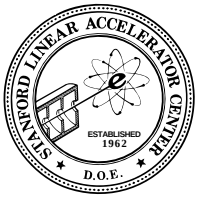
SLAC-PUB-5759
February 1992
(E/I)



SLAC B Factory Computing*

Paul F. Kunz
Stanford Linear Accelerator Center
Stanford University
Stanford, CA 94309, U.S.A.

Abstract

As part of the research and development program in preparation for a possible B Factory at SLAC, a group has been studying various aspects of HEP computing. In particular, the group is investigating the use of UNIX for all computing, from data acquisition, through analysis, and word processing. A summary of some of the results of this study will be given, along with some personal opinions on these topics.

1. Introduction

As part of the research and development work being done for a possible B Factory detector at SLAC[1], a group has been studying the computing aspects of such an effort. With no data expected before 1997, this group finds itself with a luxury of time to prepare and sufficient time to prototype possible solutions to recurring problems in High Energy Physics (HEP) computing. This paper is a report on a few aspects of the work done so far and the work that is planned to be done over the next few years.

The B Factory, like many other initiatives being planned for the late 1990's, will require a very large amount of computer processing power to analyze the expected number of events. The best estimate is about 10^4 MIPS. To achieve these resources at a reasonable cost, one expects to use high performance workstations either working together in a dedicated *farm*, or distributed over the local area network. These workstations run the UNIX operating system, so UNIX will undoubtedly be part of the analysis environment. The focus of the software engineering R&D at SLAC has been to explore using UNIX for everything else as well, that is, data acquisition, on-line real-time systems, production processing, analysis work, code development, databases, and even office work such as word processing, mail, and sending faxes.

In some of the areas mentioned above, UNIX is weak and much R&D needs to be done. However, this paper will focus on UNIX as a code development environment, where considerable work has already been done. This paper does not represent in any way final decisions that will be taken by a future B Factory collaboration as such a collaboration has not yet been formed. Furthermore, it only represents discussions and planning within a small group concerned about the computing aspects of such a detector, and even there, not everybody agrees with what might be the final outcome. However, all those involved do agree that R&D into computing is as important as the R&D into the detector components.

2. Computing Cultures

UNIX is a very different *culture* from the one to which HEP has traditionally been exposed. By culture we mean that one accomplishes tasks in certain ways, that one has certain expectations of what can be achieved, and that there is a tradition and history; just as with human cultures. To illustrate what is meant by the UNIX culture, think about how one edits a file and what features the editor has, how one compiles and links, or how one manages code development. Each computing culture accomplishes these tasks, but in ways that are somewhat different.

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

Presented at the 2nd International Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics, La Londe-Les-Maures, France, January 13-18, 1992.

HEP has the most experience with the mainframe computing culture, first with CDC and IBM, and later with DEC VAX/VMS. The HEP perspective of these computing cultures is based on the FORTRAN programming language which has served us well. However, the UNIX culture is based on the C programming language. The C language was invented to write the operating system, and C language style of accomplishing tasks dominates the UNIX environment.

Although the FORTRAN language is only one component of our mainframe culture, it is an important one because it influences other aspects of the environment. One only needs to look at the technology that is in common use. Take, for example, the set of subroutines, functions and packages that make up the CERNLIB distribution. Apart from the mathematical functions, the style of what is in this library is dominated by the FORTRAN language. Large production codes and other packages make heavy use of data management systems such as ZEBRA[2], BOS[3], or Jazelle[4]. Graphic packages such as GKS or HIGZ[5] are strongly influenced by their FORTRAN bindings and origins and the graphics technology at the time they were designed. Even code management systems such as PATCHY[6], CMZ[7], or CODEBASE[8] were designed for FORTRAN programmers by FORTRAN programmers. Also simple compiler preprocessors, such as EXPAND[9] or PREPMORT[10], were invented to satisfy certain needs of the FORTRAN programmer.

The first question that arises in considering the use of UNIX for everything, is how well would these FORTRAN-based technologies of our mainframe culture mix with the UNIX culture. Or even, should we attempt to mix them. Might it be better to learn the UNIX way of accomplishing these tasks? Before deciding on which technology to import from our mainframe culture to UNIX, we should first understand fairly well the technology that is used in the UNIX culture. The remainder of this paper focuses on this issue.

3. Programming Languages

At first appearance, there doesn't seem to be any reason not to use FORTRAN with the UNIX computing environment. However, as one looks closer, one begins to see some problems. It starts with the interface to system calls. As supplied by the UNIX vendor, they are designed to be invoked from C language programs. Certainly the interface to the windowing system and the graphical user interface (GUI) is designed for C language programming, as well as the interface to the networking system. It is also quite probable that re-training FORTRAN programmers to use C will require less effort, in the long run, then trying to support FORTRAN in an UNIX environment, at least for many programming tasks.

So in contemplating a move to UNIX, it would seem one should also consider moving to a C based language. However, it is frequently said: *The* C *language is not suitable to scientific programming*. We have found such statements overly generalized. The software tasks that one endeavors to achieve in large detector groups go far beyond the numerically intensive computation where, perhaps, C may not be well suited. Every language has its strong and weak points and one needs to take an overall look at all our computing tasks.

Where C has not been tested for large detector groups in HEP is with the off-line production codes. The way these codes are designed is strongly influenced by the handling of data structures. Packages such as ZEBRA, BOS, and Jazelle have been used to support the FORTRAN programmer to such an extent that one is no longer programming in FORTRAN, rather one uses a hybrid language that I call FORTRAN+X, where "X" is one's favorite package. We use these packages to satisfy two basic needs. The first is to handle data, such as tracks and vertices, as structured entities. The second is to be able to move the data from the computer's memory to disk in a form that can be read back for further analysis at a later time.

The use of FORTRAN with such packages has a number of problems, although not all packages suffer from the same problems. For example, one might lose access to data points by name. In some systems access to data is by pointer and numeric offset into a large common block. Such access methods have little mnemonic value. Related to that point is the limited usability of a symbolic debugger when data is accessed in that way. One might also lose portability of the code across platforms, as the auxiliary package has to be ported first. Another important point is that many of these systems are quite large and everybody needs to learn them from relatively poor documentation. One could argue that it is easier to learn another programming language than to learn all the ins and outs of FORTRAN+X.

On the other hand, the C language has the concept of data structures built into the language. Access to data in structures can be made as easily as a statement like

```
px = mctrack[itrk]->px;
```

Dynamic memory allocation for these structures is also part of the language's standard library and it is probably highly optimized by a UNIX vendor since it is important to overall system performance. Naturally, the symbolic debugger knows about structures as well. Clearly, C is a better environment than FORTRAN+X when data is best represented by structured entities.

As good as the C language is for handling data structures in memory, there is nothing built-in to the language or its library for writing the structures to disk. One needs a package to accomplish this or each application would need custom code. The Cheetah[11] package is an example of what can be done. It provides, first of all, a user-friendly way to allocate space for structures and the management of pointers to that space. Structures managed by Cheetah can be written to disk and read back again. At the beginning of a Cheetah file, there is a complete dictionary of what type of structures are contained in the file; the name of the pointer arrays; the name, type, and size of each variable in a structure; and even the comments from the file that defined the structure. For interactive applications, the structure definitions can be dynamically created, and access to the dictionary is available to the program. Cheetah is written with only 2,500 lines of C code which has shown to be very portable across such diverse platforms as IBM VM/CMS, DEC VAX/VMS, and various flavors of UNIX. The features of Cheetah compared to its size is also a testimony to the power of the C language for this kind of package.

Another important study will be the use of object oriented programming (OOP) technology. The early work at SLAC on the Reason and Gismo projects has adequately demonstrated that OOP is a significant advance over procedural languages for programming with windows and the GUI. There are now also prototypes of analysis codes using OOP as well. The MC++[12] project, for example, is an event generator toolkit written in C++. The Gismo[13] prototype for detector simulation and reconstruction written in Objective-C was completed in the spring of 1991 and is now being re-designed and written in C++. There is also the CABS[14] analysis package which uses a C++ class library for support.

It seems that the most popular OOP language and the one best suited for numerically intensive work is C++. Its C-based heritage makes it easy to interface to the operating system, the GUI, etc., while its data abstraction capabilities allow for OOP and operator overloading. C++ can be somewhat tedious, however, especially when coding such things as a base class library. On the other hand, once a good class library is established, it seems that it is not difficult for average programmers to accomplish their tasks. Prototyping work is needed to judge the suitability of C++ as the language for a large detector group.

Now that the FORTRAN '90 standard has been published and at least one compiler is available, prototyping work can begin in this *new* language. Our emphasis will be on testing those new features of the language that deal with areas where FORTRAN '77 has been weak: the handling of data structures. Are the derived data types in the language sufficient for the tasks that need to be done? Will we be able to write a general purpose I/O system, such as Cheetah, with FORTRAN '90? Can it be used when interfacing to UNIX system calls, to the GUI, or to the networking systems? The data abstraction aspects of the language are a big improvement, but it lacks inheritance, thus one can not take full advantage of OOP. Will this be significant in real world applications? In order to get some answers to these many questions, prototyping in both FORTRAN '90 and C++ will be done in parallel.

The issue of switching a HEP collaboration to a language other than FORTRAN is a difficult one. The prototyping work with other languages fits into a strategy for gaining acceptance of another language. Before people will change their habits, they must see the benefits of a change and examples of HEP analysis code in another language will be needed to show the benefits. The prototypes become these examples. One will also need a sufficiently large body of people who have tried another language and have enjoyed its benefits. Those that work on the prototypes become that group.

Another issue in switching languages that is frequently raised is one of training. So far we have found that experienced programmers have been able to learn the C language relatively quickly. The C++ language, however, is another story. It is felt that people will need to be motivated by understanding the benefits before tackling this language. In both cases, however, having example analysis codes will facilitate the learning process. It is also true, that more and more of the younger generation already know C or even C++ when they join the collaboration.

One final issue in switching languages is frequently stated something like: *You can't afford to throw out 30 years of code development and start over again*. We find this statement overly generalized. First of all, we find that in most detector collaborations a large fraction of code base is new anyway; probably greater than 50% of the total. Some existing code can be reused within new code of a different language. It is an issue of inter-language communication which is handled adequately under UNIX.

In evaluating what FORTRAN code could be reused within another language environment, we have classified codes in three categories. First, any FORTRAN code whose input and output is passed entirely via its arguments is a strong candidate for reuse. Such codes are modular and thus more easily reused. The second category is FORTRAN code that communicates via COMMON blocks. Such code is suspect and its reuse depends on the number of the COMMON blocks involved, how they are initialized, where they are used, *etc*. Finally, any FORTRAN code that uses a data management system, such as ZEBRA, implies that data structures are involved, and is a strong candidate to be re-written in another language.

A final comment is that we may throw out old code, but no one is proposing to throw out 30 years of algorithms and techniques. Rather, by re-writing these concepts in a new language one is re-casting them in a new, more productive environment.

4. Code Management

An area of concern when a group of people collectively write the software base for a collaboration is code management. UNIX has a rich set of tools with which one can set up a code management environment with only a thin layer on top and a few set of rules. First of all, there are the SCCS[15] and RCS[16] packages. They handle version control, logging, and locking. The C language preprocessor, `cpp`, can be used for configuration control (*e.g.*, taking care of architecture dependencies) in conjunction with the `make` command. Finally, there are utilities like `rdist` and `rsh` to help with the code distribution.

We have built a prototype code management and distribution system based on these native UNIX tools. In the UNIX world, SCCS and RCS are well known packages. The CMS system available on VAX/VMS is similar. Compared to what HEP has used in IBM mainframes in the past, they have an impressive set of features. They maintain different versions of a file within a single file. A working copy of any version can be checked out of the system. When files are checked back in, the differences between the new version and the previous one is automatically generated and saved. Logs are kept of when changes are made, and the user is prompted to respond why a change has been made. These logs can be included in source code when they are checked out if desired. Check-out can be done with or without locking. A locker permission list can be used to control who is allowed to lock a file. Independent of the version number, a file can be set to a *state* such as "Exp," "Dev," or "Rel." The version number can be included in the source file upon checkout. This is frequently used to initialize a static string variable. A utility exists to print the value of that variable in object code, so one can verify the version of object code. Utilities also exist to show differences between versions, and to merge different versions.

We have tentatively chosen to use the RCS package. It appears to be a well thought out and mature system. It is also freely available (although not public domain) and compiles easily on most UNIX platforms. An area missing from systems such as RCS, however, is configuration control.

The C preprocessor, `cpp`, implements the source code preprocessing defined by the ANSI standard. It provides a number of facilities for code management. The first is the inclusion of other files into the source. Another utility is the macro expansion. It is used in two ways. The first is for parameters which are valid for the scope of the whole file. The macro expansion can also be used to define inline functions although it is well known that this feature should only be used with great care. The most powerful facility consists of condition directives such as

```
#if defined(aix6000)
#include <rpc/rpc.h>
#endif
```

In this example, the file is included only if the symbol `aix6000` exists. Clearly this facility is important for configuration control.

Clearly, `cpp` works well with source code in the C and C++ languages. But `cpp` can also be used with other languages, such as FORTRAN. In fact, with most UNIX FORTRAN compilers, passing the source through `cpp` is an option to the command that runs the compiler and is even automatically invoked if the source file name ends with a `.F` suffix.

An important component of the code management system is the `make` command which handles the compilation and linking steps and can be used for other things as well. Basically, one sets up a file, named `makefile` by default, in which one lists the dependencies to build a module and its set of commands. It is similar to MMS on a VAX/VMS system, but easier to use. There are also UNIX commands to help build the dependencies for inclusion in the `makefile`.

Once the `makefile` is written, the user only needs to edit, save, and type the `make` command which takes care of building a new module by compiling only that which is necessary. To a beginner, it is a bit tricky to write a `makefile`, but it is well worth taking the time to learn. Another way to use `make` is to set up some extra targets to do some code management tasks. These targets are examples of the *thin layer* on top of UNIX. For example, one could have a target that would check-in all modified files into the RCS system, or a target that would install the module into the production directories.

For code distribution, we have put together a system that integrates with the code management system. It is called Code Distribution System (CDS) and is a further example of a thin layer on top of the native UNIX. It uses the `rdist` and `rsh` commands. At the master site, the RCS files are stored along with a reference working copy. This reference copy is simply a read-only, checked-out copy of the latest version. At remote sites, nightly updates are made of the reference copy by running `rdist` at the master site. It is smart in that it checks the dates of the files at both sites, taking into account time zone differences, and sends only those files that have been updated. The user at the remote

site can force an immediate update if he or she desires. At a remote site, a user can also perform check-in and check-out operations. The user types a command on the remote computer that uses `rsh` to issue the check-in or check-out operation on the master site.

With a system like CDS, one has a distributed code development environment. The remote collaborators are equal members of the development team. We have been using the CDS system between SLAC and McGill University over the Internet since September 1991 and are currently running it to other sites as well. Even with McGill's relatively poor connection to the Internet, response time is essentially immediate.

Our plans are to use the four components, RCS, `cpp`, `make`, and CDS for our prototype code management environment. There has been no need to import code management tools from the mainframe culture or to invent our own package.

5. Graphical User Interfaces

One of the advantages of knowing that all members of the collaboration are using UNIX is that you also know that they probably have large bit-mapped display terminal. It should be possible, therefore, to build all the analysis programs with a GUI so that using them will be as easy as using programs on a popular personal computers such as the Apple Macintosh. The Reason[17] project has demonstrated analysis codes with such a GUI and their potential.

For this goal to be achieved, one will need to learn how to program to the GUI. The Reason project used the NeXTstep environment which comes with a complete set of tools and has an object oriented programming environment, so it is relatively easy to build applications. However, most UNIX workstations are based on the X-Windows environment where programming the GUI is not so easy. Fortunately, a large number of programming tools are becoming available from the third parties, as well as the computer vendors. However, choosing from amongst them can be a long and confusing process.

To help evaluate the tools and methodologies in building an application, we have set up a software benchmark. The idea is to build the same application with a different set of tools. The benchmark was made easy enough to implement so that one wouldn't spend too much time developing code that may eventually be thrown away. It has been made difficult enough, however, to be a real test of the tools. The specification of the benchmark is given in the Appendix. When the benchmark is implemented, one will have an application for viewing data in the form of n-tuples with a minimum set of interesting features.

A number of people have taken this benchmark as an opportunity for learning how to program to a GUI environment. Others have taken it to demonstrate their previously developed toolkit. The benchmark was based on two applications that had already met the specifications. The first was HippoDraw[18] which was done with NeXTstep. The other was done on a Sun with the InterViews[19] toolkit which is based on C++. Other work is being done on DECStations, RS/6000s, SGIs, Intel '486 under Windows, and Macintosh.

6. Conclusions

An organized R&D effort into all aspects of HEP computing is being conducted as part of the preparation for a possible B Factory at SLAC. Much work needs to be done and in many areas it has just started. In other areas, such as those presented in this paper, we have reached some preliminary conclusions that set the scope and scale of a software prototyping program.

The issue of programming languages is frequently discussed during the early stages of a collaboration. It seems that since the B Factory at SLAC will base all computing on UNIX, that the C language will play the major role in areas dealing with the operating system, the GUI, and networking. The use of C or C++ for the off-line programming tasks needs to be evaluated, as does FORTRAN '90.

UNIX appears to have code management tools that are sufficient to meet our needs. We need only a thin layer on top of them. The flexibility of these tools has already allowed us to set up a distributed code development environment.

Software benchmarking will be done to focus our attention on the problems and to evaluate tools and techniques. The n-tuple viewer application described in the Appendix is an example. These benchmarks will also train a body of people who will learn a lot about the UNIX environment and culture through their participation. When the time comes to make hard decisions on the various computing issues, they hopefully will be based on real experience and knowledge.

Acknowledgments

I would like to thank the many people with whom I have had discussions of the topics presented here. They include David Aston, Adam Boyarski, Alan Breakstone, David Britton, Dieter Cords, Tom Glanzman, Fred Harris, Charlotte Hee, Terry Hung, Walter Innes, Frank Porter, and Mike Ronan.

Appendix

Specifications for B Factory Software Benchmark

Build a data visualization application for data in the form of a flat table of numbers (commonly called an n-tuple data set). This tuple viewer must have the following features and capabilities:

1. Display 1-D and 2-D histogram projections of column(s) of data.
2. May use existing histogram/n-tuple package such as HBOOK4 or Hippoplotamus.
3. Must have file browser to open n-tuple file.
4. Must have data browser to select which column(s) are used in displays.
5. Must be able to point and click to select a few display options such as error bars, joining points, *etc*.
6. For 2-D histograms must have option to display as grey scale density, color density, or scatter plot.
7. Initial range of histogram display should autoscale so all data is visible.
8. Must be able to change the histogram range and number of bins either with sliders or by typing into text fields. Histogram displays must dynamically update as one drags a slider.
9. Must be able to change the n-tuple file used for the histograms without needing to redefine the histograms parameters and attributes.

References

[1] *Workshop on Physics and Detector Issues for a High-Luminosity Asymmetric B Factory at SLAC*, Report 373 (SLAC 1991).

[2] R. Brun and J. Zoll, *ZEBRA User Guide*, Long Write-up Q100 (CERN 1988).

[3] V. Blobel, *The BOS System, Dynamic Memory Management*, Report R1-88-01 (DESY 1988).

[4] A.S. Johnson, *et al.*, in *Data Structure for Particle Physics Experiments*: Proc. 14th Workshop of the INFN Eloisatron Project, Erice, 11-18 Nov. 1990 (World Scientific, Singapore 1991), p. 7.

[5] R. Bock, *et al.*, *HIGZ: High Level Interface to Graphics and ZEBRA*, Long Write-up Q120 (CERN 1988).

[6] H.J. Klein and J. Zoll, *PATCHY: Reference Manual*, (CERN 1983).

[7] M. Brun, *et al.*, *Computer Physics Commun.*, **57** (1989) 235.

[8] K. Mueller and P. Pfeifer, *Comp. Physics Commun.* **57** (1989) 239.

[9] Fermilab Computing Division, *EXPAND*, Report PU0093 (Fermilab 1990).

[10] A.S. Johnson, private communication.

[11] P.F. Kunz and G.B. Word, in *Data Structure for Particle Physics Experiments*: Proc. 14th Workshop of the INFN Eloisatron Project, Erice, 11-18 Nov. 1990 (World Scientific, Singapore 1991), p. 19.

[12] L. Lönnblad and A. Nilsson, Preprint LU TP 91-35 (Lund 1991) and DESY 91-158 (DESY 1991) (submitted to Comp. Phys. Commun.)

[13] W.B. Atwood, *et al.*, in *Proc. Computing in High Energy Physics '91*, Tsukuba, 11-15 March 1991 (Universal Academy Press 1991), p. 433.

[14] Nobu Katayama, in *Proc. Computing in High Energy Physics '91*, Tsukuba, 11-15 March 1991 (Universal Academy Press 1991), p. 439.

[15] See for example: *Programming Utilities and Libraries*, Sun Microsystems, Inc.

[16] W.F. Tichy, in *Proc. of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982.

[17] W.B. Atwood, *et al.*, *Proc of Computing for High Luminosity and High Intensity Facilities*, Santa Fe, 9-13 April 1990 (AIP 1990), p. 320.

[18] M.F. Gravina, *et al.*, *Proceeding of the Second International Workshop on Software Engineering, Artificial Intelligence and Expert Systems for High Energy and Nuclear Physics*, L'Agelonde, France, Jan. 13–18, 1992, (to be published).

[19] M.A. Linton, *et al.*, in *Proc. of the USENIX C++ Workshop*, Santa Fe, November 1987.