Fermi National Accelerator Laboratory

FERMILAB-Conf-84/64
2380.000

SOFTWARE FOR EVENT ORIENTED PROCESSING ON MULTIPROCESSOR SYSTEMS*

M. Fischler, H. Areti, J. Biel, S. Bracker,
G. Case, I. Gaines, D. Husby, and T. Nash

August 1984

*Presented at the Symposium on Recent Developments in Computing, Processor, and Software Research for High-Energy Physics, Guanajuato, Mexico, May 8-11, 1984

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

SOFTWARE FOR EVENT ORIENTED PROCESSING ON MULTIPROCESSOR SYSTEMS

M. Fischler, H. Areti, J. Biel, S. Bracker, G. Case,
I. Gaines, D. Husby, and T. Nash

Advanced Computer Program
Fermi National Accelerator Laboratory
Batavia, Illinois 60510

ABSTRACT

Computing intensive problems that require the processing of numerous essentially independent events are natural customers for large scale multi-microprocessor systems. This paper describes the software required to support users with such problems in a multiprocessor environment. It is based on experience with and development work aimed at processing very large amounts of high energy physics data.

Introduction

We describe here the support and system software that has been developed by the Fermilab Advanced Computer Program (ACP) for users with event oriented problems to be run on ACP multiprocessor computers.[1] Supporting a system of over one hundred individual processors requires a set of efficient, flexible, and simple routines that control the movement of data within the system. To meet these requirements, the routines must be designed with particular types of applications in mind. Specifically, the software described here supports event oriented applications where the problem is naturally divided into a process requiring a single intelligence (such as reading tapes, forming statistics, etc.), and a process done once for each of many hundreds of events, which uses most of the CPU time. In high energy physics event reconstruction, the second process is the actual reconstruction procedure, which does no I/O and which takes at least 99% of the CPU time in most cases.

We are dealing here with a software structure in which a host program feeds events down to a node program, which is replicated on many node processors. The user must provide the information as to how the particular program is to be split into a part to be run on the host and a part for the node. However, the user does not have to explicitly control the detailed logic of data transmission such as deciding which node is finished and should receive the next event. Rather, a set of FORTRAN callable subroutines is provided which are simple to learn to use and which handle the transmission of events to and from the nodes. FORTRAN was chosen because of the strong commitment to FORTRAN in the physics community and the large collection of existing FORTRAN code.

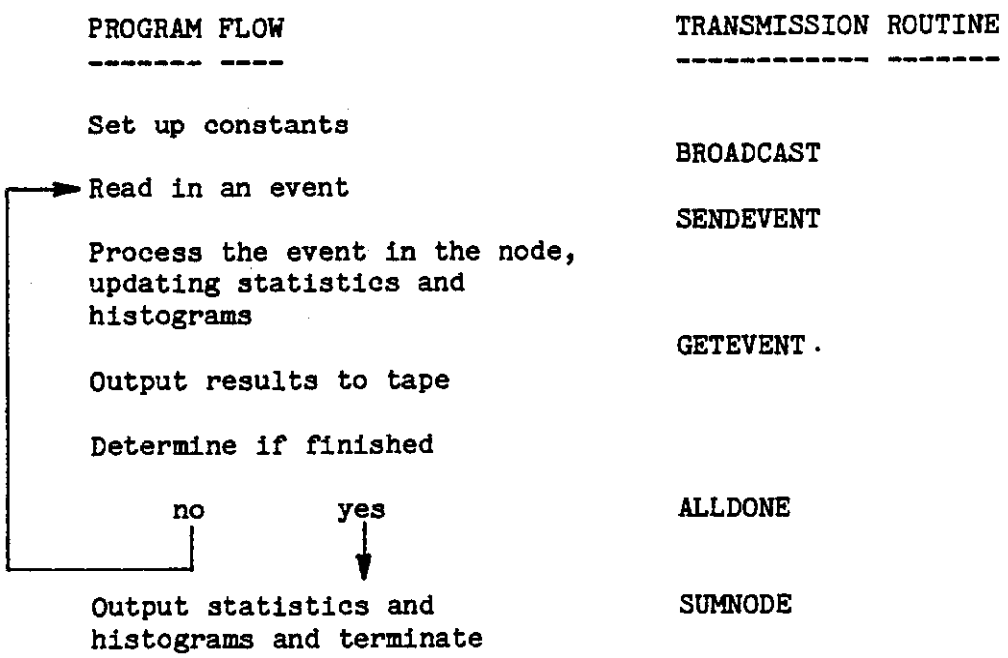
The concepts described here apply to any problem which is separable into many subproblems requiring little intercommunication. These include many lattice gauge theory applications, high dimensional integrals, tracking of particle trajectories through proposed accelerators, as well as problems outside physics such as animation and even (surprisingly) finite element calculations.[2] The system architecture that is natural for such problems is a host computer attached to the I/O devices and to a bus on which many node processors reside.

ACP Support Software

The ACP Support Software allows three modes for transmitting data between host and nodes: Constant Broadcast Mode, in which data, such as calibration constants, calculated by the host is loaded into all the nodes; Event Processing Mode, in which individual event data is loaded by the host into a single node, and the results collected by the host from the nodes as they finish; and Statistics Collection Mode, in which the host retrieves and accumulates results from all the nodes.

A typical program will have the flow shown in Figure 1.

Figure 1



The transmission routines shown in the figure have the following functions:

BROADCAST	Broadcasts constants to all nodes
SENDEVENT	Sends the event to a node
GETEVENT	Retrieves the result from a node
ALLDONE	Will see if all the nodes are finished
SUMNODE	Collects and accumulates statistics from all nodes

In Figure 2, on the following page, is an example that illustrates how a program is modified to take advantage of the routines provided. Note that about a dozen statements have been added to a program which is typically many thousands of steps long.

The user support routines have been gathered into three layers of increasing complexity. This way a user needs to be familiar with only the simplest possible set of routines. A set of Layer 1 routines, including those referred to in Figure 1 will satisfy the needs of many users with programs having basic and simple requirements. Other users may need somewhat more flexibility. Layer 2 routines allow more choices for the programmer (and have, therefore, somewhat longer descriptions to absorb). The Layer 2 routines that correspond to the Layer 1 routines listed above are:

GBROADCAST	Generalized broadcast for inhomogeneous arrays.
SENDBLOCK	Allows multiple block transmissions to nodes of one or more classes.
GETBLOCK	Retrieves multiple blocks of data from nodes of one or more classes.
CHECKNODE	Examines the complete status of nodes.
GSUMNODE	Generalized collection of inhomogeneous data with variable accumulation rules.

For example, if it is necessary to transmit multiple blocks of data for each event, calls to the Layer 2 routine SENDBLOCK replaces the single call to SENDEVENT. Particularly sophisticated users will be able to use Layer 3 routines for direct control of the traffic on the global bus, without having to write their own device drivers or system calls.

Documentation is provided in a complete and extensive "Software User's Guide."[3] A sample of a Layer 1 subroutine description from this guide appears in Figure 3.

Figure 2

Illustration of Modifications Required in User Program Running in Host

```
          ORIGINAL CODE                          MODIFIED CODE
          -------- ----                          -------- ----
1         COMMON/RAW/DAT(20000)                  COMMON/RAW/DAT(20000)
2         COMMON/ANSWER/RESULTS(10000)           COMMON/ANSWER/RESULTS(10000)
3         COMMON/CALIB/C1(100), C2(100)          COMMON/CALIB/C1(100), C2(100)
4         COMMON/STATS/HIST(10000)               COMMON/STATS/HIST(10000)
5         INTEGER DAT, RESULTS                   INTEGER DAT, RESULTS
6                                                LOGICAL LASTEVENT, SEND_DONE,
                                                   ALLDONE, GET_DONE
7                                                INCLUDE '[ACP]ACPUSER.INC'
      C   INPUT, SETUP, CONSTANTS           C     INPUT, SETUP, CONSTANTS
8                                                CALL ACPINIT
9         . . .                                  . . .
10        . . .                                  . . .
11        . . .                                  . . .
                                           C     BROADCAST CONSTANTS TO NODES
12                                               CALL BROADCAST (3, C1, 200, REAL_4)
                                           C     INITIALIZE NODE STATUS VARIABLES
13                                               SEND_DONE = .TRUE.
14                                               GET_DONE = .FALSE.
15                                               LASTEVENT = .FALSE.

      C   EVENT LOOP START                 C     EVENT LOOP START
          .                                      .
          .                                      .
          .                                      .
16      10 CONTINUE                            10 CONTINUE
17                                               IF(.NOT.LASTEVENT.AND.SEND_DONE)THEN
18          CALL READEVENT                         CALL READEVENT
19          IF (ENDOFTAPE) GO TO 20                IF(ENDOFTAPE)LASTEVENT=.TRUE.
20                                               ENDIF

      C   PROCESS EVENT                     C     PROCESS EVENT
21        CALL PROCESS                           IF(.NOT.LASTEVENT)
                                                   CALL SENDEVENT(DAT,20000,SEND_DONE)
22                                               IF(LASTEVENT)THEN
23                                                 IF(ALLDONE(NODE))GO TO 20
24                                               END IF
25                                               CALL GETEVENT(RESULTS,10000,GET_DONE)
26        CALL WRITEEVENT                        IF(GET_DONE)CALL WRITEEVENT

      C   EVENT LOOP END                   C     EVENT LOOP END
27        GO TO 10                               GO TO 10
28      20 CONTINUE                            20 CONTINUE

      C   OUTPUT HISTOGRAMS, ETC.          C     OUTPUT HISTOGRAMS, ETC.
29                                               CALL SUMNODE(4, HIST, 10000, REAL_4)
30        CALL HISTDO                            CALL HISTDO
31        . . .                                  . . .
32        . . .                                  . . .
33        . . .                                  . . .
34        END                                    END
```

Figure 3

SENDEVENT SENDEVENT

SENDEVENT passes a block of data to the first available node and starts that node running. SENDEVENT passes data to block number 1. The data will be passed, unconverted, as 32 bit binary words. (At user option a global parameter can be set at compile time to cause the routine to pass down data as unconverted 16 bit binary words.)

CALL SENDEVENT (ARRAY, LENGTH, SEND_DONE) is equivalent to the following Layer 2 call:

CALL SENDBLOCK(ARRAY,LENGTH,block_number=1,ANY_NODE,ALL_CLASSES,GO).

SENDEVENT (ARRAY, LENGTH, SEND_DONE) Layer 1

Arguments:

Input only:	ARRAY, LENGTH
Input/result:	---
Result only:	SEND_DONE

Return Variables: RETURN_STATUS

ARRAY:

The first word in an array or block of data available to the calling program on which the subroutine will act. This must be the first variable in the common block to be passed to the node.

LENGTH:

In standard usage, this is an integer scalar with the number of 32 bit words of data to be transmitted. Note that a double precision variable is two 32 bit pieces of data, and that a pair of 16 bit integers is a single 32 bit piece. LENGTH is an integer greater than zero, except in GETEVENT and GETBLOCK where LENGTH.LE.0 signifies variable length transmission.

SEND_DONE:

This is a logical variable, returned as .TRUE. if an available finished node was found, and as .FALSE. otherwise. This must be declared a LOGICAL variable in the host program.

The following are return variables available in COMMON/ACPUSER/:

RETURN_STATUS:

An integer variable that is returned to indicate the status of the subroutine's activity. For details see the section entitled, "Reserved Name Parameters and Return Status Variables."

Error Handling

Error handling for a multiprocessor can and should be more sophisticated than on a uniprocessor. Upon detecting an error, the ACP support system provides a description of the error and where it occurred. It also makes available, at user option, a memory dump of the faulting node. A third output is needed in multiprocessing environments which is not needed in uniprocessor computers. Since each node processes a different sequence of events, it is necessary to maintain a history file of which events a processor has done previously. This is made available when an error is detected so that a diagnosis can be made on the development system using utilities that automatically reproduce the errant node's history. The support software on a production multiprocessor only provides information about an error. Analysis of why the error occurred is done on a separate development system (described below) since time on the many-node production system is likely to be at a premium.

The user can specify one of the following levels of action to be taken on detection of an error: ignore the error; print a warning, but continue running; kill the node statistics, but continue running; excise the offending node from the system for the remainder of the run; or abort the run immediately.

Error detection falls in five categories: hardware failures detected by automatic bus, node, and hardware diagnostics; node software errors (divide by zero, etc.); node time out; user defined exceptions; verification exceptions. The last one is available only in a system with multiple nodes, and is a new type of error detection which can be very helpful. The same event is sent to two nodes and the results compared; we call this "verification." Verification enables the system to catch rare software "time bombs," where a logic error in the program running one event causes some area to become invalid, but the invalid area is not used until many events later. Verification will also catch infrequent non-fatal (soft) hardware errors, and enable the studying of soft error rates. This can be inconvenient to do on uniprocessor systems.

The Development System

The development system is available for writing and testing new programs as well as for analysis of errors detected in the production environment. This system will consist of a host and a few samples of each type of node that exist on production systems. The host, a commercial super-mini (VAX or similar), has compilers, a symbolic interactive debugger, file handling, editing, and all the other features of such a computer. The nodes have a node compiler and a node symbolic debugger, when available; otherwise, a cross-compiler for the node is supported on the host. (The production nodes have only a stripped-down "operating system" called the Tight Loop Monitor, which waits for the host to tell them an event is ready to be run and jumps to a program which had been downloaded over the global bus.)

Additional software is provided on the development system to support error analysis. This includes a convenient way to work through the node memory dump available when errors occur, a facility for reconstructing a

particular sequence of events from the history file to duplicate the conditions under which errors had occurred, and support for I/O directly from the node.

There is also a set of quick and simple automatic procedures for users to compile and link the node programs forming node executable images and download node programs under the control of the host FORTRAN program. These are available for use in both the development and production systems. They allow the user to select options both at compile and run time concerning how the system is to behave when handling errors, passing data to nodes, etc.

In conclusion, Fermilab's ACP has developed, in addition to the hardware, user friendly software for using its multiprocessor systems. The same software concepts can be employed over a broad range of problems, and with various implementations of the hardware architecture.

References

1. I. Gaines, et al. "Fermilab's Advanced Computer Program," Proceedings, this conference, FERMILAB-Conf-84/63. Thomas Nash, et al. "Fermilab's Advanced Computer Reseach and Development Program," Proceedings, Three Day In-Depth Review on the Impact of Specialized Processors in Elementary Particle Physics," Padova, 1983, p. 227. Hari Areti, et al. "ACP Modular Processing System: Design Specifications," Rev. April 2, 1984, FN-402. See also Reference 3.

2. John A. Swanson, talk at Forefronts of Computing Conference, NBS, Gaithersberg, Maryland, June 25-27, 1984.

3. Advanced Computer Program, "ACP Software User's Guide for Event Oriented Processing," Rev. June 18, 1984, FN-403.