ORNL/TM-9971

ornl

OAK RIDGE NATIONAL LABORATORY

MARTIN MARIETTA

Denelcor HEP Multiprocessor Simulator

T. H. Dunigan

OPERATED BY
MARTIN MARIETTA ENERGY SYSTEMS, INC.
FOR THE UNITED STATES
DEPARTMENT OF ENERGY

Printed in the United States of America. Available from
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road, Springfield, Virginia 22161
NTIS price codes—Printed Copy: A03; Microfiche A01

ORNL/TM-9971

Engineering Physics and Mathematics Division

Mathematical Sciences Section

DENELCOR HEP MULTIPROCESSOR SIMULATOR

T. H. Dunigan

Date Published - June 1986

The work was supported by the
Applied Mathematical Sciences subprogram
of the Office of Energy Research,
U. S. Department of Energy

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
operated by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-84OR21400

Table of Contents

Denelcor HEP Multiprocessor Simulator

T. H. Dunigan

Mathematical Sciences Section
Engineering Physics and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831

ABSTRACT

The structure and use of a simulator for the Denelcor HEP multiprocessor are described. The simulator provides a multitasking environment for the development of parallel programs in C or FORTRAN using a library of subroutines that simulate the parallel programming constructs available on the HEP, a shared-memory multiprocessor. The simulator also provides a trace file that can be used for debugging, performance analysis, or graphical display.

1. Overview

1.1. Introduction

Simulation of parallel processors has several important uses. First, simulation provides the computing system designer with a test bed to evaluate various parallel architectures or design decisions within a given architecture. Second, users who cannot afford a parallel computing system can develop programs on a simulator. Third, even with a parallel system available, the user may find that the simulator provides more debugging aids and performance information than the actual hardware. Finally, the user may wish to investigate new algorithms or test existing applications on proposed or theoretical architectures available only through simulation.

The objective of the simulator described in this report is to provide an environment for developing algorithms and applications for the Denelcor HEP. Although we had access to HEP systems at other sites, access was not always simple; and the machine was not always available. It was decided that a simulator would reduce program development time and provide more debugging and performance information. (Several months after the simulator was developed, Denelcor went out of business. However, there is still interest in the HEP architecture, and there are codes written for the machine. Simulation is presently the only convenient access to the HEP architecture.) In §1.2, the history and structure of the simulator are summarized. In §1.3 the architecture of the HEP is described. Section 2 is a guide to the use of the simulator with examples and sample sessions for both C and FORTRAN.

1.2. Simulator structure

The simulator is a library of subroutines for C or FORTRAN that provides HEP task and shared-memory management services. It is based on a multiprocessor simulator, the "Multitasker," developed by Brooks [1] and extended by Dunigan [3]. The simulator runs

as a single process on a Digital Equipment Corporation VAX 11/780 under the control of Berkeley's UNIX* 4.2bsd. The distribution tape (described in [3]) provides include files, makefiles, sample programs and scripts to aid in constructing an application to run under the simulator. An application may consist of up to 1000 processes or tasks that will be scheduled for "concurrent" execution by the simulator.

A set of subroutines (*tfork*, *texit*, and *twait*) provides task management services. Application subroutines that are to be executed in parallel are declared to be of type TASK; then *tfork* is used to start such subroutines in parallel. The *tfork* call is equivalent to the HEP *create* subroutine. The main entry point for the application is contained in the simulator library, so the user's "main" program is replaced by a subroutine of type TASK with the name *task0*. The simulator passes control to *task0* when the application program is started. The application can then create other parallel processes with subsequent calls of *tfork* specifying the task name, stack size, and, optionally, any arguments the task uses. The HEP shared-memory services (*set*, *inc*, *write*, *read*, *wait*, *empty*, and *full*) can then be used to synchronize access to common memory locations. An optional trace facility can be enabled to provide a history of simulator events.

Within a single UNIX process, the simulator provides a small operating system that schedules the execution of the user's application tasks. Two forms of scheduling services are provided. The simplest form is a non-preemptive scheduler. A simulator task runs until it must wait for some simulator event (for a variable to become "full", for example). When the task blocks, the next runnable simulator task is started, selected in a simple round-robin fashion. This mode requires very little simulator overhead and the application runs at normal VAX speed. A more complex, preemptive scheduling mode is available when the application is built in *aspp*-mode.

In *aspp*-mode, the application code is in effect interpreted, permitting the execution of each active simulator subtask to be interleaved. At program build time, the *aspp* module (assembler post processor) inserts calls to the simulator scheduler between each assembler instruction in the application program. The simulator then can provide concurrent execution of the application's tasks and maintain a clock (in units of VAX instructions). The clock, in turn, can be used to schedule asynchronous events such as task sleeps (*tsleep*) and can time-stamp entries in the trace file for detailed performance analyses. However, the real run time of the application will be lengthened considerably due to the additional overhead incurred.

1.3. HEP architecture

To support our local research efforts, the Brooks simulator was extended with additional subroutines to simulate the Denelcor HEP multiprocessor as well as message-passing multiprocessors [3]. The HEP (Heterogeneous Element Processor) was the first commercially available multi-instruction, multi-data stream (MIMD) parallel processor system. The system consists of up to 16 process execution modules (PEM) and up to 128 memory modules. The PEMs and memory modules are connected through a packet-switch network[7]. Any PEM can access any word of data memory through the switch.

Each PEM is designed to execute multiple independent instruction streams on multiple data streams simultaneously. A PEM is a register-to-register processor with multiple functional units pipelined with an eight-stage pipe. By providing multiple independent data and instruction streams, maximum parallelism may be achieved. While an "add" is in progress for one process, a "multiply" may be executing for another, a "divide" for a third. Since the instruction streams are independent, there are no dependencies to slow the

*UNIX is a trademark of AT&T.

pipeline. However, a single process does not achieve any speedup from the pipelining as is the case for some pipelined single-instruction stream systems. Approximately 10 processes are required per PEM to fill the pipeline and achieve the maximum execution rate. Speedup curves ([2] and [6]) flatten when the pipeline is filled.

Synchronization of cooperating processes is implemented by access to shared memory. Associated with every memory word and register is an access state that is either FULL or EMPTY. By default, the access state is ignored on all memory accesses. However, a set of subroutines are provided to test and set the access state of a memory location. though the underlying implementation is performed in hardware. When an application utilizes the access state of a memory location, a process will be re-queued in the instruction pipeline if it attempts to read from a location that is EMPTY. The instruction will not succeed until another process sets the location FULL. Similarly, the process is re-queued if it attempts to write to a location that is FULL. If one writes to an EMPTY location, the access state is changed to FULL, and a read from a FULL location changes the state to EMPTY.

The simulation models only the HEP programming environment, reflecting only the synchronization architecture associated with the access states of memory. There is no simulation of memory contention or memory switch latency. Pipeline saturation is not simulated, rather, the simulation behaves as if there were a processor available for every process.

2. User's Guide

2.1. Simulator subroutines

The simulator subroutines can be divided into three basic services — task control services (*tfork*, *texit*), information services (*strace*, *etrace*, *mark*), and HEP synchronization primitives (*read*, *write*, *inc*, *set*, *waitf*, *empty*, *full*). Appendix C summarizes these subroutines and supplemental information is available from [1], [3] and [4]. This section will illustrate how to use these services to construct parallel programs.

Rather than having a "main" procedure, a simulator application consists of a number of subroutines of type TASK that may be executed concurrently by the simulator scheduler. The "main" task has the name *task0* and may start other subroutines of type TASK with *tfork*. The first argument to *tfork* is the name of the subroutine, and the second argument specifies the number of four-byte words to be used for the stack and automatic storage for the task or process. The minimum stack size is 10,000 and may be larger if the task or subroutines it calls have large storage requirements for local variables (such as arrays). Tasks exit when they execute a *return* or *texit* or encounter the "end" of the subroutine.

Synchronization of the parallel processes created by *tfork* is controlled by a set of subroutines that manipulate the access state (FULL/EMPTY) associated with every word of HEP memory. (In the early HEP implementation, the access state of a location was provided through an extended FORTRAN notation denoting the "asynchronous variables" with a leading "$.") The simulator provides a set of complimentary subroutines for both *int* (INTEGER) and *float* (REAL*4) variables. In the following, only the *int* functions are described. Appendix A describes the calling sequence for all of the simulator subroutines. The C programmer should note that the asynchronous variables are passed by reference (&) to the HEP subroutines.

The synchronization variables (asynchronous variables) manipulated by the simulator subroutines and the HEP should be global variables or in COMMON. They should be initialized with *isete*, which sets the access state of the variable to EMPTY, or *iaset*, which stores the second argument of the function in the variable (first argument) and sets the

access state to FULL. The access state of a variable may be tested with *empty*, which returns 1 if the access state of the variable is EMPTY otherwise it returns 0, or *full*, which returns 1 if the access state of the variable is FULL otherwise it returns 0. The functions *empty* and *full* do not alter the access state of the variable.

Process synchronization can be accomplished with calls to *iaread*, *iawrite*, *iainc*, or *iwaitf*. *Iaread* blocks the process until the access state of the variable becomes FULL; it then returns the value of the variable and sets its state to EMPTY. *Iawrite* waits until the access state of the variable is EMPTY, then assigns the variable the given value and the access state is set to FULL. *Iainc* blocks the process until the access state of the variable is FULL, then increments the variable by the given value leaving the variable FULL. The function returns the value of the variable before the increment. *Iwaitf* blocks the process until the access state of the variable is FULL, then returns the value of the variable. The access state remains FULL. These functions can be used to construct critical sections to control the access to other shared variables by cooperating processes.

Figure 1 illustrates the use of some of the simulator routines in a contrived example. The program calculates the inner product of a matrix (*matrix*) with a vector (*vector*) and prints the resulting vector (*result*). The result is calculated in parallel by creating self-scheduling processes to perform the vector products. The host process (*task0*) starts the trace file, initializes the synchronization variables, and initiates several processes that will execute the subroutine *mult* concurrently. The host process then executes the subroutine *mult* and awaits the completion of all of the subprocesses. The synchronization variable *done* will be FULL when all of the columns have been multiplied. The host process then prints the resulting vector.

Rather than statically assigning columns of the matrix to specific processes, the sample program utilizes self-scheduling. The process *mult* tests a shared column pointer k to determine if there is still another column to multiply. If there is, k is incremented and the k*th* column multiplied. If no columns remain, *active* is decremented, and the last process to finish sets *done* so the host process may proceed. More substantial examples of HEP applications can be found in [2] and [6].

```
/* hepip.c  vector matrix inner product using hep self-schedule */
#include <stdio.h>
#include <hep.h>
#define STACK 10000
TASK mult();  /* the parallel subroutine */
#define DIM 5
#define CPUs 3
int vector[DIM] = {2,3,2,4,3};
int matrix[DIM][DIM] = {1,2,3,0,4, 2,3,3,1,3, 3,1,1,1,2, 4,0,2,0,1, 5,4,2,3,5};
int result[DIM];
struct Shared {
        int done;
        int k;          /* next subscript */
        int active;     /* active cpus   */
} shared;
TASK task0()
{               /* main task */
        int i;
        int val;
        int type,lth,node,pid;

        strace("hepip.trace");
        isete(&shared.done);    /* barrier */
        iaset(&shared.k,0);          /* initial subscript */
        iaset(&shared.active,CPUs+1);
        for (i=0;i<CPUs;i++)
                tfork(mult,STACK);  /* start subroutine in parallel */
        mult();                          /* let main do some work too */
        iaread(&shared.done);        /* barrier til done  */
        for(i=0;i<DIM;i++) printf(" %d",result[i]); printf("0);
}

TASK mult()
{
        /* do inner product of two vectors */
        int i, sum;
        int k, active;  /* local copies */
        for(;;){   /* keep working til done */
                k = iaread(&shared.k);
                iawrite(&shared.k, k+1);
                if (k >= DIM){   /* no more so exit */
                        active = iaread(&shared.active) - 1;
                        iawrite(&shared.active,active); /* reduce active */
                        if (active == 0 ) iawrite(&shared.done, 1); /* release*/
                        return; /* exit */
                }
                sum=0;
                for(i=0;i< DIM; i++) sum += vector[i] * matrix[i][k];
                result[k] = sum;
        }
}
```

Fig. 1. C program for matrix-vector product

2.2. Trace file and post-processors

The simulator can provide extensive debugging and performance information if one enables tracing within the application program. The trace file is initiated with *strace("filename")* where the argument is the name of a file. If the file exists, the trace information will be appended; otherwise a new file is created. Thus it is usually necessary

to remove the old trace file between successive runs of an application. The trace may be stopped with *etrace*. A program might have several calls to *strace* and *etrace* in order to trace simulator events within specific program segments or to limit the size of the trace file. If the file name given to *strace* is the null string, for example, *strace*("") then the trace output is directed to *stdout* and thus may be viewed directly on the terminal as the program runs, or, more often, piped into one of the post-processors for graphic display.

One line is written to the trace file for each simulator event such as process initiation, process termination, setting a shared variable, or waiting on a shared variable to become FULL. Figure 2 is an excerpt from a trace file. Each entry is stamped with simulator time, though the application must be built in *aspp*-mode for the simulator clock to be active. The programmer may include his own data in the trace file with the *mark* call, which writes a character string to the trace file. The *cnt* entries indicate the number of processors active at the given time. The active and waiting processors can be deduced from the "waking" and "blocking" substrings of a trace entry. In practice, the trace file can grow quite rapidly, so discretion is advised.

```
strace      tid  0 clock 4 running 0
cnt  1      clock 5
iacte tid 0 clock 6 addr 053224
iaset tid 0 clock 9 addr 053230
iaset tid 0 clock 12 addr 053234
tforktid  0 clock 18 taddr 682  stack 10000 waking 1
cnt  2      clock 19
iawrite tid 0 clock 123 addr 053230
iaread tid 0 clock 127 addr 053234 blocking 0
cnt  2      clock 127
iawrite tid 3 clock 127 addr 053234 waking 0
cnt  3      clock 128
evpost tid 3 clock 130 addr 0113314 was CLEARED
texit tid  3 clock 130 status 0
cnt  2      clock 130
```

Fig. 2. Trace file excerpt

The raw trace file can be a very useful debugging aid (see §2.5), but trace files are usually interpreted by post-processors to give performance summaries. For meaningful performance data to be obtained, the application program must have been built in *aspp*-mode. Two post-processors, *ccplot* and *trace1*, produce graphical output suitable for use by the UNIX *graph* command. For example,

ccplot tracefile | graph -b | plot -T4010

would plot processor utilization over time on a Tektronix 4010 graphics terminal. Figure 3 is an example of a plot produced by *ccplot*. The vertical axis is the number of processors active, and the horizontal axis is time measured in VAX instructions.

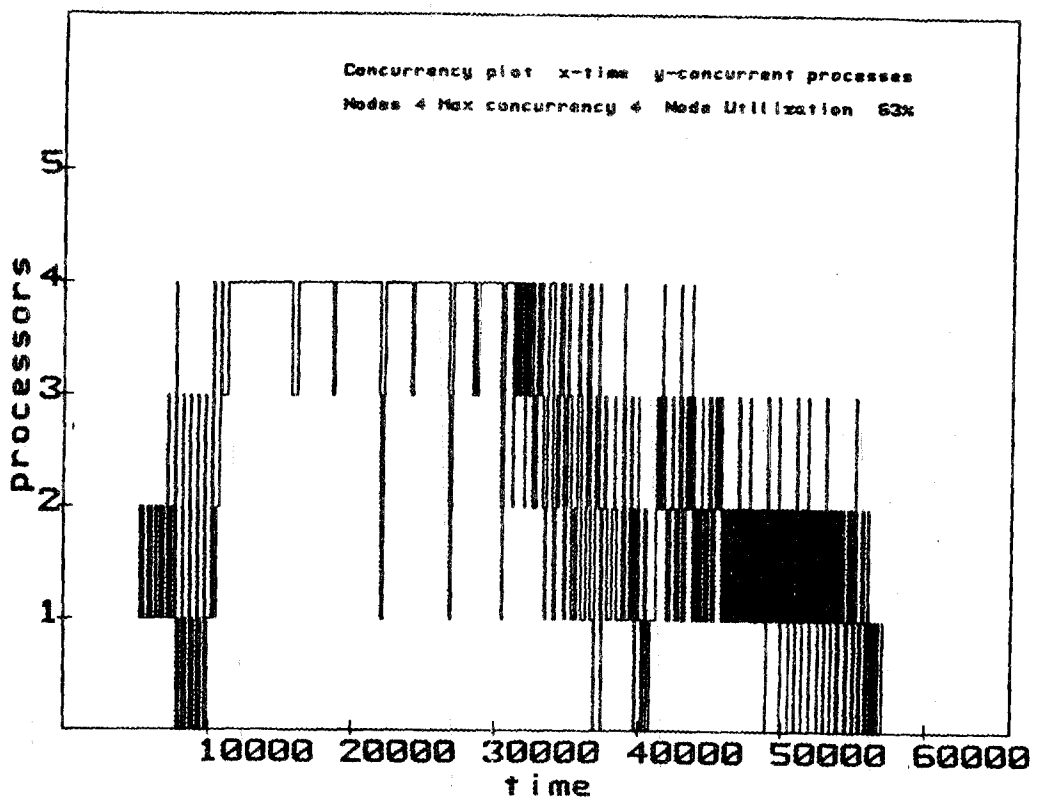

Fig. 3. Processor utilization from *ccplot*

To see specifically which processors are busy at a given time, one may use the *trace1* command,

trace1 tracefile | graph -b | plot -T4010

Figure 4 is a sample *trace1* plot, where the vertical axis is the processor id for each processor and the horizontal axis is simulator time. The horizontal lines indicate that a given processor is busy; otherwise the processor is idle (waiting for a shared variable to become FULL, for example).

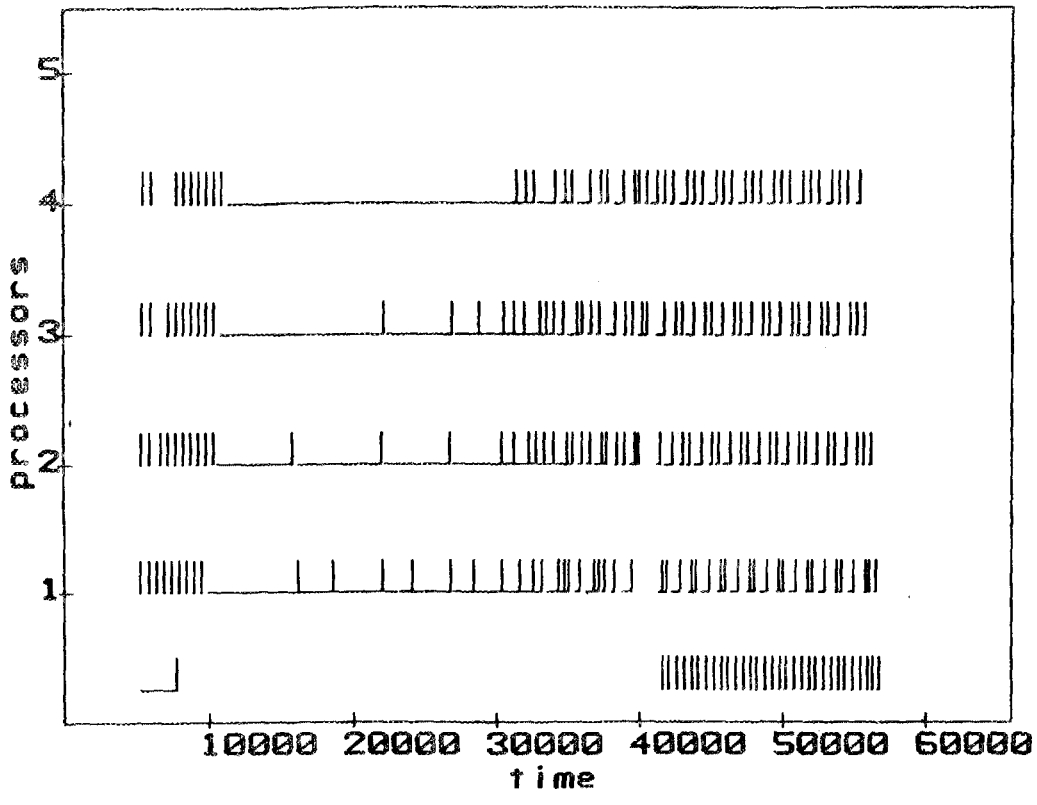

Fig. 4. Processor utilization from *trace1*

2.3. Sample session

Figure 5 is a transcript of a terminal session illustrating how one builds simulator programs and invokes post-processors. The files used are part of the simulator distribution tape. The actual location of these files is determined by how the simulator was installed at a given site.

In this sample session, the user copies three files from the simulator directory into his own working directory. The file *hepip.c* is just the vector-matrix program described in §2.1. The script *bld* is used to invoke the *makefile* using a command like *bld file* where *.c* is assumed as the extension to *file*. The executable *hepip* produced by *bld* is run and the resulting vector is printed.

The trace file is deleted and the program rebuilt in *aspp*-mode. The resulting program (*hepip*) is run again producing the same answer, but now the trace file (*hepip.trace*) has useful timing data. *Ccplot* is invoked to generate graphical data on processor utilization.

```
% cp /usr/local/intel/hepip.c .
% cp /usr/local/intel/bld .
% cp /usr/local/intel/makefile .
% bld hepip
cc -O -I/usr/local/intel -c tst.c
cc -o tst tst.o /usr/local/intel/libkernel.a -lm
% hepip
 45 27 31 14 40
% rm hepip.trace
% bld hepip aspp
cc -S -I/usr/local/intel tst.c
/usr/local/intel/aspp < tst.s > tst.tmp
mv tst.tmp tst.s
as -o tst.o tst.s
rm tst.s
cc -o tst tst.o /usr/local/intel/libkernel.a -lm
% hepip
 45 27 31 14 40
% /usr/local/intel/ccplot hepip.trace > plotdata
% /usr/local/intel/qplotit plotdata
```

Fig. 5. Sample simulator session for C

2.4. FORTRAN interface

The simulator subroutines described in §2.1 are available to the FORTRAN programmer as well. Figure 6 illustrates some of the simulator FORTRAN subroutines using the sample described in §2.1. The program calculates the inner product of a matrix (*matrix*) with a vector (*vector*) and prints the resulting vector (*result*). The result is calculated in parallel by creating processes to perform the vector products. Notice that there is an *include* statement and that those subroutines that are processes are of type TASK rather than of type SUBROUTINE. The AUTO statement declares all variables to be automatic. This permits multiple copies of a subroutine (process) to be executed concurrently — each having its own copy of local variables. (The read-only variables *matrix* and *vector* are declared *static* so they can be initialized in a DATA statement for this example.) The result is calculated in parallel by creating self-scheduling processes to perform the vector products. The host process (*task0*) starts the trace file, initializes the synchronization variables, and initiates several processes that will execute the subroutine *mult* concurrently. Notice that an EXTERNAL declaration is required for any subroutine (TASK) used in a *tfork*. The host process then executes the subroutine *mult* and awaits the completion of all of the subprocesses. The synchronization variable *done* will be FULL when all of the columns have been multiplied. The host process then prints the resulting vector.

Rather than statically assigning columns of the matrix to specific processes, the sample program utilizes self-scheduling. The process *mult* tests a shared column pointer k to determine if there is still another column to multiply. If there is, k is incremented and the *kth* column multiplied. If no columns remain, *active* is decremented, and the last process to finish sets *done* so the host process may proceed.

```
c vector matrix product
#include <task.fh>
      block data
      integer dim,cpus,k,active,done
      parameter (dim=5,cpus=3)
      integer matrix(dim,dim),vector(dim),result(dim)
      common /hep/matrix,vector,result,k,active,done
      data matrix /1,2,3,4,5,2,3,1,0,4,3,3,1,2,2,0,1,1,0,3,4,3,2,1,5/
      data vector /2,3,2,4,3/
      end
      TASK task0
      AUTO
      implicit integer (a-z)
c do simple inner product
      external mult
      integer dim,cpus,k,active,done
      parameter (dim=5,cpus=3)
      integer matrix(dim,dim),vector(dim),result(dim)
      common /hep/matrix,vector,result,k,active,done

      call strace("hepip.trace")
      call isete(done)
      call iaset(active, cpus + 1)
      call iaset(k,1)
      do 10 i=1,cpus
            call tfork(mult,10000)
10    continue
      call mult
      call iaread(done)
      write(6,*)result
      end

      TASK mult
      AUTO
      implicit integer (a-z)
c multiply vector and column of matrix
      integer localk, locact
      integer dim,cpus,k,active,done
      parameter (dim=5,cpus=3)
      integer matrix(dim,dim),vector(dim),result(dim)
      common /hep/matrix,vector,result,k,active,done

5     localk = iaread(k)
      call iawrite(k,localk+1)
      if (localk .gt. dim) then
            locact = iaread(active) - 1
            call iawrite(active,locact)
            if (locact .eq. 0 ) call iawrite(done,1)
            return
      else
            sum = 0
            do 10 i=1,dim
10                sum = sum + vector(i) * matrix(i,localk)
            result(localk) = sum
      endif
      go to 5
      end
```

Fig. 6. FORTRAN program for matrix-vector product

Figure 7 is a transcript of a terminal session illustrating how one builds FORTRAN simulator programs and invokes post-processors. The files used are part of the simulator distribution tape, and their location is determined by how the simulator was installed at a given site.

In this sample session, the user first copies three files from the simulator directory into his own working directory. Notice that *fbld* and *fmakefile* are renamed as part of the copying process. With this convention, one needs separate directories for building C and FORTRAN simulator applications. The file *hepip.m* is just the vector-matrix program described in the first part of this section. The extension *.m* is used as a reminder that the FORTRAN file is passed through a pre-processor before being compiled by the *f77* compiler. The script *bld* is used to invoke the *makefile* using a command like *bld file* where *.m* is assumed as the extension to *file*. The executable *hepip* produced by *bld* is run and the resulting vector is printed.

The trace file is deleted, and the program is rebuilt in *aspp*-mode. The resulting program (*hepip*) is run again, producing the same answer, but now the trace file (*hepip.trace*) has useful timing data. *Ccplot* is invoked to generate graphical data on processor utilization.

```
% cp /usr/local/intel/fbld bld
% cp /usr/local/intel/fmakefile makefile
% cp /usr/local/intel/hepip.m .
% bld hepip
/lib/cpp -I/usr/local/intel -DUNIX < tst.m | awk '\/^ #/ && /[^                0/' | expand -6 >tst.f
f77 -c tst.f
tst.f:
  task0:
  mult:
rm tst.f
f77 -o tst tst.o  /usr/local/intel/libfkernel.a
% hepip
 45  27  31  14  40
% rm hepip.trace
% bld hepip aspp
/lib/cpp -I/usr/local/intel -DUNIX < tst.m | awk '\/^ #/ && /[^                0/' | expand -6 >tst.f
f77 -S  tst.f
tst.f:
  task0:
  mult:
/usr/local/intel/aspp < tst.s > tst.tmp
mv tst.tmp tst.s
as -o tst.o tst.s
rm tst.s tst.f
f77 -o tst tst.o /usr/local/intel/libfkernel.a
% hepip
 45  27  31  14  40
% /usr/local/intel/ccplot hepip.trace > plotdata
% /usr/local/intel/qplotit plotdata
```

Fig. 7. Sample FORTRAN session

2.5. Debugging

The simulator provides a number of aids for discovering bugs in a parallel application. To reduce the number of initial bugs, keep the implementation simple, deferring until later tricky optimizations for speed or storage savings. Write the program so that it can run with an arbitrary number of processors, and then test it with just a few. This will keep the size of the trace file manageable, as well as any debugging output. Test and

run in non-*aspp*-mode first. This will give faster turnaround. If possible, isolate and test the synchronization logic of your application.

The trace file provides a wealth of information when things go wrong. Often synchronization problems arise from events occurring in an unanticipated order. The trace files shows what locations have been set to FULL or EMPTY and which processes are waiting. The *mark* function can be used to insert application-specific information in the trace file to assist in analyzing the state of the program.

One must estimate storage requirements for processes in the *tfork* call. The storage is used for all variables declared within the blocks of the called process as well as any procedures that process might call. Failure to provide sufficient storage produces unpredictable results! The *chkstk* function can be called from within any process or procedure to detect insufficient automatic storage. At this time, there is no way to increase the amount of automatic storage for the main program *task0*. FORTRAN programs will crash if the first argument to *tfork* has not been declared EXTERNAL.

Care must be exercised when passing arguments by reference to a parallel subroutine with *tfork*. FORTRAN passes arguments by reference, so the parallel subroutine may be making shared-memory references unknowingly through its formal parameters. Consider the following example.

```
      do 100 i=1,n
100   call tfork(work,10000,i)
```

The parallel subroutine *work* is started and passed *i*. Unfortunately, the value of *i* that *work* uses may not be the value intended, since the value will be changing as the *tfork* DO loop progresses. Using an expression, for example, *i+0*, will not change the undesired sharing. One solution would be the following

```
      do 100 i=1,n
      iarg(i)=i
100   call tfork(work,10000,iarg(i))
```

Other bugs can arise from not controlling access to shared variables with proper synchronization.

Sadly, the FORTRAN I/O routines are not re-entrant: thus concurrent use (*aspp*-mode) of FORTRAN I/O statements by two or more processes will result in a tight CPU loop. One must restrict FORTRAN I/O to one process at a time. To limit debugging output, it is a good practice to allow only one process to issue the debug output or serialize their use by using a shared variable.

The distribution also includes a version of the simulator library (*libdbxkernel.a*) that was built for use with the UNIX *dbx* debugger. The user can compile his programs with the *-g* option to the compiler and then link with the *dbx* version of the simulator library. Both the simulator and an application can be debugged in this manner.

Acknowledgements

The author is gratefully indebted to Eugene Brooks of Lawrence Livermore National Laboratory for constructing the shared memory simulator and for answering numerous questions as the simulator was extended and to Dan Pierce of North Carolina State University for his patience in testing the HEP subroutines.

References

[1] E. D. Brooks, A multitasking kernel for the C and Fortran programming languages, Tech. Rept. UCID-20167, Lawrence Livermore National Laboratory, Livermore, CA (1984).

[2] J. J. Dongara and R. E. Hiromoto, A collection of parallel linear equations routines for the Denelcor HEP, *Parallel Computing*, 1 (1984) 133-142.

[3] T. H. Dunigan, A Message-passing Multiprocessor Simulator, Tech. Rept. ORNL/TM-9966, Oak Ridge National Laboratory, Oak Ridge, TN (1986).

[4] *HEP Fortran 77 Reference Manual*, Denelcor Inc., Aurora, CO, 1984.

[5] *HEP/UPX Reference Manual*, Denelcor Inc., Aurora, CO, 1984.

[6] R. E. Hiromoto, O. M. Lubeck, and J. Moore, Experiences with the Denelcor HEP, *Parallel Computing*, 1 (1984) 197-206.

[7] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, 1984, pp. 669-684.

Appendix A

Simulator Manual Pages

NAME

hep - hep simulator routines

SYNOPSIS

```
#include "hep.h"

int isete(int *addr);
void iaset(int *addr, int value);
int iainc(int *addr, int value);
void iawrite(int *addr, int value);
int iaread(int *addr);
int iwaitf(int *addr);

float sete(float *addr);
void aset(float *addr, float value);
float ainc(float *addr, float value);
void awrite(float *addr, float value);
float aread(float *addr);
float waitf(float *addr);

int empty(int *addr);
int full(int *addr);
```

DESCRIPTION

These functions provide a HEP simulator interface to the parallel processor simulator (see *man ppsim)*. Shared variables should be external of any function defintions or be of class static. Associated with each variable on the HEP is an access state that may be either FULL or EMPTY. The access state is distinct from the value (contents) of the variable and can be used as a synchonization mechanism among cooperating parallel processes. The access state of a variable can be set or tested with the following functions.

isete and *sete* unconditionally set the state of the shared variable to EMPTY, then return the value in the memory location. One should use *sete* before referencing the variable with any of the other HEP functions.

iaset and *aset* store the value in the variable regardless of the access state and set the state of the variable to FULL.

iainc and *ainc* wait until the shared variable is FULL and then increment the variable and return the value before the increment, leaving the variable FULL.

iawrite and *awrite* wait until the shared variable is EMPTY and then set the variable to the given value and set the variable FULL.

iaread and *aread* wait until the variable is FULL and then return the value of the variable and set the variable EMPTY.

iwaitf waits until the variable is FULL before returning the value of the variable. The access stat is left FULL.

empty returns a value of 1 if the variable is EMPTY, otherwise a value of 0 is returned. *full* returns a value of 1 if the variable is FULL, otherwise a value of 0 is returned.

FORTRAN

Local variables must be declared *AUTOMATIC* in each FORTRAN subroutine. Use the include faciltiy and the macro AUTO to assist in this restriction. Shared variables should be in COMMON or passed as arguments. See the sample programs for examples.

The FORTRAN REAL functions corresponding to the C *float* functions defined above are *rsete, raread, rainc,* and *rwaitf*. FORTRAN calls to *aset, awrite* and *sete* may be used for **REAL**, but for **INTEGER** one must call *iaset, iawrite,* or *isete*.

BUGS

The shared variables are treated as 32-bit quantities for EMPTY and FULL. The HEP treats shared variables as 64-bit quantities. The choice of 32-bit for the simulator seemed more tractable for the VAX environment.

Unsigned shared variables are not presently supported. The *barrier* function is left as an exercise for the reader. The *create* function is provided by the *tfork* simulator function (see *ppsim(l)*).

The notion of PEMs is not modeled by the simulator. The simulator effectively provides a processor for every process.

SEE ALSO

Additional simulator functions are described in *ppsim(l)*. Additional HEP functional descriptions can be found in Denelcor's *HEP/UPX Reference Manual* and their *FORTRAN 77 Reference Manual*.

AUTHOR

T. H. Dunigan

NAME

ppsim - parallel processor simulator

SYNOPSIS

```
#include "task.h"

TID tfork(TASK fcn, int stacksize {,args});
void texit(int value);
void twait(TID taskid);
TID gettid();
void tsleep(int ticks);

void lock(LOCK *lockvar);
void unlock(LOCK *lockvar);

void evpost(EVENT *eventvar);
void evclear(EVENT *eventvar);
void evwait(EVENT *eventvar);

void strace(char *filename);
void etrace();
void mark(char *string);
```

DESCRIPTION

This package provides a set of function calls to implement a shared memory parallel processor and provides a means to debug and analyze parallel algorithms. Presently, up to 1000 parallel processes may be invoked. Various functions provide synchronization primitives and message-passing facilities. A trace file may be produced and plotted. The simulator is implemented within a single process on the VAX, so very large or very time consuming applications are discouraged. A sample makefile is provided to assist you in building your application. You do not provide a *main()* function, but rather provide a TASK *task0* as your application entry point. All external and global variables are known by all the processes, so proper synchronization must be used in accessing global variables. To provide full timing information and process switching after each application VAX instruction, the *aspp* build option must be used. Unfortunately, *aspp* forces a factor of 20 slowdown in the simulation.

tfork is used to initiate a process. The function initiated must be of type TASK. A stack size (in units of 4 bytes) must be provided to *tfork*, and it must be large enough to accomodate the automatic variables in the process, including those of any serial functions called by the process. A minimum stack size of 10000 is recommended. Any additional arguments to *tfork* are passed to the initiated process. *tfork* returns a TID value that can be used in a subsequent *twait* to await termination of the initiated process. A process will exit when it encounters the enclosing brace of the function definition or when a *texit* is executed. *gettid* returns the TID of the process. *tsleep* idles the process for the given number of clock ticks.

Synchronization of access to shared memory is provided by LOCK and EVENT variables and functions. *lock* locks a LOCK variable if it is not locked. If the variable is locked, the process is suspended until the lock is unlocked. The queue of processes waiting on a LOCK variable is FIFO. *unlock* unlocks a LOCK variable and releases the next waiting process on that LOCK variable. *evpost* sets an EVENT variable to POSTED and releases all processes who have issued an *evwait* on that EVENT variable. *evclear* sets an EVENT variable to CLEARED. *evwait* suspends the process until the EVENT variable is POSTED.

A trace file is initiated with *strace(filename)*, and the trace is stopped with *etrace*. If the filename is the null string, then the trace is directed to stdout. An informative string may be inserted in the trace file with the *mark* function. Various post-processing commands are available for plotting the trace file, though the *aspp* option must be used to obtain useful

concurrency information.

FILES

In /usr/local/intel the following files are available

libkernel.a	simulator functions
aspp	assembler post processor
makefile	sample make file
ccplot	trace file post processor
trace1	trace file post processor

SEE ALSO

See *man* entries for *hep* and *intel*. There is also a paper by Brooks describing the simulator. Sample programs may be found in /usr/local/intel. Other interfaces are available for f77 calls and simulating the CRAY XMP.

BUGS

The overhead of *aspp* needs to be reduced. Function calls to library routines are not accounted for in *aspp* mode. There is no easy way to detect an insufficient stack size provided to *tfork*. There is a function *chkstk()* that you may add to alert you to stack deficiencies. There is no way to set the stack size for task0.

The FORTRAN i/o routines are not re-entrant, so doing FORTRAN i/o in more than one process in aspp mode will put your code into an infinite loop. Also remember that FORTRAN passes values by reference, so passing arguments in a tfork should be done with care.

AUTHOR

Simulator is based on the "Multitasker" written by Eugene Brooks of LLNL and is available from the NESC.

DISTRIBUTION OF
ORNL/TM-9971

1. L. S. Abbott
2. J. Barhen
3. M. V. Denson
4-8. T. H. Dunigan
9. E. L. Frome
10. L. J. Gray
11-12. R. F. Harbison/ Mathematical Sciences Library
13. M. T. Heath
14. E. Leach
15. F. C. Maienschein
16. W. J. McClain
17. G. S. McNeilly
18. T. J. Mitchell
19. E. Ng
20. G. Ostrouchov
21. R. C. Ward
22. C. Weisbin
23. A. Zucker
24. Central Research Library
25. K-25 Plant Library
26. ORNL Patent Office
27. Y-12 Technical Library Document Reference Section
28. Laboratory Records--RC
29-30. Laboratory Records Department
31. P. W. Dickson, Jr. (Consultant)
32. G. H. Golub (Consultant)
33. R. W. Haralick (Consultant)
34. D. Steiner (Consultant)

EXTERNAL DISTRIBUTION

35. Dr. Donald M. Austin, Office of Scientific Computing, Office of Energy Research, ER-7, Germantown Building, U.S. Department of Energy, Washington, DC 20545

36. Dr. Robert G. Babb, Department of Computer Science and Engineering, Oregon Graduate Center, 19600 N.W. Walker Road, Beaverton, OR 97006

37. Dr. Donald A. Calahan, Department of Electrical and Computer Engineering, University of Michigan, Ann Arbor, MI 48109

38. Dr. George J. Davis, Department of Mathematics, Georgia State University, Atlanta, GA 30303

39. Dr. Jack J. Dongarra, Mathematics and Computer Science Division, Argonne National Laboratory, 9700 South Cass Avenue, Argonne, IL 60439

40. Dr. Paul O. Frederickson, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 97545

41. Dr. Jung Hong, Los Alamos National Laboratory, P.O. Box 1663, MS K488, Los Alamos, NM 87545

42. Dr. Harry Jordan, Department of Electrical and Computing Engineering, University of Colorado, Boulder, CO 80309

43. Dr. John G. Lewis, Boeing Computer Services, M/S 9C-01, 565 Andover Park, Tukwila, WA 98188

44. Dr. Joseph Liu, Department of Computer Science, York University, 4700 Keele Street, Downsview, Ontario, Canada M3J 1P3

45. Dr. Olaf Lubek, C-3, Computer Research and Applications, Los Alamos National Laboratory P.o. Box 1663 Los Alamos, NM 87545

46. Dr. James M. Ortega, Department of Applied Mathematics, University of Virginia, Charlottesville, VA 22903

47. Prof. Merrell Patrick, Department of Computer Science, Duke University, Durham, NC 27706

48. Dr. Robert J. Plemmons, Department of Mathematics and Computer Science, North Carolina State University, Raleigh, NC 27650

49. Dr. Andrew B. White, Computing Division, Los Alamos National Laboratory, Los Alamos, NM 87545

50. Dr. Arthur Wouk, Army Research Office, P.O. Box 12211 Research Triangle Park, NC 27709

51. Office of Assistant Manager for Energy Research and Development, Department of Energy, Oak Ridge Operations Office, Oak Ridge, TN 37830

52-78. Technical Information Center.