ISTITUTO NAZIONALE DI FISICA NUCLEARE

Centro Nazionale
Analisi Fotogrammi

INFN/TC-86/8
30 Aprile 1986

M.Basile, M.L.Luvisetto and E.Ugolini:

HIGH ENERGY PHYSICS EVENT PROCESSING ON A VECTOR COMPUTER: A CASE STUDY

Servizio Documentazione
dei Laboratori Nazionali di Frascati

Istituto Nazionale di Fisica Nucleare
Centro Nazionale Analisi Fotogrammi

INFN/TC-86/8
30 Aprile 1986

High Energy Physics Event Processing on a Vector Computer: A Case Study

M. BASILE
Dipartimento di Fisica, Università di Bologna and
Istituto Nazionale di Fisica Nucleare, Sezione di Bologna
M.L. LUVISETTO and E. UGOLINI
Istituto Nazionale di Fisica Nucleare, Centro Nazionale Analisi Fotogrammi, Bologna

The paper describes the problems encountered in implementing a high energy physics application on a vector computer and the steps taken to improve the code and speed–up the program over a minimum acceptable threshold. Discussion identifies the difficulties met and gives suggestions for future developments. The aim of the report is also to offer a guide to the solution of similar problems to other experimentalists tackling analogous tasks.

1. Introduction

The growth of computing power demand by the high energy physics community and the new possibilities offered by vector processing computers have been the subject of many debates in the last few years as consciousness of the approaching speed limit for traditional single–processor scalar machines spread out. The need for a jump of some orders of magnitude in processing capability, the various solutions tried or under development and the results obtained with vector and parallel processors in at least one field of theoretical physics, namely QCD on–lattice calculations, are extensively documented in the specialized literature. On the contrary, real–life experience of high energy physics experimental data processing on a vector computer is hardly traceable inside the "mare magnum" of high energy physics publications; but, as a matter of fact, "vox populi" claims that experimentalist's code both for the event processing and the statistical analysis phase is vectorization–proof (which is almost true!) and that the code–conversion effort results anyway in an unfavourable cost–over–performance ratio [1] [2].

This report was prompted by the will to document (hopefully, to other high energy physics experimentalists' benefit) the problems, solutions and results we obtained in converting and running a typical high energy physics event processing program on a vector computer. Performance of statistical analysis programs with heavy I/O load will also be briefly discussed.

The event processing program was a specialized version of the general pattern recognition (track finding) and event reconstruction (track and vertex fit) program used by the CERN–Bologna–Frascati Collaboration for the analysis of experiment R422. The physics motivation of the experiment is the study of heavy flavors production in proton–proton interactions at the highest CERN * ISR ** energy; about 32 million events were recorded with a trigger requiring the detection of a high transverse momentum electron in two $\pm 20°$ regions around 90° polar angle in the proton–proton centre–of–mass system. A large–volume 4π magnetic spectrometer (the Split–Field–Magnet), equipped with a 70000 wires system of multiwire proportional chambers (MWPCs), allowed the detection of the charged–particle trajectories originating in the interaction point. The electron–trigger regions were equipped with an auxiliary system of dE/dx chambers, gas Cerenkov counters, MWPCs and electromagnetic–shower detectors (EMSDs); a time–of–flight system allowed the low–momentum charged–particle identification in about 10% of the full solid angle.

The program version we used is the first filtering step of the off–line event processing chain; its purpose is to select those events where an electron–candidate track is reconstructed in one of the two trigger regions with a minimum–momentum request and a rough geometrical match with the EMSD module which had caused the electronic trigger. The program reads the raw event data recorded on tape by the PDP 11/60 on–line computer; each event contains about 2000 CAMAC words plus a variable number of words for the MWPCs hit–wire address information. A part of CAMAC data is decoded to flag the EMSD module which gave the trigger, then track finding starts taking into account only the wire clusters of the MWPC just in front of the flagged module. Track finding is performed on two orthogonal wire planes independently and makes use of a linear prediction procedure, i.e. the coordinate of a track on a given wire plane is related to the coordinates on a set of other wire planes through a linear relation whose coefficients (and tolerance limits) are determined via simulation methods. Projected track candidates are then matched in space by using the information of a set of inclined wire planes. Track fitting is based on a quintic spline method and allows to determine the track parameters (charge, momentum and direction cosines with the corresponding error matrix) taking into account the effect of the magnetic field on the trajectory of a charged particle.

This event–filtering program requires an average processing time of about 45 milliseconds per input–event on a powerful scalar computer as the CDC 7600; only 28% of the input events are selected and thus handed over to the next step of the software chain.

2. History

The program has been originally developed for CDC 7600 (or Cyber 70/76), a 60–bit word machine, 27.5 ns cycle time and 64 kwords very fast memory module (called "SCM" i.e. Small Core Memory) with 275 ns cycle time. SCM is interposed between a slower but much larger main memory (the so called "LCM" i.e. Large Core Memory) and the operational registers. The operations are carried out by 9 functional units, all executing concurrently and all pipelined except the divide unit which is not segmented. Functional units are specialized hardware units that implement an algorithm or a special instruction set more complex than transmit or control operations. In general the logic is independent for each functional unit, thus allowing the operation of all units at the same time. At processor level, time is measured in machine cycles, where a cycle represents the time unit for the processor. The pipeline consists of segments, each one cycle long. Functional units of pipelined processors are segmented so that a new set of operands can enter a unit at each new cycle. In this way, for each unit a new result is produced at every cycle after the first complete operation. Depending on pipeline depth (that is the number of segments in the pipeline), different overall speeds can be attained. Instructions are held in a 12 register stack with a 48 instruction capability. With this architecture a peak rate up to 35 Mips [3] could be achieved. Unfortunately, due to memory organization, for bigger applications data must be stored in LCM and, as no automatic mechanism is provided for LCM–SCM communication, the program must use explicitly special statements to move data (block transfer) from SCM to LCM and back again.

On the other hand, as high energy physics experiments are analyzed by collaboration groups belonging to various Universities and Research Institutes in Europe and elsewhere, program portability is more important than program efficiency. Thus the architectural differences among computers are hidden to the user by

* European Organization for Nuclear Research, Geneva

** Intersecting Storage Rings

general purpose libraries that handle bit management, data movement and mathematical computations such as matrix operations. In this way the operations that depend on word size and precision are hidden to the application programs, frequently at the expense of routine calling overhead.

Mainframes are designed as number crunching machines with high processing capabilities in a batch environment but, as we have mentioned, the experiments take data in a real time environment where 16–bit machines are the favourite task solvers. Data coming from the experimental apparatus are processed by complicated electronic set–ups, where the steering task is accomplished by a 16–bit mini or 32–bit super–mini machine that writes tapes to be further analysed off–line by the mainframe machine.

As a consequence, when the mainframe is a 60–bit machine, part of the processing time in the off–line programs is spent in bit manipulation. And this time increases in case of memory shortage, when data packing and unpacking takes place at each event.

Another problem caused by limited memory is magnetic field handling. As mentioned before, events are built up by tracks moving in a magnetic field that must be taken into account for track reconstruction in space. In case of inhomogeneous field its shape and characteristics are described by a map that is made of a great amount of coefficients stored as bidimensional arrays, that are not always full, with rows either completely missing or partially filled. For a large–volume highly–inhomogeneous field like the one of the Split–Field–Magnet at the CERN ISR, the size of the array, without taking into account the empty elements, is 70000 coefficients. Therefore complicated algorithms are devised to read and store the magnetic field information with the bulk of data stored in LCM and an array of pointers describing the "shape" of the data array, obviously stored in SCM. When the field components at a given space point are required, the pointer is used to find out which row, if any, is to be extracted from the array in LCM.

The above considerations apply specifically to our case. A more general problem is represented by memory management that affects heavely program efficiency.

3. Memory Management

Events produced by high energy physics experiments generate very complicated data structures with variable size information ranging from a few to a hundred computer words per event or more up to thousands depending on event topology, energy, physics process involved, just to mention some parameters that contribute to data structure and consequently influence the programming technique best suited to cope with data analysis. Fortran array definition is not the ideal way to handle this kind of data as no dynamic memory allocation is provided. The typical event structure is a tree structure that has as root "the event" and as branches "the tracks". Event information is usually run and event number, data acquisition date, general set–up characteristics, and the like. The branches are made of more or less complicated structures that contain track information such as coordinates in plane and space. The number of tracks per event and of data per track changes from event to event, therefore a static dimension means that in some cases memory is wasted (i.e. empty) while in other cases the event must be discarded due to memory lack. Furthermore, every time a change is necessary to increase array dimensions, the whole program must be recompiled and, even worse, any correlated library must also be changed accordingly.

Thus to handle memory in a dynamical way, for high energy physics computation have been developped "ad hoc" packages that interface standard Fortran statements with a special subsystem allowing structure generation in the form of routine calls and pointer maintenance in Fortran arrays. This high energy physics memory handling system is much more flexible than any other dynamic allocation supported by structured languages. The system is conceived to be able to handle different kind of links connecting both serial and parallel structures, that is a structure can refer to tree elements in the usual way or to other data (structured or not) that do not belong to the structure itself. In this way a high flexibility is achieved and the structure layout is suitable for any other data representation that implies logical connection among the entities describing the data as a whole. Therefore this way of data representation is used to describe any complicated and logically related objects that contribute to the event computation, particularly experimental apparatus with its variety of detector modules. Furthermore the system is able to discard empty branches recreating a new compressed structure when space is needed. This last feature is extremely important as processing can alter the amount of data in the structure element. In general data reduction is operated at the various stages while new elements may arise. Complicated structures that are changed or newly

created during processing are handled by special routines called "processors" which work on pointers to the dynamical storage.

Data analysis is a statistical process and final information on data behaviour is plotted in the form of histograms enabling a fast check of correct elaboration. Also histogramming technique is best exploited by dynamic structures where different information sizes fit in a easier and more adequate way to the huge amount of information handled.

The interface between application program and memory management is an array in blank common, whose size is declared at initialization and therefore can be adjusted to the peculiar needs of the current processing. The drawbacks of this system are:

1. data items are always seen as contiguous words in storage all belonging to the same array addressed by different pointers,
2. data type is not easily known at any processing step,
3. information about structure manipulation in the routines that handle only pointers to the dynamical memory is hidden at statement level and, last but not least,
4. data structure are not represented by the program definitions, thus somewhere else there must exist a documentation file describing data.

Unfortunately, this last drawback presents serious problems when a structure layout must be altered and whenever updating, changes, corrections and debug are performed on the program that should also be reflected by the documentation file.

The memory management system is realised in the form of library routines that are linked to the application program at execution level. There are essentially two libraries, one contains general purpose routines that can be called also as stand alone, while the other contains the memory handling routines that are partly hidden to the user who calls only those required to create, update, fill and eventually purge branches in the structure. General purpose routines are used to copy data from one memory area to the other, to fill arrays, to process data in the form of bit strings, and so on.

As described so far, provided that data structures are well documented, the system enables a simple program layout with a limited number of processors, each one playing its special and well confined role, acting over a limited set of structures with well defined input and output data, corresponding to a flexible prototype, whose size and contents can be changed at user's need without altering the general layout.

To sum up, this system, for well documented applications, improves program readability and design but, on the other hand, program speed is getting worse due to routine calling overhead and data processing mechanism. And this overhead is especially heavy for parallel and vector computers, as we will see when describing the steps gone through trying to speed–up our test program.

4. Vector Computer Architecture

The computer used for our test program is CRAY X–MP model 12 *, a single–processor machine with 2 million 64–bit word random–access solid–state memory. The CPU clock period is 9.5 ns with 105 million floating–point operations per second. Central Memory is divided into 16 banks that are independent of each other with sequentially addressed words residing in sequential banks. Central Memory cycle time is 8 clock periods. Instructions are 1 or 2 parcel long where a parcel is 16–bit long [4] [5] [6] [7] [8].

Instructions are fetched from memory and stored in the Instruction Buffers. CPU has four such buffers, each holding 128 consecutive 16–bit instruction parcels. Delays are generated if any of the following is met:

1. a 2–parcel instruction is met and a blocking mechanism sets a wait state until the first parcel is issued,
2. the instruction is in a different buffer than the previous one and a change of buffers occurs requiring a 2 clock period delay,
3. the instructions must be loaded from memory at the rate of eight words per clock period. The first group of 32 parcels delivered always contains the next instruction, thus limiting the time delay.

Forward and backward branches are allowed within buffers. Large loops containing up to 512 consecutive instruction parcels can be mantained in the four buffers and programs can take advantage of this possibility.

* located at CINECA, Bologna

The CPU consists of operating registers and functional units associated with three types of processing: address, scalar, and vector. Address processing operates on 24–bit registers and integer arithmetic functional units performing addressing and control operation. Vector and scalar processing act on data. A vector is an ordered set of elements. A vector instruction performs the same function repeatedly on a series of elements and produces an analogous series of results. Scalar processing acts on one or two operands producing a single result. Scalar processing operates on 64–bit scalar registers and four functional units. Vector processing acts on a set of 64–element vector registers, 64–bit each and seven functional units, three of which are floating–point and support both scalar and vector operations.

To achieve the highest possible speed data should not be stored in memory for temporary results, for this reason besides primary registers there is a set of intermediate registers that are not directly accessible to the functional units but act as buffers for the primary registers. Primary registers are designated as A, S, and V. The intermediate registers for A registers are designated as B, those for S registers are designated as T. There are 8 24–bit A registers and 64 24–bit B registers. The main task of A registers is address computation for memory reference, but also integer arithmetic can be performed with A registers enhancing program speed. Scalar arithmetic and logical processing are performed by the 8 64–bit S registers with temporary buffering accomplished by the 64 64–bit T registers. The major computational registers are the 8 64–bit 64–element each V registers. The length of the vector operation is set by the vector length register VL. Vector lengths greater than 64 must be taken into account at software level. Vector operations have a dead time to get started therefore, usually, speed efficiency is achieved only for vector lengths greater than some operation dependent threshold. The Cray Fortran compiler has default threshold values ranging from 2 to 14, but a good compromise is represented by a vector length above $\simeq$ 10. Vector operations can be chained (see §6) so that two or more results can be produced at every cycle after start up (one result per functional unit per clock period). Functional units perform algorithms in a fixed amount of clock periods, delay is impossible once the operands have been delivered to the unit. Functional units are fully segmented. This means that new operands can enter a unit at each cycle.

From the above follows that bidimensional arrays should never have the second dimension multiple of 16 otherwise bank access conflict arises thus increasing the time required for memory access. Vector registers should not be used when arrays are too short. A good use of address registers and interleaving of short instructions with vector chaining can result in better program efficiency. Short routines should be avoided as they can be responsible for frequent loading of the instruction buffers from memory.

5. Code Conversion

The first task that we had to accomplish was the implementation on Cray of a working program starting from the CDC production version. The code is nearly all Fortran and the changes needed belong essentially to one of the two classes: local changes or library changes. With local changes we mean handling of differences that depend strictly on the peculiar application program, while library changes involve changes due to the general purpose and the memory management libraries.

The architectural differences between the two machines that affect Fortran code are essentially due to the different internal representation of data. The word length is 60—bit on Cyber and 64–bit on Cray, alphanumeric data are represented in ASCII on Cray (1 byte per character) and in a special internal code on Cyber (6 bits per character), thus up to 10 characters can be packed in one Cyber word, while only 8 fit in one Cray word. Both floating and integer data have different representations. Statements involving Hollerith data, such as format definitions, must take into account the different packing factor; conversion routines are needed for floating and integers.

Part of these differences represent a minor problem as portability is already handled at a very high degree for all high energy physics software. In fact the input data for our test program were either in the form of cards acquired from the standard input stream or binary records stored on PDP tapes and therefore handled in a special way on every machine (see § on "Tape Handling"). For the card information no format conversion for floating, etc. is needed, while tape data contain only 16–bit binary words and character strings. For character data the situation on Cray is simpler than on CDC as such information was recorded on PDP tapes in ASCII, therefore once identified the CDC character handling routines, these were eliminated and replaced by shifts and logical operations to pack PDP 16–bit words to Cray 64–bit. The following example can be used as a guide–line to solve this kind of problems. PDP words are Fortran format A2, furthermore on PDP

bytes are stored in reverse order with respect to Cray. Let us suppose that the string "NT" has been stored as a Cray integer at location IBUF(n). The hexadecimal contents is "000000000000544E", where "54" is the ASCII code for "T" and "4E" is the code for "N". The following statements store the correct Cray result at location IQ(lbk+m). The constants IA2 and BLK6 are used to pad blanks to produce A2 left–justfied format and contain the following hexadecimal values "FFFF000000000000" and "0000202020202020".

```
IQ(lbk+m)=SHIFTL(IBUF(n),56).OR.SHIFTL(IBUF(n),40)
IQ(lbk+m)=(IQ(lbk+m).AND.IA2).OR.BLK6
```

Local changes, besides the just mentioned one, were limited essentially to the magnetic field handling routine. As we have said earlier this had been tailored on Cyber 70/76. Firstly we had to transfer the magnetic field map, that is the coefficient tape. As this operation is made only once, we prepared "ad hoc" utilities on both machines, the one running on Cyber and producing an output tape with variable length records without control words (this is achieved using the Fortran BUFFEROUT statement together with the Record Manager parameters RT=U, BT=K), the other running on Cray to read and convert the tape. Data conversion is handled by three Cray Benchlib routines: U6064 to unpack 60–bit to 64–bit Cray words, INT6064 to convert integers, FP6064 for floating–point data. The calling sequences for the routines are:

1. **U6064**(INBUF,INBIT,OUTBUF,NUM), where:
   INBUF is the input buffer,
   INBIT is the first useful bit of INBUF (zero–based),
   OUTBUF is the resulting buffer (64–bit left–justified zero–filled words),
   NUM is the number of words stored in OUTBUF.
2. **INT6064**(INBUF,OUTBUF,NUM), where:
   INBUF is the input buffer (CDC left–justified integers),
   OUTBUF is the resulting buffer (64–bit Cray integers),
   NUM is the number of items to convert.
3. **FP6064**(INBUF,OUTBUF,NUM), where:
   INBUF is the input buffer (CDC left–justified floating),
   OUTBUF is the resulting buffer (64–bit Cray floating),
   NUM is the number of items to convert.

Finally the routine in the application program was partially rewritten. Coefficients were no longer packed, movement between memory levels had to be taken out, data were now in Cray format.

The remaining changes belong to the library difference handling [9]. The modifications needed were two–fold, the ones needed because of missing routines in the Cray version of the libraries and the ones due to library evolution. The general purpose libraries must both support existing programs and evolve to manage new experiments. Usually the life time of a production program lasts years, while at the same time the libraries are upgraded more than once. As the Cray versions are fairly new, some routines have been cancelled, others have been developped instead and so on. Thus we had to face the problem of upgrading the memory management package interface that handles the "processors" calling mechanism.

A processor is made of one or more related routines working on data structures. To offer dynamic handling of lengthy, variable size actual arguments, a processor acts on a calling structure that links it to the memory management system as a whole permitting access to other active processors and to the data structures. The original release used by the program we inherited handled the processor calling mechanism with a special three step protocol. A routine call was required to book a structure for each processor, another one was required to call the processor itself, a third routine was supplied to exit the processor. Furthermore the processor interface structures could be deleted to free memory. The new release creates instead a special structure for processor handling with its own root permanently residing in memory. If, by chance, a branch of this structure is deleted the program will crash at the next request for a new branch. Booking of new processors and processor exit are handled in a different way with a handshaking protocol that grants correct execution of processor entry–exit sequence; finally processors are called directly by the Fortran call statement.

Problems arose with calling sequences. On CDC routines could be called leaving out arguments. As on Cray no argument counting mechanism is implemented this short–hand notation is not allowed and, if

matching between formal and real arguments in the call fails, the job aborts in strange ways that are often difficult to debug. Thus calling sequences must be checked with care.

The input package used to read PDP data tapes was missing in the Cray library routines and ought to be newly implemented. Fortunately the Cray standard library offers a package of routines that do the job in a clean and efficient way and, last but not least, the resulting program is portable, while the use of similar Fortran statement like BUFFERIN–BUFFEROUT is restricted to Cray and CDC. As tape handling is of great interest to high energy physics programs and our solution is extremely general, we will deal with it in detail when discussing I/O problems.

A more detailed description of the modifications and implementations completed by an exhaustive list of routine calls and related arguments together with entire programs plus Cray control cards is given in [10].

Besides the time required to install the libraries and to detect how to change the code, the overall modifications took nearly 30 man–days to have a working version. No optimization was tried in the unchanged routines, while the new code was written to perform at best on Cray.

6. Optimizing Methodology

Optimization is a complex task that involves a good knowledge of the program and a deep insight of the target machine both of hardware and software, with special regard to the compiler and to any available timing tools. The task is especially hard when the program has been developed for a machine with a different architecture and is made of some 15 thousand Fortran lines plus a variable amount of library routine calls, as in our case. A clean program would be highly desirable to start with, but usually this is not the case in particular for those parts that had been optimized for the source machine, as tricky programming is frequently needed for best results. Furthermore, vector processors optmizing requisites are, usually, exactly the opposite of scalar machines requirements [11].

From a theoretical point of view it is simple to detect the worse performing routines, but it is not at all easy to predict the gain, if any, that can be reached applying the improvements to a real life problem; thus some criteria must be identified before attacking such a difficult task that requires critical handling of complicated and extensive code.

We will consider first the architectural characteristics that contribute to program performance and how to take advantage in speeding up the code. In general basic instructions are seen by programmers as primitive, indivisible operations, but this concept is not at all true. Even the simplest instruction consists of several sequential steps, that could be summarized as follows:

1. compute instruction address,
2. fetch instruction from storage,
3. decode instruction,
4. compute operand address,
5. fetch operand from storage,
6. execute instruction,
7. store result.

Some of this steps are slightly different whether fetch/store operations are accomplished on memory or on registers.

To obtain higher performance in the execute instruction phase, Cray X–MP implements the parallel operation of functional units and the pipelining technique (both successfully experimented in the CDC 7600) [12]; thus it is able to execute multiple instructions concurrently on the different functional units. Functional units are all segmented, that is a single instruction is executed as a sequence of steps, each step corresponding to a unit segment which acts as a specialized facility of a pipeline. At any given moment there might be the same instruction executing on several (pairs of) operands, each one occupying a different segment of the pipeline, with one clock period time per segment. Cray X–MP allows also the chaining of vector instructions, that is the result coming out of a functional unit (one result per clock period) can go directly as input operand into another functional unit (i.e. the same V register is used as result and operand register for two different functional units). If the instructions are badly queued, conditions may arise that prevent the chaining of two vector instructions until some other operation is accomplished. This situation is met when the second instruction needs an unavailable register for its operand or a busy functional unit. The delay magnitude

can be comparable to the execution time of the complete instruction and can be very high if it applies to vector operations at full vector length. Therefore great care must be taken when mixing address and scalar instructions to vector operations not to miss vector chaining. At the same time, moving address and scalar instructions in between vector operations, provided that free clock periods are filled correctly, will produce code that executes efficiently. Unfortunately adherence to these guidelines reduces program clarity. To best match cycle interleaving Cray stores instructions in the instruction buffer and uses intermediate registers for partial results. Even so, branch instructions affect badly pipelined processors performance as they reduce the ability to prefetch instructions, therefore the code should be arranged so that the most frequently used code is kept together. Another time consuming task is random access to arrays. Cray has an efficient algorithm to access both sequential and constant stepped storage locations, provided that the step does not generate bank access conflicts; nevertheless large disconnected arrays should be avoided.

Usually the Fortran programmer is not so deeply involved in some of the above considerations as the compiler should do all the job. Vector registers are used when DO loops in agreement with vectorization rules are met. Cray Fortran compiler allows some interaction to alter the machine code produced by the compiler in the form of directive statements. In any case structured IF statements should be preferred to logical ones, branches should be avoided, array dimensions must not generate bank access conflict, Cray library routines should be used where possible, algorithms should be implemented accordingly. If, however, Fortran performs too badly and machine language is the last hope, cycle usage must be evaluated carefully to really achieve maximum speed.

Up to now the optimizing process is an empirical one that is managed as a compromise between achievable results and programmer time required. From experience [13] rules of thumb have been established such as "80% of the time used is due to 20% of the code", "do–loops are the field where most of the gain can be obtained", "loop unrolling is very efficient". So we tried to investigate the behaviour of scalar and vector processing of do–loops and any gain given by loop unrolling.

Let us consider the following code fragment, in which the loop at statement 200 is a example of one level of loop unrolling with respect to the loop at statement 100.

```
      PARAMETER (MAX=5000)
      REAL X(MAX), W(MAX)
C
C     SIMPLE DO LOOP
C
      DO 100 K=1,MAX
      W(K) = X(K)*X(K)
100   CONTINUE
C
C     DO-LOOP WITH UNROLLING DEPTH 2
C
      DO 200 K=1,MAX,2
      W(K) = X(K)*X(K)
      W(K+1) = X(K+1)*X(K+1)
200   CONTINUE
```

Timing measurements have been done also with a two–level unrolling, both for vector and scalar compilation (parameter OFF=V for the CFT — Cray Fortran compiler). The results and gain are shown in the following tables.

Cray Vector & Scalar Loop Time (sec)			
Level	Scalar	Vector	Speed–up
0	1.33E–2	6.71E–4	19.82
1	7.84E–3	5.56E–4	14.10
2	5.71E–3	6.31E–4	9.05

Cray Unrolling Speed–up		
Level	Scalar	Vector
0/1	1.70	1.21
0/2	2.33	1.06

We tried to run the same test on scalar computers with a highly optimized Fortran compiler (i.e. VAX 11/750*, VAX 11/780 and VAX 8600 all running VMS 4). Accurate timing gives rise to serious problems as these machines are very sensitive to the overall load due to their virtual memory architecture, thus to get reproducible results the CPU–time must be increased usually raising the number of statements in the loops. Therefore we used the simple trick of writing nested DOs as shown in the code below.

```
      PARAMETER (MAX=5000)
      REAL X(MAX), W(MAX)
C
      DO 100 J=1,MAX
      DO 100 K=1,MAX
      W(K) = X(K)*X(K)
100   CONTINUE
```

The loop is obviously dummy but it must be time consuming without the need to increase memory allocation thus creating a penalty both on Cray (program too big) and on VAX (too many page faults). When we ran the program we found surprisingly that Cray code was 1.5 times slower than the VAX 11/750 one, that is .331 seconds on Cray and .217 on 11/750. Thus we noticed that the Fortran compiler on VAX is so optimized to realize that the external loop is completely useless. Therefore rules of thumb must be taken into account very carefully and verified before undertaking costly program modifications.

From our inquiry, it is obvious that the power of Cray is best exploited only when vector registers can be fully used. Just to show an example the following code performs 45 times faster on Cray X–MP/12 than on Cyber 70/76.

```
      PARAMETER (IVEC=64,NTIM=100,ABEG=7279.,BBEG=3523.,XBEG=3.7859)
      PARAMETER (NTOT=(IVEC-1)*NTIM)
      REAL X(NTOT),Y(NTOT),A(NTOT),B(NTOT)
C
      DO 100 K=1,NTOT
      A(K)=ABEG*(1./FLOAT(K))
      B(K)=BBEG*(1./FLOAT(K))
      X(K)=XBEG*FLOAT(K)+XBEG
100   CONTINUE
C
      DO 200 K=1,NTOT
      Y(K)=A(K)*X(K)+B(K)*X(K)*X(K)
200   CONTINUE
C
```

Thus the only successful changes that require a reasonable programming investment are limited to inhibit vectorization for short loops and reduce routine overhead. While vectorization acts only on vector computers, routine overhead reduction improves performance also on scalar machines, even if vector machines are more sensitive to routine calls. In our test we had a gain of 2.3 for scalar machines and 3.7 for Cray on the same code by taking out a call from a do loop.

7. Timing Tools on Cray

Before affording the optimizing task on vector computers one needs to know the vectorization efficiency and the time spent by the program. The CFT compiler gives information on the vectorization level by

* without floating point accelerator

printing a list of vector loops, short vector loops (i.e. loops with vector length ≤ 64), loops that inhibit vectorization and, at user request through the CFT option ON=D, a table of all DO–loops. To trace the flow of Fortran programs on Cray there exist library and compiler options to determine the most time–consuming parts of a program. The simplest use of the flow–tracing facility is given by specifying the option ON=F at compiler level, thus obtaining a global information. If the user wants to limit the tracing information to selected parts of the program, inner compiler directives can be invoked. The directives are in the form of comment cards to be typed at the beginning and at the end of the code that must be traced. This convention on directive syntax keeps the code portable. The tracing directives are:

```
CDIR$ FLOW      !  ENABLE FLOW TRACE
CDIR$ NOFLOW    !  DISABLE FLOW TRACE
```

Another way to use selective flow–trace is offered by a package of Fortran callable routines included in the standard Cray library. The routines are:

```
CALL FLOWENTR    enable flow--trace
CALL FLOWEXIT    return from traced subroutine
CALL FLOWSTOP    end program execution
CALL GETNAMEQ    return name of traced subroutine
CALL GETREGS     return register usage statistics
CALL SETPLIMQ    initiate detailed tracing
CALL ARGPLIMQ    initiate listing of argument values
CALL FLOWLIM     set tracing limits
```

In order to produce the trace output, immediatly after the card invoking program execution the following control cards must be present:

```
EXIT.
DUMPJOB.
FLOWDUMP.
```

The above options print a summary containing the timing information for each subroutine and subroutine linkage overhead for the whole program. The items listed are:

1. time spent in the routine,
2. percent of total time spent in the routine,
3. number of times the routine was called,
4. average time per call spent in the routine,
5. a list of the first 14 routines called by the subroutine,
6. a list of the first 14 routines that call the subroutine,
7. total number of routine calls,
8. register usage statistics.

The overall trace option is generally preferred as no physical changes to the source code are nedeed, on the other hand a careful use of the other timing facilities enables the tracing even of single DO–loops.

Finally the library routine SECOND can be used to measure the elapsed time between any two user selected statements. This routine gives reliable and reproducible results and can prove extremely useful in detecting critical areas of the program.

8. Code Optimization

From the above considerations and from time measurements carried out on vector loops, we may argue that high computational speed is achieved when a high degree of low–level parallelism is inherent to the problem under investigation. This means that short invariant pieces of code can be translated into short loops involving long vectors. This is not the case with high energy physics data analysis programs that are complex, big and full of decision–taking algorithms, while arrays that can be processed using vector registers are very short, so that in many instances vector registers perform worse than scalar ones. In our test program some degree of parallelism is achieved only at event level. Furthermore not all events are processed in the

same way, because at this stage of data analysis nearly one third of the events represent good physics, while the other two thirds are discarded as background, so that processing time differs greatly in the two situations. The ideal computer for this kind of programs should therefore allow a two–level parallelism, a low–level one with vector units for the occasional long arrays (mainly input–output buffers and data conversion routines) and a higher–level parallelism, accomplished by multi–CPU architecture, enabling concurrent processing on the event basis [14] [15] [16] [17].

For the above reasons, it is obvious that, unless the program is drastically rewritten, one can reasonably expect a speed–up of 20% with $\simeq$ 30 man–days investment. These data are in agreement with data published for similar jobs done in other laboratories [18]. In general a longer time spent for optimization is not justified, the improvement factor being not greater than 35–45%. Bench marks run on Cyber 70/76 and Cray X–MP show that programs run from 1.5 to 53 times faster on Cray, strictly depending on application [19].

The program we used as prototype belongs to the typical high energy physics event filtering processing and is made of some 100 routines to which general purpose and memory management routines are to be added reaching a total of 285 routines plus some 80 routines that are linked from the Fortran library. The first step, when trying to optimize some code, is the use of flow–trace facilities that give information on time spent in each routine and time percentage per routine so that optimization is done at first on the worse behaving parts. Our test program does not follow, as one could easily infer from the size of the source code, the general empiric rule that asserts that 80% of time is spent in less that 20% of code. To run trace monitoring, the program must be recompiled with tracing options, thus we decided to leave unchanged the memory management system and to improve only the application program and the general purpose routines. A first run using the flow trace information on 133 routines with 450 input events declared that only two routines used more than 15% of the time, a few routines ranged from 3 to 7%, while the rest was below 2%; routine calls summed up to 984662.

As the CFT compiler prints loop information but does not produce a statistics on vectorization, we have written a Fortran program that reads the list produced by CFT and writes a statistics of vector loops, short vector loops, total loop amount, number of routines compiled, total amount of loopless routines. We ran this program on the 187 most called routines; only 107 (57.2% of the total) contained at least one DO–loop. A summary of vectorized loops is shown in the next table.

Vector Rates Over 478 Loops		
Vector Length	No. of Loops	Percent
>64	21	4.4
$\leq$64	48	10.0

Therefore improving the code of existing loops would have a negligible effect on the overall performance, thus we decided to concentrate our attention on the flow trace information.

The longest time was spent in unpacking and packing PDP input data. These operations are carried out by Fortran routines that, even if working on any bit pattern, were initially developped for 60–bit target machines where problems arise handling byte–oriented data, as a consequence their performance on Cray is very poor. With regard to the remaining routines, we decided to investigate the most called ones and to try timing considerations on simulated situations instead of acting on the whole program. At the same time, as some improvements were applicable also to scalar code, we tried to estimate the time gain both on vector and scalar machines. The same improvement, as already mentioned, produces a code running nearly 4 times faster on Cray and only 2 times faster on a scalar computer. To avoid calling overhead, all those routines that could be changed to statement functions were translated accordingly, even if sometimes we were obliged to change also the calling sequences [20]. Examples are given by the following code.

As bit handling is used both to check the status of branches in structures and to unpack data, two functions to get one bit (JBIT) or a string of bits (JBYT) are frequently called. The calling overhead can be greatly reduced by introducing the two statement functions instead.

```
JBIT(IZW,IZP) = SHIFTR(IZW,IZP-1) .AND. 1
JBYT(IZW,IZP,NZB) = SHIFTR(IZW,IZP-1) .AND. MASK(128-NZB)
```

Another similar case is given by the dot product of 3–element arrays. This is usually done by a general routine that makes use of vector registers. The overhead due both to routine call and inefficient performance of vector registers, can be reduced by a factor 21.4 by using a statement function. The original call is:

```
      REAL X(3), Y(3)
C
      SDOT=VDOT(X,Y,3)
```

where the dot product is carried out by the following Fortran code:

```
      FUNCTION VDOT(A,B,N)
      DIMENSION A(N),B(N)
C
      XX= 0.
      DO 9 I= 1,N
9     XX= XX + A(I)*B(I)
      VDOT= XX
      RETURN
      END
```

As statement functions cannot reference array elements, the optimized code needs also rewriting of the calls, as follows. Furthermore the statement function must have a different name to avoid clashes with the general routine VDOT.

```
      REAL X(3), Y(3)
      DOT(A1,B1,A2,B2,A3,B3)=A1*B1+A2*B2+A3*B3
C
      SDOT=DOT(X(1),Y(1),X(2),Y(2),X(3),Y(3))
```

The next step involved changes of logical IF to IF–THEN–ELSE structures, modifications to include short routines directly in the calling programs, reordering of tests and DO loops, especially in the general purpose library. To check speed–up, we made time measurements on typical calls emulating the situation in real life programs using different vector length values. The emulation test on the new library gave the following results.

General Purpose Library Speed–up	
Vector Length	Speed–up
3	1.3
10	2.0
100	7.7

After these changes we reached a total gain of 10.5% over the unoptimized Cray version. Then we changed all the packing and unpacking routines, using the Cray Fortran library optimized versions. This was not a straightforward operation as the two versions were not exactly equivalent and some code management was necessary to switch from the old to the new ones. The total gain reached now 27.5%.

Track parameter computations are done by means of matrix operations that involve multiplication, inversion and solution of linear equations applied to variable size arrays. The matrix package makes use of the address of Fortran arrays to compute the stride between array elements, therefore, as it is practically impossible to switch to Cray Library routines due to the completely different calling sequences, we used instead a machine language package that had been developped for theoretical physics applications [21], thus gaining nearly another 7% and reaching a final improvement of 34.3%. The flow trace information showed now only one routine using more than 10% of total time; subroutine call overhead was reduced to 835037 calls.

The following table summarizes the optimizing steps and related results. Time is event processing time in milliseconds. On Cyber 70/76 the average time per event amounts to 45.33 millisecs. Timing is related to 450 input events. Speed–up represents the total percent improvement, while gain is the percent improvement

from one step to the next. CDC/Cray is the speed factor between the two CPU times. Optimizing level has the following meaning. Level 0 gives CPU time after code conversion, level 1 is reached after library optimization, level 2 is reached after improving input handling routines, finally level 3 shows the results attained including CAL (Cray Assembly Language) code for matrix operations.

Cray Total Gain (millisec)				
Level	Time	Speed-up	Gain	CDC/Cray
0	33.37	–	–	1.36
1	29.86	10.5	10.5	1.52
2	24.19	27.5	19.0	1.87
3	21.92	34.3	9.4	2.07

9. Tape Handling

High energy physics experiments require the handling of thousands of tapes from data taking to data summary tapes (DST) at the end of the analysis. Tapes are recorded at various computers and interface packages must be developped for portability. Essentially three types of tapes are produced: 16–bit tapes in the experimental area, 32–bit IBM–compatible tapes created by the memory management system and machine–dependent tapes at summary stage. IBM–compatible tapes are handled by the high energy physics libraries, but both raw data tapes and DST tapes, produced as in our case by the PDP and the Cyber Record Manager respectively, need I/O package implementation to be processed on Cray. Both problems are solved making use of Cray standard I/O library routines. Tape handling for PDP tapes needs only call substitution in the CDC input routines, while for DST tapes special routines must be written to unblock physical records taking into account system control words.

Raw data tapes are written with the aim of portability, therefore no system control words are recorded on tape and each physical record represents also the logical information. DST tapes on the contrary are machine and system dependent, therefore logical records are mapped to physical records with control words describing the blocking factors and structures. Raw data tapes for present experiments are frequently written on VAX/VMS based data acquisition systems. Data information can be stored as 16–bit or 32–bit words depending on experiment set–up. The only problem is represented by VAX addressing that is the opposite of Cray and other mainframes. Usually byte swapping is already performed at data–taking level with the aim of portability (IBM format is the common facility); in this case the following code fragment works directly for 16–bit words and needs only the minor change of substituting 32 for 16 in the computation of the variable MXBITS (see below) for 32–bit words. In case byte swapping is required, calls to the Cray Fortran routines UNPACK and PACK solve the problem in a nearly straightforward way.

In our test we decided to stage tapes on Cray mass storage through the IBM VM station in order to preserve the physical record structure. Physical records and files are copied from tape to disk without any conversion. Experimental (raw data) tapes were written on a PDP and contained variable length records up to $\simeq$ 16000 bytes each without control words. The input routine must be able to acquire physical records and to detect their length in bits–per–record, before unpacking from 16–bit to 64–bit integers can take place. On Cyber the read operation was done by general purpose high energy physics routines, for Cray to following code fragment is used.

```
C
      COMMON /INPUT/ BUF(10000)
      PARAMETER (MXPDP=8192,NQBITW=64,NTIN=50,IEOF=2,IERR=4)
C
      MXBITS=MXPDP*16
      MAXPDP=MXBITS/NQBITW + 1
      NCOUNT=MAXPDP
      CALL READ(NTIN,BUF,NCOUNT,ISTAT,IUBC)
      IF(ISTAT.EQ.IEOF) GO TO 2000
```

```
NBITS=NQBITW*NCOUNT-IUBC
```

The common array BUF is filled with the input record and the total number of bits in the record is returned in NBITS. The code is extremely general and brief and can be directly placed where required, thus avoiding the overhead caused by interface routines. The parameter statement defines the maximum size of the input record, the number of bits in the Cray word, the input unit and the error codes. The records are read by the following call:

CALL READ (UNIT,BUF,NCOUNT,ISTAT,IUBC), where:

UNIT is the logical unit number or a character string declaring the local dataset name,
BUF is the input array,
NCOUNT is the maximum 64–bit word record length in input and the actual number of transmitted words in output,
ISTAT is the status flag whose value gives information if read operation successful, end–of–file or end–of–dataset met, error condition etc.
IUBC is the unused bit count.

DST tapes were written by the Cyber Record Manager with the default file attributes (i.e. RT=W, BT=I) and are made of several partitions. Tape structure is the same for different Operating Systems; the following description applies both to SCOPE 2 and NOS/BE and takes into account the most general situation. Nearly a hundred tapes written on both systems have been successfully read using our package. The changes from CDC to Cray involve Fortran differences, unpacking routines and data transformation from CDC to Cray format. The steps involved to create the Cray version are the following: analyse tape contents, then write general purpose library routines to read CDC standard tapes.

The Cray routines of our package read physical records, build logical records, handle end–of–partition information, detect end–of–dataset and support floating point conversions for packed data. In our case to reduce tape occupancy three 20–bit floating point words were packed in one 60–bit word, thus we had to unpack these CDC information to Cray floating point, and conversely pack Cray floating point to CDC format (four empty bits followed by three 20–bit CDC floating point words). Conversion for 60–bit integer and floating are accomplished by the Benchlib routines already described (§5). A detailed description of tape format and utilities is given in Appendix A.

Some timing considerations on I/O operations are reported. The time required to "FETCH" a tape from the VM station to the Cray mass storage ranges from 7 min. to a maximum of 35 min. depending on tape unit load. The average value over several jobs is 9 min. The CPU time required to read a tape consisting of 28532 physical records 512 60–bit word fixed record length is $\simeq$ 1 sec.

With regard to CPU time, Fortran read and write statements are usually more time consuming than the READ and corresponding WRITE Cray library routines. When a long array is written or read the speed–up reaches a factor of 5, but for short arrays time for Fortran I/O can increase up to prohibitive levels such as 58 times longer.

10. Conclusions

Code conversion for Fortran programs that use a common subset of ANSI Fortran is usually neither too difficult nor time consuming. Optimization to a low gain, as the one we operated is also feasible in some 30 man–days. In this way a speed twice the Cyber 70/76 is easily achieved. High energy physics programs are essentially scalar in nature, thus a gain of the order of cycle time ratio between Cray and other machines with similar scalar architecture can be reached. A further gain is strictly dependent on the specific application. For example, running a typical statistical analysis program with heavy histogramming load on DSTs, related to the R422 experiment, we reached a speed–up factor of 2.5 as compared to Cyber 70/76 without any optimization effort. Timing is sometimes critical to evaluate, as on small CPU–time measurements, minimal gains cannot be noticed. In fact on some hundred events the analysis histogramming process gave rise to the same CPU time for Cyber and Cray, while the statistics on 20000 events showed the already mentioned factor of 2.5 improvement.

Compiler efficiency is a bottleneck, as evidenced by the 7% gain of CAL routines that were just translations of the Fortran ones without extremely tricky programming, therefore a higher optimization level of CFT should avoid the need of cumbersome, non-portable machine language programming.

Even more striking is the difference in optimizing efficiency among different compilers, so that it is very difficult to make reliable comparisons. Hopefully, these differences should be kept to a minimum by the new standard forseen for Fortran 8x, that is also designed with the aim of enhanced vector operation performance [22].

In the case of high energy physics programs for event analysis better results will be probably reached by multi–CPU machines with each CPU processing one event. At present Fortran for these machines allows multitasking operations through library calls, therefore the program must be tailored to the hardware at the expense of portability and more complex program design.

However the user must be aware that vector computers show a large gap between peak speed obtainable for highly vetorized programs and the average speed reached by those routines that cannot be vectorized, as scalar speed influences heavily the overall performance of the application programs. Only better Fortran compilers, Fortran enhancements to handle multitasking in some standard and portable way and more careful program engineering will enable the user to take full advantage of the most powerful architectures of present and future supercomputers.

References

[1] R.Pike, Physics vs. Computer Science, Workshop on Software in High Energy Physics , CERN 82–12, Pagg. 13–18.

[2] M.Metcalf, The Role of Computing in High Energy Physics , CERN Preprint DD/83/15.

[3] V.Zacharov, Parallelism and Array Processing, Cern DD/82–20.

[4] R.W.Hockney, C.R.Jesshop, Parallel Computers, Adam Hilger Ltd., Bristol.

[5] F.Walkden, A user's view of parallel processors, Proceedings of the 1976 CERN School of Computing, CERN 76–24, Pagg. 213–226.

[6] T.Bloch, Data Processing in High Energy Physics and Vector Processing Computers, CERN Preprint DD/84/3.

[7] T.Bloch, Trends in Supercomputers and Computational Physics, CERN DD/85/4.

[8] K.Kwang, Briggs, A.Fayé, Computer Architecture and Parallel Processing, McGraw–Hill.

[9] CERN Program Library, HYDRA Topical Manuals

[10] M.L.Luvisetto, E.Ugolini, A guide to code conversion from Cyber to Cray, on–line documentation stored at VXCNAF::[AID.CRAY]HYDCRAY.DOC.

[11] G.Waldbaum, Tuning Computer Users' Programs, IBM Research Laboratory, San Jose, California.

[12] J.W.Rymarczyk, Coding Guidelines for Pipelined Processors, IBM Technical Report.

[13] D.Kracht, J.van Kats, FORTIM — a tool to identify the time–consuming parts of a Fortran program, Supercomputers, July–Sptember 1985.

[14] D.J.Paddon, Supercomputers and parallel computation, Clarendon Press, Oxford.

[15] H.Lorin, Parallelism in Hardware and Software, Prentice–Hall Inc., Englewood Cliffs, New Jersey.

[16] V.Zacharov, Invariant Coding and Parallelism in Data Processing, CERN Preprint DD/82/1.

[17] C.Verkerk, Use of Intelligent Devices in High Energy Physics Experiments, CERN DD/81/03.

[18] J.Arbocz, T.van Baten, R. LLurba, Shell collapse load calculations on supercomputers, Supercomputer, March 1985.

[19] MSC/NASTRAN, Time Estimation and Problem Execution, MSR–is 70, The MacNeal–Schwendler Corporation, February 1985.

[20] M.Metcalf, Fortran Optimization, Academic Press.

[21] F.Semeria, Private Communication and CAL Implementation of Matrix Computation.

[22] X3J3, Proposals approved for Fortran 8x, ANSI X3J3 Committee

Appendix A

Tapes produced by the Cyber Record Manager with file attributes RT=W, BT=I, contain fixed–length records (except for the last record of a partition) of 512 60–bit words corresponing to 480 64–bit words. The logical records are therefore packed to this physical record length with the following criteria:

1. each physical record begins with a control word (ICW = Internal Control Word),
2. each logical record begins with a control word (WCW = W Control Word),
3. each partition is defined by a special pattern of WCWs (see later).

The contents of the control words is the following, the bit field description is right justified, bit 0 to 59, bit 0 being the least significant one.

ICW

1. Location of Next Logical Record (18 bits – bit position 0–17).
2. Number of Next Record (24 bits – bit position 18–41).
3. Block Ordinal (12 bits – bit position 42–53).

WCW

1. User Record Length $l1$ (18 bits – bit position 0–17).
2. Unused Bit Count of Last Record (6 bits – bit position 18–23).
3. Previous Record Length $l2$ (18 bits – bit position 24–41).
4. Continuation flag (2 bits – bit position 42–43), as follows:

0	Full Record	binary = 00,
1	Beginning Portion	binary = 01,
2	Middle Portion	binary = 10,
3	End Portion	binary = 11.

5. Record flag (2 bits – bit position 57–58), as follows:

0	Data Follows	binary = 00,
1	Deleted Record	binary = 01,
2	End–of–Partition	binary = 10,
3	End–of–Section	binary = 11.

6. Parity.

The records can be spanned, that is only full records are present, therefore from the above description, we can see that the record pattern is the following: for the first record in any partition WCW follows immediately ICW, and at any record, but for the first in the partition, $l2 = l1 + 1$. The records, on the other hand, can be divided into fragments, that means that only short logical records are recorded as full records, the others are divided into portions according to the above flag. Our routine therefore handles the record collating information through the following quantities:

1. the address of the next WCW (= 1 always for fragmented records),
2. the continuation flag (as above),
3. the length of the current portion (see $l1$).

Usually the End–of–Partition (EOP) record is shorter than the other records. The EOP configuration, as derived from a statistical tape analysis, is given by the 3 last words in the physical record, whose contents and meaning are:

1: WCW with record length = 1 and record flag = 1 (deleted record),
2: empty word,
3: WCW with record length = 0 and record flag = 2 (EOP).

If the physical record has not enough words left to fit the EOP information, the pattern can be one of the following.

1. The empty word is the last in the record. In this case a short record (4 words) follows. In this record ICW is followed by the above pattern.

2. The last word in the record is the "deleted record" WCW. In this case the record length is 0 and a short record (as described at 1. above) follows.

As tapes can occasionally give rise to problems, both to detect tape errors and to create debugging tools for our tape handling package, the following utility programs have been developed [10] :

1. BADEOP to dump first and last 6 words of physical records in an user defined range,
2. VFYEOP to verify EOPs,
3. ANALTAPE to verify tape structure,
4. DUMPTAPE to dump tape according to input directives.