SLAC – PUB – 4589
March 1988
(E/I)

THOUGHTS ON SOFTWARE AND COMPUTING*

PAUL F. KUNZ
Stanford Linear Accelerator Center
Stanford University, California 94309
⟨PFKEB@SLACVM.BITNET⟩

ABSTRACT

This talk has three distinct parts. The first two parts are on vector and parallel processing and their success, or lack thereof for HEP. The third part is an analysis on the software situation in HEP. These topics have been chosen because of the frequency with which they are discussed in the hallways of our laboratories and institutions. This review looks at the field from a particular point of view: that of an experimental physicist working with a large detector at a collider and, in addition, only considers the offline processing aspects of the field.

I. VECTOR PROCESSING

I.1 WHAT IS VECTOR PROCESSING?

The term *vector processor* and the allied term *array processor* are somewhat misnomers for a style of computer architecture that is based on a simple fact: there is no way with a given technology that floating point arithmetic is going to be as fast as a binary add. Also, there is no practical way that random memory access time is going to be as fast as a binary add. Thus, in any computer, a single floating point operation is going to take multiple CPU cycles. For example, a floating point add may be divided into a number of cycles as shown in Fig. 1. The operations performed in each cycle are as follows:

1. Fetch operands from memory and/or register files.
2. Prenormalize the mantissa with the smallest exponent.
3. Add the mantissi.
4. Postnormalize the resulting mantissa and correct the exponent if necessary.
5. Store the results in memory or register file.

Invited talk Presented at the 8th International Conference on High Energy Physics at Vanderbilt, Nashville, TN, October 8-10, 1987

*Work supported by the Department of Energy, contract DE–AC03–76SF00515.

With an appropriate computer architecture, this operation over a number of operand pairs can be made faster by overlapping the steps or pipelining them. Thus, as shown in Fig. 2, one can do three floating point adds in only seven cycles instead of the fifteen it would take if done sequentially.

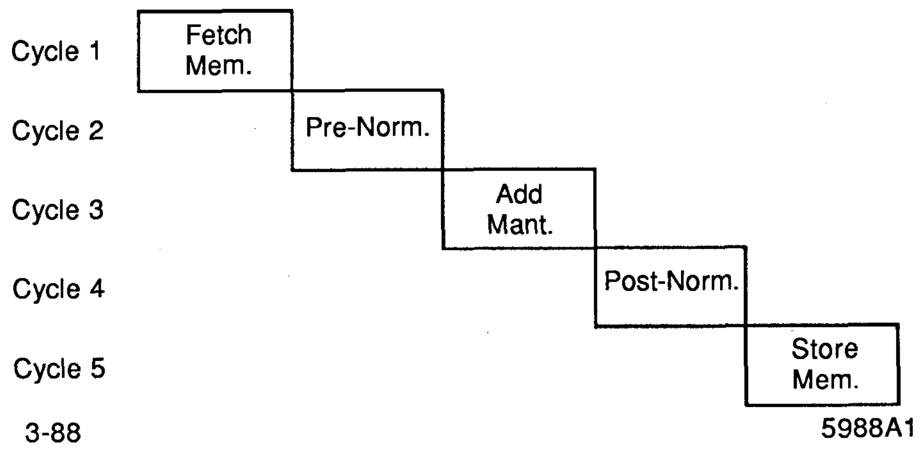


Fig. 1. Example of multiple cycles of floating point add.

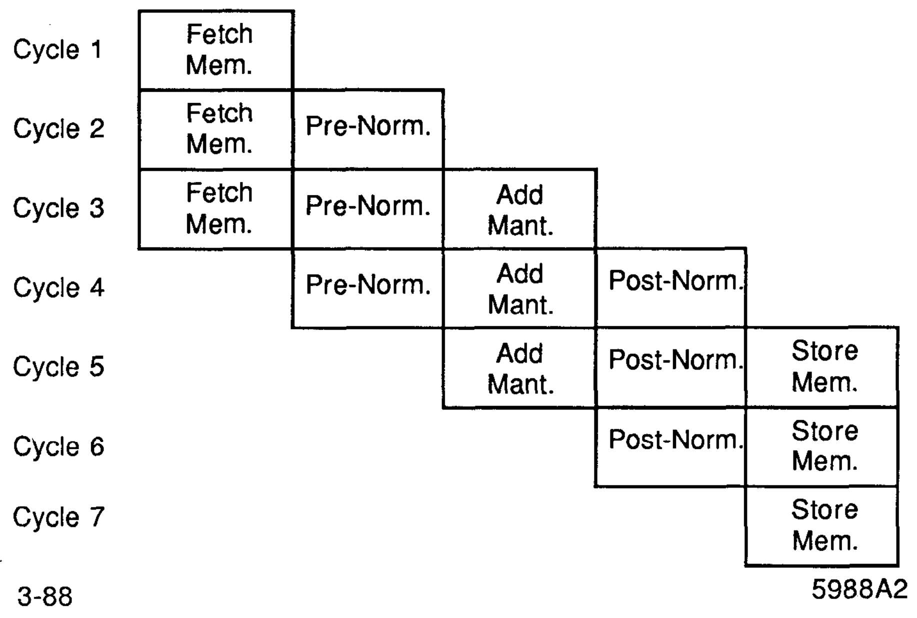


Fig. 2. Example of pipelined cycles of floating point add.

This pipelining only works well if the data operands are in an orderly pattern, which the FORTRAN programmer knows as a vector or an array; thus, the term vector or array processor is used for a processor that can perform in this manner.

I.2 HEP Use of Vector Processing Computers

All modern supercomputers have vector processing capabilities from which a lot of their processing speed depends. However, for the HEP experimentalist, early attempts to use these new machines have been disappointing. An example of what happens was given by Kenichi Miura.[1]

Miura worked with the FOWL Monte Carlo tracking code he obtained from CERN. He first compiled the code, with all vectorization in the compiler turned off, and ran it on the FACOM VP-200. With vectorization turned off, the compiler does not generate any vector machine instructions, thus measuring the speed of the scalar processor. A run lasted 105.9 seconds instead of 753.6 seconds on an IBM 370/168; a speedup of about a factor of seven. Then, without changing the code, he turned on the vectorization in the compiler and found that the code ran slower (111.9 seconds)!

Not surprisingly, there was no speedup since this type of code does not deal much with vectors. It seems a bit strange that the code was slower with vectorization turned on until one thinks about it. The only vectors in the code are the three-vectors of the particle momentum; but the vector pipelines of the machine, although fast, are relatively long, so there is a startup time penalty. Thus, operation of three-vectors using the vector pipeline instructions takes longer then doing three sequential scalar operations, and the full potential of the vector machine is not realized. This result is typical of all supercomputers, not just the FACOM VP-200.

I.3 Need for New Methods to Use Vector Machines

To realize the full potential of vector machines, so that the compiler will efficiently handle vector lengths in the hundreds, one needs to use new methods of programming. There are two approaches to the problem. The first is the micro approach in which one re-codes the problem so that the inner do-loop will have long vectors. The other is the macro approach in which one brings an outer do-loop (say, over particles) into an inner do-loop.

Micro Approach to Re-coding

An example of re-coding HEP code with the micro approach comes from the track reconstruction code of the Mark III detector.[2] The innermost time-consuming do-loops are in the basic pattern matching for finding tracks. The technique, described below, was also applied to track finding for a Fermilab fixed target experiment by the Florida State group.[3]

The first step is to generate a track dictionary by a "geometry" program which draws circles from the beam line through the detector in the $r-\phi$ plane and notes which sets of drift chamber cells lie on each. This method is illustrated in Fig. 3. Each dictionary entry is one distinct set of these cells. To keep the dictionary small, only circles which correspond to transverse momenta of greater than 50 MeV are drawn. Because the data from the detector is unpacked cell-by-cell, it is natural to structure the dictionary not only as a list of cells on each track, but also inversely as a list of tracks that pass through each cell.

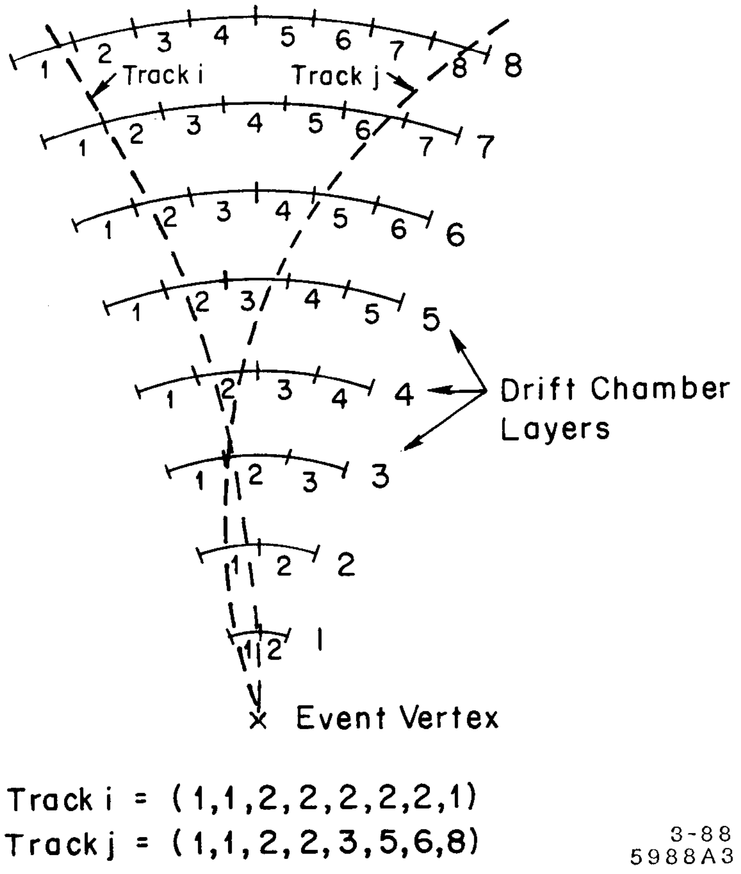
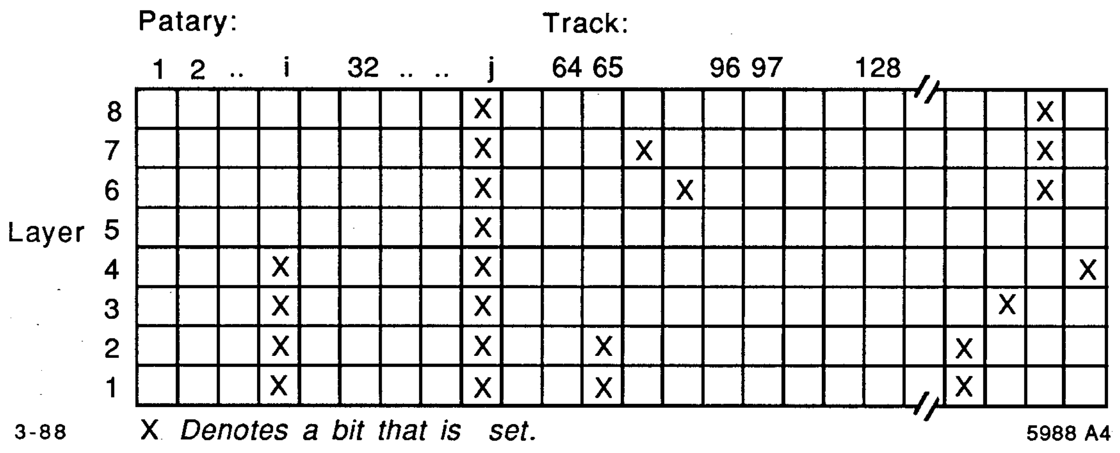


Fig. 3. Schematic representation of dictionary generation.

Having set up these tables once, the pattern recognition is ready to begin on events. During this phase, as each cell is unpacked and identified, the program sets bits in a two-dimensional bit array called PATARY with one row for each layer in the drift chamber and one column for each track in the dictionary. For each hit cell, one bit is set for each track that might have caused the hit. These bits then indicate which of the drift chamber layers on any given track are actually hit, as shown in Fig. 4.



Patary:
Track:
1 2 .. i 32 .. .. j 64 65 96 97 128
Layer
8
7
6
5
4
3
2
1
X
3-88
X Denotes a bit that is set.
5988 A4

Fig. 4. PATARY table generation.

Note that, at this lowest level of reconstruction, the code is already amenable to exploiting the vector instructions of some supercomputers. This is so because one can take as one long vector the list of all hit cells, and operate on that vector to fill the PATARY array.

These ideas may seem trivial, but they are critical to exploiting vectorization. Rather than doing pattern recognition serially (track-to-track), information is developed and stored from primitive operations on all cells (as described above), then all clusters of cells in layers (named objects), then all clusters of objects over layers (named bundles of track candidates), and finally the isolated tracks themselves. At each step, long vectors can be made up of objects, bundles, or tracks.

Of course, the real-life situation isn't that simple. For example, the actual code is more complex in order to allow for cell inefficiencies which leads to tracks without the full complement of hit cells. Also, unlike the Florida State Group, the Mark III group never had a chance to actually run this vectorizable code on a computer with vector capabilities. Nevertheless, they predicted that such methods could save up to a factor of five in the track finding and fitting time. As others have found, the code ran faster on scalar processors as well, with a measured three-fold increase in speed over a conventional approach.

In summary, with the micro approach, one can find tremendous speed gains for the inner do-loops by restructuring the code to deal with a large number of items at a time. However, to find a speedup in the overall code, one must do such restructuring for all such do-loops in the code.

Macro Approach to Re-coding

The basic strategy of the macro approach is to process many events or tracks in one pass; that is, to bring the event or track loop to the innermost DO-loop instead of the outermost one. An excellent example of this approach is work done by Kenichi Miura of Fujitsu Limited to vectorize the EGS4 shower program.[4]

The standard EGS4 program works from a stack initially loaded with the one incident particle as illustrated in Fig. 5. One particle from the stack is processed at a time. The shower subroutine decides which of the many physical processes will be in effect, calls that subroutine, calculates it and stores the results, rearranging the new particles on the stack so that the particle with the lowest energy is at the top of the stack. This is equivalent to tracing the shower tree in Fig. 6 toward the shortest path until all particles are absorbed. In this form, the program has almost no vectorization potential.

To achieve vectorization, Miura changed the whole program flow so that instead of a stack, there are queues of particles waiting to be processed by some physical process as shown in Fig. 7. With each step, the queue with the largest number of particles is chosen and these particles are taken as a vector. The resulting particles are put back into their appropriate queues. To make the vector length even longer, the initial particle stack is changed to a queue of particles from multiple events.

For a simple case of 1 GeV electrons incident on an infinite lead brick, the vectorized version of EGS4 achieved a speed up of a factor of 8 over the scalar version. The size of the code increased about 30% due to the extra bookkeeping involved and, since most of the scalar variables in the physics routines had to be changed to array variables, the size of the memory space required went from 1/2 MegaByte to 6 MegaBytes.

In summary, with the macro approach, one can achieve a significant speedup of the code at the cost of more complex control structure and significantly more memory usage. Memory usage should not be a problem as vector processors come with very large memories. Time will tell if the control structure will be tolerated by our HEP users.

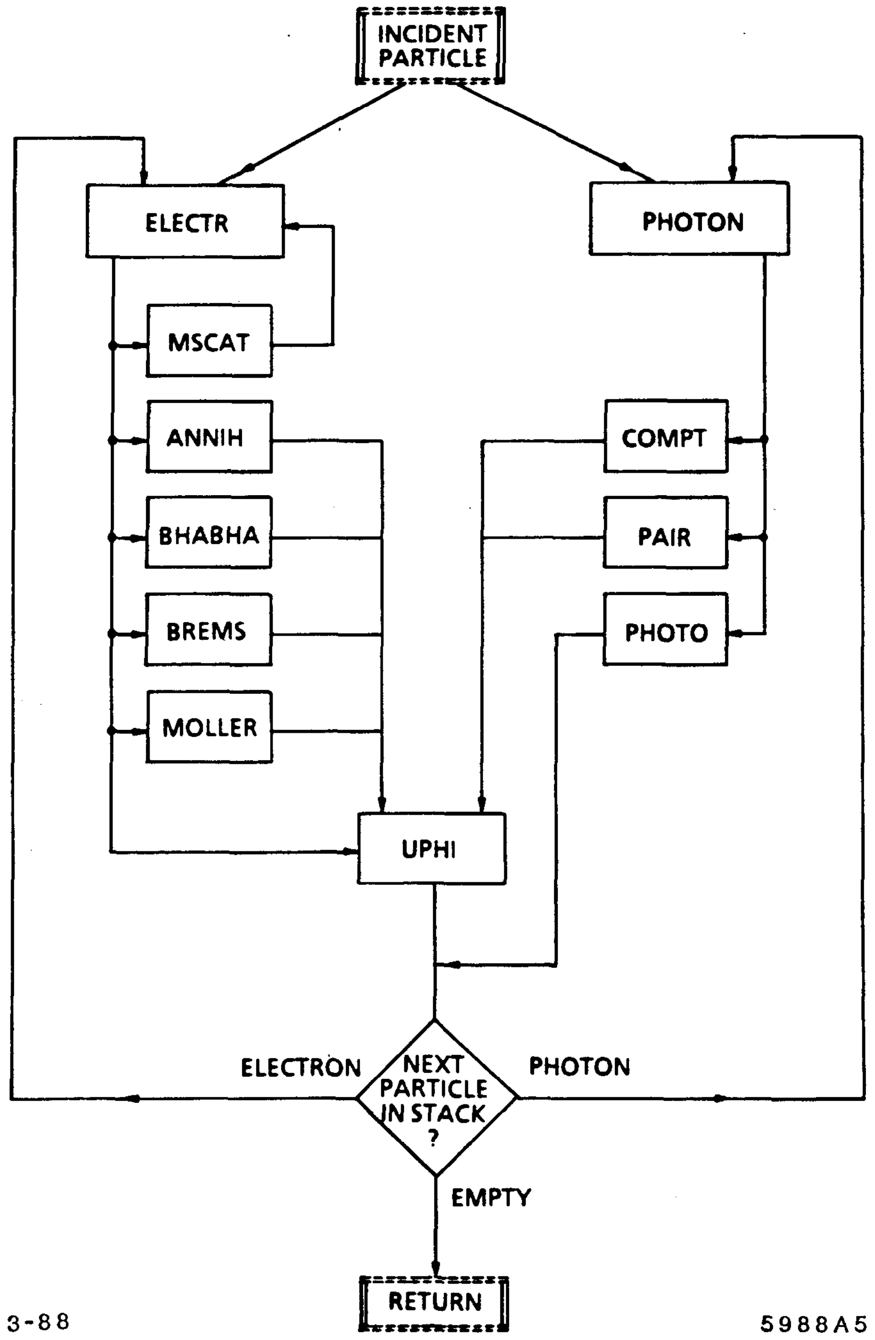


Fig. 5. Control flow in standard EGS4 program.

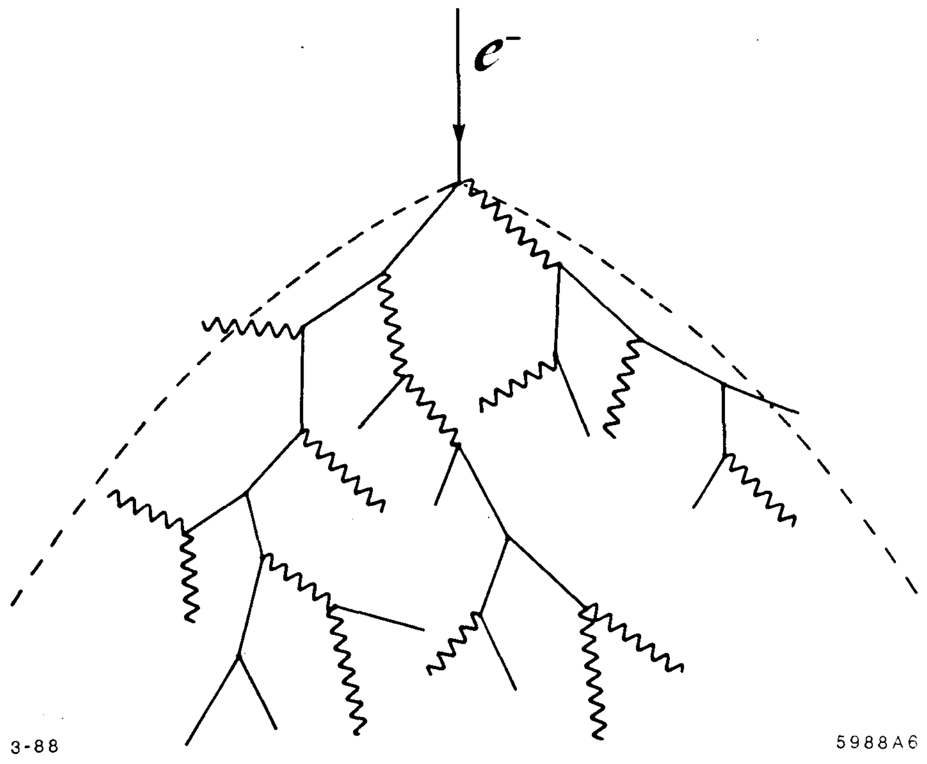


Fig. 6. An electromagnetic cascade shower.

I.4 Summary on Vector Processing

It is clear that experimentalists will need to take new approaches in structuring their code to make good use of vector processors. Even with this restructuring, there remain some questions, however. For example, is the speedup one obtains greater than the increased cost of the vector machine? A speedup of a factor of three, say, on a machine that costs three times as much is no net gain. Then there are other operational considerations; for example, one usually needs to learn another operating system to use the vector machine and may also need to export the raw data tapes to another site where a vector machine is located. Nevertheless, we are in the early stages of experimenting with vector processors and no final conclusions on their usefulness to the HEP community can be made yet.

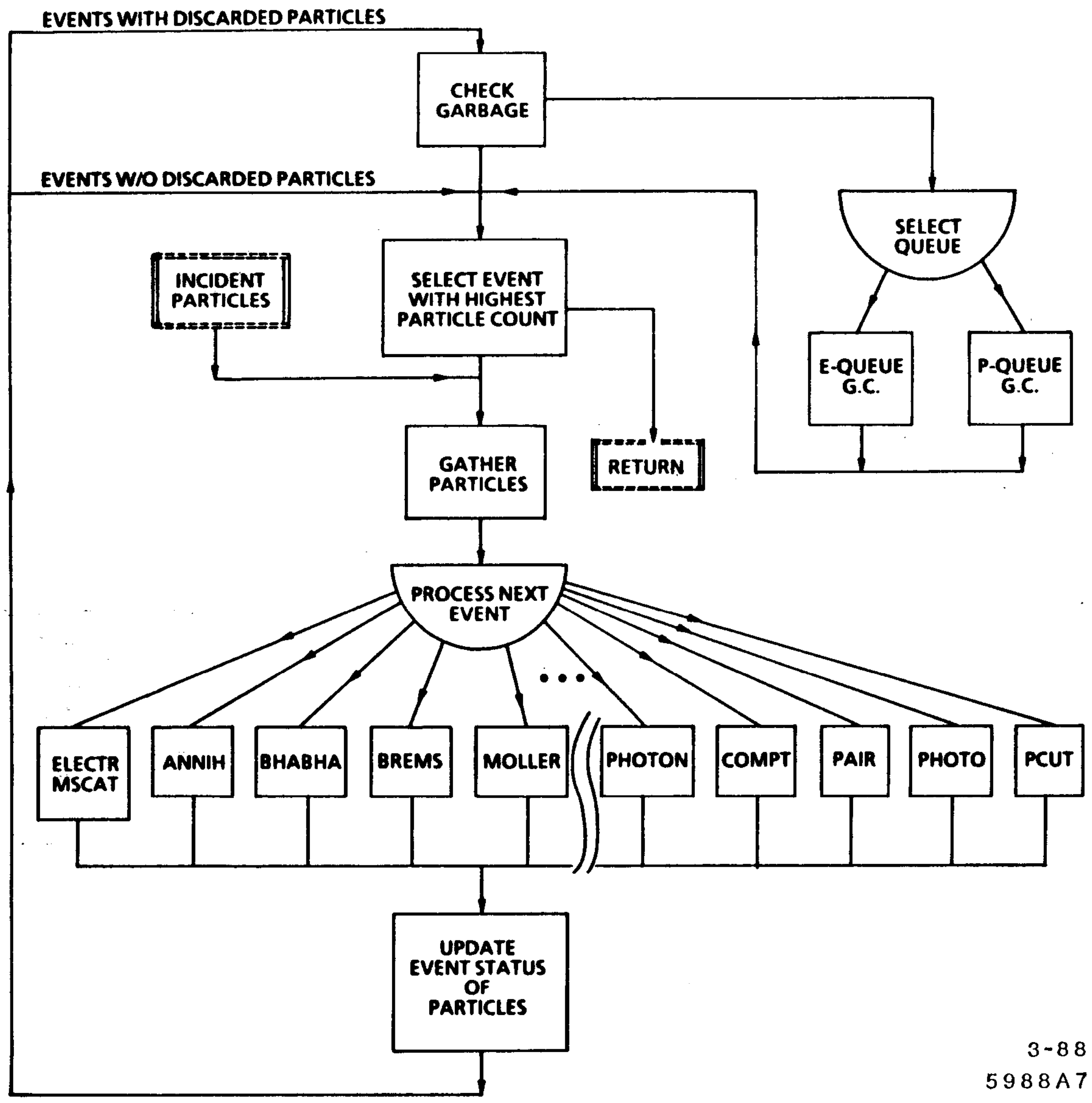


Fig. 7. Control flow in vectorized version of EGS4.

II. Parallel Processing

The other method of getting more performance is to try to exploit the inherent parallelism of workload and have these parts run in parallel on separate processors. These processors can be either tightly coupled or loosely coupled; in many cases it doesn't matter. They don't even have to be complete computers as long as they are cost-effective processors.

For experimental HEP offline processing, the main workload is event processing, either raw data or Monte Carlo generation. Three methods of using parallel processing have been and continue to be thought about.

Within an event, there are many parts of the program which are independent of the results from all other parts. This leads to the idea of running these parts on parallel processors, thus reducing to the total time to process an event. There have even been ideas to make specialized processors with some of the algorithms in hardware, to speed up the parallel parts even further. One problem with this method is there remains large parts of the code that can not be run in parallel. Not only does this limit the overall speed but, because the parallel parts will probably not take an equal amount of time, there is a loss of efficiency due to idle processors. There is also a loss of efficiency as data is moved around among the processors.

Another method of parallel processing is to move events through a pipeline of processors, each processor doing one part of the overall job. Again, specialized processors with algorithms in hardware are frequently mentioned. One fundamental flaw in this scheme is that not all events take the same amount of processing time, so there will always be a "longest event" that will clog the pipeline. In addition, the time spent passing data down the pipeline can be quite serious, because the temporary data set generated and used during the processing of an internal program is generally much larger than either the initial raw data or the final DST output data.

To date, the only successful way to introduce parallelism for event processing is feed an event to one processor and let that processor work on that event alone, while the next and subsequent events are fed to additional processors. This method keeps each processor fully occupied except for the minimal communications time inputting the raw data and outputting the results. This method of parallelism is popularly called the *microprocessor farm.*

The method of having one event processed by one processor *works* as was clearly demonstrated as far back as 1979 by users of the 168/E.[5] It has also been shown that this method is not sensitive to the method of coupling. It is working equally well with tightly coupled processors such as the Elxsi computers or loosely coupled processors such as the FPS-164.[6] Trying to exploit parallelism within one event, however, has so far been less effective because the overall execution time can easily be slowed down by the nonparallel part of the program, even with tightly coupled processors.

II.1 PARALLEL PROCESSORS IN THE SSC ERA

The question has been frequently raised of whether the parallel processing technique will continue to be valid in the SSC era where a single detector will require 1000–2000 VAX 11/780 equivalent processing power. The answer seems to be affirmative and can be understood from the following simple arguments:

- A data acquisition computer with a certain I/O bandwidth recorded the data at the detector.
- Whatever the power of the parallel processors (as long as they can run the complete program), one will add enough of them to obtain the required total CPU power.
- As long as rate of event processing is not greatly different from the original data acquisition rate, then the host computer with I/O capacity at least equal to the data acquisition computer will be sufficient to run a processor *farm.*

III. SOFTWARE ISSUES

Today, in high energy physics, software is generally a "mess." That is to say, most experimental groups, especially the new large detector groups, are having a difficult time developing and managing their software. As each new large detector comes online, the software effort becomes increasingly more difficult. This leads to the conclusion that we will have a major problem with the software for the Superconducting Super Collider (SSC). Although not explicitly stated, there also seem to be many in our community that believe the reason software will be a problem at the SSC is that "we need to develop large (200–500K lines of FORTRAN) complex code for the detector, with 400 physicists at 50 institutions."[7] We first explore whether the above reasoning is fact or fiction.

First of all, let's look at the size and complexity of the code for a very large detector. The size and complexity of the code should scale with some aspects of the detector. If we can find the scaling laws, we should be able to estimate the size of the problem for an SSC detector by extrapolation from our current detectors.

The size and complexity of the code should scale, for example, with number of different kinds of detector elements in the detector. This is because each detector type will need its own pattern recognition code and there will be some code that links tracks between the detector elements. For an SSC detector, however, there is no reason, necessarily, that there should be more different kinds of detector elements than a large Tevatron or SLC detector. Therefore, this scaling law would say that an SSC detector would not be a more difficult problem.

Another scaling law is the size and complexity of the code scales with the number of boundaries in the detector. This is because each irregular boundary takes additional code to calculate the position of the boundaries, cross the boundary and, in general, makes for a lot of exceptional case handling in the code. There is no reason that an SSC detector should have more boundaries than existing large detectors, so the software problem for an SSC detector may not be more complex because of this scaling law.

The size and complexity of the code should scale with the track density, due to the many alternative possibilities a pattern recognition program must try to resolve. However, these problems are in a limited area of the detector code and the effect is not very strong. Thus, we would not expect a great deal of size and complexity from this effect alone.

So far, we have seen areas where an SSC detector is not necessarily very different from our present day detectors. However, there is still a feeling shared by many that the large physical size of an SSC detector is going to lead to a larger software code problem. For example, an SSC detector will have many more detector channels. But the size and complexity of the code should not scale with the number of channels; only the size of the arrays should grow, not the size of the code. The same could be said about the number of tracks in the detector, except for the second order effect that with a large number of tracks one expects to have areas of higher track density. The change in scale of the energy of the particles in the detector should also not have a strong effect on the code. And certainly, the total amount of iron in the detector doesn't affect the size and complexity of the code.

Thus, we see that, because an SSC detector is very large compared to our current detectors, there is no inherent reason that the code for the detector be any larger than current detectors. The first part of the above reason to worry about SSC detector code seems to be mostly fiction. The second part of the reason is the people factor, which we now explore a bit further.

Over 400 physicists are expected to be collaborating on a large SSC detector. Getting so many people working on a software project is clearly a problem. But in our modern era, it seems that only about 10% of them actually work on the Monte Carlo and event reconstruction code. This means a software team of about 40 software people; a much more manageable number. Of these 40, we might expect them to split up among four to six detector types. That is, for example, the vertex detector, central drift detector, particle Id device (if one exists), calorimetry detector, and muon chambers. If equally divided, there would be seven to ten software people per detector type. People in industry experienced with managing large software projects tell us that this is about the right size for a software team. In fact, today's large detectors are usually undermanned in software development efforts, leading to software teams which are too small.

Thus, it appears that the second part of the above reason for major software problems with SSC detectors seems to be also mostly fiction. And yet we know it is a fact that with each generation of large detectors, the software problem is growing. The discrepancy between what we have concluded is fiction and the fact that software is an increasing problem lies in correctly identifying the causes of our software problems, which we explore in the next section.

III.1 Causes of our Software Problems

I do not profess to understand all the causes to the software problem. Nevertheless, I will discuss some causes I have identified. I do not pretend to understand their relative importance. With different large collaborations, in fact, their relative importance may be different.

The first cause is the some physicists in a collaborations don't take software seriously enough. The software system is a very important part of a modern detector and yet there is not enough effort put into the software at an early enough stage in the detector development, constructing and commissioning. There is, in general, little monitoring of the progress that is being made in the software effort compared to monitoring of the progress in building the detector itself. There is sometimes also an attitude of many of the key potential software writers that "I'm too busy now to worry about software," or "A software error now is not serious, we can fix it later." Sometimes younger, less experienced physicists are the only ones writing the code at an early stage, with little guidance, monitoring, or control by more experienced physicists. This leads to large, important parts of the coding being rewritten after the first real data is taken.

Another cause is that some physicists in a collaboration take software too seriously. By this I mean that some collaborations spend an excessive amount of time discussing what software tools and methodologies are thought to be *necessary* for the success of the software effort. Frequently, religious wars break out between proponents of competing techniques. The excessive time spent on discussing whether to use FORTRAN or another more modern language, or the discussions on the *best* operating system to use, or which code management system to use are all symptoms of this problem. Frequently, when software is taken too seriously, a group builds an overly complex and/or fancy foundation on which to build their physics code. This is caused by allowing an abundant amount of creativity to run free in the tools and utilities. This creativity doesn't always seem to be aware of making normal engineering tradeoffs. That is, frequently, worrying about what the system should do in some 5% detail effect rather than what users need 95% of the time.

There are other areas which may be the cause of the software mess. One of them is how software development teams are organized. Ideally, one would like to see a clear chain of command from top level manager to individual software writers. Frequently, however, one finds a set of people from the various collaborating institutions, and at various stages in their professional career. The software manager, thus, doesn't necessarily have the same level of control, authority, or influence over the software team as, say, a head of an engineering department.

This structure leads to another area. One might like to see the software manager provide strong leadership in order that the software efforts lead to a coherent, well-engineered whole. However, one might find that the software manager is acting only as a coordinator between development teams with different styles, methodologies, and preferences, leading to an overall package that barely works together. Even with an attempt of strong leadership, one still needs to realize that these large software systems are never built from scratch. This leads to code in different areas that could have quite different styles, internal rules, and methods. Temporary interfaces are made between these different areas which may never be eliminated. In practice, one may never be able to achieve a desired level of uniformity of the code.

In general, there is little professionalism in managing the software effort, compared to that found in the building of the hardware. All of the factors mentioned above, plus a tendency on the part of most HEP software writers to work very independently, contribute to this lack of professionalism.

III.2 Some Better Ways to Manage Software Issue

The question is, then, what do we need to do and what tools to we need to keep our software efforts from being such a mess? I don't pretend to know all the answers, but will mention two possibilities below.

First, a large software effort needs a good design. Good design comes with the proper modularity, which may not be as simple as division by detector type. Between the modules, there should be well-designed interfaces, which usually come in the form of COMMON blocks and/or data banks. A good design of a large project cannot be laid down correctly from the start; a certain amount of prototyping needs to be done. When a software team knows it is building a prototype, the whole attitude of approaching decisions change for the better.

Second, in any large project one needs to have progress and quality controls. Unlike hardware, it is much harder to quantify progress or quality with software. Although difficult, it is not impossible to invent some measurement tools, with which a software manager can judge the rate of progress. At the very least, peer review of software modules should be done systematically to judge progress and quality.

III.3 Conclusion of the Software Situation

It is generally felt in our community that, with each generation of large detector, software is becoming a bigger and bigger problem. If we extrapolate this trend to the SSC era, software would be a very big problem indeed. However, software, even for an SSC detector, should not be such a big problem. But we need to understand the fundamental causes of our current problems, before we can find the solutions.

REFERENCES

1. Miura, K., *Proc. Computing in High Energy Physics,* Amsterdam (1985).
2. Becker, J., *Nucl. Instr. Meth.* **A235** (1985).
3. Georgiopoulos, G. et al., *Nucl. Instr. Meth.* **A249** (1986).
4. Miura, Kenichi, *Computer Physics Communications* **45** (1987) 127.
5. Kunz, P. et al., *IEEE Trans. Nucl. Sci.* **27** (1980).
6. Pohl, M., *Computer Physics Communications* **45** (1987) 47.
7. Report of the Task Force on Detector R&D for the Superconducting Super Collider, SSC–SR–1021, June 1986.