Distribution Category:
Mathematics and Computers
(UC-32)

ANL--83-96

DE84 005440

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

An Approach to Programming Multiprocessing Algorithms on the Denelcor HEP*

E. L. Lusk and R. A. Overbeek

Mathematics and Computer Science Division

December 1983

* This work was partially supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under Contract W-31-109-Eng-38, and also by National Science Foundation grant MCS82-07496.

An Approach to Programming Multiprocessing Algorithms on the Denelcor HEP

E. L. Lusk
R. A. Overbeek

ABSTRACT

In the process of learning how to write code for the Denelcor HEP, we have developed an approach that others may well find useful. We believe that the basic synchronization primitives of the HEP (i.e., asynchronous variables), along with the prototypical patterns for their use given in the HEP FORTRAN 77 User's Guide, form too low-level a conceptual basis for the formulation of multiprocessing algorithms. We advocate the use of monitors, which can be easily implemented using the HEP primitives. Attempts to solve substantial problems without introducing higher-level constructs such as monitors can produce code that is unreliable, unintelligible, and restricted to the specific dialect of FORTRAN currently supported on the HEP. Our experience leads us to believe that solutions which are both clear and efficient can be formulated using monitors.

1. Introduction

Recently, we began to investigate the degree to which parallelism exists in many of the fundamental algorithms of automated reasoning. While studying (over a number of years) the potential use of multiprocessing in the implementation of automated reasoning software, we gradually developed the view that, due to the synchronization overhead, very little true benefit could be obtained by implementing algorithms in which the ratio of computation to communication was small (algorithms with "small granularity"). However, after listening to Burton Smith discuss the architecture of the HEP, we came to seriously question our initial view. While a great deal of parallelism in an automated reasoning system can be exploited via processes which must be synchronized only after large amounts of separate computation, it is also possible that significant parallelism

within low-level algorithms might be exploited in a suitable hardware environment.

In order to investigate whether or not the use of tightly-coupled multiprocessing algorithms could enhance performance of automated reasoning systems, we implemented a version of the unification algorithm[7,8] on the HEP. The existence of low-overhead synchronization mechanisms adds a new and potentially very exciting dimension to the creation of automated reasoning software. It appears to us that the Denelcor HEP is one of the first operational machines to provide such mechanisms.

Our results concerning the degree of parallelism that can be exploited in the unification algorithm will be reported elsewhere, and will be of interest to a fairly limited audience. However, our experiences with the HEP and our implementation techniques may be of interest to a large number of other researchers. Essentially, we have translated existing tools for the formulation of multiprocessing algorithms into a form usable in the HEP environment without sacrificing the low-overhead of the HEP primitives. Furthermore, the code produced by our approach is essentially portable in that all of the peculiarities of the synchronization mechanisms offered in HEP FORTRAN[1] can be hidden using a trivial macro processor.

We believe that the basic tools offered by HEP FORTRAN should not be used to formulate algorithms. A person attempting to conceptualize the solution to even a moderately complex multiprocessing problem in terms of asynchronous variables will tend to make a great many errors. We speak from bitter experience. Impressed by the elegant solutions to several common synchronization problems done using asynchronous variables, we decided to approach our problem at the same level. Because we consider ourselves reasonably familiar with the intellectual traps of multiprocessing algorithms, we thought that the relatively simple task we were attacking would be easily solvable. In the process of trying to debug an early attempt at solving the problem we:

1) developed a version using two processes that gave us the correct answer on exactly 16 of 17 test cases (after giving the correct answer on all 17 when run with a single process),
2) achieved a deadlock, and
3) developed a version that worked with one process, ran marginally faster with two processes, and "hung" with three processes.

These errors resulted from attempting to formulate a solution in terms of the HEP primitives alone. Once we decided to retreat to familiar intellectual ground, a straightforward solution utilizing monitors resulted. This solution was then easily translated into a version based on the HEP primitives. This paper

describes our experience in detail.

2. The Basic Problem

The general problem that we attacked may be described as follows:

1) A sequence of tasks must be solved. We shall refer to these as the "major" tasks T_1, T_2, ...

2) Each major task Ti may be decomposed into one or more minor problems t_{i1}, t_{i2}, ... The solution of a task will take one of two forms, *success* and *failure*. The solution of a major task T_i will be success if and only if the solutions of t_{i1}, t_{i2}, ... are each success. The minor tasks may interact via shared variables, and they may be solved in any order.

3) A minor task may itself be decomposed. If it is, then its solution will be success if and only if the solutions to the tasks into which it is decomposed are each success.

In our specific problem, each major task involved the search for a common instance of two logical formulas. Such a search frequently decomposes into an attempt to show that two subterms have a common instance.

The basic approach to solving a major problem is to utilize a *stack* of minor problems remaining to be solved. Independent processes claim stack entries resulting directly from the decomposition of the original major task, or from the decomposition of other minor tasks. If any single minor task is solved with failure, the current major task has been solved with failure. This requires some careful synchronization to clear the stack, and wait for the currently operating processes to finish their (no longer interesting) minor tasks. On the other hand, a successful solution can be detected only when the stack is empty, and no processes are currently working on an outstanding minor task.

A natural way to think of solving such problems is to have a *master* process which creates a number of *slave* processes. The master process is responsible for decomposing the original major task, initiating the activity of the slaves, waiting for a solution to be computed, and reporting the solution. There is an objection to this approach: To debug the algorithm requires a minimum of two processes (the master and one slave). We have found it more convenient if the whole problem runs correctly with a single process (and, hopefully, faster if more processes are used). This objection can be overcome if the master joins the slaves in working on solutions to the minor tasks. This introduces the slight synchronization question of reactivating the master when a successful solution has been detected (since it will quite likely be blocked—probably waiting on the contents of an asynchronous variable, if a straightforward implementation of the stack is utilized).

Before going on to consider a solution to this class of problems, we should note that the solutions to the minor tasks may interact, as long as no backtracking is required. That is, the solution of any minor task may introduce constraints on the solutions of other minor tasks (through a shared data structure peculiar to the specific problem), as long as alternative solutions do not have to be considered. If alternatives must be considered via backtracking, the whole situation becomes significantly more complex. We have restricted our attention to the class of problems in which the minor tasks can be solved in any order. That is, if two minor tasks can be solved, they can be solved in either order, or simultaneously.

3. The Concept of Monitor

The basic concept of a monitor and how it can be used to impose structure on the specification of multiprocessing algorithms is well known by those working in operating systems or in language design[2, 3, 4, 5, 9]. The concept is quite simple: the critical sections associated with some data structure are coded as a set of procedures called a *monitor*. It must be guaranteed that only one process can be active in a single monitor. There must also be some mechanism for blocking and restarting processes.

The details of monitor implementation differ among the existing languages which include monitors as a basic construct. The outlook that we chose is similar (but not identical to) the one described by Per Brinch Hansen[3]. If a process that is currently active in a monitor must be blocked, it should issue a command

delay(<queue>)

where <queue> is a "delay queue" capable of holding any number of delayed processes. The effects of issuing a delay(<queue>) are as follows:

1) The process relinquishes control of the monitor.
2) The process is blocked in <queue> until activated by a continue(<queue>) command.

A process that is active in a monitor may reactivate one other process that is blocked on a queue associated with the monitor. It achieves this with

continue(<queue>)

The execution of a continue causes the currently active process to lose control of the monitor and return. In addition, if there are blocked processes on <queue>, one of them will be reactivated. Note that these two mechanisms retain the feature that only one process can be active in a monitor at one point in time.

Perhaps a simple example is in order: the following one is adapted from one in [3]. Suppose we wish to synchronize access to a single buffer, which must not be written to unless it is empty and must not be read unless it is full. (Note the similarity to the facilities provided directly by HEP asynchronous variables.) We will call this monitor BUFFER. The data structures associated with BUFFER consist of the buffer itself (say, an array of characters), and a boolean variable FULL to indicate the state of the buffer. The queues associated with with BUFFER will be SENDQ, where processes are held while waiting to write into the buffer, and RECEIVEQ, where processes are held while waiting to read the buffer. The two procedures of BUFFER are SEND and RECEIVE, which are called by processes which want to communicate through the buffer by writing into it and reading from it, respectively.

The algorithms for SEND and RECEIVE are then:

```
SEND:    procedure(message)
         if FULL is true then
             delay(SENDQ)
         endif
         move message to the buffer
         set FULL to true
         continue(RECEIVEQ)
         end procedure


RECEIVE: procedure(message)
         if FULL is false then
             delay(RECEIVEQ)
```

```
        endif

        move the contents of the buffer to message

        set FULL to false

        continue(SENDQ)

        end procedure
```

The initialization required for BUFFER is just

```
    set FULL to false
```

It is relatively easy to see that a process attempting to write to the buffer while it is non-empty will be delayed in SENDQ until restarted by the continue issued by another process which has just emptied the buffer. The precise way in which the queues are managed is a separate issue, and may vary from one application to another. It is not critical to the understanding of monitors.

Our delay/continue mechanism differs from Brinch Hansen's in that we allow <queue> to hold an arbitrary number of processes. He did implement FIFO blocking queues using his basic constructs. In our case, the process reactivated by a continue operation is completely arbitrary. You could, of course, use our mechanism to implement FIFO queues by following the same approach that Brinch Hansen used.

The approach we use hides the specific HEP primitives not only from the mainstream of the algorithm (since it only makes monitor calls), but also from the code in the monitors themselves (since they are written using "delay" and "continue", together with a mechanism for providing exclusive access to the monitor. Only in the implementation of "delay" and "continue" and the exclusive access mechanism does the specific syntax of HEP asynchronous variables appear. Most of the code (including the implementation of the monitors) thus becomes portable to any machine that supports primitives with which "delay" and "continue" can be implemented.

4. Solution of the Basic Problem Using a Monitor

The solution of our problem of computing solutions to major tasks can now be described. There are four basic operations on the stack:

stack-a-problem(<problem>)

post-failure()

post-no-more-major-tasks()

ask-for-task(<returned-task>,<return-code>)

These four operations can be coded as subroutines, along with an initialization subroutine, and grouped as a monitor.

Before considering the logic required by the monitor subroutines, let us present the overall logic of the solution:

```
master: procedure

        initialize the monitor

        create the desired number of slave processes

        try to acquire the next major task

        do while (there is a current major task)

            decompose the major task and invoke
                stack-a-problem(<minor task>) for each
                of the resulting minor tasks

            call work(master-identifier,return-code)

            process the return-code

            try to acquire the next major task

        enddo

        invoke  post-no-more-major-tasks() to signal completion
            to the slave processes
```

```
        end procedure

slave:  procedure

        call work(slave-identifier,return-code)

        end procedure


work:   procedure(identifier,return-code)

        set done to false

        do while (not done)

            ask-for-task(next-task,return-code)

            do while ((return-code = 'solved with success') or
                      (return-code = 'solved with failure')) and
                     (identifier = slave)

                ask-for-task(next-task,return-code)

            enddo

            if (return-code = "acquired a task") then

                process the minor task.  This may lead
                    to decomposing the task (stacking
                    more minor tasks), detecting failure
                    (and, hence, invoking post-failure),
                    or successful solution of the task.

            else

                set done to true

            endif

        enddo
```

```
        end procedure
```

Now let us consider the routines in the monitor. The monitor will have a single delay-queue, which is represented by two variables:

<delay-queue> represents the queue of blocked processes

<delay-queue-count> contains a count of the number of processes held in <delay-queue>

The delay(<delay-queue>) operation automatically increments the count, and continue(<delay-queue>) will decrement it (if a processes is reactivated). In addition to the <delay-queue> and the <delay-queue-count>, the algorithm requires two more variables:

no-more-major-tasks is a boolean variable initialized to false. It is set to true when the master process has determined that there are no more major tasks to solve.

problem-status is a variable that can take on three values: "open", "solved with success", and "solved with failure". It is initialized to "open" and reflects the status of the current major task. Now we can specify the essential logic of the procedures in the monitor:

```
monitor initialization procedure:

        set the stack to empty

        set no-more-major-tasks to false

        set <delay-queue> to empty

        set <delay-queue-count> to 0

        end procedure


stack-a-problem: procedure(<problem>)

        if (problem-status is "open") then

            add <problem> to the stack of minor tasks
```

```
            continue(<delay-queue>)
        endif
        end procedure

post-failure: procedure
        set problem-status to "solved with failure"
        end procedure

post-no-more-major-tasks: procedure
        set no-more-major-tasks to true
        continue(<delay-queue>)
        end procedure

ask-for-task: procedure(<returned-task>,<return-code>)
        if ((not program done) and (problem done)) then
            if (there are other nondelayed processes) then
                delay(<delay-queue>)
            endif
        else
            <return-code> <- "undetermined"
            while ((not program done) and (not problem done) and
                    (<return-code> = "undetermined")) do
```

```
        try to claim a problem
        if (success) then
            continue(<delay-queue>)
        else
            if (this is the last active process)
                set problem done (set code to "exhausted")
            else
                delay(<delay-queue>)
            endif
        endif
    enddo
endif
if (program done) then
    <return-code> <- "program done"
    continue(<delay-queue>)
else
    if (problem done) then
        <return-code> <- "problem done (done-code)"
        if (no more delayed processes) then
            reset variables (for next problem)
        endif
```

```
            continue(<delay-queue>)

        endif

    endif

    end procedure
```

We would like to reemphasize that only one process may be active in the monitor. Even so, the argument that these procedures accomplish the desired objective is nontrivial.

5. Implementation in HEP FORTRAN

Suppose that you have convinced yourself that an algorithm based on a monitor of the sort described above will work. How can such a solution be translated into HEP FORTRAN? The rules are really quite simple:

1) There must be a designated asynchronous variable used as a "lock" for access to the monitor. Call this variable <monitor-lock>. <monitor-lock> must be set to 0 during the initialization. The first instruction at the start of a monitor procedure should then be

   <local-variable> = <monitor-lock>

   where <local-variable> is any local variable.

2) Falling through to the end of a procedure causes a return, preceded by

   <monitor-lock> = 0

3) The execution of

   delay(<delay-queue>)

   is achieved with the following code

```
<delay-queue-count> = <delay-queue-count> + 1
<monitor-lock> = 0
```

```
<local-variable> = <delay-queue>
```

4) The execution of

continue(<delay-queue>)

is achieved with the following code

```
if (<delay-queue-count> = 0) then
    <monitor-lock> = 0
else
    <delay-queue-count> = <delay-queue-count> - 1
    <delay-queue> = 0
endif
return
```

The use of asynchronous variables for <monitor-lock> and <delay-queue> make the implementation essentially trivial.

There might well be some objection to the use of monitors on the grounds of efficiency. The overhead of the subroutine calls for the monitor procedures may in some cases be excessive. This can easily be obviated by replacing the call with the in-line expansion of the monitor procedure. The elegant way to achieve this is with a simple macro processor. Using this technique achieves an efficient, comprehensible, and reasonably portable implementation. The implementation is portable in that the same code can be used on a system with a similar architecture by replacing the code generated by the "enter a monitor", "exit from a monitor", "delay(q)", and "continue(q)" operations. These four operations, which we have implemented via asynchronous variables, represent the only operations utilizing specific features of the HEP instruction set.

6. Implementation of HEP Synchronization Patterns

Is it possible to implement the synchronization operations presented in the HEP user's manual conveniently with monitors? Obviously, operations requiring creation/deletion of processes (such as the fork/join example) cannot be handled with the concept of monitor as we have described it. The creation/deletion of processes is not achievable with the primitives that we have described for use with monitors. However, the other types of synchronization are easily achievable. One example, is the implementation of "barrier synchronization" with a monitor. The code given in the HEP User's Guide is as follows:

```
      IF (WAITF($INLOCK)) CONTINUE
      N=$NP+1
      IF (N .NE. IP) GO TO 5
      PURGE $INLOCK
      $OUTLOCK=.TRUE.
5     $NP=N
      IF (WAITF($OUTLOCK)) CONTINUE
      N=$NP-1
      IF (N .NE. 0) GO TO 10
      PURGE $OUTLOCK
      $INLOCK=.TRUE.
10    $NP=N
```

A barrier can be implemented with a monitor that contains a single <delay-queue> and its associated <delay-queue-count>. The code for such a monitor is simply

```
barrier: procedure(N)

         if (<delay-queue-count> < (N - 1)) then

             delay(<delay-queue>)

         endif

         continue(<delay-queue>)

         end procedure
```

This code translates into the following HEP FORTRAN:

```
      BARR = $BARR
      IF (BC1 .LT. (N-1)) THEN
          BC1 = BC1 + 1
          $BARR = 0
          BD1 = $BD1
      ENDIF
      IF (BC1 .GT. 0) THEN
          BC1 = BC1 - 1
          $BD1 = 0
          GO TO 10
      ELSE
          $BARR = 0
      ENDIF
10    CONTINUE
```

We certainly do not claim that this FORTRAN is more intelligible than the implementation in the HEP manual. It is, however, just as efficient. Furthermore, when expressed in the monitor notation, we feel that it is noticeably easier to grasp.

A second example given in the manual involves the general pattern for "self-scheduling DO loops". The code given in the manual is as follows:

```
      PROGRAM XXXX
      LOGICAL $DONE,DUMMY
      COMMON/ /$K,N,$DONE,$IACTIVE
      PURGE $K,$DONE,$IACTIVE
      $K = 1
      $IACTIVE = NPROC
      DO 10 J=1,NPROC-1
      CREATE SUB
10    CONTINUE
      CALL SUB
      DUMMY = $DONE
      STOP
      END
```

```
      SUBROUTINE SUB
      COMMON/ /$K,N,$DONE,$IACTIVE
 5    LOCI = $K
      IF (LOCI .GT. N) GO TO 10
      $K = LOCI+1


      .
      GO TO 5
10    K1 = $IACTIVE-1
      IF (K1 .EQ. 0) $DONE = .TRUE.
      $IACTIVE = K1
      RETURN
      END
```

(As an aside, it might be noted that the above code contains a bug. The statement following statement 5 above can transfer control to statement 10. When it does so, the asynchronous variable $K is never filled, and so processes will wait forever at the statement before statement 5.)

The monitor that would achieve such self-scheduling is composed of a single procedure: gettask is invoked at the head of SUB to get the next available subscript. Initially, the variable J is 1. The parameter I will be set to the next available subscript in the range 1 to N. NPROC is the number of processes competing for tasks.

```
gettask: procedure(I,N,NPROC,J)

         if (J <= N) then

            set I to J

            increment J by 1

          else

            set I to 0

            if (<delay-queue-count> < (NPROC - 1)) then

               delay(<delay-queue>)
```

```
            endif

            set J to 1

            continue(<delay-queue>)

        endif

        end procedure
```

This procedure allocates the next available subscript. If there are no more, a barrier is in effect until all of the processes have unsuccessfully attempted to get a task. Then they are all released (with I = 0). Using this monitor, the code would be

```
      PROGRAM XXXX
      COMMON/ /NPROC, J, <variables for the monitor operations>
      J = 1
      NPROC = <number of processes>
      <initialize the monitor>
      DO 10 I=1,NPROC-1
      CREATE SUB
10    CONTINUE
      CALL SUB
      STOP
      END

      SUBROUTINE SUB
      COMMON/ /NPROC, J, <variables for the monitor operations>
      N = <maximum subscript>
 5    gettask(LOCI,N,NPROC,J)
      if (LOCI .EQ. 0) GO TO 10
         .
         .
         .
      GO TO 5
10    RETURN
```

END

The monitor is initialized by setting the asynchronous variable associated with it (as the lock to allow only a single process to be in the monitor) to 0. This code appears at least as clear to us as the code which was not built around the concept of a monitor procedure.

7. Advantages of Using Monitors

Parallelism introduces complexity into algorithms. It is desirable to localize this complexity, lest it interfere with understandability and maintainability of the code that is not involved with synchronization. Monitors gather the complexity introduced by multiprocessing and hide it from the main, problem-solving sections of the code.

In the approach described above, the monitors themselves are written in terms of lower level primitives (delay and continue) which hide from the monitor code the specifics of HEP FORTRAN. Thus even those sections of code which do synchronization (the monitor procedures) are portable to any machine for which delay and continue can be implemented. An interesting special case is that of a uniprocessing machine. If an algorithm is organized in such a way that it can be carried out by a single process, then the code can by run and at least partially debugged on an ordinary single-processor machine. (This was why we chose to have the master in our unification algorithm join the slaves in solving the problem.) The authors have developed a macro language for specifying monitors with two sets of macro definitions: one for our VAX and one for the HEP. The source code for a given program is exactly the same whether the program is to be run on the VAX or the HEP; it is expanded using the appropriate macro definitions just prior to compilation on either machine. This has allowed us to do significant testing on our local VAX. The macro expansion approach leads to roughly the same number of lines of HEP FORTRAN as hand coding, thus providing portability with no cost in efficiency. The details of the this macro package are provided elsewhere[6].

A third advantage of monitors is that it may be possible to develop a useful library of standard, well-understood monitors which will be used in a wide variety of algorithms. The monitors for self-scheduling DO loops and barriers are examples of such patterns; the central example of this paper is another. We are currently developing others. The macro package described in [6] provides a mechanism for a user to customize a version of the "ask-for-task" monitor procedure for his own task structure, without having to rewrite (or even be aware of) the synchronization logic. Once a sufficiently large set of standard monitors

has been developed, algorithms that utilize multiprocessing can be implemented with a minimum of errors resulting from synchronization problems, and the resulting code will be portable from one multiprocessing architecture to another.

8. Summary

After experimenting with the HEP primitives for synchronization, we have found them to be excellent tools for implementing monitors. We do not feel that most algorithms can be safely formulated using such low-level primitives. We advise programmers of the HEP to utilize monitors, expanding the monitor procedures in-line when efficiency is critical. This approach will lead to clearer, more portable code, and will not impair the efficiency of the code.

There is currently a great deal of discussion concerning which extensions should be added to FORTRAN to allow the specification of multiprocessing algorithms. Since our backgrounds are not in language design, we do not feel qualified to give a completely specified set of language enhancements. However, we do believe our experience indicates that the basic notion of monitors is far preferable to lower-level constructs such as asynchronous variables.

References

1. *HEP FORTRAN 77 User's Guide*, Denelcor, Inc., Aurora, Colorado (1982).
2. Per Brinch Hansen, "The programming language Concurrent Pascal," *IEEE Transactions on Software Engineering SE-1* **2** pp. 199-207 (June 1975).
3. Per Brinch Hansen, *The Architecture of Concurrent Programs*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey (1977).
4. C. A. R. Hoare, "Monitors: an operating system structuring concept," *Communications of the ACM*, pp. 549-557 (October 1974).
5. R. C. Holt, G. S. Graham, E. D. Lazowska, and M. A. Scott, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley Publishing Co., Menlo Park, California (1978).
6. Ewing L. Lusk and Ross A. Overbeek, *Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors*, (preprint)
7. J. Robinson, "A machine-oriented logic based on the resolution principle," *Journal of the ACM* **12** pp. 23-41 (1965).

8. J. A. Robinson, "Computational logic: the unification computation," pp. 63-72 in *Machine Intelligence 6*, ed. B. Meltzer and D. Michie,American Elsevier, New York (1971).

9. N. Wirth, "MODULA: a language for modular programming," *Software Practice and Experience* **7** pp. 3-35 (January-February 1977).

Distribution for ANL-83-96

Internal:

K. L. Kliewer
A. B. Krisciunas
E. L. Lusk *(72)*
P. C. Messina
R. A. Overbeek (10)
D. M. Pahis
T. M. Woods (2)
G. W. Pieper
ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (6)

External:

DOE-TIC, for distribution per UC-32 *(186)*
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
J. C. Browne, U. Texas, Austin
S. Gerhart, Software Research Associates, Culver City
L. P. Kadanoff, U. of Chicago
W. C. Lynch, Xerox Corp., Palo Alto
J. M. Ortega, U. Virginia
D. L. Wallace, U. of Chicago
M. F. Wheeler, Rice U.
D. Austin, Office of Basic Energy Sciences, DOE
G. Michael, LLL