



Fermi National Accelerator Laboratory

FERMILAB-Conf-89/124

High Level Language Memory Management on Parallel Architectures *

P. Lebrun and A. Kreymer
Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510 U.S.A.

May 1989

* Presented by P. Lebrun at the 1989 Conference on Computing in High Energy Physics, Oxford, England, April 10-14, 1989.



Operated by Universities Research Association, Inc., under contract with the United States Department of Energy

High Level Language Memory Management on Parallel Architectures

P. Lebrun, A. Kreymer
Computing Department
Fermi National Accelerator Laboratory*
P.O. Box 500 Batavia, Illinois 60510 USA

Abstract

HEP memory management packages such as YBOS and ZEBRA have been implemented and are currently running on a variety of mainframe computers. These packages were originally designed to run on single CPU engines. Implementation of these packages on parallel machines, loosely or tightly coupled architectures is discussed. ZEBRA (CERN package) on ACP (Fermilab) is presented in detail. Design of memory management system for the new generation of ACP systems or similar parallel architectures are presented. The future of packages such as ZEBRA is not only linked to system architecture, but also to languages issues. We briefly mention penalties in using F77 with respect to other increasingly popular languages in HEP, such as C, on parallel systems.

INTRODUCTION

Years ago, the need for "pointer based" FORTRAN packages, such as HYDRA or BOS, later on ZBOOK [1], ZEBRA [2] or YBOS, became a necessity to efficiently manage the user heap space on a tight fixed size physical memory, in order to support large applications, such as code management systems or histogram packages.[1] Later on, these memory management systems were used not only to allocate dynamically space in a fixed size heap space, but also as a tool to organize and manage sensibly a complicate set of structures describing a detector, or an elementary particle collision. These packages are the essential building blocks for HEP data bases. They were designed to run mainly on single CPU systems and ignored entirely the existence of virtual memory available within FORTRAN through - system dependent ! - system calls.

Because of the increasing demand on memory size and CPU load, advanced hardware architectures - parallel or vector machine - must be used in HEP analysis. . As a consequence, current FORTRAN based memory management systems such as YBOS or ZEBRA had to be implemented on machines they were not originally designed for. This paper describes some of the issues in doing these implementations, on parallel architectures, and discuss design rules for future systems (not necessarily using F77 as the basic programming language). Emphasis can not be placed only on high level software considerations, since high performance can only be achieved on these systems if the user understands and controls the parallelism, i.e., if he is able to map the hardware architecture to his particular problem.

* Operated by the Universities Research Association Inc, under contract with the U.S. Department of Energy.

This paper is organized as follows : We start by reviewing some of the relevant concepts used in describing parallel architectures. We then proceed to describe how the high level application deals with memory transfers on parallel systems. We then discuss in more detail the implementation of ZEBRA on ACP . We stress the inherent limitation of F77, with respect to better suited language for memory management, such as C, which is now widely and cheaply available.

Relevant concept on parallel architectures.

a. Concept of distributed memory [3]:

A physical memory unit can be a simple data register in a CPU, or a first level cache, a second level cache (if any), the main memory itself and finally a disk . Of course, since we are describing systems with many CPU's, there will be more than one cache, and the "main memory" can refer either to the memory of a single node, or alternatively, to a single central memory shared by all the nodes in the system. These devices are connected to each other through a - sometimes intricate - succession of Bus. From a software point of view, a data structure can be as elementary as a bit, a word, a "bank" or a F77 array, a structure of banks or arrays, or a complete data base. The key issue is to keep track of these logical constructs in the machine and be aware when multiple copies of the same structure start diverging from each other. Indeed, it is usually more efficient to keep multiple copies of the same information at different physical locations, in order to access this information concurrently from different processors, to avoid bus or memory contention.

b. Private (or local) versus shared memory :

When we refer to the "private" memory with respect to a given processor, we simply expect to be able to READ/WRITE from or to this memory without the other processor's (or other Bus controller's) permission or knowledge. A memory is shared if more than one processor has WRITE access to this memory.

c. "granularity" of the parallelism : tightly versus loosely coupled systems

If the local memory is small, for instance, if it simply consists of a limited size fast cache, the CPU is likely to "miss" that cache frequently, causing lots of small transfers across the Bus. Such systems are called tightly coupled. On the contrary, if the local memory is as big as the central, shared memory, the entire application can run "locally" and the transfers will be "rare" and "organized".

In general, tightly coupled systems imply a "fine grained" parallelism, where small tasks are executed concurrently, for instance, at level of F77 loops.. Loosely coupled systems can be easily programmed if the parallelism is "coarse" i.e., when the tasks running concurrently are time consuming with respect to the time taken by a single transfer from shared memory to local memory.

Because of the complexity involved in managing a large number of micro-tasks running concurrently, fine grained parallelism is usually implemented at the compiler level, or, alternatively, the user must often completely rewrite his application in a non standard "parallel FORTRAN" [4]

Although we do not intend to describe in detail fine grained parallelism, we would like to stress the crucial difference between genuine FORTRAN and a ZEBRA based application with respect to parallel FORTRAN dialects. Let us consider the following application:

```

.....
COMMON /CALIB/HA(20),HB(20)
COMMON /TRACK/XYZ(3,10), VAL(10,20)
----
DO 1 I = 1,10
DO 2 J = 1,20
    VAL(I,J) = HA(J)*HB(J)*SQRT(XYZ(1,I)**2 + XYZ(2,I)**2)
-----

```

These loops can be executed concurrently. There are absolutely no dependencies involved. A good parallel compiler will detect this without programmer help, because the arrays involved are clearly and unambiguously distinct from each other.

Unfortunately, many of us do think that managing hundreds of little distinct and unstructured F77 COMMON BLOCKS in a big application leads to considerable difficulties. These data structures are more elegantly maintained in a single COMMON, with the help of ZEBRA. This same code becomes [2]

```

.....
COMMON /ZZANAL/ Q(100000)
DIMENSION IQ(1), LQ(1)
EQUIVALENCE (Q(1), IQ(1))
EQUIVALENCE (IQ(1), LQ(9))
.....
COMMON /LNKANA/ JCALIB, JTRACK
----
DO 1 I = 1,10
    JLI = LQ(JTRACK-I)
DO 2 J = 1,20
    JLJ = LQ(JCALIB -J)
    Q(JLI+30+J) = Q(JLJ+1)*Q(JLJ+11)*
&    SQRT(Q(JLI+I)**2 + Q(JLI+10+I)**2)

```

Here is the problem : the user is of course aware that the pointer JCALIB and JTRACK have distinct values, as well as the sub-bank indexes JLI and JLJ. Unfortunately, the parallel compiler must be conservative, and assume these pointers - which are nothing but indexes to the Q array - can take equal or "dependent" values, and there must start to either give up on implementing concurrent sub-tasks or install an intricate sets of locks to ensure that data won't be overwritten. This is not achievable in general, and "automatic" parallelism is doomed ... Thus, the programmer must rewrite the code, and define the inner loop instruction as a "worker" or "micro-tasks" specific routine[5].

This simple example also clearly demonstrates why automatic parallel FORTRAN compiler do exist for some commercial machines, but are not available - or not really worth writing - for the C language : with pointer based language, one only knows at run time where the relevant data is located; unfortunately one needs this information at compile time to generate the appropriate code. Thus, if pointer based language is used (FORTRAN/ZEBRA, F8X or C), the parallelism must be explicit. The user must have control over concurrency and location of his data. If so, life is certainly easier if the concurrent tasks are as big as they can be, to avoid a large amount of micro memory management . That explains why efficient use of ZEBRA on ACP (a very loosely coupled architecture) is possible, and has been achieved with no rewriting of the core of ZEBRA.

Case Study : ZEBRA on ACP (I)

The ACP (generation I) [6] is made of a host computer (a VAX or a micro-Vax) and nodes (M68020 VME boards, their number ranges from 1 to 100) performing the bulk of the CPU intensive operations. The host controls the I/O and acts as a master with respect to these nodes. The local memory on the nodes is large (2 Mbyte or 6Mbyte), and can accommodate large applications, such as the complete GEANT3 code.[7] Thus, the parallelism is coarse, always explicit, and is achieved by invoking F77 callable functions from the ACPSYS software package [8]. The implementation of ZEBRA on ACP (generation I) consists of:

1. Install the memory manager MZ package, the ZEBRA debugger DZ package, and the core of the I/O package FZ. Note that the node is not able to service interrupts, but FZ can write a complete, self consistent buffer into a dedicated section of the local memory. This buffer can be fetched from the host upon termination of an event.

2. Provide communication tools between the host (and I/O devices connected to it) and the nodes, in order to transfer complete data structures from or to the nodes. In order to avoid complex - and therefore slow - host/node communications, only a complete store (an entire F77 COMMON BLOCK managed by ZEBRA) or an entire DIVISION (a contiguous section of a store) can be carried from/to the nodes. If a particular substructure has been received (or sent) to the nodes, the user must use the FZ internal memory I/O to a dedicated set of buffers, which themselves are contiguous in real memory on the node.

A complete description of these host routines is available from the Fermilab Computer Department library [9] We briefly describe the following modules :

- * INIT_ZEBRA_STORE : Install a ZEBRA store on the nodes. Note that all these store copies do have exactly the same size, and that the system division is broadcast to all nodes. This routine must be called prior to bank manipulation on the node.

- * BROADCAST DIVISION : Users DIVISION can be broadcast at the beginning of the job, before creation or deletion of structures on the node.

- * SEND/GET DIVISIONS : During the event analysis phase, entire DIVISIONS can be sent/received from generic nodes. Note that the location of these DIVISIONS in the store are different on every node.

- * READ_SEND and GET_WRITE FZ buffers : the host can READ or WRITE entire internal FZ buffers, in order to manage very efficiently the I/O in such a complex environment.

This package has been used by many groups, mostly at Fermilab. It allowed us to run the HBOOK4 histogram package and the Monte-Carlo package GEANT3 on the ACP. This installation was straightforward, we encountered no major difficulty. However :

1. Because ZEBRA accesses user data internally by doing its own mapping of the memory across F77 COMMON BLOCKS boundaries, ZEBRA does not respect F77 arrays bound checks. The ABSOFT compiler had to be modified to allow "32 bit addressing range" for every memory access, even if the array is "short" and needs only 16 bit arithmetic in computing indexes. This is a very unpleasant feature of ZEBRA. Also, ZEBRA has implicit store to store dependencies or communication paths: the heap space in the primary store can be used by an other store, causing link relocation in the primary store, without explicit warning to the user.

2. There is no way to enforce or check the variable type declaration in a ZEBRA store, causing possible confusion if structures are moved across machines with different data representations (such as VAX and M68020). Such variable declaration should be made mandatory. Unfortunately, because of the F77 EQUIVALENCE, the user will always be able to misrepresent data. Nevertheless, tools could be provided, such as checks for legal floating point data if the bank is real or double precision. This problem is specific to F77, and can be alleviated by using C.

3. ZEBRA has very flexible data structures, allowing very powerful organization of the data. Such flexibility has a price : link relocation during garbage collection becomes very complex, especially with links pointing outside the DIVISION being reorganized. These reference links can point to physical memory on other processors. For the sake of simplicity, such links were ignored in this implementation, which only "contained" DIVISIONS that are supported.

CONCLUSION

*High level memory management packages are currently running efficiently on parallel machines, if memory and processors are loosely coupled and if the parallelism is explicit, controlled by the user. On a tightly coupled architecture, efficient management becomes much harder.

*These implementations (YBOS, ZEBRA) are no longer laboratory exercises, they became an essential software tool to analyze HEP data at Fermilab (CDF,E706,E705,E772..)

*Correct design of the data structure type and organisation matters more than the language used in coding such a package. Although C- or F8X - are language more suitable for memory organization, the crucial concept is data dependancies among pointer and structures, not the language by itself. Nevertheless, fixed size COMMON BLOCKS without dynamical memory capability is a pretty big penalty for F77. Hopefully, F77 absolute supremacy will not dictate our software environment for ever...

Literature cited.

1. R. Bock, E.Pagiola, J. Zoll et al., HYDRA Topical Manual, CERN Program Library.

R. Brun, F. Carena, H. Grote, M. Hansroul, J. Lasalle, W. Wojcik, ZBOOK User Guide and Reference Manual, CERN Program Library.

2. R. Brun, R. Goosens and J. Zoll, ZEBRA user's Guide , CERN Program Library DD/EE/85.6 (1987)

D. Quarrie and B. Troemel, YBOS programmers Reference Manual, CDF Compt. Group, FERMILAB (1988).

3. S. Otto, Shared Store versus Message Passing - halftime score, to be published in the proceeding of this conference.

4. Alan H. Karp, Programming for Parallelism, Computer, May 1987 pp. 457

5. Alan H. Karp, Robert G. Babb II, A comparison of 12 Parallel FORTRAN Dialects, IEEE software, Vol 9, P 52 (1988).

6. Donaldson, R., Kreisler M. N., eds Proc. Symp. on Rec. Dev. in computing , Processor and Software Res. for High Energy Phys., Guanajuato, Mexico, 1984, Batavia, Ill, Fermilab.

7. P. Lebrun, eds. Workshop on detector simulation for the SSC. Anl-HEP-CP-88-51, Argonne National Laboratory, Ill.

8. I. Gaines et al, ACP Software User's Guide for event oriented processing, Computer Department library, Fermilab.

9. P. Lebrun et al, ZEBRA on ACP, Computer Department Library,