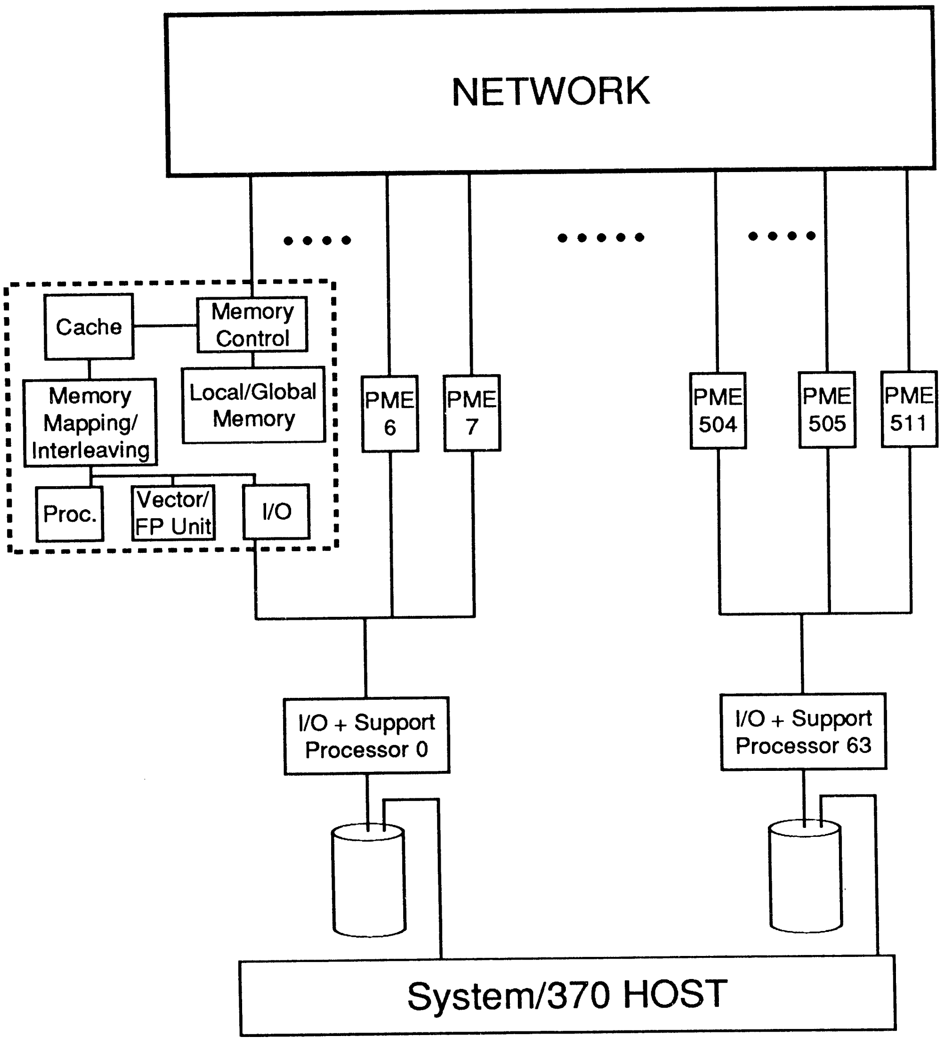

Figure 9 Outline of the RP3 Architecture

Networks

Having supported almost every conceivable combination of local and global memory, the RP3 designers went on to include almost every conceivable capability in the networks linking the PME's and memories. This led quickly to a need for two networks, one of which would be stupid and fast, and another which would be clever and slower.

The stupid and fast network is more correctly called the 'low latency non-combining network'. This network carries the main data flow. The term 'low latency' just means that the traversal time has been minimised, and the term 'non-combining' means that it does not do all the clever tricks performed by the other, 'combining' network. The non-combining network is a shuffle network of 4×4 switches and 4×2 concentrators. There are two 8-bit paths from each PME to each memory, and there are separate networks for carrying requests (CPU to memory) and responses (memory to CPU). To make it as fast as possible, the non-combining network is built from water-cooled TTL electronics, packaged in the same 'thermal conduction modules' used in IBM's largest mainframes. A one way trip through the network involves a concentrator, four switches, and a de-concentrator, and takes about 500 nanoseconds.

The clever, slower 'combining network' is intended for synchronisation and control rather than data-flow. The construction of a separate and complex network for synchronisation information came after simulation studies had shown that this vital element of any parallel computation could cause paralysing congestion at 'hot-spots' in the non-combining network. The combining network avoids hot-spots by intelligently combining requests as they pass through the switches.

The heart of the RP3 combining network are 2×2 combining switches, whose behaviour is shown in Fig. 10. When the combining network starts to become congested, due to many CPU's addressing a few memory locations, queues begin to build up at the input to the 2×2 switches. The combining switches are sufficiently intelligent to recognise queued requests which refer to the same memory location and to combine these requests before sending them onwards throught the network. The switches must keep a record of combined requests, so that when the response comes back, it can be de-combined to satisfy the multiple requests. Implicit in these ideas is the integration of the request and response networks. 'Fetch-and-op' instructions are very valuable tools in parallel computing, since they can be used where necessary to impose synchronisation and sequencing on the work of many processors. It is logically possible to merge fetch-and-op instructions in a combining network and Fig. 10 shows how the RP3 combines two fetch-and-add instructions.

The RP3 combining network is composed of 2×2 switches and 4×2 concentrators. To keep the power consumption and cost of the switches at a reasonable level they are constructed using NMOS rather than TTL technology. The slower switches, and longer path, make the transit time of the combining network several times greater than that of the non-combining network.

RP3 Memory Hierarchy

The sizes and access times of the components of the RP3 memory are shown in Table 2. To me, the most striking feature of the access times is the almost negligible extra overhead involved in accessing global instead of local memory. I would not be in the least surprised to hear that an upgraded RP3 was planned, using the same network, but much faster CPU's and somewhat faster memories.

Status of the RP3 Project

The RP3 is intended to be adventurous in its possibilities as a research tool in parallel

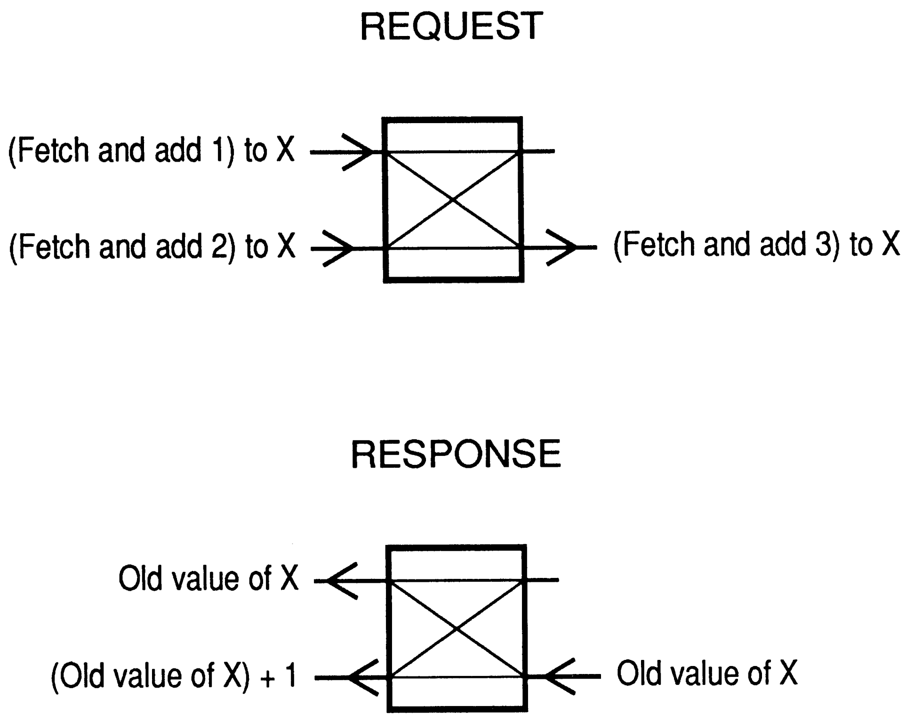

Figure 10 Principle of the RP3 Combining Network

Table 2

RP3 Memory Hierarchy

Memory	SIZE	ACCESS-TIME
CACHE	32k bytes	1 cycle 200 nanoseconds
LOCAL	4M bytes	10 cycles 2 microseconds
GLOBAL	2000M bytes	15 cycles 3 microseconds

computing, and a definite effort has been made to be unadventurous in constructing the hardware. Wherever possible standard IBM packaging techniques, power supples, cabinets etc. have been used. The use of a cabinet about 60 cm higher than the IBM standard was only unwillingly accepted (it won't fit into most elevators). The result of this approach is that the RP3 is huge and expensive, although no precise cost figures are available. The final machine will be composed of eight 'octants', each containing 64 processors, covering a total of about $100m^2$. The first octant has recently been completed, and the IBM internal cost of replicating this octant is in the region of \$2M per copy.

6. ARCHITECTURES IN THE REAL WORLD

This section is the place for relating real 'hands on' experience, and concrete plans arising from the search for a solution to the computing needs of the L3 experiment. Almost all of this section remains unchanged from last year where it appears as section 4, 'Benchmarks and Anecdotes', and I will not repeat it here. The new material this year is a description of the L3 computing system LEPICS.

6.1 LEPICS

LEPICS is the L3 Parallel Integrated Computer System. The priority for LEPICS is to produce physics results; achievements in computer science will be welcomed, but must take second place. These priorities, together with the need to have the system working as early as December 1987, dictated the choice of a host plus attached processor system. The main functions of the host are to provide entirely conventional interactive, disk, and tape support, and to offer sufficient high-speed communications channels to the attached processors.

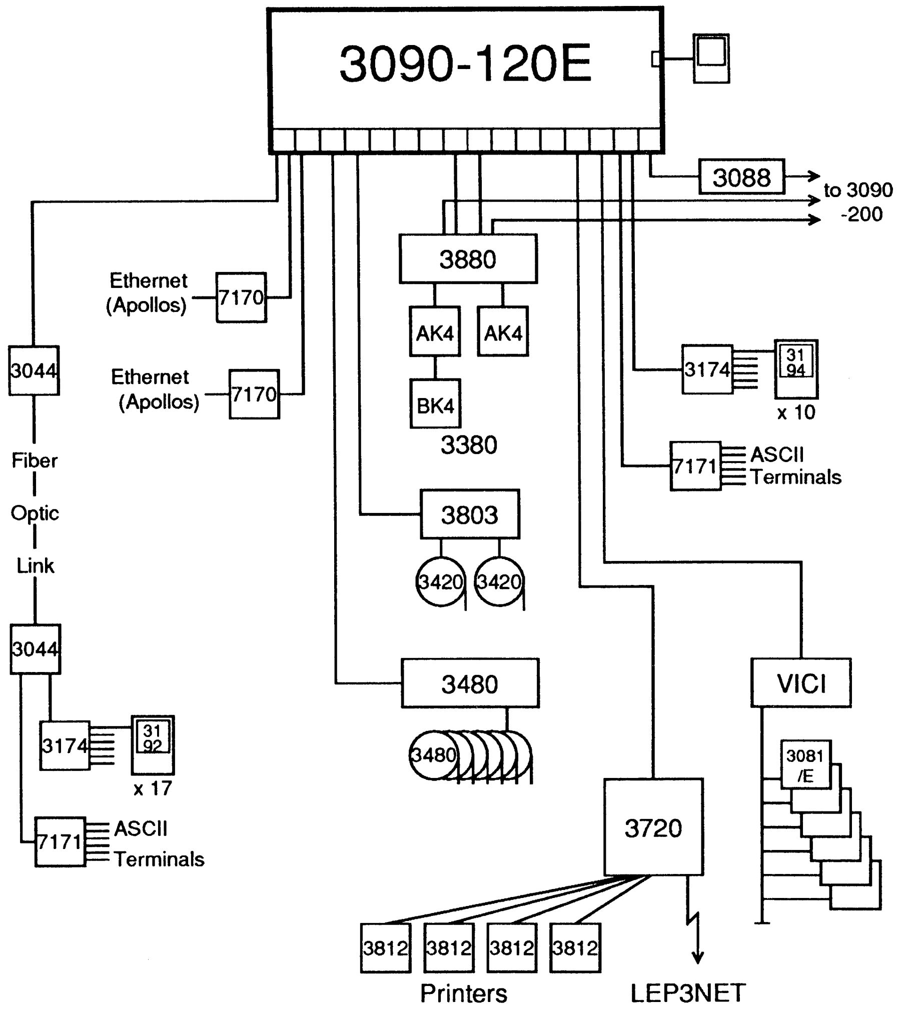

Figure 11 LEPICS - December 1987

Figure 11 shows LEPICS as it will be by the end of 1988. The system consists of an IBM 3090-120E processor with peripherals and with an initial complement of six 3081/E emulators. During 1988 the 3090-120E will be upgraded to a 3090-180E with additional memory.

In Figure 11 it seems that the importance of the peripherals overwhelms that of the attached processors. To some extent this is true. LEPICS will have 22.5 gigabytes of disk space, eight tape drives, two Ethernet interfaces, four printers, over 100 terminal ports, and several network connections. Connectivity is also emphasised in Fig. 12 showing how LEPICS will be linked to terminals and computers at CERN and outside. Thus LEPICS will be immediately capable of becoming the centre of all L3 computing.

The 3081/E emulators will be connected through a VICI (VME to channel) interface already developed as a joint CERN-IBM project. During 1988, L3 will work on

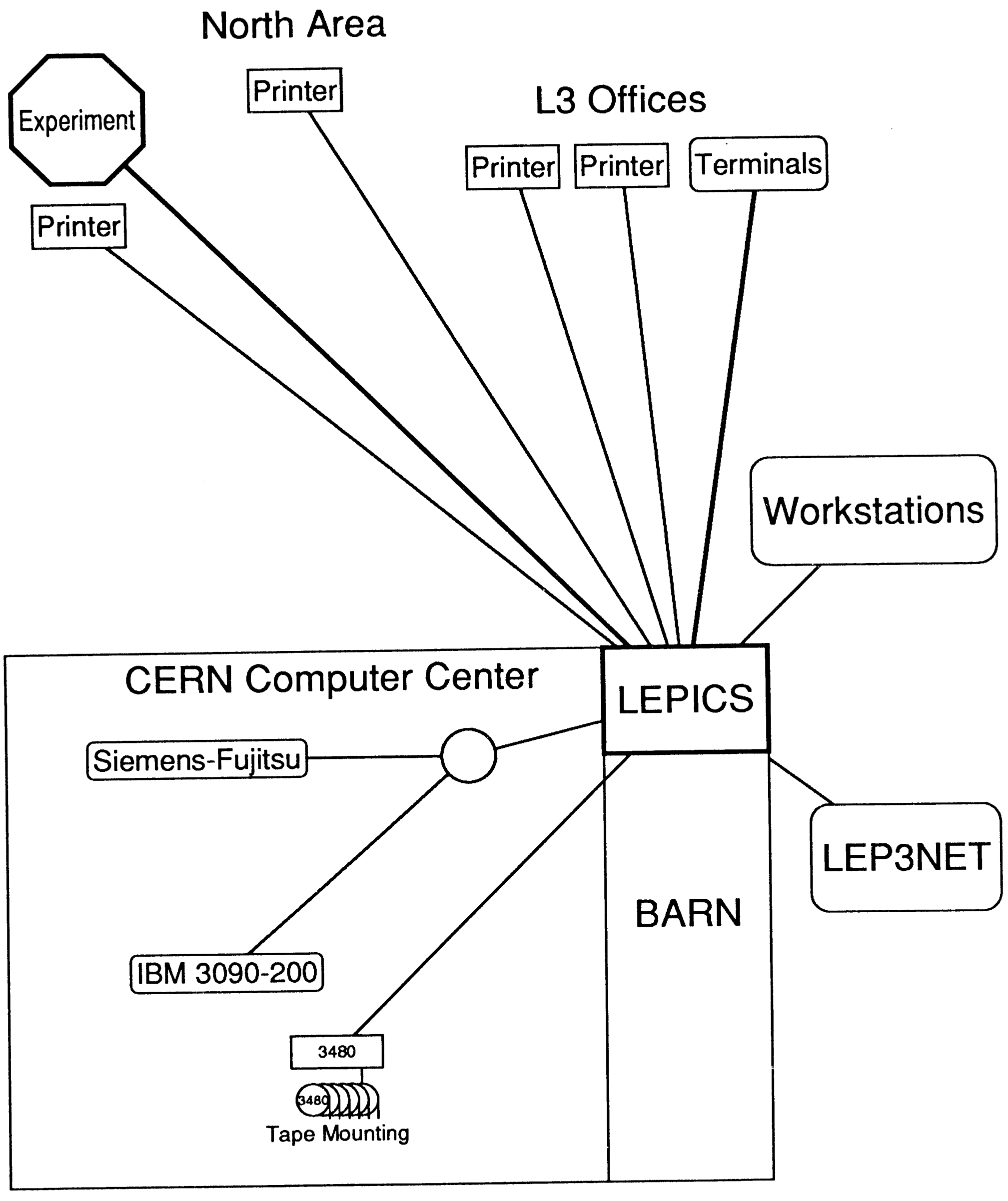

Figure 12 LEPICS Communications

integrating the attached processors into the batch subsystem of LEPICS, so that by the time LEP starts running, LEPICS will be able to add substantial parallel computing resources to its already significant conventional computing facilities. It is important to note that even on the machine initially installed, there are several free channels, each capable of 3 megabytes/second communication with attached processors. Once the attached processors have been successfully integrated, it will be relatively easy to expand the parallel processing capability by adding more processors on the spare channels.

Bolting LEPICS together is a relatively trivial operation; the interesting and difficult part of the project will be developing software to support parallel processing. The L3 plans will be outlined later in these lectures.

7. COMMENTS ON PRICE/PERFORMANCE

This is not an attempt to select a 'best buy' as in a report on video recorders or washing machines. The aim of this section is to try to show what you get for your money as a function of the computer architecture chosen. It is not easy to discuss price/performance in abstract terms alone, so I have chosen to centre the discussion on the systems I have reviewed. The performance of some systems has changed since the L3 tests, and prices (after discount) can vary by amazing factors, particularly for special customers like CERN and universities. No price I give is accurate to better than 30%.

Having absorbed these caveats, please turn to Table 3, which summarises my price/performance comparisons. I have already commented on the variability of prices. The performance of machines which depart from the classical scalar architecture depends greatly on the code, and on the amount of re-writing we are prepared to do. For most machines I have expressed this dependence as a range, and I have taken the middle of the range when calculating '$ per VAX'. For the CRAY, as for other vector computers, I cannot do this because I really have no idea what range of performance we will finally get. I have therefore been a little unfair in taking the worst case of unmodified HEP code.

It is not meaningful to compare systems without taking into account how complete each system is (from a hardware point of view), and how easy each system is to program. I give four blobs to systems which have all the disks, tapes, printers and terminal lines needed to be considered complete, whereas I give three blobs to a CRAY which normally

Table 3

Approximate Price/Performance Comparisons

Computer	Price $M approx	HEP VAX equiv	$k per VAX (batch)	How complete a system?	How easy to program?
Scalar and Vector Mainframes					
IBM 3090-200	6	40	150	●●●●	●●●●
CRAY X-MP *(4 CPU)*	14-17	100 $\longrightarrow$?	155	●●●	●●●
Pipelined Attached Processors					
FPS 164	0.25	3-6	56	●●	●●●
3081/E	0.033	5	7	●	●●
SISD					
AMT DAP 510	0.2	5-200	6	●	●
MIMD					
ELXSI 6400 *(10 CPU, 1985 model)*	2	40	50	●●●●	●●●
RP3	10-20	500-1000	20	●	●●
Clementi Machine *(1984 version)*	6	40-70	109	●●●●	●●●
LEPICS *(Large host plus 6 emulators)*	2	41	49	●●●●	●●●
3081/E Farm *(small host plus 6 emulators)*	0.7	32	22	●●	●●
Don't Forget					
Motorola 68020	0.0001	1	0.1		●●●●

expects most users to work on an addtional front-end computer and I give one blob to a bare emulator or AMT DAP.

When assessing how easy each processor was to program, I gave four blobs to anything which, when provided with any missing peripherals, would become as easy to use as the large mainframes.

To emphasis the importance and logic of the 'how complete' and 'how easy to program' criteria, I have included an entry for a bare CPU chip, the Motorola 68020. The bare chip is, of course, totaly incomplete and merits no 'completeness' blobs at all. Conversely, when provided with the necessary memory and peripherals, it can form the core of a very pleasant system (like the Apollo on which I am typing this text.)

Table 3 shows that by moving away from the conventional scalar mainframe solution to HEP computing we may be able to get up to three times as much CPU power for our money, while still having a complete system which is only moderately painful to use. Futher savings or performance improvements by the use of highly specialised machines would be counterbalanced by the need to buy a conventional mainframe to do the rest of the work.

8. SOFTWARE

Success in a computing task is by no means assured simply by taking delivery of a large quantity of hardware. This is especially true if the hardware departs in any way from the conventional scalar processing systems which we are accustomed to use.

Before I review the software problems in more detail, let me air my prejudices. Although both vector and parallel computing systems are tricky to use efficiently, I believe that computing for experimental HEP can be implemented in a much more natural fashion on parallel computers than on vector computers.

8.1 Software for Vector Computers

HEP can use two, almost distinct, approaches:

1. Totally re-think the algorithms so that 'von Neumann' (one thing at a time) code becomes efficiently vectorisable. The work on the Fermilab E-711 experiment track-finder is an example of this.

   This approach is likely to result in clear, intelligible code which is efficient on a specific vector architecture.
2. Parallelise the logic so that many unrelated but similar operations are performed at the same time. For example, re-organise a Monte-Carlo tracking program so that all the trivial co-ordinate transformations involved in tracking N particles through one step can be performed simultaneously.

   The likely result is tricky code, difficult to write and maintain, and still tied to a specific vector architecture.

I have spent years preaching the necessity of writing clear, maintainable, portable Fortran code, so both of these approaches worry me. However, if we ignore these problems, what speed-ups are we likely to get as a result of a vector-specific software effort?

It has already been demonstrated that specific algorithms can be speeded-up by around a factor of 10. How much can complete programs be speeded-up? My own feeling is that a factor of 3 is the likely maximum.

8.2 Software for Parallel Computers

The general problem — how to apply N processors to a single task — is one of computing science's most important challenges. I think that this will still be considered an important challenge well into the 21st century.

I will only concern myself with HEP specific solutions which are far from general. However, the foundation of my approach is parallelisation by a minimal reorganisation of the computing task. In other words, I advise stepping back and thinking about the task for a few minutes, rather than rushing in with any existing automatic parallelisation tools.

After a quick look into the long term future, I will concentrate on the practical problems of implementing parallel processing on the range of systems currently accessible to HEP, including, of course, the LEPICS system which will be used by the L3 collaboration.

The Long Term Future: Parallelising Compilers

Parallelising compilers recognise code that can run in parallel. The most obvious way to do this is to examine Fortran DO loops to see whether one iteration could be calculated at the same time as another. The compiler must find any ways in which one iteration uses variables calculated by another. If such dependencies are found, it may still be possible to allow parallel execution with the insertion of a synchronisation point where one process waits until the variable it needs is ready. Hardware supporting fetch-and-op instructions may also be valuable in removing possibilities of error due to simultaneous attempts to update a memory location.

The parallelising compiler has to identify each variable as private to a particular parallel process, or shared, and for shared variables it has to decide whether any process can ever be allowed to use its cache memory. Many machines allow individual memory pages, or even locations, to be declared cacheable or not cacheable.

Finally, a parallelising compiler inserts the necessary system service calls to create and communicate with parallel processes.

The parallelising compilers which exist today can analyse only relatively small loops. As a result, they produce code which can only run efficiently on tightly-coupled, low overhead architectures. Many highly parallel applications, including HEP, require recognition that blocks of 20,000 lines and 200 subroutines are parallelisable. There seems little doubt that parallelisation on this macroscopic scale will not be automated for some decades.

The Structure of HEP Batch Jobs

Figure 13 shows the very simple structure of most HEP batch jobs. Almost invariably such jobs involve processing events. Processing one event may take minutes (for Monte-Carlo simulation) or milliseconds (for Data Summary Tape analysis) but the job structure is the same.

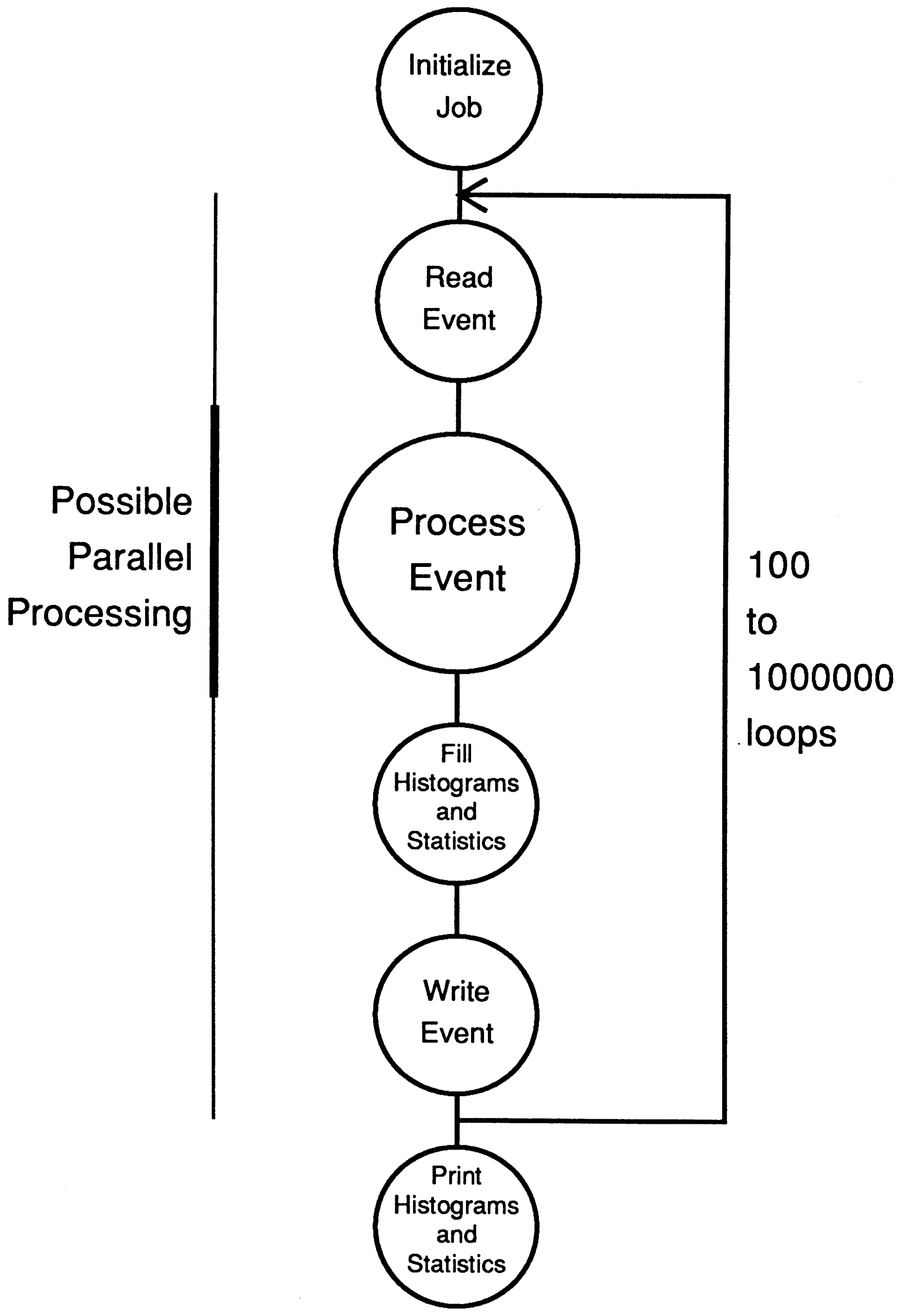
Initialize Job
Read Event
Possible Parallel Processing
Process Event
100 to 1000000 loops
Fill Histograms and Statistics
Write Event
Print Histograms and Statistics

Figure 13 The Structure of HEP Batch Jobs

Since the events are independent, they may easily be processed in parallel. Man-power, organisation and sanity dictate that we should try to manage the parallel processing within one job, rather than running many jobs at once.

Parallel processing becomes trickier when we want more than one process to access a single peripheral or a single memory location. The simplest way to solve this is to allow only one process to read and write events, and fill histograms. In most cases this works perfectly, since the 'process event' kernel still dominates the CPU requirements.

On tightly-coupled machines, it is usually possible to let the parallel 'child' processes fill the histograms and statistics in shared memory. Some machines even offer 'fetch and op' instructions which require no memory locking. On loosely-coupled machines it is usually better to let the host handle the histograms.

How to use an Emulator Farm

An emulator farm consists of a small host computer coupled to a number of attached processors. The smallness of the host computer means that it must be given relatively little work to do if the farm is to work efficiently. In the context of HEP batch jobs, this means that the 'process event' kernels which are farmed out to the emulators must involve seconds or minutes of computation.

L3 has had access to a farm cosisting of an IBM 4361 host, and five or more 3081/E emulators. This resource has encouraged the creation of a parallel version of the L3 Monte-Carlo simulation program, and has made us aware of the potential and the problems of systems using cheap attached processors.

These are the components of the software system which we have used:

- Source code pre-processor (PATCHY [7]).

  It is highly desirable to maintain the parallel version of a program together with the scalar version. The code which will run in parallel is flagged and extracted automatically by a pre-processor for separate compilation and linking.

- Data-structure manager (ZEBRA [8]).

  By using ZEBRA to manage data within a program it becomes simple to transport any data to another processor at any time. ZEBRA knows all about the data types (floating, integer, etc.) of the data it manages, so automatic format translations are possible.

- Cross compiler.

  This is provided by the combination of the IBM compiler and the 'translator'. (Note that the 370/E emulator, developed by the Weizmann Institute, Rutherford Lab. etc., operates directly on IBM object code).

- Cross linker.

- Debugging Tools.

  The best place to debug the Fortran code is on the host system. It is also advisable to debug the parallelised version, using a system like VM/EPEX (VM Environment for Parallel Execution [9]), to simulate a multi-processor system without using real emulators.

  After debugging on the host system, the code should run on the emulators without problems. This will probably be almost true, eventually, but at present occasional problems with the translator or with the emulator hardware are tricky to debug and are best handed over to the few experts. From necessity, the experts have created a limited set of hardware and software tools to help with this debugging.
- FORTRAN calls to:
  - download a program module
  - download data
  - start emulator
  - wait for emulator to finish
  - upload data
  - interrupt emulator
- Resource Management.
  - Phone number of the system manager.

When an emulator farm is used by one or two teams to run week long chains of production jobs, this method of resource management is quite adequate. It is clearly inadequate to support attempts to make an emulator farm a more general purpose resource.

How to use a Shared Memory System

A shared memory system is suitable for almost 'mindless' parallel processing of HEP events. The overheads are sufficiently small that few users need be concerned about whether they are doing enough computation on each event to make parallel processing worthwhile. Computational kernels of a few tens of milliseconds can already make efficient use of parallel processing.

Although L3 has never used such a machine as a serious resource, we did sufficient work on the ELXSI 6400 to understand quite well how we would use such a machine. As for the emulator farm, I list the necessary elements of the software system:

- Source code pre-processor (as for an emulator farm).
- FORTRAN calls to:
  - create child processes

  - start children
  - share memory with children
  - control cache memory
  - control synchronisation (semaphores and locks)
- Programmer discipline
  - access to shared memory
  - cacheing: what and when?
- Resource management
  - multiprocessing CPU's
  - load balancing by process migration
  - otherwise anarchy???

These limited resource management tools are effective in making sure that all the CPU's are busy, provided that the users create at least as many processes as there are CPU's in the system. They do not solve all the problems of providing efficient and prioritised parallel processing for a large anonymous user community.

To make it clearer what we really have to do, I list below the Fortran 'intrinsics' available to support parallel processing on the ELXSI system. Figure 14 shows how these intrinsics can be used to set up a simple parent-child system, which may be readily generalised to an arbitrary number of children.

<u>ELXSI Parallel Processing Intrinsics</u>

Most applications need only the following calls:

STATUS =MT$ShareMemory (Address, Length, Cacheable?)

MT$SetupSemaphore (Name_of_Semaphore, QueueLength)

MT$SetupTasks (Number_of_Tasks)
: creates identical copies of the parent at this instant.

MT$StartTask (Task_no., Subroutine_Name, Subroutine_Arguments)
: calls a subroutine within a child process.

MT$SignalSemaphore (Semaphore_Name)
: increment the named counting semaphore.

MT$WaitOnSemaphore (Semaphore_Name)
: if the named semaphore is greater than zero, continue, otherwise sleep until the semaphore reaches zero. (Sleeping processes consume no CPU cycles.)

MT$WaitOnTask (Task_no.)
: sleep until the child task RETURN's (or STOP's) from the subroutine called by MT$StartTask.

MT$DestroyTasks ()
: kill the children.

MT$WhatTaskAmI ()

returns its task number to a child.

Enthusiasts can also play with:

MT$SetupLock (Lock_Name)

MT$Lock (Lock_Name)

if the named lock is open, lock it and continue,
if the named lock is locked, spin in a NOP loop
until the lock is opened, then lock it and continue.

MT$Unlock (Lock_Name)

open the named lock.

MT$FlushMemory (Address, Length)

flush the cache-memory belonging to the calling process to ensure that the main memory contents are updated.

MT$FlushMemoryAllSharers (Address, Length)

as above, but for all processes sharing this memory.

How to use LEPICS: Parallel Processing in a Computer Center Environment

In the longer term, all the greediest users of computer power will be forced into parallel computing, and all those who, like HEP, also use peripherals heavily will be best served by computer centres.

For the next few years, serious parallel processing is avoidable by spending money on a few of the fastest available CPU's. Like most HEP experiments, L3 never has remotely enough money. This together will a need for massive expandability, and a clearly parallelisable computing load, makes a parallel processing computer center our best choice now.

While discussing emulator farms, and simple shared memory systems, I have begun to point out what is special about a computer center. With a large, possibly competing, user community, resources can no longer be scheduled by manual intervention or verbal agreements. As I have indicated above, the existing support for parallel processing largely ignores this resource management problem.

LEPICS will be a multi-CPU computer system, and will immediately have all the basic hardware and software tools for communication between the processors. The L3 software which will run on LEPICS is fundamentally parallelisable, and we believe that writing and maintaining software in a form immediately ready for parallel or scalar processing will have almost no negative impact on the intelligibility or efficiency of the code.

However, it will have escaped nobody that LEPICS is simply an emulator farm with a large host computer, and the initial resource management tool for LEPICS parallel

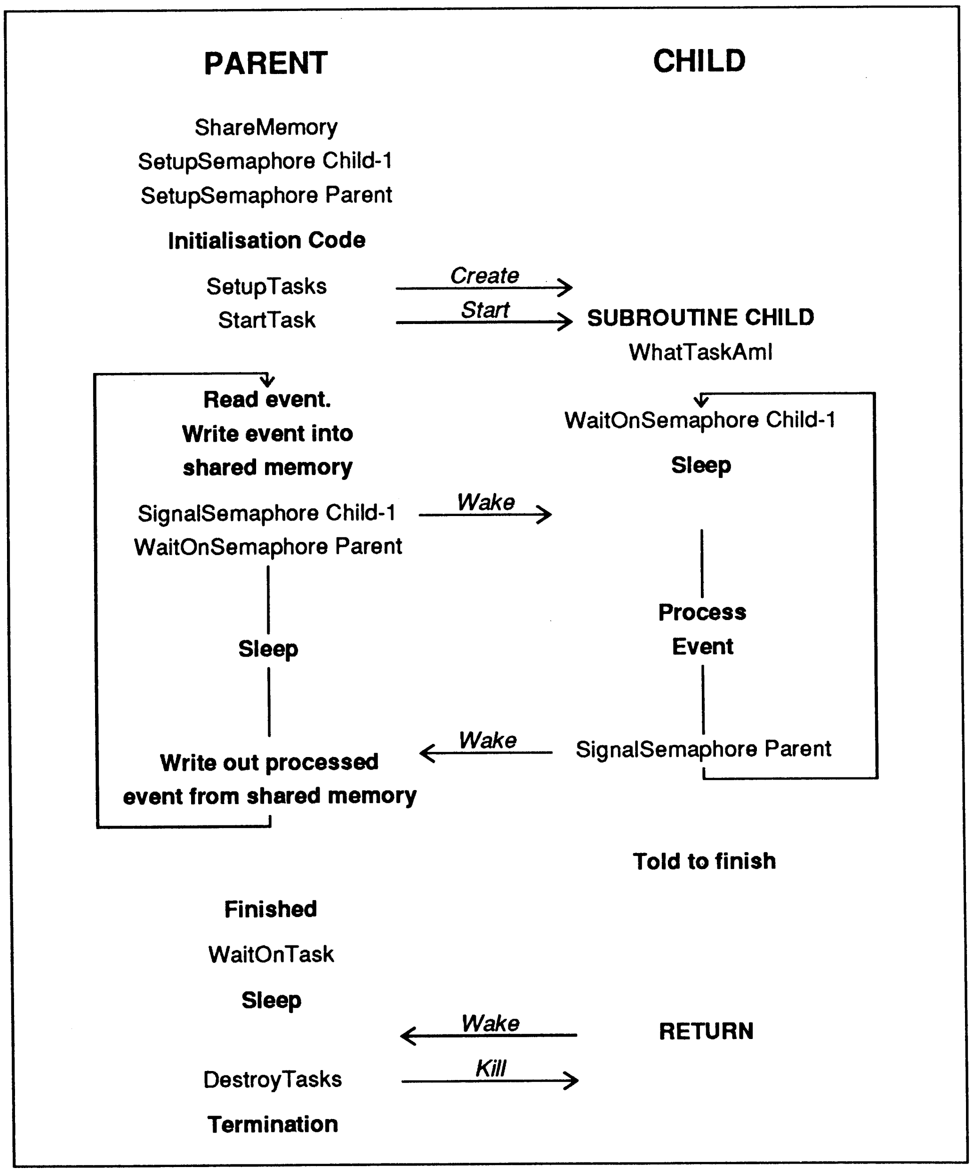

Figure 14 A Parent-child System using ELXSI Intrinsics

processing will be just the phone number of the system manager. In the one to two years which remain before LEP data begin to flow freely, we have to turn LEPICS into a parallel computer center. The core of the LEPICS Parallel Processing Project will be to make enhancements to the batch system which will:

1. Schedule batch jobs taking into account declared intentions to use attached processors.
1. During execution, allocate processors dynamically based on relative task priority and throughput considerations.......
   - offer executing tasks more processors if they become available.
   - request the release of processors by lower priority tasks.
   - force release of processors by un-cooperative tasks, for example by total or partial swap-out to disk.

2. Encourage throughput, for example by penalties for letting attached processors lie idle.

An outline of the way in which LEPICS parallel processing will be managed is shown in Fig. 15. LEPICS will run the IBM VM/CMS operating system, which is notable for its use of 'virtual machines' which were once so robustly separated from each other that they could only communicate by real I/O. Recent developments allow virtual machines to communicate through shared memory, but the operating system architecture still encourages a clear separation of functions between virtual machines. In the existing batch system (developed at SLAC), each batch job runs in its own virtual machine under control of the batch monitor machine. To support parallel processing on attached processors we must create a Parallel Processing Manager VM, and Interface VM's to control the VICI interfaces.

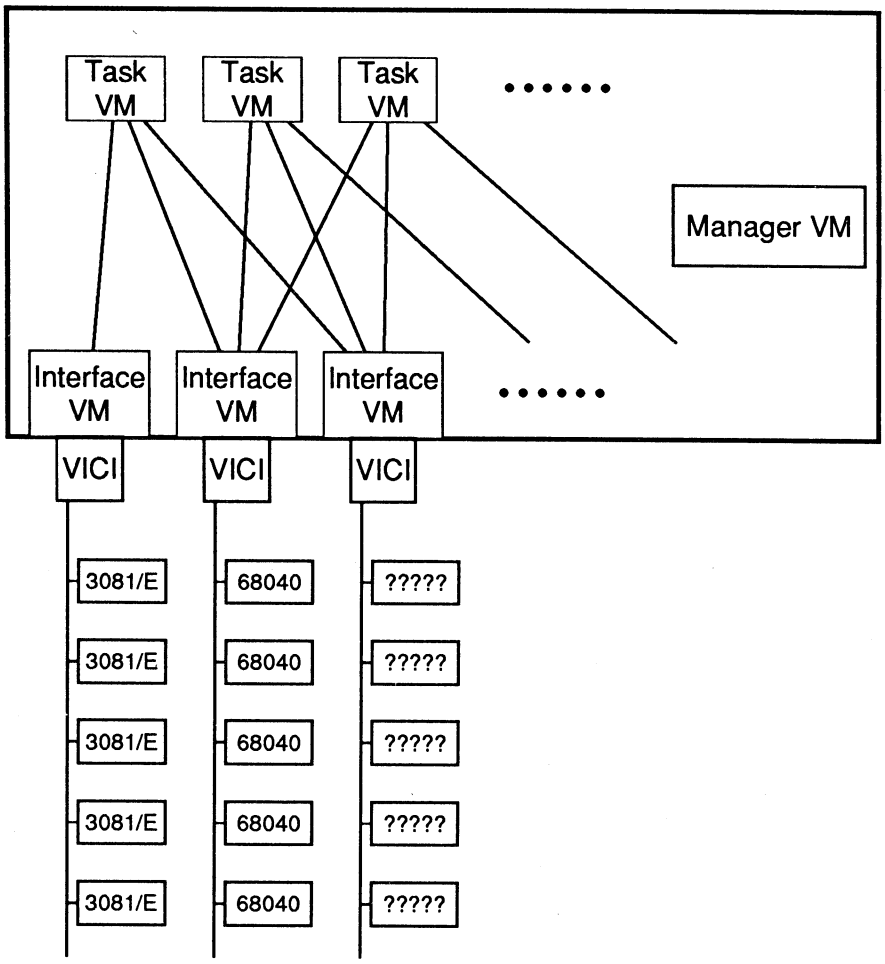

Figure 15 LEPICS: Attached Processor Management

The Manager VM will receive requests for attached processors, and send out offers or demands. It will also communicate with the interface VM's to monitor levels of traffic and CPU utilisation in the attached processor system. The use of Interface VM's brings several advantages. Firstly, the current version of the VICI does not allow more than one host VM to communicate with the devices attached to the VICI. Indirect communication through the Interface VM's can work around this problem. Secondly, the interface VM can present a standardised software interface to the task VM's without the need to freeze all developments of the interface and the attached processors. Finally, leading on from the last point, it should be possible to connect new, cost effective, VME based processors to LEPICS, without the need to modify user jobs. For example, if Motorola produces a '68040' with the power of a 3090 and costing $500, one of the Interface VM's could be modified to provide automatic translation of ZEBRA data from IBM to IEEE format, and the Manager VM could be modified to bear in mind the extra overheads of format translation when deciding which tasks should use the new processors.

9. CONCLUSIONS

I have satisfied myself, and I hope the reader, that increasingly parallel computing is inevitable in the long term future. Parallel computing is itself rather ill-defined given the amazing diversity of parallel architectures which it is technically possible to construct. I predict that the dominant forces will be economic, and to find which parallel architectures will succeed by the end of the century, we must ask which systems could be most effectively used in banking and insurance, as well as in HEP.

Most tasks run serially on computers are either unrelated, or naturally parallelisable. However, software, both for automatic parallelisation of large tasks, and for management of large scale parallel computing, will remain a major challenge for decades to come.

* * *

REFERENCES

[1] M.J. Flynn, Very High-Speed Computing Systems, Proc. IEEE, 54 (1966), no. 12.

[2] R. Bock, R. Dobinson, private communication concerning Real Time Data Processing Studies within the LAA Project.

[3] J. Beetem, M. Denneau and D. Weingarten, J. Stat. Phys., 43 (1986) 1171.

[4] D. Toussaint, Supercomputation in QCD, Computer Physics Communications, 45 (1987) 111.

[5] P. Bacilieri et al., The APE Project: A Gigaflop Parallel Processor for Lattice Calculations, Computing in High Energy Physics, L.O. Hertzberger and W. Hoogland (Editors), North Holland, 1986.

[6] G.F. Pfister et al. The IBM RP3: Introduction and Architecture, Proceedings of the 1985 International Conference on Parallel Processing, IEEE Comp. Soc. 637 (1985) 764.
W.C. Brantley et al., RP3 Processor-Memory Element, Proceedings of the 1985 International Conference on Parallel Processing, IEEE Comp. Soc. 637 (1985) 782.

[7] J. Zoll et al., PATCHY, CERN Program Library.

[8] R. Brun, J. Zoll et al., ZEBRA, CERN Program Library.

[9] F. Darema-Rogers et al., VM/EPEX - A VM Environment for Parallel Execution, IBM Research Report RC11225 (#49161) 1985.