IMPLEMENTATION OF MONITORS WITH MACROS:
A PROGRAMMING AID FOR THE HEP AND
OTHER PARALLEL PROCESSORS*

E. L. Lusk and R. A. Overbeek
Mathematics and Computer Science Division

December 1983
Revised July 1984

* This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under contract W-31-109-Eng-38.
Implementation of Monitors with Macros:

A Programming Aid for the HEP and Other Parallel Processors

E. L. Lusk
R. A. Overbeek

ABSTRACT

In a previous paper[3] we delineated the advantages of using monitors when implementing multiprocessing algorithms for the Denelcor HEP. In this report we give a detailed presentation of how monitors can be implemented on the HEP using a simple macro processor. We then develop the thesis that a small body of general-purpose monitors can be defined to handle most standard synchronization patterns. We include the macro packages required to implement some of the more common synchronization patterns, including the fairly complex logic discussed in [3]. Code produced using these macro packages is portable from one multiprocessing environment to another. Indeed, by recoding the set of basic macros (about 100 lines of code for the Denelcor HEP), most programs that we are now writing could be moved to any similar multiprocessing system.

1. Introduction

In a previous paper we discussed the motivation for using monitors when programming an MIMD computer such as the Denelcor HEP[3]. In this paper we will present the details of how monitors can be implemented using a simple macro processor. We believe that there are a small number of general types of synchronization patterns that suffice for the implementation of most algorithms. Several such prototypes were analyzed in early work by Jordan based on the HEP[1, 2]. For example, "barrier synchronization" and "self-scheduling DO-loops" were found to be particularly useful. However, in the earlier work, no attempt was made to develop portable code. Nor was there any attempt made to cast these patterns in the more general conceptual framework of monitors.

In the process of implementing both numeric and non-numeric algorithms, we have found it necessary to develop other general patterns of synchronization. We have developed the macros to support portable code based on monitors to implement these patterns. In addition we have implemented a monitor that can
be used to support the intercommunication of a number of processes in what might be referred to as a "data flow" synchronization pattern. We believe that the addition of these basic types of monitors to the patterns already well understood, along with the macro implementations of these monitors, have resulted in a useful set of tools for programming multiprocessing algorithms.

Our intent in this paper is to present all of the coding techniques in complete detail. We risk boring those who wish a cursory understanding in hopes that others will gain more benefit from our experiences. Our entire implementation of monitors for the HEP is based on the m4 macro processor provided by UNIX. The programs are all written in FORTRAN, because that is the only language currently supported on the HEP. However, nothing in our approach would not carry over naturally into an environment supporting equivalent macro and language processors.

2. Basic Concepts of Multiprocessing

Fundamental to any discussion of multiprocessing are the basic notions of process, program, and processor. A process can be thought of as an abstract machine with predictable behavior. The behavior is described precisely by a program. In order for the process to carry out its behavior, it must be assigned to a hardware device, the processor. The three concepts are quite independent. Two different processes may be described by the same program (e.g., two "sort" processes may run simultaneously, using the same program). A collection of processes may be run on a single processor by assigning them each to the processor one after the other, or by interleaving short periods of time during each of which a specific process is assigned to the processor. This is the familiar time-slicing operation of large single-processor computer systems. For this reason the theoretical study of multiprocessing has been largely confined to the field of operating systems. The users of such systems have always seen their own tasks as single processes.

As long as processes do not communicate with one another, the programs describing their behavior look just like programs describing the behavior of one process assigned to one processor which has no other processes assigned to it. This has always been the most common situation, and in it the distinctions made above among process, program, and processor are often blurred. In fact, it is easy to think of a program as a set of instructions for the processor to carry out. This point of view is not "wrong", but it does make it difficult to find the right abstract concepts for formulating an algorithm for a communicating collection of processes without being led astray by the language peculiarities of a specific machine.
As we approach the limits of computational speed obtainable on a single processor, it will become increasingly necessary to reformulate algorithms so that they consist of multiple processes, each of which may be assigned to a processor in a machine with many processors. Note that it still may be the case that many processes share a processor. There is not nearly the body of programming experience built up for algorithms organized around a collection of processes as there is around algorithms for single processes. Since processes cooperating on a problem must communicate, an important part of the programs that describe the behavior of such processes is taken up with interprocess communication and synchronization. In an earlier paper[3] the authors described a method for organizing such a program, which was indeed borrowed from the operating system literature. The central concept is that of a monitor, which can be viewed as a collection of program fragments that are shared by several processes. The complexities of interprocess communication are localized in the monitors, which in turn can be implemented in terms of synchronization mechanisms that are independent of any specific machine. Thus the goal of that paper was to present an approach to programming which would result in programs for multiprocessing algorithms which were not only easy to understand and maintain, but also portable to a variety of machines. Thus a program developed for a specific multiprocessing computer like the Denelcor HEP could be moved to another multiprocessor (or, if certain conditions are met, to a uniprocessor) with a minimum of change.

In this paper we take the approach a step further in level of detail, and present a technique which makes the implementation of multiprocessing algorithms particularly straightforward. By using the monitor structure described in [3], together with a collection of macros which we present in detail here, the programmer of many complex algorithms can be spared altogether the coding complexities introduced by multiprocessing, as well as the particular way they are specified on a given machine.

3. Implementing Send/Receive

3.1. Send/Receive Implemented as Monitors

The early work of Jordan[1] offers a clear explanation of how to implement many of the basic synchronization primitives in the HEP environment. In particular, his paper contains a nice explanation of how to implement send/receive primitives. Our work has centered on the creation of portable, reliable code. We have found that (in the limited set of real programs that we have implemented and evaluated) portability can be achieved with very little loss of efficiency. By portability, we mean the ability to transfer the code to another multiprocessor machine which supports similar capabilities (shared memory with
synchronization primitives) with minimal recoding. In our case we have limited the required recoding to fewer than a hundred lines of low-level macros. Thus, any of the programs that we have written in HEP FORTRAN can be converted to run on another MIMD machine by recoding this small macro library. In this section we will give the basic details of our approach, illustrating our techniques by implementing the send/receive primitives. In some sense this may appear to be a poor starting point, because Jordan gives a significantly more efficient (and elegant) implementation based on peculiarities of the HEP asynchronous variables. We urge the reader to defer a detailed investigation of efficiency issues until after studying implementations of several more of the prototypical synchronization patterns.

We will start this discussion with a presentation of the send/receive logic in the form given in our previous paper:

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
    delay(RECEIVEQ)
  endif
  move the contents of the buffer to message
  set FULL to false
  continue(SENDQ)
end procedure

The initialization required for the monitor is just

set FULL to false
These two procedures, along with the initialization code, are a **monitor**. A monitor is a set of procedures and initialization logic. The initialization logic must be invoked by some process before any process attempts to invoke any of the procedures. The procedures can be invoked from any process. The procedures represent **critical sections**. That is, they are sections of code that must not be executed simultaneously by several processes. Thus, in a loose sense, the procedures in a monitor may be thought of as the critical sections associated with processing a shared data structure. In fact, our approach towards implementing the critical sections of monitors involves the use of macros to generate the code for a monitor operation ("procedure") in-line. By definition, only one monitor procedure can be executed at a time. Thus, if process A has invoked one of the monitor procedures, and if process B attempts to invoke a procedure in the same monitor, process B will be **blocked** until A relinquishes the monitor. Only one process may be "inside the monitor" (i.e., executing a monitor procedure) at any point in time.

Because processes normally must synchronize on the shared data associated with a given monitor, it is sometimes the case that a process will have to **delay** until some event occurs. To delay a process means to associate the process with a **delay queue**. This causes the process to cease forward progress until it is removed from the queue. The act of putting the process into the delay queue causes the process to relinquish ownership of the monitor. Thus, a delay operation will allow other processes to execute monitor procedures while the delayed process remains in the queue.

A delayed process is reactivated when another process executes a **continue** operation (which can only be issued from a procedure in the monitor). The continue operation specifies a delay queue that may or may not be empty. The effect of executing a continue is to cause the process that issued the continue to immediately exit from the monitor procedure. In addition, if the specified delay queue is not empty, one of the delayed processes will be reactivated. Note that the continue operation maintains the property that only one process may be active inside the monitor. Also note that the reactivated process gains "ownership" of the monitor - no other process can enter between the exit of the process that issues the continue and the reactivation of the delayed process.

With these comments in mind, the logic given above for the send/receive logic should be understandable. Such a monitor might be used to implement a "pipelined" communication between two processes. One process performs an initial computation and then sends the result to the second process. The second process uses the receive procedure to access the data items sent by the first process. Suppose that there were multiple processes performing the initial computation and multiple processes issuing receives to acquire the results.
Would the above monitor still work?

To implement the above monitor, we have created several macros that can be used for purposes of assuring exclusive ownership of the monitor, for delaying processes, and for reactivating processes. An example of how to code such a monitor using these macros would be as follows: (SR can be thought of as the monitor's name, for Send/Receive)

```
SUBROUTINE SEND(<message>)

LOGICAL FULL

COMMON /UCOMM/ FULL, <buffer>

decvar(SR,2)
decom(SR,2)

menter(SR)

IF (FULL) THEN

    delay(SR,1)

ENDIF

<move message to the buffer>

FULL = .TRUE.

continue(SR,2)

next(SR)

RETURN

END
```
SUBROUTINE RECEIVE(<message>)

LOGICAL FULL

COMMON /UCOMM/ FULL, <buffer>

decvar(SR,2) deccom(SR,2) 
menter(SR)

IF (.NOT. FULL) THEN

delay(SR,2)

ENDIF

<move the contents of the buffer to message>

FULL = .FALSE.

continue(SR,1)

mexit(SR)

RETURN

END

The initialization is

COMMON /UCOMM/ FULL, <buffer>

moninit(SR,2) 
FULL = .FALSE.

The initialization code would have to be executed before any monitor calls were executed. To understand our implementation, it is necessary to grasp both the logical function of each macro in the code, and the HEP FORTRAN code
generated by each macro. The logical functions of the macros are as follows:

*decvar and deccom:* These macros are used to declare and generate the variables required to support the monitor operations. For example, decvar(SR,2) declares the variables required for a monitor named SR with two associated delay queues. The deccom(SR,2) generates the specifications for the COMMON area required to hold the variables shared between the monitor routines.

*menter and mexit:* These macros generate the code required to assure exclusive access to the monitor. In the example, menter(SR) generates the code required to make sure that the process is blocked if another process owns the monitor. The execution of mexit(SR) relinquishes control of the monitor.

*delay:* This macro delays the current process in the designated delay queue. Thus, delay(SR,1) delays the process in the first delay queue associated with the monitor SR.

*continue:* If there are delayed processes in the specified delay queue, one of them will be reactivated. In addition, this macro causes the process to GO TO just past the next mexit statement. For example, continue(SR,2) in the send procedure will cause a GO TO just past the next mexit in that procedure. In addition, if any process is delayed in delay queue 2, one of the delayed processes will be activated.

*moninit:* This macro generates the code required to initialize any of the variables used to implement the monitor.

### 3.2. Synchronization on the HEP

Before describing in detail our implementation of these concepts on the HEP, let us describe briefly the very few aspects of the HEP instruction set required to understand the generated code. Synchronization on the HEP is achieved via *asynchronous variables*. Such variables are represented by an identifier name that begins with a dollar sign. An asynchronous variable is really a 2-tuple, a *value* and a *state*. The value is simply the usual representation of a variable, while the state can be either *full* or *empty*. Consider the assignment
Here we are assigning the value of an asynchronous variable to that of a normal FORTRAN variable. The following rule applies:

When the value of an asynchronous variable is referenced, it will be accessed only when the state is full. If the state of the asynchronous variable is empty, the process "waits" for another process to insert a value (changing the state to full). Furthermore, accessing the value alters the state to empty.

Thus, asynchronous variables offer a particularly convenient way to pass values between processes. The HEP implements the state with an extra bit in memory and machine instructions which utilize it, thus providing extremely efficient synchronization mechanism.

To initialize the contents of an asynchronous variable, one would first use

```
PURGE $AS
```

which sets the state to empty. Then, an assignment statement can be used to insert the first value. The initialization via the PURGE, along with the simple "access-when-full-then-empty" and "fill-when-empty" rules, are the only features of the HEP that we utilized in constructing our implementation of the macro primitives.

### 3.3. Macro implementation on the HEP

The actual HEP FORTRAN code generated by the macros for the HEP implementation of the SEND/RECEIVE routines would be as follows:

```
SUBROUTINE SEND(<message>)
LOGICAL FULL
COMMON /UCOMM/ FULL, <buffer>
INTEGER $SR, SR
INTEGER $SRD1, SRD1, SRC1, $SRD2, SRD2, SRC2
COMMON /SRCOM/ $SR, $SRD1, SRC1, $SRD2, SRC2
```
SR = $SR$

IF (FULL) THEN

SRC1 = SRC1 + 1
$SR = 0
SRD1 = $SRD1$

ENDIF

<move message to the buffer>

FULL = .TRUE.

IF (SRC2 .EQ. 0) THEN

$SR = 0
ELSE

SRC2 = SRC2 - 1
$SRD2 = 0
ENDIF

GO TO 800

$SR = 0

800 CONTINUE

RETURN

END

SUBROUTINE RECEIVE(<message>)

LOGICAL FULL

COMMON /UCOMM/ FULL, <buffer>

INTEGER $SR, SR

INTEGER $SRD1, SRD1, SRC1, $SRD2, SRD2, SRC2
COMMON /SRCOM/ $SR , $SRD1, SRC1, $SRD2, SRC2

SR = $SR

IF (.NOT. FULL) THEN

    SRC2 = SRC? + 1
    $SR = 0
    SRD2 = $SRD2

ENDIF

<move the contents of the buffer to message>

FULL = .FALSE.

IF (SRC1 .EQ. 0) THEN

    $SR = 0
ELSE

    SRC1 = SRC1 - 1
    $SRD1 = 0
ENDIF

GO TO 801

$SR = 0

801 CONTINUE

RETURN

END

The initialization code is

PURGE $SR
This code is certainly not as elegant as that given by Jordan. Neither is it quite as efficient (although this is the only standard monitor so far that is less efficient than those customized to the HEP instruction set, and the difference in efficiency will normally be negligible). The code (in the form with the unexpanded macros) is, however, portable. The actual macros that generate HEP FORTRAN code are given in Appendix A. There are about 100 lines of macro definition that generate code for the HEP. These are the only lines of code that are not entirely portable in all of the programs that we have coded for the HEP. The macros are based on the m4 macro processor, which is available under UNIX. Any comparable macro processor work work just as well, and in fact we use only the most basic features of m4. When studying the macros, the reader is warned to note the use of the "changequote" statement that changes the definition of the quote symbols. This was necessary in that some of our terminals did not support the default symbols. In addition, in later macro examples, the reader should note that commas intended to be in the code generated by macros must be surrounded by quotes (i.e., [,]).

Some concern might exist that the macros occasionally generate statements that cannot be reached (e.g., the last "$SR = 0" in each of the monitor operations). This will happen when a "continue" macro immediately precedes a "mexit". A "mexit" is always required to terminate a monitor operation (among other things, it generates a CONTINUE statement that may be referenced from within the monitor operation). When a "continue" immediately precedes the "mexit," a superfluous GO TO and an unreachable assignment statement are generated. One could avoid this by using a slightly more complex definition of the macros, but for our purposes this seemed unwarranted.

There are two points which should be made at this stage in our development:
1. If macros for send and receive are coded, the "monitor procedures" may be thought of instead as monitor operations generated in-line by invoking macros. This eliminates the overhead of the procedure invocation.

2. The only difference between different incarnations of the send/receive monitor will be the specific code to move messages to and from the shared data area ("the buffer"). This means that general send/receive macros can be coded that invoke user supplied macros to generate the fairly small amount of code that differs between applications.

These points are worth covering in detail, because we plan to supply a standard library of macros to generate prototypical monitors. Users will then be able to invoke portable synchronization mechanisms by supplying only some very minimal code in the form of macros that generate application-dependent code. Appendix B contains four macros: srdec, srinit, srsend, and srrec. These four macros implement the prototypical monitor for send/receive logic. Thus,

\[
\text{srsend(SR, moveto(MESS))}
\]

could be used to send a message, assuming that "moveto" is a user-supplied macro that generates the code to move MESS to a shared buffer area (which must be in a COMMON area established for this specific instance of the send/receive logic).

4. Barrier Synchronization and Self-scheduling DO-loops

The early work of Jordan identified a number of basic synchronization patterns. Two of those identified were barrier synchronization and self-scheduling DO-loops. In this section we discuss the macros for generating these two monitor prototypes.

The concept of barrier synchronization is quite simple. A "barrier" for IP processes causes (IP - 1) processes to be delayed until the remaining process reaches the barrier. Then all IP processes continue execution. The code for barrier synchronization given in the HEP User's Guide is as follows:

\[
\begin{align*}
\text{IF (WAITF($\text{SINLOCK}$)) CONTINUE} \\
N &= \text{SNP} + 1 \\
\text{IF (N .NE. IP) GO TO 5} \\
\text{PURGE $\text{SINLOCK}$} \\
\text{SOUTLOCK} &= .\text{TRUE.} \\
\text{SNP} &= N \\
\text{IF (WAITF($\text{SOUTLOCK}$)) CONTINUE}
\end{align*}
\]
This code uses the built-in function \texttt{WAITF}, which delays until \texttt{SINLOCK} has a state of full (but does not empty \texttt{SINLOCK}). Note that this barrier for \texttt{IP} processes assumes that \texttt{IP} is the total number of processes (e.g., it cannot be used as a barrier for 3 of 7 processes).

The macros implementing barrier synchronization are given in Appendix C. Note that the code for the barrier macro references a variable \texttt{<mon>C1}, where \texttt{<mon>} is the name of the monitor, which is used as the count of the number of processes delayed in the first (and only) delay queue associated with the monitor. By convention in the macro package for any new machine, this name will always reference the count for the specified queue (the variables for other queue counts would be \texttt{<mon>C2}, \texttt{<mon>C3}, etc.) Thus,

\begin{verbatim}
bardec(B1)
barinit(B1)
barrier(B1,NPROC)
\end{verbatim}

would implement a barrier for \texttt{NPROC} processes. Here the \texttt{bardec} declares the required variables, the \texttt{barinit} initializes the monitor, and \texttt{barrier} generates the single monitor operation (which creates a barrier for \texttt{NPROC} processes). The most common use of such a barrier would be in a program that is simultaneously being executed by \texttt{NPROC} processes. However, the distinction between a process and a program must always be kept in mind. For example, would the above code work if the

\begin{verbatim}
barrier(B1,NPROC)
\end{verbatim}

were used in two distinct subroutines being executed by two processes? The answer is yes, and the reader should make sure at this point that he clearly
understands exactly why (the routines share variables in a COMMON area; this works whether or not the processes are executing the same routine).

The expansion of the barrier macro yields the following code:

\begin{verbatim}
enter(B1)
  IF (B1C1 .LT. (NPROC - 1)) THEN
    delay(B1, 1)
  ENDIF
  continue(B1, 1)
exit(B1)
\end{verbatim}

This code is expanded using the basic macro package into the following FORTRAN code:

\begin{verbatim}
B1 = $B1

  IF (B1C1 .LT. (NPROC - 1)) THEN
    B1C1 = B1C1 + 1
    $B1 = 0
    B1D1 = $B1D1
  ENDIF

  IF (B1C1 .EQ. 0) THEN
    $B1 = 0
  ELSE
    B1C1 = B1C1 - 1
    $B1D1 = 0
  ENDIF
  GO TO 800

  $B1 = 0
800 CONTINUE
\end{verbatim}

Note that it is not less efficient than the corresponding code from the IEP FORTRAN manual.

A self-scheduling DO-loop is used when a number of processes (NPROC) wish to cooperate in executing the body of a loop. Here it is assumed that execution of the body of the loop can be carried out in parallel for different values of the subscript (LOCI in our example). Subscript values in our example range from 1 to N. The code given in the IEP manual for a self-scheduling DO-loop is as
follows:

```fortran
PROGRAM XXXX
LOGICAL $DONE, DUMMY
COMMON /SK, N, $DONE, $IACTIVE
PURGE SK, $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 /SK, N, $DONE, $IACTIVE
5 LOCI = $K
IF (LOCI .GT. N) GO TO 10
$K = LOCI+1

<body of the loop>

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

This code is worth noting for several reasons. Perhaps the most outstanding feature is that it contains a serious bug: ($K$ is not "unlocked" when $LOCI$ is assigned a value greater than $N$). We wish to emphasize that such bugs are extremely easy to introduce and very difficult to locate. This is one of the key reasons that we advocate the use of macros to hide the complexity of such synchronization mechanisms. It is also worth noting that the code for self-scheduling DO-loops is given correctly in Jordan's early work[1]. The macros to
implement the self-scheduling DO-loop monitor are given in Appendix D. Using these macros the corresponding code would be as follows:

```
PROGRAM XXXX
COMMON/ /NPROC, J
deccvar(SD,1)
decccrn(SD,1)

J = '
NPROC = <number of processes>
moninit(SD,1)
SDSB = 1

DO 10 I=1,NPROC-1
CREATE SUB
10 CONTINUE
CALL SUB
STOP
END

SUBROUTINE SUB
COMMON/ /NPROC, J
deccvar(SD,1)
decccrn(SD,1)

N = <maximum subscript>
5 menter(SD)
IF (SDSB .LE. N) THEN
    LOC1 = SDSB
    SDSB = SDSB + 1
ELSE
    LOC1 = 0
    IF (SDC1 .LT. (NPROC - 1)) THEN
        delay(SD,1)
    ENDIF
    EXIT(SD)
ENDIF
mexit(SD)
```
IF (LOCI .EQ. 0) GO TO 10
.
<body of loop>
.
GO TO 5
10 RETURN
END

This code can be expanded using the basic macros into the following FORTRAN code:

PROGRAM XXXX
COMMON /NPROC, J
gsdec(SD)
J = 1
NPROC = <number of processes>
gsinit(SD)
DO 10 I=1,NPROC-1
CREATE SUB
10 CONTINUE
CALL SUB
STOP
END

SUBROUTINE SUB
COMMON /NPROC, J
gsdec(SD)
N = <maximum subscript>
5 getsub(SD,LOCI,N,NPROC)
IF (LOCI .EQ. 0) GO TO 10
.
<body of loop>
.
GO TO 5
10 RETURN
END
5. A More Complex Prototypical Monitor: the "ask-for-task"

In our studies of both nonnumeric and numeric algorithms we encountered a fundamental synchronization pattern that we described in our previous paper. We repeat the description of the pattern here for completeness. The general problem may be described as follows:

1) A sequence of computational tasks (i.e., problems) must be solved. We shall refer to these as the "major" tasks $T_1, T_2, ...$

2) Each major task $T_i$ may be decomposed into one or more minor tasks $t_{i1}, t_{i2}, ...$

3) A minor task may itself be decomposed.

4) At any point in the computation, the solution of a minor task may result in a solution for the current major task. Thus, the current major task is thought of as "unsolved" until either a subcomputation produces a solution, or until all subcomputations are completed. We refer to this latter situation as a solution via exhaustion.

In one of our specific problems, each major task involved the search for a common instance of two logical formulas. If two formulas have such a common instance, they are said to be unifiable. Such a search frequently decomposes into an attempt to show that two subterms have a common instance. If at any point in the computation two subformulas cannot be unified, then the major task is solved (with "failure to unify"). If exhaustion occurs without failure, then a successful unification will have been computed.

One 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. The detection of the end of a computation 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 solution due to exhaustion 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). The objection can be overcome if the master joins
the slaves in working on solutions to the minor tasks. This introduces the syn-
chronization difficulty of reactivating the master when a successful solution has
been detected (since it will quite likely be blocked—probably waiting on the con-
tents 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 back-
tracking is required. That is, the solution of any minor task may introduce con-
straints 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 imposed the restriction
that minor problems may be solved in any order. That is, if two minor tasks can
be solved, they can be solved in either order, or simultaneously.

The logic for the "ask-for-task" operation is subtly complex. We found that
it required a substantial amount of effort to construct a solution that functions
properly. We believe at this point that a great deal of effort is justified to create
and prove the validity of monitors for such prototypical operations, with the
expectation that these operations can be used over and over again. The logic we
used to implement the operation is as follows:

```
ask-for-task: procedure(<returned-task>,<return-code>)
  if ((not program done) and (problem done)) then
    if (other nondelayed processes) then
      delay
    endif
  else
    <return-code> <- "undetermined"
    while ((not program done) and (not problem done)) do
      try to claim a problem
      if (success) then
        continue
      else
        if (last active process)
          set problem done (set code to "exhausted")
        else
          delay
        endif
      endif
  enddo
```
endif
if (program done) then
    <return-code> <- "program done"
    continue
else
    <return-code> <- "problem done (set code)"
    if (no more delayed processes) then
        reset variables (for next problem)
    endif
    continue
endif
end procedure

The creation of this monitor to handle the "pool of outstanding minor tasks" was difficult. However, once we had completed it and used it (twice) in our program that computes common instances of logical formulas, we found that the same synchronization pattern could be used in a program that computed QR factors for matrices. The only difference between the two applications centered on the representation of the "pool of outstanding minor tasks". This difference resulted in application-dependent code to enter a new task into the pool, claim a task from the pool, to initialize the pool, and to reinitialize the pool between major tasks.

Our experiences caused us to believe that this is a very commonly required form of synchronization. Hence, we have coded a macro package for implementing this monitor, as well. To use the package, one must provide only the application-dependent code (in the form of user-supplied macros). All of the synchronization is managed by the standard macros. Appendix F contains the package of macros required to implement this "ask for next task from a pool" type of synchronization. Appendix G contains an example of a program that utilizes the macros. The program computes QR factors. The interested reader will be able to glean a number of useful techniques for programming the HEP from the source code. Appendix H contains the user-supplied macros required to actually compile the program.

6. Event Synchronization

Occasionally it is necessary for a process to pause until another process signals an event. To implement this very simple synchronization mechanism we wrote four macros: pauseinit, pausedec, pause, and pauseevent. To illustrate when these macros are useful, consider the situation where a main program
creates a number of identical "worker" processes. Assume that parameters are to be passed to the worker processes. If the parameters are passed via a common storage location, it will be necessary for a newly-created worker to copy its parameters into local storage before the main program creates the next worker. To accomplish this, the main program would use the pause macro to generate the code to pause. This pause would occur immediately after the creation of the worker. The code in the worker process would contain instructions to first copy the parameters. Then it would use the pauseevent macro to generate the code required to signal the "event" to the waiting main program. The code for these macros is given in Appendix 1.

7. The Implementation of Delay/Continue

The reader should be aware by this point of how most common synchronization primitives can be built up naturally from the few primitive notions included in the implementation of monitors. This careful "pyramiding" (or layering) of the implementation provides a number of advantages. To illustrate, we will discuss the particular implementation on the Denelcor HEP.

The implementation of the primitive notions is included in Appendix A. This implementation works quite well on a single-PEM HEP. However, it can lead to substantial bottlenecks in a multiple-PEM system. Harry Jordan discovered, while working with barrier synchronization, that allowing a large number of processes (say 50-100) spread over several PEMs to wait on a single asynchronous variable can lead to severe congestion in the switch. There is no known way to overcome this difficulty within HEP FORTRAN. That is, neither Jordan nor ourselves know how to implement barrier synchronization without encountering this problem, as long as one is restricted to HEP FORTRAN. Jordan did, however, provide a perfectly reasonable solution based on assembler language routines.

The solution provided by Jordan is an assembler language routine that implements barrier synchronization effectively on the HEP. Similar routines could be created to handle each of the synchronization primitives that we have discussed. Our approach would be somewhat different. By implementing the delay/continue primitives using Jordan's technique, an acceptable solution is automatically obtained for all of the constructs built with monitors. We have not yet written the assembler routines required to solve the problem properly, because a major change in the Denelcor software environment is imminent. We are sure, however, that Jordan's approach does offer a satisfactory approach to implementing delay/continue (i.e., one that overcomes the specific congestion problem discovered by Jordan).
8. Summary

In this document we have described a technique for writing FORTRAN code that is portable between MIMD machines. A set of macros is provided for generating code for the Denelcor HEP, which we regard as the forerunner of wide class of machines. By recoding the basic macros (approximately 100 lines of code), programs written and debugged on the HEP should be ready for recompilation on any new machine with a FORTRAN compiler.

In our opinion the HEP should be viewed as a machine on which multiprocessing algorithms can be developed, debugged, and evaluated. At this stage, it should not be viewed as the ultimate target system. The creation of nonportable HEP code strikes us as a rather speculative venture. It is based on the premise that the ultimate supercomputer (which we believe will be an MIMD machine) will support the peculiar syntax of HEP FORTRAN. While it is quite possible that the HEP II will support the current dialect of FORTRAN and that it will be the ultimate machine, even in this case it might be desirable to create software suitable for execution on alternative machines (without substantial recoding).

We advocate the construction of multiprocessing algorithms formulated using monitors, with machine-independent primitives used in the construction of specific monitors. Beyond this, we believe that it is possible to define a small class of general-purpose monitors that will substantially reduce the coding effort required to implement multiprocessing algorithms. In this report we have attempted to lay the groundwork for such implementation efforts by defining some of the basic synchronization monitors in terms of a portable macro package.

References


Appendix A
Basic Monitor Macro Package

1. The decvar and deccom Macros

The `decvar` and `deccom` macros are used to specify the variables required to implement a monitor. They should be used in any routine containing a monitor operation. Because they generate the COMMON area required for the monitor, it is a good idea to use them in the root module of any process utilizing the shared monitor. Their formats are as follows:

```
decvar(<mon>,<number-of-queues>,<number-of-monitors>)
```

1. `<mon>` gives the two-character "name" of the monitor.
2. `<number-of-queues>` gives the number of delay queues used by the monitor.
3. `<number-of-monitors>` may or may not be present. If it is omitted the variables for a single monitor are generated. If it is included, the variables for an array of identical monitors are generated. This operand would normally be used only in fairly esoteric situations (such as some data flow situations).

```
deccom(<mon>,<number-of-queues>,<number-of-monitors>)
```

1. `<mon>` gives the two-character name of the monitor.
2. `<number-of-queues>` gives the number of delay queues used by the monitor.
3. `<number-of-monitors>` may or may not be omitted. If it is omitted, the variables for a single monitor are generated. If it is included, the variables for an array of identical monitors are generated. This operand would normally be used only in fairly esoteric situations (such as some data flow situations).

As an example, the code to generate the code for a monitor named MO with a single delay queue would be as follows:

```
decvar(MO,1)
deccom(MO,1)
```

The only variables specified by these macros that may be examined by user code (assuming portability is to be maintained) are `<mon>C1, `<mon>C2, etc. These variables contain the number of processes currently delayed in delay-queue 1, delay-queue 2, and so forth. For example, MOC1 would be generated by the above macro invocations. It would be initialized by
moninit(MO,1)

Thereafter, it will always be maintained by the delay and continue operations.

2. The moninit Macro

The moninit macro generates the code to initialize a monitor. Thus,

moninit(MO,1)

could be used to initialize a monitor “named” MO. Note that it only initializes the “standard” fields shared by the monitor. For any monitor that includes user-defined fields, the user will follow a reference to moninit with the statements that initialize the problem-specific data areas. An optional third operand can be used to specify the number of monitors, if an array of monitors is being used. Thus,

moninit(AR,1,10)

could be used to initialize 10 identical monitors, each of which has a single delay-queue.

3. The menter and mexit Macros

The menter and mexit macros are used to generate the entry and exit code for a monitor operation. They normally take a single operand, the name of the monitor. For example,

menter(MO)

mexit(MO)

gives the code for a monitor operation in the monitor “named” MO. If an array of identical monitors is being used, a second operand giving a subscript should be specified. Thus,

menter(AR,1)

mexit(AR,1)
would be used around an operation on AR(I).

4. The delay and continue Macros

The delay macro is used to delay a process in one of the delay-queues associated with the monitor. The process relinquishes ownership of the monitor and remains inactive until a continue is used to reactivate the delayed process. The continue causes a process to immediately leave the monitor operation (i.e., GO TO the first statement past the mexit(<mon>) command). In addition, if there is a delayed process in the designated queue, then an arbitrary single process from the queue is reactivated. No other process could gain access to the monitor between the exit of the process issuing the continue and the reactivation of the delayed process. Thus, control of the monitor is passed from the process issuing the continue to the reactivated process. To see how this works, study the SEND/RECEIVE example given in the body of this paper.

If an array of monitors is being used, a third operand (the subscript) should be included. Thus,

delay(AR,1,1)

would be used to delay the process in the first delay-queue associated with the monitor AR(I).

5. Macro Definitions

Here are the macros themselves in the language of the UNIX m4 macro processor. The clock and create macros are included as basic operations that would normally be used to create a process or access the system clock.

define (endlab,5283) dnl
define (mlabel,4222) dnl
define (changequotu([.]) dnl
define (delay,
    [$1CS2 ifelse($3...(3)) =$1CS2 ifelse($3...(3)) + 1
    $21 ifelse($3...(3)) = 0
    $1DS2 ifelse($3...(3)) =$3DS2 ifelse($3...(3))]
)
define (continue,
    [ IF ($1CS2 ifelse($3...(3)) .EQ. 0) THEN
      $31 ifelse($3...(3)) = 0
    ]
)
ELSE
    $1C$2 ifelse($3,,($3)) = $1C$2 ifelse($3,,($3)) - 1
    $$1D$2 ifelse($3,,($3)) = 0
ENDIF
GO TO endlab]
)
define(menter,
    [$1 ifelse($2,,($2)) = $1 ifelse($2,,($2))]
)
define(mexit,
    [$1 ifelse($2,,($2)) = 0]
[endlab] CONTINUE
    [define([endlab],eval(endlab+1))]
)
define(decvar,
    [INTEGER $1LOC [], $1 ifelse($3,,($3)) [], $1 ifelse($3,,($3))
        decvgen(31,82,33,1)]
)
define(decvgen,
    [ifelse($2,,
        ifelse(eval($2 < $4),1,,
            [INTEGER ] $1D$4 ifelse($3,,($3)) [], $1D$4 ifelse($3,,($3))
            - [], $1C$4 ifelse($3,,($3))
        )
    )decvgen(31,52,33,eval($4 + 1))]]]
)
define(deccom,
    [COMMON /$1COM/ $1 deccgen($1,$2,$3,eval($4 + 1))]]
)
define(deccgen,
    [ifelse($2,,
        ifelse(eval($2 < $4),1,,
            [.] $1C$4 deccgen($1,$2,$3,eval($4 + 1))]]]
)
define(moninit,
    [ifelse($3,,
        PURGE $1;
        $1 = 0
        purgen($1,$2,1),
        DO mlabel $1LOC=1[,,]$3
        PURGE $1($1LOC)
$S1(S1LOC) = 0$
purgen($S1, S2, 1, (S1LOC))

mlabel    CONTINUE
    define([mlabel], eval(mlabel + 1)))
)
define(purgen,    [ifelse($2,,,
    [ifelse(eval($2 < $3),1,,
    PURGE $S1$D$S3$S4
    $S1C3$S4 = 0
    [purgen($1, $2, eval($3 + 1), $4)])])])
)
define(create,    [CREATE $1]
)
define(clock,    [CALL CLOCK($1)]
)
define(env,    [SUBROUTINE CANCEL(FMAX)
    DIMENSION A(1)
    I = FMAX
    clock(1START)
    10 CONTINUE
    clock(1STOP)
    IF (FLOAT(1STOP - 1START) .GE. FMAX) B = A(I)
    GO TO 10

C
C     LAST CARD OF CANCEL
C
     END]
)
define(initenv,    [READ (5, 1111) MINITS
  1111 FORMAT (13)
    IF (MINITS .GT. 0) THEN
      WRITE(6, 1112) MINITS
  1112 FORMAT ('TIMEBOMB SET FOR .13,. MINUTES')

C
C    THIS CREATE CANCEL(FMAX) WILL STOP THE PROGRAM
C    IF NUMBER OF CLOCK CYCLES EXCEEDS FMAX
C

FMAX = MINITS*60.0E7
create(CANCEL(FMAX))

ELSE

WRITE(6,1113)

1113 FORMAT(' TIMEBOMB NOT SET; BE VERY CAREFUL. ')

ENDIF

)
Appendix B  
Send/Receive Monitor Macro Package

1. The srdec Macro

The srdec macro can be used to generate the specifications for the variables required to support the Send/Receive monitor operations. It should be used in the root of any process that uses the monitor, as well as in any routine that actually invokes a send or receive operation. Thus,

```
srdec(SR)
```

could be used to declare the variables to support a send/receive monitor "named" SR. To use an array of send/receive monitors (which is useful in some data flow applications), one can specify the size of the array with a second argument. Thus,

```
srdec(AR,10)
```

would declare an array of 10 send/receive monitors.

2. Initializing a Send/Receive Monitor

A send/receive monitor can be initialized by the code generated by the srinit macro. The monitor should be initialized once before any send/receive operations are performed. As an example,

```
srinit(SR)
```

could be used to initialize the SR monitor. If an array of send/receive monitors is being used, a second argument giving the size of the array should be included. Thus,

```
srinit(AR,10)
```

would initialize 10 identical send/receive monitors.
3. The Send/Receive Operations

Assuming that the SR monitor has been declared and initialized, a message may be "sent" (i.e., placed into the shared buffer, which must be part of an application-dependent COMMON area) by using the code generated by the \texttt{srsend} macro. The first operand is just the monitor name. The second operand must be a macro invocation (of a user-defined macro) that will generate the code required to move the message into the shared buffer. We require the user to create such a macro, because the code required to move the message depends on the data items that together are treated as a single message. Thus,

\begin{verbatim}
    srsend(SR,movein(I,J,MYVAL))
\end{verbatim}

could be used to move a message into the shared buffer, assuming that

\begin{verbatim}
    movein(I,J,MYVAL)
\end{verbatim}

will generate the appropriate code to move data into the buffer. Similarly,

\begin{verbatim}
    srrec(SR,moveout(I,J,MYVAL))
\end{verbatim}

might be used to "receive" the data from the buffer. Again, the second operand must generate the code required to move the data.

To use an array of send/receive monitors, use a third operand as a subscript into the array (for both the \texttt{srsend} and \texttt{srrec} macros). Thus,

\begin{verbatim}
    srsend(SR,movein(I,J,MYVAL),K)
\end{verbatim}

would generate a send operation against the Kth send/receive monitor.

4. Macro Definitions

\begin{verbatim}
define(srinit, [moninit($1.2.3) $1FL = .FALSE.] )
define(srddec, [decvar($1.2.3) ]
\end{verbatim}
LOGICAL $1$FL ifelse($2\ldots($2))
deccom($1,2,.$2)
- . $1$FL]
)
define(srsend,
    [menter($1$3)
     IF ($1$FL ifelse($3\ldots($3)) ) THEN
      delay($1,1,$3)
    ENDIF
$2
$1$FL ifelse($3\ldots($3)) = .TRUE.
continue($1,2,$3)
exit($1,$3)]
)
define(srrec,
    [menter($1$3)
     IF (.NOT. $1$FL ifelse($3\ldots($3))) THEN
      delay($1,2,$3)
    ENDIF
$2
$1$FL ifelse($3\ldots($3)) = .FALSE.
continue($1,1,$3)
exit($1,$3)]
)
Appendix C
Barrier Synchronization Macros

1. The bardec Macro
The bardec macro can be used to generate the code to support a barrier monitor. Thus,

    bardec(B1)

could be used to declare a barrier "named" B1.

2. The barrier Macro
The barrier macro generates the code to act as a barrier. The first operand is the monitor name, and the second gives the number of processes to be held at the barrier (actually, this value minus one are held; the last process to arrive just moves through, releasing the blocked processes). Thus,

    barrier(B1,NPROC)

would be used to generate a barrier that will cause (NPROC - 1) processes to pause until the last process reaches the barrier. Then they will all be released. Normally, all of the cooperating processes will be processing an identical program segment, which will include the code generated by the barrier macro. This need not be the case, however. As long as all of the programs use a commonly declared monitor, the barrier operation can be performed from distinct program segments.

3. Macro Definitions

    define(barrier,
            [mandatory($1)
             IF ($1C1 .LT. ($2 - 1)) THEN
                 delay($1,1)
             ENDIF
             continue($1,1)
             exit($1)]
           )
define(bardec,
    [decvart($1,1)
     deccan($1,1)]
  )
define(barinit,
    [moninit($1,1)]
  )
Appendix D
Macros for the Self-scheduling DO-loop Monitor

1. The gsdec Macro
The gsdec macro can be to declare and generate the variables required to support a monitor used for a "self-scheduling DO-loop". (gs stands for "get subscript"). The only operand is the name of the monitor. Thus,

    gsdec(GS)

appears in the module that initialized the monitor, as well as in the module containing the self-scheduling DO-loop.

2. The gsinit Macro
The gsinit is used to initialize a monitor for managing a self-scheduling DO-loop. For example,

    gsinit(GS)

would generate the code to initialize the monitor GS.

3. The getsub Macro
The getsub macro is used to claim a subscript in a self-scheduling DO-loop. For example,

    getsub(GS,1,N,NPROC)

sets I to the "next" subscript in the range 1 through N, where NPROC is the number of processes competing for subscripts. I will be set to 0, if all of the subscript values have been processed.

4. Macro Definitions

    define(gsdec,
        [deccrn($1,1)
         - , $1SB
        deccrn($1,1)
define(gsinit,
    [moninit($1,1)
    $1SB = 1]
)
define(getsub,
    [menter($1)
    IF ($1SB .LE. 3) THEN
        $2 = $1SB
        $1SB = $1SB + 1
    ELSE
        $2 = 0
        IF ($1C1 .LT. ($4 - 1)) THEN
            delay($1,1)
        ENDIF
    ENDIF
    $1SB = 1
    continue($1,1)
    ENDIF
    mexit($1)]
)
Appendix E
An Example Illustrating Barriers and Self-scheduling DO-loops

The following program to sort an array using the Shell sort algorithm provides examples of both the barrier and self-scheduling DO-loop synchronization patterns.

```
define (mode, REAL)

* THIS PROGRAM DEMONSTRATES THE "BARRIER" AND "SELF-SCHEDULING DO-LOOP"
* SYNCHRONIZATION PRIMITIVES. IT FILLS IN A VECTOR (A) WITH VALUES IN
* DESCENDING ORDER. THEN IT USES A SHELL SORT (SEE KNUTH'S 3RD VOLUME
* ON SORTING AND SEARCHING ALGORITHMS) TO SORT THE VALUES INTO
* ASCENDING ORDER. TIMES ARE ACQUIRED FOR TABLE SIZES OF 100, 1000, AND
* 10000.
*
* PROGRAM SRTPGM
*
* COMMON AREA VARIABLES
*
* mode A(10000)
INTEGER NPROCS, N, INC
LOGICAL PGDONE
COMMON /MAINC/ INC, A, PGDONE, N, •NPROCS
*
gsdec(GS)
bardec(B1)
*
INTEGER I, J
*
* INITIALIZE THE ENVIRONMENT (ON HEP THIS MEANS SET THE TIMEBOMB)
*
* \texttt{\_initenv} *

* INITIALIZE THE BARRIER AND SELF-SCHEDULING DO-LOOP MONITORS *

* \texttt{gsinit(GS)} \texttt{barinit(B1)} *

\texttt{PGDONE = .FALSE.}

* READ IN THE NUMBER OF PROCESSES TO RUN IN PARALLEL *

* \texttt{READ (5,10) NPROCS} \texttt{10 FORMAT(14)} \texttt{WRITE(6,20) NPROCS} \texttt{20 FORMAT('NPROCS =',14)} *

* \texttt{DO 15 i=1,NPROCS-1} \texttt{create(SLAVE)} \texttt{15 CONTINUE} *

* THE MAIN LOGIC JUST FILLS IN THE TABLE AND SORTS IT. TIMINGS ARE TAKEN FOR TABLES OF 100, 1000, AND 10000. *

* \texttt{N = 10}
DO 100 I=1,3
   N = 10 * N
   CALL FILL

   IF (I .EQ. 1) THEN
      DO 101 J=1,N-4,5
         WRITE (6,30) (A(K), K=J, J+4)
      30 FORMAT(E12.5,' ',E12.5,' ',E12.5,' ',E12.5,' ',E12.5)
      101 CONTINUE
   ENDIF

   clock(J)
   T1 = J

   CALL SORT

   clock(J)
   T2 = J - T1
   WRITE(6,110) N, T2
   110 FORMAT('SIZE = ',I5,' TOTAL TIME = ',E12.5)
   IF (I .EQ. 1) THEN
      DO 102 J=1,N-4,5
         WRITE (6,30) (A(K), K=J, J+4)
      102 CONTINUE
   ENDIF
100 CONTINUE

ONE LAST CALL TO LOOP IS REQUIRED TO FREE THE OTHER PROCESSES FROM THE BARRIER (SO THEY CAN EXIT).

PGDONE = .TRUE.
CALL LOOP(Q)
STOP
END
THE FOLLOWING LITTLE ROUTINE JUST FILLS THE VECTOR WITH VALUES IN DESCENDING ORDER.

SUBROUTINE FILL

INTEGER NPROCS, N, INC
LOGICAL PGDONE
COMMON /MAININC/ INC, A, PGDONE, N, NPROCS

INTEGER I

DO 10 I=1,N
    A(I) = (N - I) + 1.0
10 CONTINUE
RETURN
END

THE SLAVE PROCESSES JUST HANG ON THE BARRIER IN THE "LOOP" AND HELP WHEN A TABLE IS TO BE SORTED.

SUBROUTINE SLAVE

CALL LOOP(1)
RETURN
END

THE SORT ROUTINE IS EXECUTED BY THE MASTER PROCESS. IT JUST CALCULATES THE RADIX FOR EACH PASS OF THE SHELL SORT, AND JOINS
THE SLAVE PROCESSES WHEN WORKING ON EACH PASS.

THE RADIX VALUES ARE HT, ..., H2, H1: H1 IS 1; H1 IS (3*H(I-1) + 1);

H(T+2) >= N. SEE KNUTH FOR ARGUMENTS IN FAVOR OF THESE VALUES.

SUBROUTINE SORT

INTEGER A(10000)
INTEGER NPROCS, N, INC
LOGICAL PDONE
COMMON /MA/ INC, A, PGDONE, N, NPROCS

I1 = 1
I2 = (I1 * 3) + 1
I3 = (I2 * 3) + 1

10 CONTINUE
IF (I3 .GE. N) GO TO 20
   I1 = 12
   I2 = 13
   I3 = (I2 * 3) + 1
   GO TO 10

20 CONTINUE

INC = I1

30 CONTINUE
IF (INC .LE. 0) GO TO 90

CALL LOOP(0)
   INC = (INC - 1) / 3
   GO TO 30

90 CONTINUE
RETURN
END
THE LOOP ROUTINE IS THE CODE REQUIRED TO COORDINATE THE NPROCS
PROCESSES AS THEY EXECUTE ONE PASS OF A SHELL SORT. NOTE THE
BARRIER AT THE TOP, WHICH IS USED TO CAUSE THE PROCESSES TO
WAIT FOR THE VECTOR TO BE SET UP AND THE INCREMENT CHOSEN.
THEN A SELF-SCHEDULING DO-LOOP IS USED TO ALLOCATE SUBSCRIPTS.
NOTE THAT THE MASTER PARTICIPATES IN THIS LOGIC, SO THE PROGRAM
CAN BE RUN WITH NPROCS SET TO 1.

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

SUBROUTINE LOOP(WHO)
INTEGER WHO

mode A(10000)
INTEGER NPROCS, N, INC
LOGICAL PGDONE
COMMON /MAINC/ INC, A, PGDONE, N, NPROCS

LOGICAL DONE
mode T

bardec(B1)
gsdec(GS)

10 CONTINUE
barrier(B1,NPROCS)
IF (PGDONE) GO TO 90

20 CONTINUE
getsub(GSJ,INC,NPROCS)
IF (J EQ. 0) GO TO 80

K = J + INC
40 CONTINUE
IF (K .GT. N) GO TO 70
K1 = K - INC
DONE = .FALSE.
50 CONTINUE
IF (DONE .OR. (K1 .LT. 1)) GO TO 60
IF (A(K1) .LE. A(K1+INC)) THEN
DONE = .TRUE.
ELSE
  T = A(K1)
  A(K1) = A(K1+INC)
  A(K1+INC) = T
  K1 = K1 - INC
ENDIF
GO TO 50
60 CONTINUE
  K = K + INC
  GO TO 40
70 CONTINUE
  GO TO 20
80 CONTINUE
  IF (WHO .EQ. 1) GO TO 10
90 CONTINUE
  RETURN
END

THE FOLLOWING MACRO GENERATES THE ROUTINES REQUIRED AS PART
OF THE "ENVIRONMENT". ON THE HEP THIS AMOUNTS TO THE TIMEBOMB
SUBROUTINE, WHICH CAUSES ABNORMAL TERMINATION AFTER SOME
SPECIFIED NUMBER OF MINUTES (A VALUE THAT GETS READ FROM AN
INPUT RECORD.

env
Appendix F
Macros to Implement the "Ask for Next Task" Monitor

1. The adec Macro
The adec macro is used to declare and generate the variables required to support an "ask for next task" monitor. A task here has no relation to the HEP concept of task. Rather, a task is a subproblem as described in Section 5. The only operand gives the name of the monitor. For example,

    adec(MO)

would be used for a monitor "named" MO.

2. The ainit Macro
The ainit macro is used to generate the code required to initialize an "ask for next task" monitor. Thus,

    ainit(MO)

could be used to initialize the monitor.

3. The askfor Macro
The askfor macro is used to generate the code required to "ask for the next task". The first operand gives the name of the monitor. The second operand gives a variable that is set to reflect the outcome of the request. A value of 0 reflects a "task" has been successfully claimed; a value of -1 means that there will be no more tasks (end-of-program has been signaled via a progend operation); a value of 1 means that the current pool of tasks was exhausted (which represents an end-of-problem condition); any value greater than 1 represents an end-of-problem value set via a probend operation. The third operand gives the number of processes claiming tasks from the pool (this value is used to detect a solution via exhaustion of the pool -- if the pool is empty and all processes are waiting, there is a solution due to exhaustion of the pool). The fourth operand is a user-defined macro invocation that will generate the code required to claim a problem from the pool. This generated code should inspect the user-defined variables required to manage the pool. If a task can be claimed, the return code (second operand) should be set to 0. Finally, the last operand is a user-defined macro invocation to generate the code required to reinitialize the pool of tasks. Thus,
might be coded to claim a task. Here "checkpool" and "reset" are user-defined
macros that must generate the code to "check for an available task" and "reset
the pool of tasks", respectively.

4. The progend Macro

The *progend* macro is used to signal end-of-program. This will cause all
processes to receive a -1 return code on subsequent "askfor" operations. The
only operand is the name of the monitor. Thus,

```
progend(MO)
```

would be used to signal "no more major tasks".

5. The probend Macro

The *probend* macro is used to indicate that a major task has been solved. The
first operand is the name of the monitor. The second is a "problem end code"
that should be greater than 1 (1 is used to reflect solution due to exhaustion of
the pool of tasks). Thus,

```
probend(MO,2)
```

indicates an "end of problem with return code 2". An end-of-problem condition
will cause each process to receive the given return code on the next "askfor"
operation. After all processes have received the return code, the user-defined
code to reset the pool of tasks will be executed.

6. Macro Definitions

```c
define(alabel,2222) dnl
define(blabel,3222) dnl
define(ainit,
    [moninit($1,1)
    $1PG = 0
    $1PB = 0]
)
define(adec,
```
\begin{verbatim}
[decvar($1,1)
  - , $1PG , $1PB
deccom($1,1)
  - , $1PG , $1PB}
)

define(askfor,
  [menter($1),
   IF' (($1PG .EQ. 0) .AND. ($1PB .NE. 0)) THEN
    IF ($1C1 .LT. ($3 - 1)) THEN
      delay($1,1)
    ENDIF
  ELSE
    $2 = -2
  ENDIF
  alabel CONTINUE;
   IF (($1PG .NE. 0) .OR. ($1PB .NE. 0)) THEN
    GO TO blabel
  ENDIF

$4
  IF ($2 .EQ. 0) THEN
    continue($1,1)
  ELSE
    IF ($1C1 .EQ. ($3 - 1)) THEN
      $1PB = 1
    ELSE
      delay($1,1)
    ENDIF
  ENDIF
  ENDIF
  GO TO alabel
blabel CONTINUE
  define([alabel],eval(alabel+1))
define([blabel],eval(blabel+1))
ENDIF

  IF ($1PG .NE. 0) THEN
    $2 = -1
    continue($1,1)
\end{verbatim}
ELSE

\$2 = \$1PB

IF (\$1C1 .EQ. 0) THEN

\$5

\$1PB = 0

ENDIF

continue(\$1,1)

ENDIF

mexit(\$1)]

)
define (probend,

[\menter(\$1)

\$1PB = \$2

mexit(\$1)]

)
define (progend,

[\menter(\$1)

\$1PG = -1

continue(\$1,1)

mexit(\$1)]

)
Appendix G

An Example Illustrating the "ask-for-task" Monitor

Here we give an example which illustrates the use of the macros just described. It is an algorithm to perform QR-factorization of a matrix. The problem-specific macros are given in the next Appendix.

* THE WORK SUBROUTINE *

SUBROUTINE WORK(FLAG)
   INTEGER FLAG

   REAL A(301,300), B(301)
   INTEGER NPROC, LDA, M, N
   COMMON /MAINC/ A, B, N, M, LDA, NPROC

   INTEGER ITAG(1000), LSTREF, PGST, PBST, NXTSUB, NXTSTR
   COMMON /SYNC/ ITAG, LSTREF, PGST, PBST, NXTSUB, NXTSTR

   adec(ST)

   INTEGER I

   INTEGER RC, K, L, TRC

* DECLARATIONS FOR CREF AND APREF *

   REAL ZERO, TAU
   INTEGER NK, KM1
   REAL ENORM
   REAL THETA
   DATA ZERO/0.0/
askfor(ST,RC,(NPROC+1),getprob(N,RC,K,L,TRC),reset(N))

* RC = 0 MEANS THAT IT GOT A PROBLEM
* RC = 1 MEANS THE PROBLEM IS COMPLETED
* RC = -1 MEANS THE PROGRAM IS COMPLETED

* *** IF RC = 0, THEN
  *
  * TRC = 0 MEANS CREATE A REFLECTION IN COLUMN K
  * TRC = 1 MEANS APPLY REFLECTION L TO COL. K
  *
  * N IS THE NUMBER OF COLUMNS IN THE MATRIX
  *
  * K IS SET TO THE COLUMN UPON WHICH A REFLECTION IS TO BE
  * CREATED OR APPLIED
  *
  * L IS MEANINGFUL ONLY WITH AN RC OF 1 (APPLY A REFLECTION).
  * IT THEN GIVES THE REFLECTION NUMBER TO APPLY
  *
  * while (you got a task to do)
  *
10 IF (RC .NE. 0) GO TO 100
   IF (TRC .EQ. 0) THEN

                      CREATE THE REFLECTOR FOR THE K-TH COLUMN

C KM1 = K - 1
NK = N - K + 1

C NOW COMPUTE AND STORE THE K-TH REFLECTOR
C
  TAU = ENORM(NK,A(K,K))
  TAU = SIGN(TAU,A(K,K))
  B(K) = -TAU
  A(K,K) = A(K,K) + TAU
NOW SIGNAL THAT THE REFLECTION HAS BEEN CREATED

DONE CRT
IF (K EQ. N) THEN
    PROBEND(ST, !)
ENDIF

NOW GET THE NEXT TASK

GO TO 5
ELSE

APPLY THE NEXT REFLECTION (THE L-TH)
TO THE K-TH COLUMN

THETA = ZERO
DO 50 I = 1, M
    THETA = THETA + A(I, K)*A(I, I)
50 CONTINUE
THETA = THETA/(B(I)*A(I, I))
DO 60 I = 1, M
    A(I, K) = A(I, K) + THETA*A(I, I)
60 CONTINUE

DONE R(T, TRC, L, )

TRC = 0 MEANS CREATE A REFLECTION IN COLUMN K
TRC = 1 MEANS APPLY REFLECTION L TO COL. K
TRC = 2 MEANS GET ANOTHER TASK

IF (TRC EQ. 2) THEN
   GO TO 5
ELSE
GO TO 10
END IF
END IF

100 CONTINUE
IF ((RC .EQ. 1) .AND. (FLAG .EQ. 1)) GO TO 5
RETURN
END

SUBROUTINE QRMON

REAL A(301,300), B(301)
INTEGER NPROC, LDA, M, N
COMMON /MAINC/ A, D, N, M, I.DA, NPROC

INTEGER ITAG(1000), LSTREF, PGST, PBST, NXSUB, NXSSTR
COMMON /SYNC/ ITAG, LSTREF, PGST, PBST, NXSUB, NXSSTR

adec(ST)
probstart
CALL WORK(0)
RETURN

C
LAST CARD OF QRMON
C
END

Q SLAVE
SUBROUTINE QSLAVE
REAL A(301,300), B(301)
INTEGER NPROC, LDA, M, N
COMMON /MAINC/ A, B, N, M, LDA, NPROC

INTEGER ITAG(1000), LSTREF, PGST, PBST, NXTSUB, NXTSTR
COMMON /SYNC/ ITAG, LSTREF, PGST, PBST, NXTSUB, NXTSTR

adec(ST)

CALL WORK(1)
RETURN
END

THE MAIN LOGIC

REAL A(301,300), AA(301,300), B(301), MFLOPS
INTEGER WSIZE, NPROC, LDA, M, N, I
COMMON /MAINC/ A, B, N, M, LDA, NPROC

INTEGER ITAG(1000), LSTREF, PGST, PBST, NXTSUB, NXTSTR
COMMON /SYNC/ ITAG, LSTREF, PGST, PBST, NXTSUB, NXTSTR

adec(ST)

initenv

stkinit
ainit(ST)

NOW CREATE THE QRMON WORKERS

READ (5, 1111) NPROC
DO 600 I = 1, NPROC
  create(QSLAVE)
  600
\begin{verbatim}
600 CONTINUE
*
*
LDA = 301

C

WRITE(6,40)
40 FORMAT(' QRFAx DECOMPOSITION TIMING')
DO 200 N = 50,300,50
   DO 20 J = 1,N
      DO 10 I = J,N
         AA(I,J) = -I*J
         AA(J,1) = 2*AA(1,J)
      10 CONTINUE
      AA(J,J) = 0.0
   20 CONTINUE
WRITE(6,50)LDA,N
50 FORMAT(/' SP SIZE OF THE ARRAYS',15,' AND ORDER IS ',15/)
DO 70 J = 1,N
   DO 60 I = 1,N
      A(I,J) = AA(I,J)
   60 CONTINUE
70 CONTINUE
*

DO 103 J = 1,N
   DO 102 I = 1,N
      A(I,J) = AA(I,J)
102 CONTINUE
103 CONTINUE
WSIZE = 3
clock(I)
T1 = I
*
M = N
CALL QRMON
*
clock(I)
T2 = 1 - T1
MFLOPS = ((4.*FN**3)/3. + (3.*FN**2)/2. + FN/6.)/T2/1.0E6
WRITE(6,110) T2,MFLOPS
110 FORMAT(/' TIMING FOR WSIZE = 3, N = ',I5,
       
IF( N .LE. 50 ) WRITE(6,1000) (B(I),I = 1,N)
1000 FORMAT(5X,E12.5)
\end{verbatim}
DO 113 J = 1,N
   DO 112 I = 1,N
      A(I,J) = AA(I,J)
   112 CONTINUE
113 CONTINUE
200 CONTINUE
}
progend(ST)
STOP
END

REAL FUNCTION ENORM(N,X)
INTEGER N
REAL X(N)

********

FUNCTION ENORM

GIVEN AN N-VECTOR X, THIS FUNCTION CALCULATES THE
EUCLIDEAN NORM OF X.

THE EUCLIDEAN NORM IS COMPUTED BY ACCUMULATING THE SUM OF
SQUARES IN THREE DIFFERENT SUMS. THE SUMS OF SQUARES FOR THE
SMALL AND LARGE COMPONENTS ARE SCALED SO THAT NO OVERFLOWS
OCUR. NON-DESTRUCTIVE UNDERFLOWS ARE PERMITTED. UNDERFLOWS
AND OVERFLOWS DO NOT OCCUR IN THE COMPUTATION OF THE UNSCALED
SUM OF SQUARES FOR THE INTERMEDIATE COMPONENTS.

THE DEFINITIONS OF SMALL, INTERMEDIATE AND LARGE COMPONENTS
DEPEND ON TWO CONSTANTS, RDWARF AND RG1ANT. THE MAIN
RESTRICTIONS ON THESE CONSTANTS ARE THAT RDWARF**2 NOT
UNDERFLOW AND RG1ANT**2 NOT OVERFLOW. THE CONSTANTS
GIVEN HERE ARE SUITABLE FOR EVERY KNOWN COMPUTER.

THE FUNCTION STATEMENT IS

REAL FUNCTION ENORM(N,X)

WHERE

N IS A POSITIVE INTEGER INPUT VARIABLE.
X is an input array of length N.

Subprograms called

Portran-supplied ... ABS, SQRT

Burton S. Garbow, Kenneth E. Hillstrom, Jorge J. More

**********

INTEGER I
REAL AGIANT, FLOATN, ONE, RDWARF, RG1ANT, S1, S2, S3, XABS,
*   X1MAX, X3MAX, ZERO
DATA ONE, ZERO, RDWARF, RG1ANT /1.0E0, 0.0E0, .294E-38, .17E39/
S1 = ZERO
S2 = ZERO
S3 = ZERO
X1MAX = ZERO
X3MAX = ZERO
FLOATN = N
AGIANT = RG1ANT/FLOATN
DO 90 I = 1, N
   XABS = ABS(X(I))
   IF (XABS .GT. RDWARF .AND. XABS .LT. AGIANT) GO TO 70
   IF (XABS .LE. RDWARF) GO TO 30

SUM FOR LARGE COMPONENTS.

IF (XABS .LE. X1MAX) GO TO 10
   S1 = ONE + S1*(X1MAX/XABS)**2
   X1MAX = XABS
   GO TO 20
10 CONTINUE
   S1 = S1 + (XABS/X1MAX)**2
20 CONTINUE
GO TO 60
30 CONTINUE

SUM FOR SMALL COMPONENTS.

IF (XABS .LE. X3MAX) GO TO 40
S3 = ONE + S3*(X3MAX/XABS)**2
X3MAX = XABS
GO TO 50
40 CONTINUE
   IF (XABS .NE. ZERO) S3 = S3 + (XABS/X3MAX)**2
50 CONTINUE
60 CONTINUE
   GO TO 60
70 CONTINUE

C C SUM FOR INTERMEDIATE COMPONENTS.
C
S2 = S2 + XABS**2
80 CONTINUE
90 CONTINUE

C C CALCULATION OF NORM
C
   IF (S1 .EQ. ZERO) GO TO 100
   ENORM = X1MAX*SQR((S1+(S2/X1MAX))/X1MAX)
   GO TO 130
100 CONTINUE
   IF (S2 .EQ. ZERO) GO TO 110
      IF (S2 .GE. X3MAX)
 *     ENORM = SQR((S2*(ONE+(X3MAX/S2)*(X3MAX*S3)))
      IF (S2 .LT. X3MAX)
 *     ENORM = SQR(X3MAX*((S2/X3MAX)+(X3MAX*S3)))
   GO TO 120
110 CONTINUE
    ENORM = X3MAX*SQR(S3)
120 CONTINUE
130 CONTINUE
RETURN

C C LAST CARD OF FUNCTION ENORM.
C
END

env
Appendix H
Macros Specific to QR-factorization Program

define(reset,
     [NXTSUB = 1
     DO 704 LSTREF = 1[,] $1
        ITAG(LSTREF) = 0
    704 CONTINUE
     LSTREF = -1]
     )

define(stkinit,
     [moninit(ST,1)
     DO 700 LSTREF = 1[,] 1000
        ITAG(LSTREF) = 0
    700 CONTINUE
     LSTREF = -1
     NXTSUB = 1]
     )

define(getprob,
     [IF (LSTREF .GE. 0) THEN

        IF (ITAG(LSTREF + 1) .EQ. LSTREF) THEN
           $2 = 0
           $5 = 0
           $3 = LSTREF + 1
           ITAG(LSTREF+1) = -(LSTREF+1)
        ELSE
           NXTSTR = NXTSUB
    702 CONTINUE
        IF (NXTSTR .EQ. 0) GO TO 703
        IF ((ITAG(NXTSUB) .LT. LSTREF) .AND.
           - (ITAG(NXTSUB) .GE. 0)) GO TO 703

        IF (NXTSUB .LT. $1) THEN
           NXTSUB = NXTSUB + 1
        ELSE
           NXTSUB = LSTREF + 1
        ENDIF
     ENDIF]
IF (NXTSUB .EQ. NXTSTR) NXTSTR = 0
GO TO 702

703 CONTINUE
IF (NXTSTR .NE. 0) THEN
  $2 = 0
  $3 = NXTSUB
  $4 = ITAG(NXTSUB) + 1
  ITAG(NXTSUB) = -$4
  $5 = 1
ENDIF
ENDIF
END IF

define(doneref,
[menter(ST)
  ITAG($1) = -ITAG($1)
  IF (ITAG(LSTREF+1) .EQ. LSTREF) THEN
    $1 = LSTREF + 1
    $2 = 0
    ITAG(LSTREF+1) = -(LSTREF+1)
  ELSE
    IF (ITAG($1) .LT. LSTREF) THEN
      $2 = 1
      $3 = ITAG($1) + 1
      ITAG($1) = -$3
    ELSE
      $2 = 2
    END IF
  END IF
ENDIF
mexit(ST)]
)

define(donecrt,
[menter(ST)
  LSTREF = LSTREF + 1
  ITAG(LSTREF) = LSTREF
  IF (NXTSUB .LE. LSTREF) NXTSUB = LSTREF + 1
  continue(ST, 1)
  mexit(ST)]
)

define(probstart,
[menter(ST)
LSTREF = 0
continue(ST,1)
mexit(ST)]
)

Appendix I
Macros for Pause and Event Synchronization

1. The pausedecl Macro

The pausedecl macro can be to declare and generate the variables required to support a monitor used for pause/event synchronization. There is one required operand, the name of the monitor. If an array of similar monitors is required, the second operand can be used to give the number of monitors that will be used. Thus,

pausedecl(PE)

in the module that initialized the monitor, as well as in the module containing the self-scheduling DO-loop.

2. The pauseinit Macro

The pauseinit is used to initialize a monitor for managing pause/event synchronization. For example,

pauseinit(PE)

would generate the code to initialize the PE monitor.

3. The pause Macro

The pause macro is used to pause for an event to be signaled. For example,

pause(PA)

would cause execution to be suspended until the event is signaled by another process. The event can be signaled before the pause occurs, but an attempt to signal twice before a pause occurs will cause the signaling process to be suspended (until the pause occurs). If an array of similar monitors is being used (where the subscript would indicate an event type), a second argument giving the subscript should be included.
4. The \texttt{pauseevent} Macro

The \texttt{pauseevent} macro is used to pause for an event to be signaled. For example,

\begin{verbatim}
pauseevent(\texttt{PA})
\end{verbatim}

would signal an event. The event can be signaled before the pause occurs, but an attempt to signal twice before a pause occurs will cause the signaling process to be suspended (until the pause occurs). If an array of similar monitors is being used (where the subscript would indicate an event type), a second argument giving the subscript should be included.

5. Macro Definitions

\begin{verbatim}
define(pauseinit, 
   [\texttt{moninit($1,2,$2)}
   \texttt{ifelse($2,$1,$1) = .FALSE. ,}
   \texttt{DO mlabel $1:LOC=1[,]$2}
   \texttt{$1$FL($1:LOC) = .FALSE.}
   \texttt{mlabel CONTINUE}
   \texttt{define([mlabel],eval(mlabel+1))}]
]

define(pausevec,
   [\texttt{decover($1,2,$2)}
   \texttt{LOGICAL $1$FL ifelse($2,..($2))}
   \texttt{decover($1,2,$2)}
   \texttt{- . $1$FL}]
]

define(pauseevent,
   [\texttt{enter($1,$2)}
   \texttt{IF ($1$FL ifelse($2,..($2)) ) THEN}
   \texttt{delay($1,1,$2)}
   \texttt{ENDIF}
   \texttt{$1$FL ifelse($2,..($2)) = .TRUE.}
   \texttt{continue($1,2,$2)}
   \texttt{mexit($1,$2)}]
]

define(pause,
\end{verbatim}
IF (.NOT. $1FL ifelse($2,$2)) THEN
    delay($1,2,$2)
ENDIF

$1FL ifelse($2,$2) = .FALSE.
continue($1,1,$2)
rmexit($1,$2)
Distribution for ANL-83-97 Rev. 1

Internal:

K. L. Kliever
A. B. Krisciunas
E. I. Lusk (91)
P. C. Messina
R. A. Overbeek (91)
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 (173)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
J. L. Bona, U. Chicago
J. C. Browne, U. Texas, Austin
S. Gerhart, Wang Institute, Tyngsboro, MA
G. H. Golub, Stanford U.
W. C. Lynch, Xerox Corp, Palo Alto
M. F. Wheeler, Rice U.
L. P. Kadanoff, U. of Chicago (Board of Governors liaison)
D. Austin, Office of Basic Energy Sciences, DOE
G. Michael, U.M.