COMPUTER ARCHITECTURES FOR HIGH ENERGY PHYSICS

Richard P. Mount

California Institute of Technology, Pasadena, California, USA

ABSTRACT

The increasing cost of HEP computing leads us to seek more cost-effective sources of computing power. Vector, deep-pipeline, and parallel computer architectures are reviewed, and the results of perfomance tests are given, leading to a cost/performance/completeness analysis for a range of available systems. The software needed to make efficient use of vector and parallel computers is then examined in more detail.

1. INTRODUCTION

1.1 Why am I here at all?

I am a physiscist, and as such, I am more accustomed to being a user of computer systems than a buyer or creator of them. My interest in the computing needs of HEP, and of peculiar ways in which these needs might be met, stems from my involvement, from its earliest stages, in the L3 experiment at the LEP accelerator.

LEP is the 27km circumference electron-positron colliding-beam accelerator now being constructed in a circular tunnel straddling the Swiss-French border near Geneva. When the construction is complete in early 1989, four large 'experiments' will measure the products of the electron-positron collisions occurring at four equally spaced positions around the LEP ring. L3 is physically the largest experiment, and arguably the most computationally demanding due to its many very high precision detector systems.

A few numbers may help to impress and, perhaps, even inform. The L3 experiment will be assembled in a huge artifical cavern, 50m below the surface. Externally, L3 will appear to be a cylindrical magnet, 14m long and 17m in diameter, with a tube, 32m long and 4.4m in diameter running along the axis of the magnet. The magnet, containing 6000 tons of steel, and 1000 tons of aluminium coil will have a free bore of nearly 12m diameter by 12m length and a field of 5000 gauss. The tube will support the 700 tons of instrumented detectors which cover almost all the solid angle around the 'interaction point' at the centre of the structure.

Particles emerging from the central interaction point will be measured by tracking chambers and by calorimetry (total absorption in energy-sensitive devices). Most of the L3 detector systems are breaking new ground in the precision which will be attained. For example, the massive 'muon chamber' system, which fills the magnetised volume outside the support tube, will measure the position of particle tracks to better than 30 microns. Such precision has only become possible by resorting to ambitious and, naturally, expensive, engineering techniques.

It is considered foolish 'to spoil a good ship for a ha'p'orth of tar'. In this context the 'tar' is the provision of computing resources which will enable us to realise the potential of the L3 experiment, and the 'halfpennyworth' may be several million dollars worth. From the very beginning, we have worried about the computing resources we would need and how they might be provided. Here is the chronology of the L3 computing plans:

1982

Based on experience with earlier, smaller, experiments, we estimated that we would need a at least 50 VAX 11/780 equivalents on the CERN site for rapid off-line analysis of L3 data. (I will explain later why this is just a minimum).

CERN planning, based on available finance and not on expected need, put the likely CERN contribution to L3 at about 12 VAX equivalents.

1982-1986

L3 searched for ways to provide the missing computing power. Firstly by trying to get as much finance as possible from our funding body (in this case the US DoE.). Secondly by studying cost-effective architectures and benchmarking systems and components whenever possible.

1986-1987

A 'Request for Proposals for an Integrated Computer System for the L3 Collaboration', has recently been issued. Proposals are sought for a system comprising a Host plus attached processors which L3 will develop into a flexible parallel processing system. Offers are expected before the end of 1986 and installation will start in June 1987.

As a result of these activities, several L3 physicists have acquired an unexpected experience and understanding of computing architectures, and are frequently invited to give lectures such as these.

1.2 What is HEP Off-line Computing?

High energy physicists are greedy consumers of computing power. However, HEP cannot be satisfied by MIPS alone; for most tasks a full configuration involving expensive peripherals is a necessity. To explain and illustate this I will outline the activities typical of HEP computing.

Software Development

In most HEP experiments, the majority of the code used to process and analyse the data is written specially for the experiment. L3 will have a software base of between 300,000 and 500,000 lines of FORTRAN, of which 80% will be written by L3 members. The computers used for software development must support many users (30 to 100 for L3). Typically the users develop and test code using Edit/Compile/Link/Run cycles requiring both good interactive response and substantial CPU power. Productivity is improved by making plenty of disk space available to each user, and by having good printing facilities. Tape usage is not heavy, but it is needed for serious tests of software approaching its final state.

Production Processing

Production processing involves using the data from the experiment to reconstruct particle tracks and energies. It is also necessary to generate a large volume of simulated data by Monte-Carlo techniques, and pass this through the same reconstruction program.

All production processing requires massive CPU power and a large I/O volume. However, the I/O to CPU ratio is never very high. A 50 VAX equivalent system might perform I/O at a rate below 100k bytes per second for production processing. Monte-Carlo event generation, due to its high CPU requirements, will perform an order of magnitude less I/O.

HEP data do not come in a continuous stream, but in the form of distinct entities called 'events' which contain all the data relating to a single collision between (in LEP) an electron and a positron. Events are independent, and may in principle be processed in any order, or in parallel.

Physics Analysis

Physics analysis starts when the routine production processing is finished, and uses all the facilities at our disposal to reduce a massive amount of data to (we hope) startling revelations of the new physics accessible to our experiment. Since we don't know what new things we will discover, we cannot write all our analysis tools in advance, and much software development activity is included in 'physics analysis'. In addition, to manipulate the massive data samples in imaginative ways, physicists need large CPU and I/O resources available on demand. As in production processing, the data are still in the form of events, which can, in principle, be processed in any order.

Naturally, as scientists, high energy physicists spend most of their computing power on massive floating point calculations.......or do they? Figure 1 shows a randomly selected page of code taken from the GEANT program, which is a component in the L3 Monte-Carlo event simulator. Although my example (it really was chosen randomly) is perhaps a little extreme, more systematic analysis shows that floating point computations do not dominate HEP code, and that 'IF's and subroutine calls are very frequent. Vector arithmetic using vectors and matrices larger than 4×4 is uncommon even within floating point code.

1.3 Where Does the Money Go?

Computing systems which support the full range of HEP computing activities must have many peripherals. If you get a manufacturer's price list, and calculate the cost of a complete HEP computing centre, you will find that about 50% of your money is required for peripherals.

Looking at this in another way, even if you can reduce the cost of CPU power to near zero, with the aid of a bag of chips and a soldering iron, you still need a lot of money to set up an HEP computing centre.

In reality, the potential savings from cheap CPU power are even lower. Although it is realistic to do a lot of 'production processing' on cheap, home-made processors, it is very difficult to move highly interactive or peripheral-intensive work off commercial processors. Production processing uses 60 to 70% of the CPU power, but the remaining interactive and I/O intensive activities use at least 60 to 70% of the money.

Our search for new architectures for HEP must keep these cold realities in sight. However, although

```
      ENDIF
      IF(NVTX.EQ.1)TOFG=0.
      DO 4 I=1,3
    4 Q(JV + I) = V(I)
      Q(JV + 4) = TOFG
      Q(JV + 5) = NTBEAM
      Q(JV + 6) = NTTARG
      NTK=0
      IF(JKINE.GT.0)NTK=IQ(JKINE-2)
      IF(NTBEAM.GT.NTK)GO TO 90
      IF(NTBEAM.LT.0)GO TO 90
      IF(NTTARG.GT.NTK)GO TO 90
      IF(NTTARG.LT.0)GO TO 90
      IF(NTBEAM.NE.0)THEN
         JK = LQ(JKINE- NTBEAM)
         IF(JK.EQ.0)GO TO 90
         NVG = Q(JK + 7)
         NFREE=IQ(JK-2)-7-NVG
         IF(NFREE.LE.0)CALL MZPUSH(IXDIV,JK,0,2,'I')
         Q(JK + NVG + 8) = NVTX
         Q(JK + 7) = NVG + 1
      ENDIF
C
      IF(NTTARG.NE.0)THEN
         JK = LQ(JKINE- NTTARG)
         NVG = Q(JK + 7)
         NFREE=IQ(JK-2)-7-NVG
         IF(NFREE.LE.0)CALL MZPUSH(IXDIV,JK,0,2,'I')
         Q(JK + NVG + 8) = NVTX
         Q(JK + 7) = NVG + 1
      ENDIF
C
      NVERTX = NVTX
      IQ(JVERTX+1)=NVERTX
      GO TO 99
C
C             Error
C
   90 NVTX   = 0
   99 END
      SUBROUTINE GTAU
C.
C.    ************************************************
C.    *
```

Figure 1 A Randomly Selected Page of Code from the GEANT Program

smart computing ideas are unlikely to save much money, they may make feasible smart physics ideas, and thus greatly increase the real productivity of high energy physics experiments.

1.4 HEP CPU Needs and Conventional Solutions

Nobody need journey to a remote corner of the Netherlands to learn that HEP computing needs are increasing, whereas the cost of a unit of computing power is decreasing. However, if I can make some more quantitative statements, the travel might start to be vaguely justified.

HEP computing needs have continued to increase for many reasons. Here are some of them:

- Higher energy collisions generating more and more particles,
- Larger detectors (to contain higher energy particles),

- Fancier electronics, particularly 'flash ADC's',
- More simulated data needed to match high-statistics, high-precision real data.

I will pick on one of these reasons for futher explanation. In the old days (all of ten years ago) when we built a 'wire-chamber' we would send the signal from each wire to very expensive electronics which would set a single bit if a particle passed near the wire. These days, if we build a similar wire-chamber we will probably send the signal to a 'flash-adc' which digitises the signal on the wire to a precision of 6 or 8 bits every few nanoseconds. Instead of one bit, we now have thousands.

The rapid increase in needed computing power might not be a major problem if the decreasing cost of computing always made it possible for us to buy all the power we needed. Sadly, computers are not getting cheap fast enough. Figure 2 shows both the evolution of computing costs and the evolution of experimental needs. The computing costs in $/MIPS* (as quoted in 'Datamation') are shown for IBM mainframes over the last 20 years. Computing needs of HEP experiments are not reviewed in Datamation, so I drew on my own experience and estimated the MIPS per physicist for the five experiments in which I have had a substantial involvement. It seems clear from this figure that conventional computers are indeed not getting cheap fast enough. For those who wonder whether the experiments in which I work are perhaps weird and atypical, Fig. 3 compares the slope of the MIPS per physicist in Fig. 2 with estimates of computing needs of the US national laboratories given in a recent report[1] by a HEPAP subpanel. The long term extrapolation from my experience seems, if anything, low.

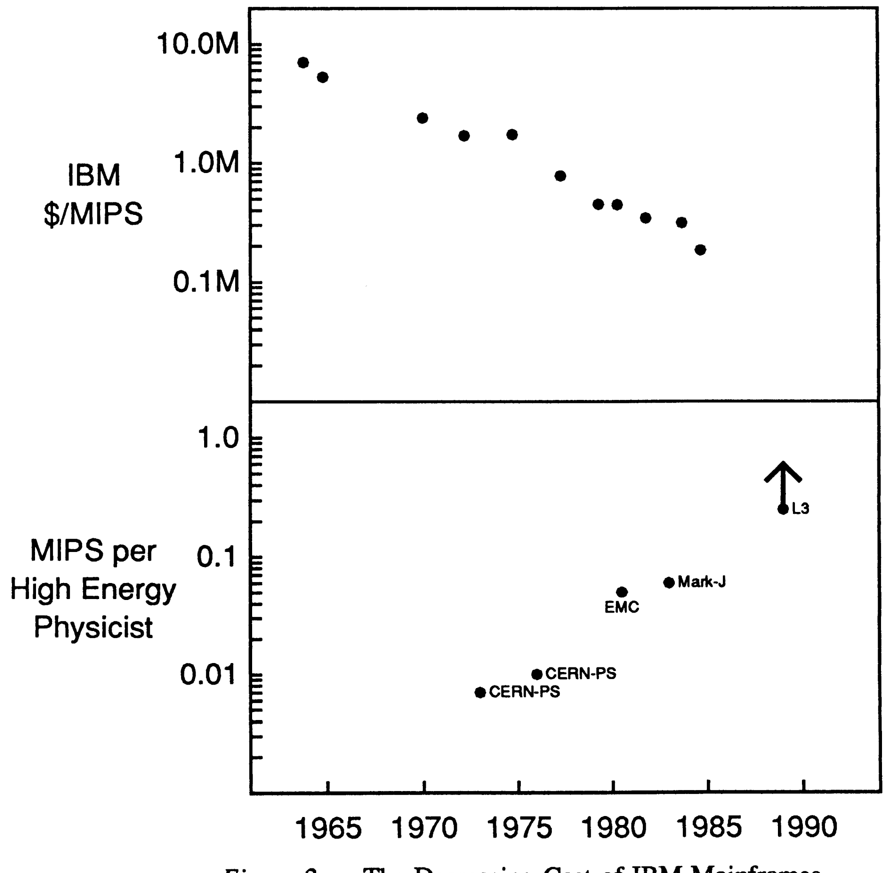

Figure 2 The Decreasing Cost of IBM Mainframes Compared with the Increasing Needs of High Energy Physicists

⋆ I do agree that the MIP is a 'Meaningless Indicator of Performance' under most circumstances, but in the case of comparison between IBM machines (mainly systems 360/370) it is not too distorted, at least on a logarithmic plot.

The price of IBM mainframes was about the most relevant quantity I could have plotted in Fig. 2. The conventional solution to HEP computing needs have always been to go out and buy the fastest scalar processor you could afford. This was quite a good idea, since as we have already begun to see, HEP code is really quite well optimised for conventional scalar machines.

Only the spectre of a total insufficiency of computing resources has led us to examine more cost-effective ways of doing our computing. Saving money has, of course, never been the prime motivation, but the prospect of getting vastly more computing power for very little more money proves very exciting. Many HEP experiments have been seriously compute-bound (I won't give any names here):

- data analysed once only
- insufficient simulated data
- mistakes not allowed (to be admitted)
- new ideas for analysis not encouraged.

Removing these restrictions can increase the quality of physics results just as surely as building a better detector.

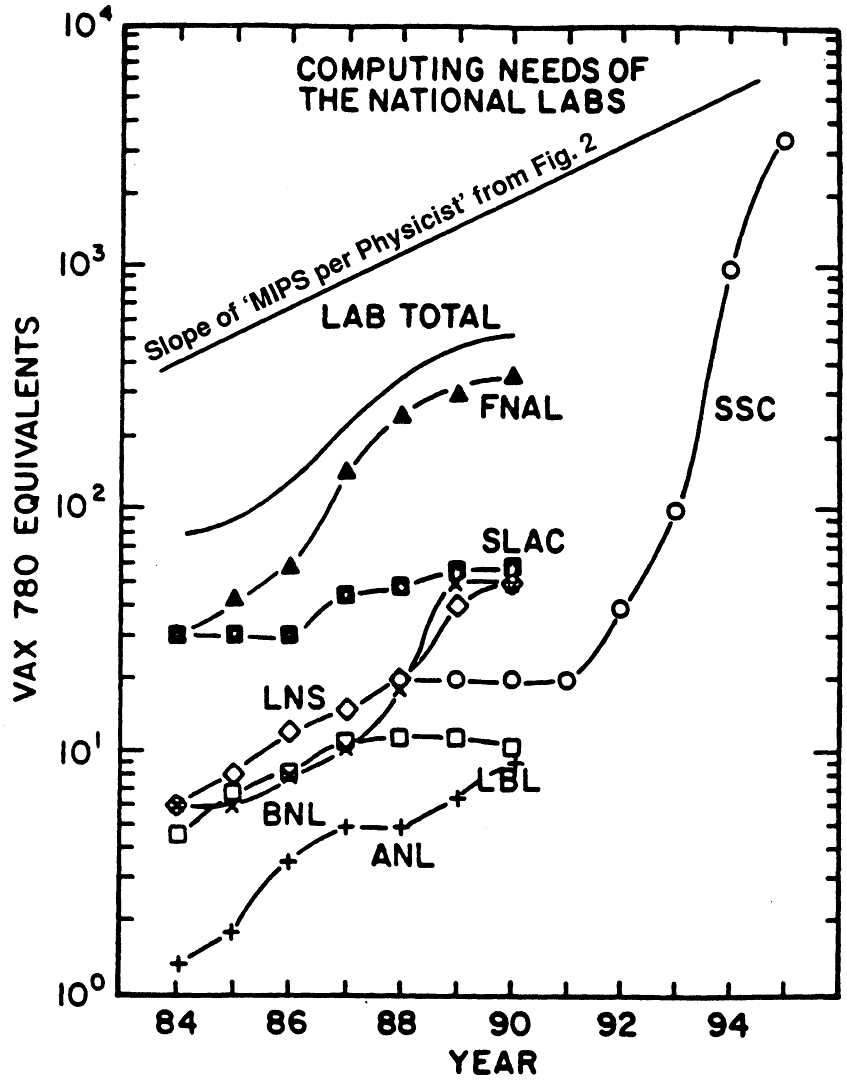

Figure 3 Computing Needs of US National Laboratories

2. THE FUNDAMENTAL PROBLEM

The fundamental problem facing all users of massive computing power is that the CDC 7600 is still a fast machine. Let me explain this cryptic statement.

CERN installed a CDC 7600 in 1972. It was the fastest computer in existence. When it was switched off, in August 1984, it was still one of the fastest scalar CPU's available. Even today, the fastest scalar CPU's only beat the CDC 7600 by a factor of two.

This story is partly a tribute to the genius of Seymour Cray, the designer of the CDC 7600, but it also illustrates how hard it has become to make faster scalar CPU's. Construction and maintenance costs continue to fall by about a factor of 2 every 4 years, but the cycle times of the fastest CPU's now decrease very 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. ARCHITECTURAL REVIEW

This review of computer architectures will be pragmatic and simple enough for even HEP experimentalists to understand.

All the architectures I will describe are attempts to solve 'The Fundamental Problem' by performing many operations in parallel. The technology of the machines is the same as that used in fast scalar computers.

3.1 Vector Computers (e.g CRAY, CYBER 205)

The so-called vector computers achieve their speed by performing many identical operations (almost) simultaneously. Since this is typical of what is required in vector and matrix arithmetic, the machines have long been named 'vector computers'.

Figure 4 shows the time relationship between successive multiply instructions on a vector machine. Although a multiply takes typically 7 cycles, it is possible to initiate a multiply every machine cycle. Provided that the operands are made available in the appropriate 'vector registers' it takes just 63+7 cycles to perform 64 multiply operations.

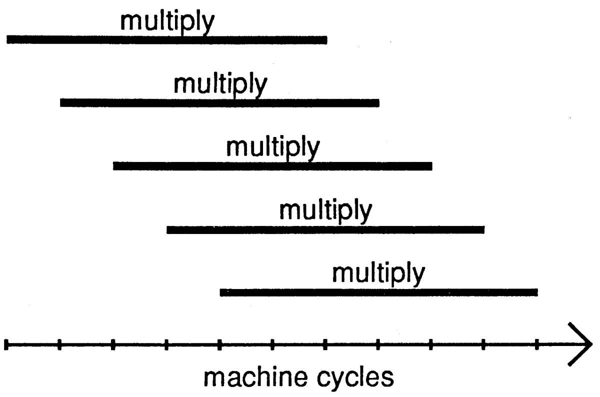

Figure 4 Example of Successive Instructions on a Vector Computer

These machines are expensive (in the 10M$ to 20M$ range), but they can be very cost-effective for programs which perform many identical operations in succession.

For comparison with other architectures I note some important features of vector computers:

- Single instruction stream.
- Parallelism invisible to the programmer.
- HEP code will run correctly on these machines, but making it efficient (known as 'vectorising') usually means re-writing it.
- Program speed-up depends mainly on how much 'unvectorised' execution remains.

3.2 Deep-Pipeline Computers (e.g. FPS 164, 264, 364)

I consider it much more 'natural' to build a computer which can perform many different operations (almost) simultaneously than to concentrate entirely on repetitions of identical operations. The FPS deep-pipeline computers have ten complete execution units enabling up to ten operations to be in progress at the same time.

Figure 5 shows the time relationship between successive instructions on an FPS computer. Obviously it is only possible to initiate an operation if it does not require the results of any operations still in progress.

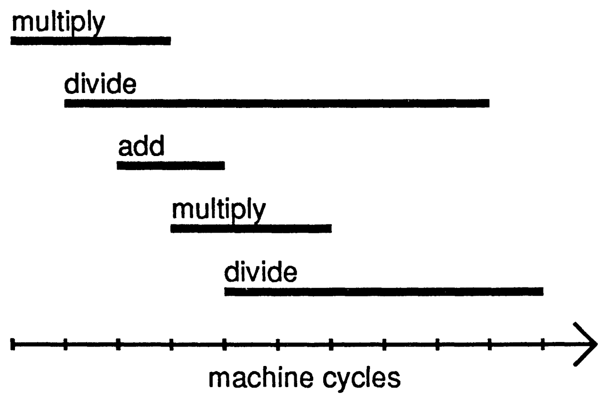

Figure 5 Example of Successive Instructions on a Deep-pipeline Computer

FPS computers are cost-effective for programs which perform a lot of (not necessarily vector) arithmetic. Code with many conditional branches (see Fig. 1) executes inefficiently.

Once again I note some important features:

- Single instruction stream.
- Parallelism invisible to the programmer.
- Typical HEP code can keep two or three of the ten execution units busy.

3.3 Tightly-Coupled Parallel Computers (e.g. ELXSI 6400, Alliant FX/8)

Parallel computers execute several instruction streams simultaneously. Tight coupling normally means memory sharing, and overheads can often be kept low enough so that short sections of code can execute in parallel quite efficiently. The conceptual structure of a tightly-coupled system is shown in Fig. 6.

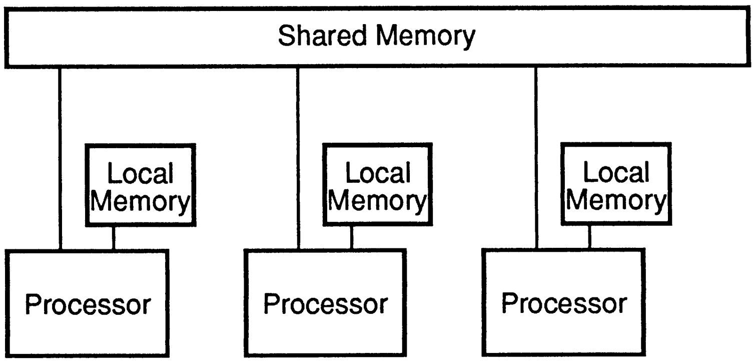

Figure 6 The Conceptual Structure of a Tightly-coupled Parallel Computer

High bandwidth connections to shared memory are costly and work only over limited distances. Hence these machines are not necessarily cheap or infinitely expandable.

You may have noticed that the hardware of a tightly-coupled parallel computer looks just the same as that of a traditional multi-CPU mainframe such as an IBM 3084Q. The distinction is largely a matter of software, and recent developments have made it possible to use the 3080 and 3090 series computers for parallel processing. This can make single tasks execute up to four times faster. It does not, of course, make the computers any cheaper.

The important features of tightly-coupled parallel computers are:

- Single instruction stream parallelised by a special compiler,

or

- Parallel instruction streams controlled by the programmer. (Can get very tricky unless the memory sharing is disciplined.)
- Process-process communications overhead is negligible.
- Parallelising HEP code is best done by hand.
- The processors can have vector or pipeline features, (the CRAY-XMP can be used as a 4-processor tightly coupled parallel system).

In my OPINION, tightly-coupled systems with many processors are the computers of the future. Although the tight coupling costs money and is not essential for most HEP event processing, it gives the machines a wide enough range of potential applications to make commercial success a real probability.

3.4 Loosely-Coupled Parallel Computers (e.g. Clementi Machine, Emulator Farm)

In a loosely-coupled system, inter-processor communication is by I/O only with maximum speeds in the region of a few megabytes per second. To make parallel processing efficient, relatively large sections of code must run in parallel on each processor; such large parallel 'subprograms' may not be recognisable by a parallelising compiler.

Figure 7 shows the conceptual structure of a loosely-coupled system. The driving principle in the design is usually the minimisation of cost. This leads naturally to relatively cheap attached processors with little or no ability to drive peripherals, coupled to a conventional 'host' mainframe system which provides the necessary disks, tapes and terminals, but little CPU power.

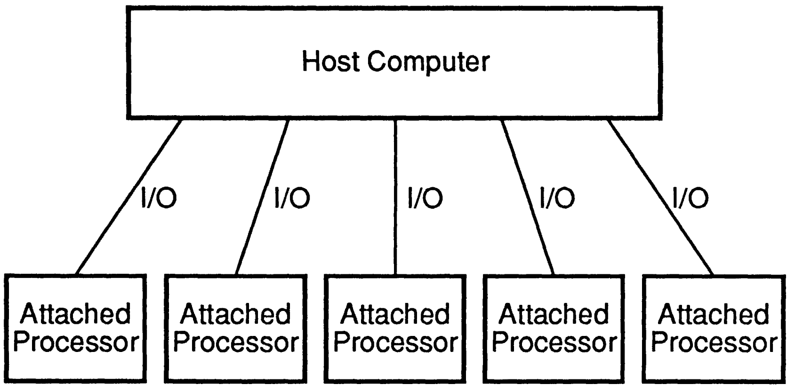

Figure 7 The Conceptual Structure of a Loosely-coupled Parallel Computer

If the application requires little host-AP communications, AP's can be added almost indefinitely. A loosely-coupled system can be constructed by any do-it-yourself enthusiast with a little bit of money and a lot of enthusiasm. It is possible for the host and the AP's to have different instruction sets, data representations, and operating systems (if any). Making the whole thing work is 'only a matter of software'.

The important features of loosely-coupled systems are:

- Parallel instruction streams generated by the programmer. The instruction streams communicate only by I/O.
- Processor-host communications overhead is significant; don't try to parallelise small blocks of code.
- Parallelising HEP event-processing code is easy but the communications overheads limit the range of applicability.

In my OPINION, loosely-coupled systems are a good short-term solution for HEP provided that we use cheap AP's and beware the overheads.

4. BENCHMARKS AND ANECDOTES

Most of the information in this section has been gathered by myself and other members of L3 in the course of our search for a powerful, affordable, computer system.

4.1 Vector Computers (CRAY 1, CRAY X-MP, unmodified code)

The L3 collaboration has not seriously considered the purchase of a CRAY X-MP, so I can only report other people's benchmarks. The results obtained by the OPAL collaboration[2)] are typical. The benchmark program was an 'unvectorised' shower Monte-Carlo. The only modification made for the benefit of the vector machines was to replace calls to matrix multiplying subroutines by in-line FORTRAN code.

Table 1 shows the CRAY performance in comparison with a conventional (IBM compatible) main-frame. The CRAY performance is close to that of a scalar CPU with the same cycle time, showing that the compiler was almost unable to use the vector hardware when faced with unmodified HEP code.

Table 1

Performance of Cray computers: unmodified HEP code

Processor	Time	VAX 11/780/FPA Equivalents
Cray X-MP *(on one CPU)*	33 seconds	25
Cray 1	47 seconds	17
Siemens 7890 *(on 1 CPU; scalar machine for comparison)*	32 seconds	26

4.2 Vector Computers (Cyber 205, re-written code)

If we are prepared, not just to re-write, but also to re-think, tracking and reconstruction algorithms, execution speeds on vector machines can increase dramatically. A group at Florida State has re-thought a track-finding algorithm within the reconstruction code for Fermilab experiment E-711.[3]

The traditional way to find projected tracks in a wire-chamber pack is something like this:

- for each 'hit' in the first plane of the pack,
- and for each hit in the last plane in the pack,
- define a 'road' running from the first plane to the last plane,
- look in all the intermediate planes and select the hits lying within the road.

The totally different approach which was devised to take advantage of the large memory and the vector hardware of the Cyber 205 was:

- pre-generate all 89784 possible projected tracks in the chamber pack. Store all the projected track hit-patterns in memory.
- compare data with all the projected tracks.

A speed comparison between the original code running on a VAX, and the re-written code running on a Cyber 205 is shown in Table 2. For normal scalar HEP code, the expected Cyber/VAX speed-up would be about 20. Accordingly it is estimated that re-thinking the code gave a further speed-up of a factor of 10.

Table 2

Performance of a vector computer: re-written code

Processor	Time	VAX 11/780/FPA Equivalents
CYBER 205 *(re-written code)*	7.7 msec	208
VAX 11/780 *(original code)*	1.6 seconds	1

I will discuss the merits of re-thinking code later. For the moment we should simply note that the large speed-up was achieved on a track-finding kernel, and not on a whole program, and that the identical approach would have been a disaster on a CRAY 1 or X-MP, both of which have very limited memories.

4.3 Deep-Pipeline Attached Processor (FPS 164)

The FPS 164 (and 364, 264) are 64-bit computers designed to be used as attached processors.

They feature:

- 10 pipelined execution units.
- FPS-specific data format and instruction set.
- Good cross-compilation and cross-linking support, (at least on the IBM host system we tested).
- Good software and hardware support for automatic data translation provided you don't use EQUIVALENCE.

In HEP Fortran code, it is normal to use EQUIVALENCE to allow the mixing of Real and Integer data-types in the same FORTRAN array. This gives the FPS software no chance of setting up the correct format conversions when data are to be transferred.

Fortunately, although the true data-type cannot be determined by the Fortran compiler, in well organised HEP programs the data-types are available so that the data can be translated to a machine-independent format when it has to be output. The code used to write and read magnetic tapes can also be used to 'write' to and 'read' from attached processors like the FPS 164.

L3 benchmarked the FPS 164 as a component of the 'Clementi Machine'.[4)] The performance is shown in Table 3. Differences of more than an order of magnitude were found between the extremes of the 'EMC DECODING' package and 'WIRCHA' using the FPS Mathematical Library.

The 'EMC DECODING' unpacks data written by a 16-bit mini-computer. All the arithmetic is integer, there is a lot of bit manipulation and many conditional branches. It would be hard to find any code less suited to the 64-bit pipelined FPS.

The 'WIRCHA' program calculates the electric field in a wire-chamber particle detector. Most of the CPU-time is used to invert a 250 x 250 matrix, and performing this matrix inversion with a specially optimised FPS subroutine gave a very impressive result.

Between these extremes, the 'GHEISHA' hadronic shower simulator, and the 'LUND' electron-positron collision simulator are more typical of the bulk of HEP code.

Table 3

Performance of the FPS 164

Benchmark Program	Type of Code	VAX 11/780/FPA Equivalents
EMC DECODING	Integer arithmetic, bit manipulation, many 'IF's.	2
GHEISHA	Single precision arithmetic, many 'IF's.	3
LUND	Some double precision arithmetic, fewer 'IF's.	6
WIRCHA	Double precision arithmetic, including 250 x 250 matrix inversion.	9
WIRCHA	As above, but using the FPS Mathematical Library.	26

4.4 Scalar Attached Processor (3081/E Emulator)

Emulators are 'home made' CPU's that run IBM object code translated 'on the fly', or in a previous step, into emulator microcode. Although commercial attached processors have become increasingly attractive in the 6 or 8 years since the emulators were first conceived, emulators remain one of the cheapest and easiest ways to add CPU power in a IBM computing environment.

The 3081/E has been developed in a joint CERN-SLAC project. Several processors are now in active service, for example in an emulator 'farm' run by the UA1 collaboration at Harvard, and in a farm run by CERN, L3 and UA1 at CERN.

The features of the 3081/E are:

- Limited pipelining, as in most modern 'scalar' processors. For example, the multiply and divide units can be in use at the same time.
- IBM data format.
- 64-bit hardware.
- Runs translated IBM object code. The translator re-arranges instructions to optimise pipelining.
- Interfaces are available to
  - — CAMAC
  - — VME (plus VME to IBM channel)
  - — Fastbus (under development).

The performance of the 3081/E on the GHEISHA, LUND and WIRCHA benchmarks is given in Table 4. The WIRCHA code can take full advantage of the pipelining and the 64-bit floating-point hardware, and thus shows a large speed-up relative to the solidly 32-bit architecture of the VAX 11/780.

Table 4

Performance of 3081/E emulator

Benchmark Program	VAX 11/780/FPA Equivalents
GHEISHA	4
LUND	4
WIRCHA	10

4.5 Loosely-Coupled Parallel Computer (Clementi Machine)

In the summer of 1984, L3 was invited to be one of the first outside users of the 'Loosely Coupled Array of Processors' assembled by Enrico Clementi at IBM Kingston.[4] Clementi is primarily a theoretical chemist, and he leads an IBM supported group doing calculations of molecular structure from first principles. However good a theoretical chemist you are, such a study needs almost unlimited computing power.

Figure 8 shows a highly simplified diagram of the 'Clementi Machine' as it was when we visited Kingston. The 'host' processor is an IBM 4381 Model 3 (about 8 VAX equivalents) running VM/CMS. Attached to the host are 10 FPS 164 processors.

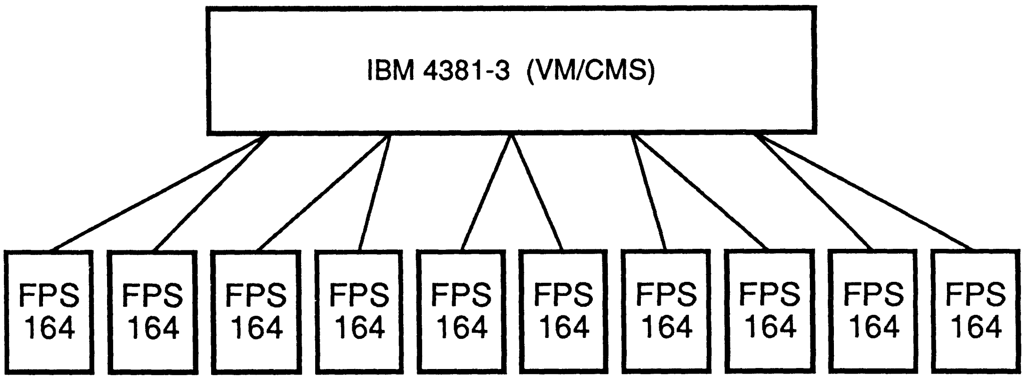

Figure 8 A Simplified Diagram of the 'Clementi Machine'

Physically, two FPS 164's are attached to each block-multiplexor channel; the maximum data transfer rate for an IBM channel is 3 megabytes per second. The FPS software required that each FPS 164 was connected to a separate 'Virtual Machine' running under VM. Thus to achieve parallel processing it was necessary to set up communications between a 'Parent-VM' and ten 'Child-VM's.

Apart from the benchmark of the FPS 164's, which I have already described, our main interest was to study how effective this machine could be for HEP parallel processing. This study required the measurement of the various overheads which would limit efficiency, followed by implementation of a parallel version of one of our benchmark programs.

The overheads are shown in Table 5. First we note that the real speed of communications over the channel is quite close to the theoretical maximum. The relative slowness of communications between processes running within the 4381 might be a surprise to the uninitiated. This slowness is due to the rigid software barriers between virtual machines which were only allowed to communicate by real I/O.

New IBM software now allows virtual machines to communicate using shared memory, and FPS have also reduced many of the overheads in downloading and starting processors.

Table 5

Clementi Machine communications overheads (measured 1984)

VM-VM communications	40 msec. plus 500 msec/Mbyte
VM-FPS communications	20 msec. plus 500 msec/Mbyte
Program downloading	several seconds

Parallel Computing

L3 'production' processing was simulated by running the core of the GHEISHA Monte-Carlo simulation on several AP's, under the control of a single parent running on the host.

Each AP executed the following loop:

- receive 100 kbytes of data from the parent process,
- generate an event (taking about 170 seconds),
- send 100 kbytes of data back to the parent.

There were no synchronisation points other than at the beginning and end of the job. In other words, each AP was given more work to do as soon as it had finished, without regard to how far the other AP's had progressed.

To express the effectiveness of the parallel execution we chose the expression:

$$Fraction\ of\ parallel\ execution = \frac{elapsed\ time\ (one\ AP)}{N \times elapsed\ time\ (N\ AP's)}$$

In this expression 'N' is the maximum number of AP's which could have been used. In this case N is equal to the number of child processes. If the AP's can multi-process, N is the lesser of the number of child processes and the number of AP's.

Figure 9a shows the elapsed time as a function of the number of AP's for a run generating 100 events. The elapsed time falls almost as 1/N. This is confirmed by the fraction of parallel execution shown in Fig. 9b, which stays close to 100% even for N=10. This is not really a surprise, given the measured overheads and the additional fact that only 10% of the host CPU power was needed to manage the 10 AP's.

Had we tried to run the LUND generator in parallel, we would have obtained rather dismal results, since an FPS can generate a LUND event in about 40 milliseconds which is comparable with the communcations set-up overheads.

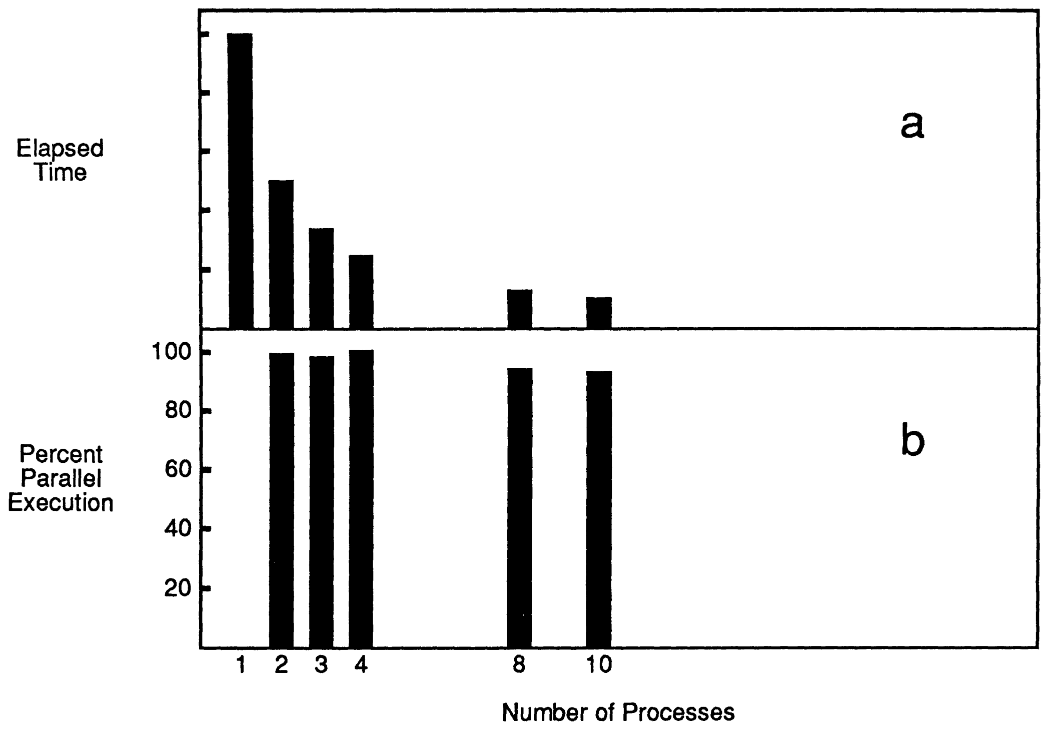

Figure 9 Parallel Execution of GHEISHA on the Clementi Machine

4.6 Tightly-Coupled Parallel Computer (ELXSI 6400)

The architecture of the ELXSI 6400, summarised in Fig. 10, is centred round the 'GIGABUS', capable of transporting between 160 and 213 megabytes per second. Physically the GIGABUS is a backplane, up to two of which may be linked together when more than 5 CPU's are required. Up to 10 CPU's may share the GIGABUS with up to 192 megabytes of memory and with I/O processors. When we benchmarked the machine in early 1985,[5] the CPU's were each equivalent to between 3.5 and 4.5 VAX 11/780 equivalents; the latest version is somewhat faster.

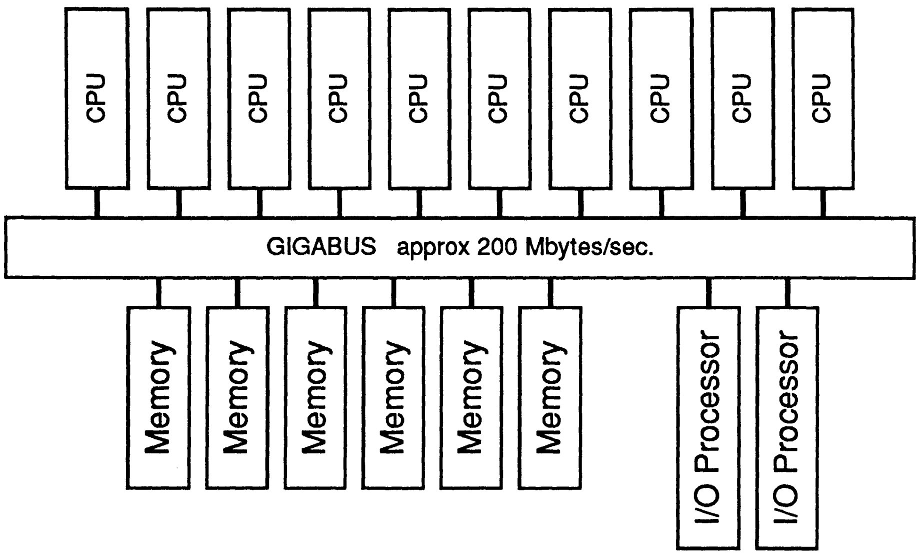

Figure 10 The Architecture of the ELXSI 6400 Computer

The ELXSI 6400 was not conceived for parallel computing. Anyone who displays the processes running on an average super-mini will find several processes queueing for the CPU. The multi-CPU ELXSI was designed to increase throughput by migrating active processes to available CPU's.

Interest in parallel computing was first generated by a customer, Sandia Laboratories. Sandia runs very time consuming calculations on a CRAY X-MP processor. To reduce the real time spent waiting for results, the Sandia programmers use the parallel processing 'intrinsics' provided in CRAY Fortran, to make use of all four processors of the CRAY for one job.

Programming parallel processing on a shared memory system can be tricky, and there is a danger that many CRAY hours might be wasted in debugging the parallel logic. Sandia resolved this problem by persuading ELXSI that they should implement a CRAY-like set of intrinsics on the 6400, which could then be a 'cheap' test bed for CRAY programs.

I will describe the intrinsics in more detail later. At this point it is sufficient to point out that they provide good support for child creation, memory sharing, and inter-process message passing.

From the operational point of view, two features of the ELXSI make it easy to use as the foundation of a 'parallel computing service':

1. The processors are multi-processors, and can execute a large number of tasks apparently simultaneously.
2. The operating system migrates tasks automatically to balance the load on the CPU's.

Overheads

Data-transfer via shared memory takes no time at all. Waking up a sleeping child or parent took about 250 microseconds.

Parallel Computing

In contrast to the Clementi Machine, the minimal overheads of the ELXSI 6400 made it possible to test parallel computing for a wide range of applications.

The plots of elapsed time and fraction of parallel execution versus the number of processes is shown in Fig. 11 for the GHEISHA program. The implementation was exactly the same as that on the Clementi Machine; each GHEISHA event took 130 seconds on a single ELXSI CPU. Clearly, the communications overheads for GHEISHA were negligible, and any inefficiencies must be due to interference with, and inefficient management by, the operating system. Only when the number of processes exceeded the number of physical processors, was any inefficiency noticeable, and even then the parallel execution was only slightly less than perfect.

LUND events took 70 milliseconds on a single CPU, so with 10 child processes, the parent had to be woken to attend to a child every 7 milliseconds. Figure 12 shows how the system performed. The fraction of parallel execution is always over 90%, and apart from the measurement with 9 and 10 children, it is close to 100%. No care was given to give the parent any special status (although it is possible to lock important processes in one of the 16 register sets on a CPU). The efficiency drop is probably caused by the parent sharing a CPU part or all of the time with one of its children.

The range of timing, from GHEISHA to LUND, covers most of HEP event processing. GHEISHA is typical of Monte-Carlo simulation and production processing. The LUND timing is typical of 'Data Summary Tape Analysis' where up to millions of events are read sequentially and subject to simple processing.

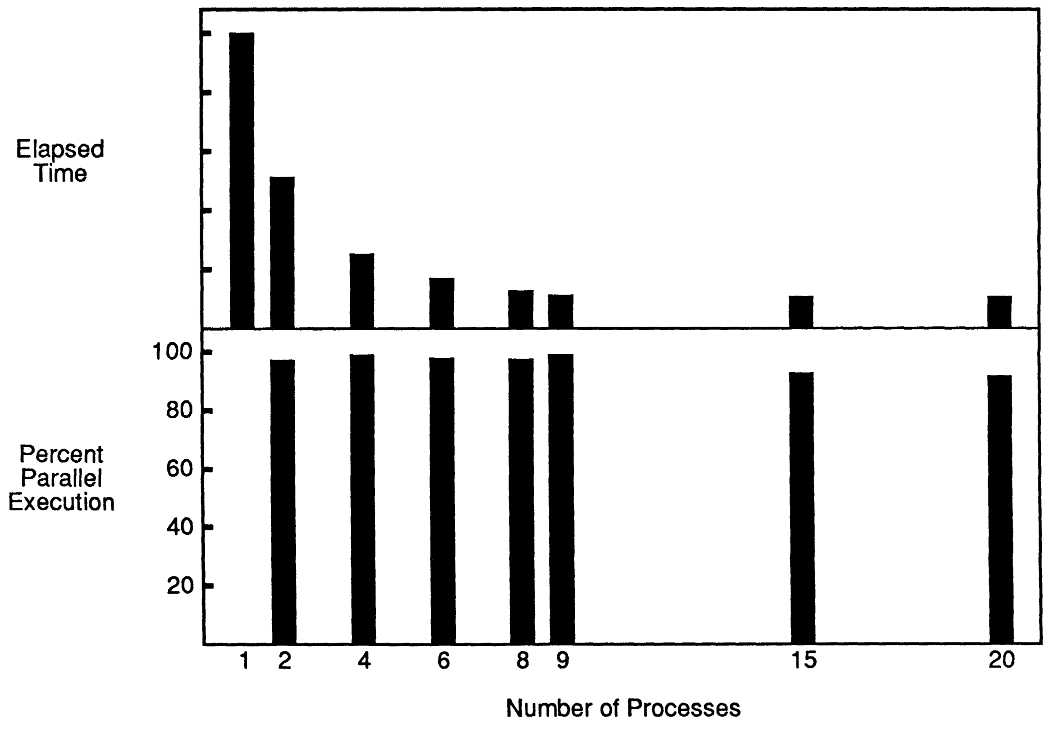

Figure 11 Parallel Execution of GHEISHA on the ELXSI 6400

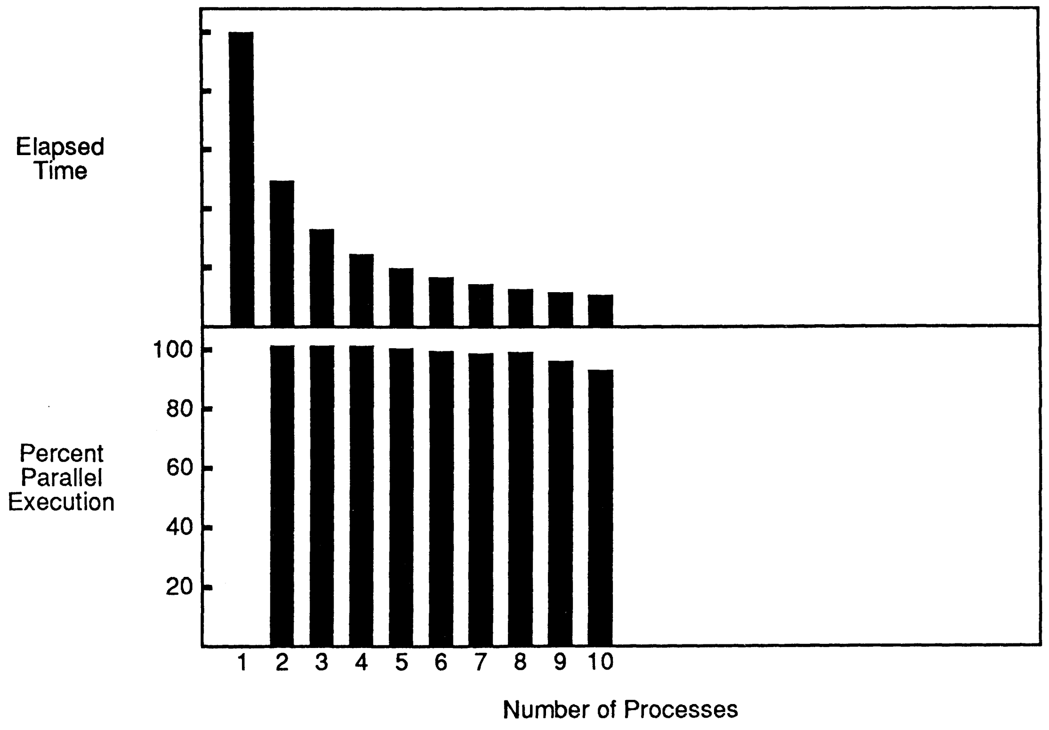

Figure 12 Parallel Execution of LUND on the ELXSI 6400

Since perfect results rapidly become boring, we decided not to limit ourselves only to tests typical of HEP computing, and to parallelise the WIRCHA matrix inversion at the loop level. First the matrix inversion algorithm was changed to Gaussian Elimination, which is fairly easily parallelised. This initial change reduced the speed by 21%. Then a varied number of processes was used to perform the matrix

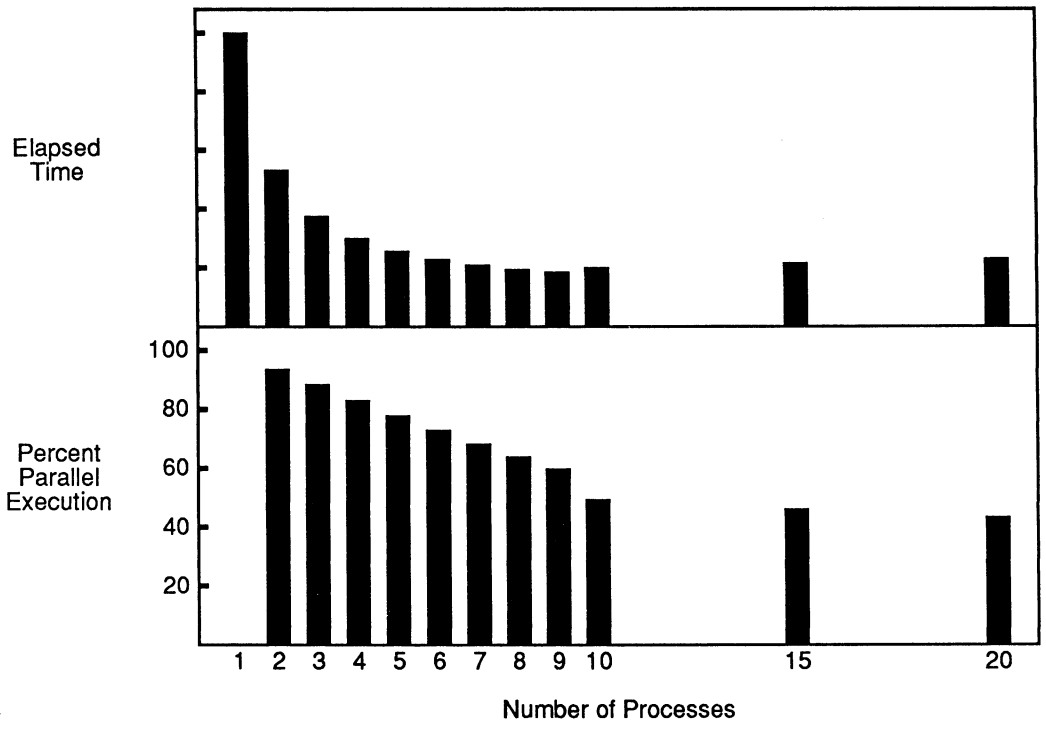

Figure 13 Parallel Execution of WIRCHA on the ELXSI 6400

inversion, leaving the rest of the program shell unchanged and un-parallelised. In contrast to event processing, each process helping in the matrix inversion had to wait for the others when it had finished. The result of all this is shown in Fig. 13. The imperfections are clearly visible, but are mainly due to the shell of the WIRCHA program which we did not bother to parallelise.

4.7 MIMD Computer (Denelcor HEP-1)

For the benefit of all those only marginally less well informed than I am:

MIMD = 'Multiple Instruction, Multiple Data'

HEP = 'Heterogeneous Element Processor'

The L3 benchmarkers visited Denelcor late in 1984.[6] Although Denelcor is now defunct, their unusual machine remains of some interest.

The HEP-1 contains up to 16 processors (PEM's) communicating and sharing data memories over a packet-switched network. A PEM can pipeline instructions from separate processes, and 10 to 12 processes are needed to use a PEM to the full. When fully loaded, a PEM delivers about 13 VAX 11/780 equivalents.

The support for parallel processing is in the form of Fortran callable intrinsics with similar functions to those of ELXSI. Two important differences are:

- memory access synchronisation; a process can wait for another process to fill a memory location.
- Mandatory sharing of all COMMON blocks by parent and children.

The memory access synchronisation feature lends itself well to the automatic generation of parallel code by compilers. Conversely, the mandatory sharing of COMMON blocks made it very difficult to parallelise existing HEP code.

During the L3 tests, the machine made available to us suffered many hardware and software failures. As a result the number of benchmarks we could run was somewhat limited.

Parallel Processing

The limited results obtained with the GHEISHA benchmark are shown in Fig. 14. There was no time to re-structure GHEISHA to avoid the conflicts produced by mandatory COMMON block sharing, so all that was measured was the elapsed time to generate 100 events by running independent jobs on one PEM. It is hard to know what to expect from such a measurement; to me, the relative lack of interference between jobs running on the same PEM is remarkable. The reported CPU usage with one job running was 10%. With seven jobs it had incresed to 61%.

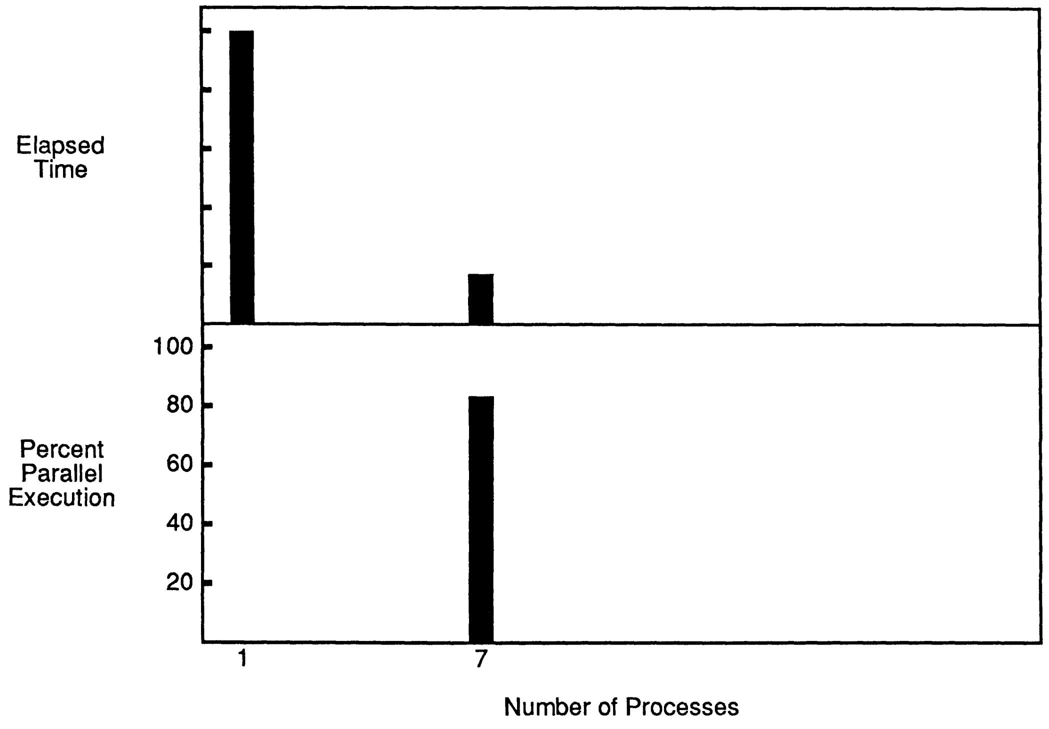

Figure 14 Parallel Execution of GHEISHA on the Denelcor HEP-1

The existence of a Denelcor matrix inversion subroutine made it very interesting to try loop-level parallelism within WIRCHA. The Denelcor matrix inverter was 8% slower than the original code but it allowed the programmer to specify how many parallel processes would be used. Figure 15 shows how the elapsed time varied with the number of processes. The 'percent parallel execution' has been calculated making the assumption that one PEM is equivalent to 10 independent processors. The results show that a PEM does not become inefficient even when executing 40 processes; the deviation from perfect parallel execution is consistent with the fraction of the WIRCHA code which was not parallelised.

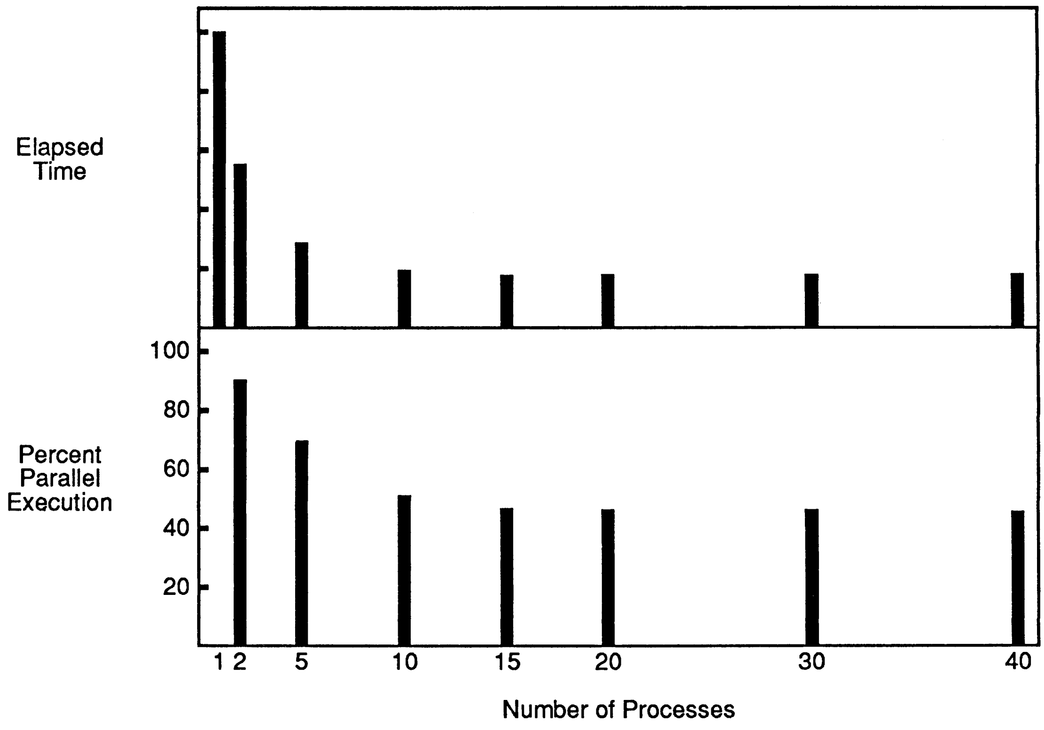

Figure 15 Parallel Execution of WIRCHA on the Denelcor HEP-1

4.8 LEPICS, the L3 Off-line Computer

The wisdom accumulated over the long series of benchmarks and investigations, some of which have been described above, has led us to the specification for the L3 off-line computer. LEPICS means something like 'L3 Parallel Integrated Computing System'. Although we believe that the longer term future lies with tightly-coupled parallel computers, we have to have our system fully operational by the end of 1988. On this time scale it seemed that a system with a large host and attached processors was the best choice. A summary of the specifications is shown in Table 6.

Table 6

Specifications for the L3 Off-line Computer

Installation	Start during 1987
Host	IBM or DEC compatible $\geq$ 12 VAX 11/780 equivalents $\geq$ 10 Gigabytes of user disk space $\geq$ 8 Tape drives Printers, Terminals, Networks
Attached Processors	$\geq$ 30 VAX 11/780 equivalents total 3081/E (for IBM) Micro VAX (of some sort) for DEC
Host-AP Communications	$\geq$ 2 Megabytes/second

5. 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. Some of these systems are no longer available, 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 7, which summarises my price/performance comparisons. I have already commented on the variability of prices. The performance figures are mainly those measured by L3, and are, of course, totally reliable. Nevertheless, 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.

In comparing computer systems it is vital to take into account how complete the system is. To emphasise the importance of this I have included an entry for 'a bag of chips', specifically a Motorola 68020. This excellent processor is nothing like a complete computer system, and its price is about as relevant as the price of copper or silicon.

A 'complete' system includes disks, tape drives, printers, and terminal lines. Systems which are close to complete have been given four 'blobs'. The CRAY is normally sold with a limited peripheral configuration, expecting that most of the more trivial (but by no means cheap) I/O operations will be handled by, for example, an IBM mainframe as a front-end. This fact makes the CRAY lucky to get three blobs.

The FPS 164, which is designed to be an attached processor, but can access its own disks, gets two blobs, whereas a bare 3081/E emulator merits only one. Adding a minimal sized host to the emulator raises it to two blobs. In every case I have only estimated the hardware completeness of a system. Whether it has an operating system or not is a separate consideration.

The message which I draw from Table 7 is that there are no miracle solutions. Improvements of up to a factor of 3 in price/performance may be available by departing from the classical, easy to use, scalar mainframe solution. The task facing HEP is to choose an architecture which works well for our sort of computing. The task facing computer manufacturers is to choose architectures which work well for a wide range of applications.

Table 7

Approximate Price/Performance Comparisons

Computer	Price $M approx	HEP VAX equiv	$k per VAX (batch)	How complete a system?
CRAY X-MP *4 CPU*	14-17	100 $\longrightarrow$?	155	● ● ●
IBM 3090-200	6	40	150	● ● ●●
FPS 164	0.25	3-6	56	●●
Clementi Machine *1984 version*	6	40-70	109	● ● ●●
3081/E	0.033	5	7	●
3081/E Farm *(small host plus 6 emulators)*	0.7	32	22	●●
3081/E Integrated System *(large host, 6 emulators (LEPICS?))*	2	41	49	● ● ●●
ELXSI 6400 *(10 CPU, 1985 model)*	2	40	50	● ● ●●
Denelcor HEP-1	1.3	7-13	130	● ● ●●
Motorola 68020	0.0001	1	0.1	

6. 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.

6.1 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.

   The likely result is 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 (and I am not alone) is that a factor of 3 is the likely maximum.

6.2 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 existing automatic parallelisation tools.

Rather than make myself look ridiculous by trying to create a theory of parallel computing, I will restrict myself to examples of what L3 is doing, and intends to do, to make parallel computing work.

A High Energy Physics Batch Job

Figure 16 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.

Since the events are independent, they may easily be processed in parallel. Manpower, 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.

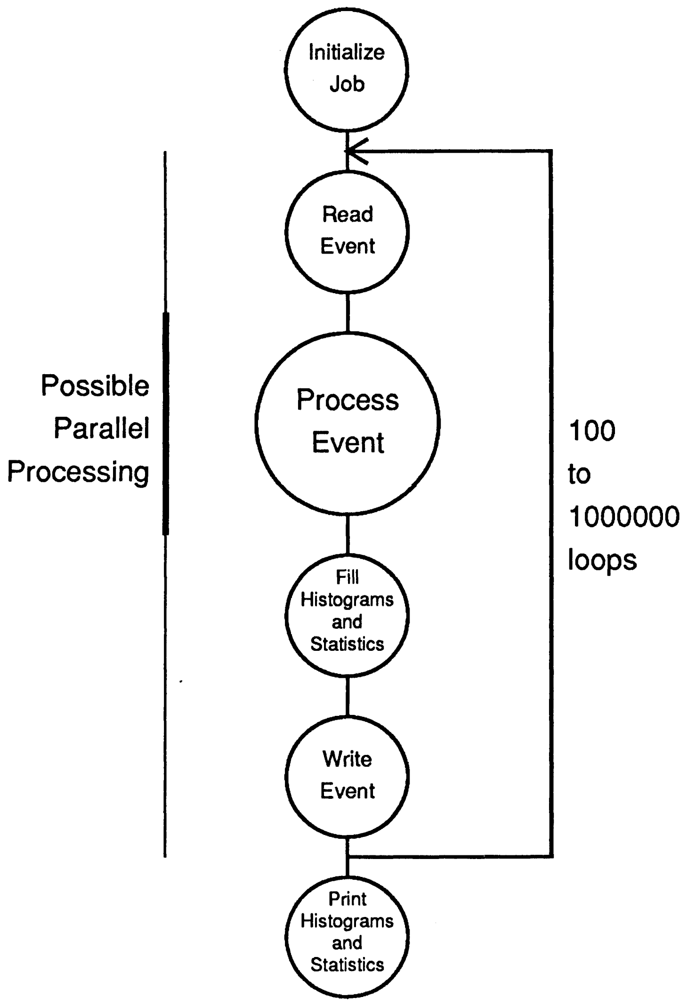
Initialize Job
Read Event
Process Event
Fill Histograms and Statistics
Write Event
Print Histograms and Statistics
Possible Parallel Processing
100 to 1000000 loops

Figure 16 The Structure of HEP Batch Jobs

Parallel Processing Examples — Emulator Farm

L3 is now one of the users of an emulator farm comprising an IBM 4361 host, and five 3081/E emulators. The L3 Monte-Carlo event simulator has been parallelised in a rather simple way so that events can be generated on an arbitrary number of emulators, but program initialisation and termination, and all I/O, are done by a single process on the host.

The host currently accesses the emulators through CAMAC. This works perfectly for Monte-Carlo simulation. For event processing with a higher I/O to CPU ratio, and with more work to be done by the host, a more powerful host and faster host-emulator communications will be needed.

From this experience, we know the software tools needed to make such a system work. We also have a fairly good idea about the additional software facilities needed to make a host-AP system into an integrated HEP computing environment.

Here is a commented list of the requirements:

1. Cross compiler. (exists)

   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).

2. Cross linker. (exists)
3. Debugging Tools. (some exist)

   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), 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.
4. FORTRAN calls to: (exist)
   - download a program module
   - download data
   - start emulator
   - wait for emulator to finish
   - upload data
   - interrupt emulator
5. Resource Management (does not yet exist)
   - assign emulators to requesting jobs on the basis of the job's relative priority
   - request a job to release (some) emulators
   - forcibly remove emulators from uncooperative jobs

It makes an enormous amount of sense to put emulators on an IBM compatible host and avoid any data-format incompatibilities. With some types of attached processors, for example a 68020 on a card, format incompatibilities are nearly inevitable. This should not be a problem for well organised software. For example, all L3 software uses the ZEBRA data structure manager. At any stage in the processing, ZEBRA data can be transported to another processor in a machine-independent format.

Parallel Processing Examples — Tightly-Coupled Systems

Tightly-coupled systems can do everything an emulator farm can, but the lower overheads make it attractive to try to parallelise a wide range of tasks. When the 'process event' kernel shrinks to a few tens of milliseconds, the programmer has to be careful even on a shared memory machine.

Discipline in the use of shared memory must usually be imposed by hand; one process must tell another that the shared memory is available. To squeeze out the last 10% of performance, the programmer may have to make his own decision about which data may be held in a processor's cache memory, and which data must always be read from and written to the main shared memory.

'Fetch and op' instructions (e.g. fetch and add) can be used to implement inter-process synchronisation, and for common histogram and statistics accumulation without explicit discipline.

An emulator farm operating system has to take great care to ensure that the number of processes exactly matches the number of processors. Most tightly-coupled processors can multi-process, so the operating system can exert a much looser control over the total number of processes.

To make some of these points a little clearer, I list below the Fortran 'intrinsics' available to support parallel processing on the ELXSI system. Figure 17 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.

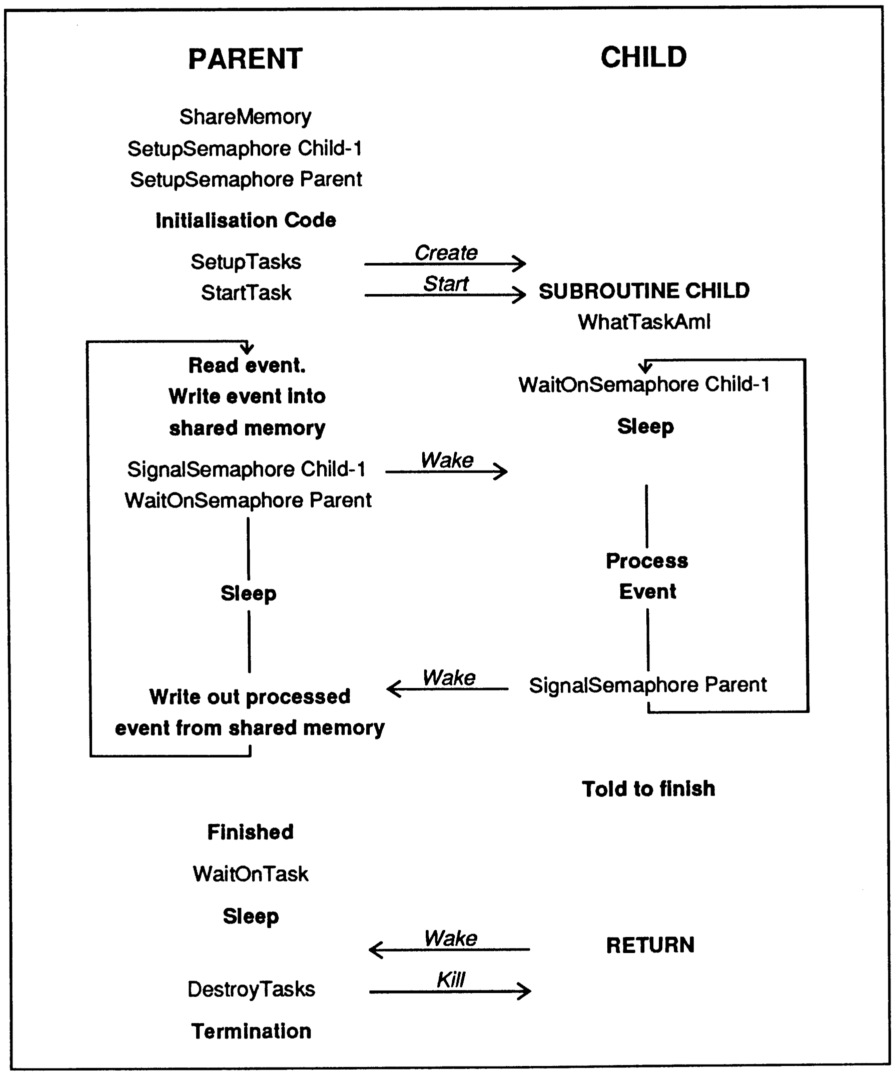

Figure 17 A Parent-child System Using ELXSI Intrinsics

<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.

Parallel Processing Invisible to (Most HEP) Users

After reading the previous section, and struggling to understand what the meaning of locks and semaphores might possible be, an HEP user might be forgiven for abandoning parallel processing in total disgust. In reality, most users (even those who write analysis programs) need hardly notice parallel processing at all.

I will describe briefly the simple program organisation which makes this possible. More ambitious and more flexible schemes are under discussion[7] but I have to keep these lectures to a finite length.

The keys to invisible parallel processing are clearly visible in the example of an ELXSI implementation shown in Fig. 17. Although the parallel processing logic can be moderately complex, it does not invade the 'Initialisation', 'Read-Event', 'Process-Event', 'Write-Event' and 'Termination' phases. Most of the programming effort goes into writing code which fits cleanly within one of these phases, and the writer of this code can forget about the parallel processsing environment.

The overall parallel processing framework can be generalised and made portable by inventing a set of generic parallel processing intrinsics which collapse to nothing on a scalar machine, and call the appropriate machine dependent services elsewhere.

All this sounds too good to be true, and to some extent it is. Adapting many existing programs to such a scheme would be a nightmare combination of major surgery and nasty little fixes. L3 is in the fortunate position of having few existing programs, and we can ensure that further software development obeys principles which will make it suitable for parallel processing. The principles are quite simple:

- it must be easy to identify the data-flow into and out of a processing 'module'. Ideally the input and output should each be a single data-structure.

- any writes into variables which are not private to the event being processed must be clearly identified and localised. Filling of common histograms and statistics is the most obvious example of this.

7. CONCLUSIONS AND SUMMARY

I have shown that, in spite of the continuing fall in the cost of MIPS, HEP computing needs are growing so rapidly that they need either more and more money, or a more imaginative approach. Furthermore, not even money can save large CPU users like HEP from parallel processing in the near future.

In some sense, all the more imaginative approaches involve computing in parallel. In vector and deep-pipeline computers the parallelism is on a microscopic scale, but these machines do not appear to be well matched to HEP needs. Using the larger scale parallelism of tightly- and loosely-coupled parallel systems normally requires intervention by the programmer, but these architectures match well the intrinsic parallelism of HEP computing.

For immediate relief for under-financed HEP experiments, loosely coupled host-AP systems appear a natural choice. The longer term future of commercially successful parallel computing probably lies with much more tightly coupled, and thus much more versatile, machines.

Parallel computing could be viewed as the solution of the hardware problems by the creation of software problems. In the general case, this is not so inaccurate, but in HEP computing it is quite possible for most programmers to ignore the parallel processing framework.

REFERENCES

1. J. Ballam et al., Computing for Particle Physics; Report of the HEPAP Subpanel on Computer Needs for the Next Decade, DOE/ER-0234, 1985.
2. R. Brun, A. Rossi et al., CERN report in preparation.
3. C. H. Georgiopoulos et al., A non-numerical method for track finding in experimental high energy physics using vector computers, FSU-SCRI-85-11, 1985.
4. F. Carminati et al., Technical report on the evaluation of the parallel computing system in Kingston N.Y., L3 report No. 313, 1984.
5. R. Ball et al., L3 evaluation of the ELXSI 6400 multiprocessor system, ETH/L3 Technical Report, 1985.
6. G. Massaro et al., Report on the L3 benchmark of the Denelcor HEP-1 computer, NIKHEF/L3 Report, 1984.
7. M. Pohl, Data driven parallelism in experimental high energy physics applications, ETH/L3 report in preparation.