Use of Monitors in Pascal on the Lemur:
A Tutorial on the Barrier,
Self-scheduling FOR-Loop,
and Askfor Monitors

J. A. Clausing, R. T. Hagstrom, E. L. Lusk, and R. A. Overbeck
Mathematics and Computer Science Division

July 1984

*High Energy Physics Division
Use of Monitors in Pascal on the Lemur:

A Tutorial on the Barrier,
Self-scheduling FOR-Loop,
and Askfor Monitors

J. A. Clausing
R. Hagstrom
E. L. Lusk
R. A. Overbeek

ABSTRACT

A set of macro libraries has been developed that allows programmers to write portable Pascal code for multiprocessors. This document presents, in tutorial form, the macros used to implement three common synchronization patterns: self-scheduling FOR-loops, barrier synchronization, and the askfor monitor.

1. Introduction

Monitors have been used for many years as a basis for process synchronization[1, 3, 4, 5, 6, 10]. After some experimentation, we have found that portable programs for use on multiprocessors can be written using monitors implemented by means of macros. Furthermore, once a programmer is familiar with the use of the macros, they offer a relatively convenient mechanism for writing programs that require the synchronization of multiple processes.

Two of our earlier publications[8, 9] discussed the basic concept of macros and briefly described their use. We are now writing a series of tutorials that more thoroughly explain how to use our macros to implement a variety of common synchronization patterns. This document is the first of the series and is

This work was supported in part by National Science Foundation grant MCS83-07496 and in part 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.
intended for anyone who wishes to write portable Pascal for multiprocessors. Our earlier reports are not prerequisites to this document, but can be accessed for more of the implementation details. A similar tutorial has been written to aid those wishing to code in FORTRAN[7].

One problem faced by a person beginning to code for multiprocessors is a choice of "synchronization primitives". A large number of primitives have grown up from applications in systems programming, telecommunications, and shared databases. The set of proposed alternatives may well appear bewildering to the uninitiated, and arguments between advocates of the different primitives can become quite heated. Unquestionably, the particular synchronization primitives that you select will have important consequences. Choosing primitives that are too "low-level" will almost inevitably lead to avoidable errors. On the other hand, utilizing primitives that are too "high-level" can preclude efficient implementation on many machines.

This second point is somewhat controversial, so let us briefly amplify our position. The use of a rendezvous mechanism of the sort included in Ada will (we believe) prevent the efficient implementation of an algorithm based on small-granularity parallelism. That is, the cost of your synchronization mechanism determines a lower bound on the size of the "work units" that are done in parallel.

We believe that monitors offer a reasonable conceptual framework for writing Pascal code for multiprocessors, without introducing overhead that unnecessarily restricts the class of algorithms that can be implemented.

Before offering a precise definition of what we mean by a monitor, we present a few programming examples that illustrate common synchronization patterns. These examples can then be used as a basis for forming a precise notion of the concept. Please note that our examples are (for the most part) relatively trivial. They are intended to clarify synchronization patterns and use of our macros; they are not considered of interest in themselves. The examples all illustrate "small-granularity" parallelism. That is, the computational tasks that are executed in parallel are very small. However, exactly the same synchronization patterns would be used in programs that capitalized on "large-granularity" parallelism.

To make our discussion concrete, we may occasionally discuss some aspect of a specific machine. In this document we shall present techniques for coding the Hagstrom Lemur, although our techniques are equally applicable to most multiprocessors that feature a number of processors operating on a globally shared memory.

In the programs covered in this tutorial, we use lowercase names to represent macro invocations. In addition, after each listing we briefly comment
2. Example 1: Self-scheduling FOR-loops

Our first program illustrates a number of important concepts. The task is to read in two vectors of integers, add the two vectors, and write out the resulting vector. To illustrate the use of parallelism, we coordinate a number of processors that add different elements of the vectors in parallel. That is, one process computes \( c[i] = a[i] + b[i] \), while a separate process adds \( a[2] \) and \( b[2] \).

(Adding two integers is a very small task; that is why our example illustrates small-granularity parallelism. If, instead, each element in the vector \( c \) were set to the smallest prime number greater than the product of the corresponding elements in the \( a \) and \( b \) vectors, the granularity would be considerably larger.)

Use of the Lemur requires writing programs to be run on specified processors which coordinate through a globally shared memory. In our examples, only one processor is active when the program begins execution. The program for this processor will be called the master program. Normally, a single, separate program is initiated and executed on the remaining processors. This program will be called the slave program. The program used to add the two vectors is as follows:

```
program addtwo(input,output);

This program demonstrates the "self-scheduling FOR-loop"

synchronization primitive. Reads in two vectors, adds them,
and prints the results.

type

dectype;

set up the variables in the shared memory

globmem = record;

  case char of
    'L': (a : array [1..1000] of integer;
    b : array [1..1000] of integer;
    c : array [1..1000] of integer;
    n : integer;
    nprocs : integer;
    gadsce(GS));
  end;
```
var
import large: globmem;
{i declare any local variables}
i: integer;

{ the next statement is required in the lemurenvironment }
lemurenv;

procedure work;
var
i: integer;

begin
  with large do
    begin
      getsub(GS,i,n,nprocs);
      while i <> 0 do
        begin
          c[i] := a[i] + b[i];
          getsub(GS,i,n,nprocs);
        end;
      end;
    end;
end;

begin
  with large do
    begin
      { initialize the monitor for the self-scheduling FOR loop }
gsinit(GS);
      { read in the number of processes to run in parallel }
      write('Input number of processes : ');
      readln(nprocs);
      writeln('nprocs = ', nprocs);
      { * * read in the two input vectors * * }
      write('Input vector length : ');
      readln(n);
program slave;

type
  declare
    the
  monitor
  type
dectype;

  set
  up
  the
  variables
  in
  the
  shared
  memory
  globmem = record,
    case
      char
    of
      'L' : (a : array[1..1000] of integer; b : array[1..1000] of integer; c : array[1..1000] of integer; n : integer; nprocs : integer; gsdec(CS));
      end;
  end;

var
  import large : globmem;

  { declare any local variables }

  writeln(' Input vector a ');
  for i := 1 to n do
    readln(a[i]);
  writeln(' Input vector b ');
  for i := 1 to n do
    readln(b[i]);

  { creation of the slave processes }
  for i := 1 to nprocs-1 do
    create(slave.exe, i);

  { now join the slaves to help with the work }
  work;

  { the master writes out the result }
  writeln('The resultant vector c = ');
  for i := 1 to n do
    writeln('c[i -> c[i+1]);

end;
Note that we have included code for two programs. The "master" program begins execution initially. This program activates other processors by "creating" processes which execute on the previously idle processors. A number of features of these programs require discussion:

1. Some of the peculiar characteristics of the Lemur must be understood to make the code intelligible. First, the Lemur is a machine composed of 8 processors sharing a global memory. Each processor also has a somewhat limited local memory. In the environment we are discussing here, the creation of a process results in the allocation of a processor. While a multiprocess environment could, in principle, be implemented within each of the processors, we believe that such a subdivision of processor capabilities would not justify the added system overhead. Thus, each process is executed on a dedicated processor. This means that there can be at most 8 processes in existence at any point in time. These processes operate within the different processors so that they
must communicate through the global memory, which is declared as a special type `globmem` (see lines 11-22). A single variable called `large` of type `globmem` must be declared via a line similar to line 25. The specific format of the shared data in the global memory is determined by the type declaration, so the user will wish to give distinct versions for each program that is coded. Note that this program declares the variables `a`, `b`, `c`, `n`, and `nprocs` in the global memory. In addition, the macro invocation `gsdec(GS)` is used to generate the variables required to manage a self-scheduling loop. Finally, lines 29-30 illustrate a macro invocation which is required in every Lemur program. The `lemurenv` should immediately precede the first procedure declaration.

2. The macro invocation `dectype` (lines 11-12) is used to define the type used by our standard macro packages. This type is called `monitor` and contains the data fields required to manage the synchronization for the monitor. The `gsdec` macro is used to define a monitor that can be used to coordinate a "self-scheduling FOR-loop". We will show you exactly what that means later in the code; for now all you really need to know is that the `gsdec(GS)` is used to generate the data structures required to coordinate processes to work on a self-scheduling FOR-loop. The GS is an arbitrary two-character "name" of the monitor. It can be anything, but the names used in the `gsdec`, `gsinit`, and `getsub` invocations must agree.

3. Line 56 contains `gsinit(GS)`, which initializes the variables that will be used to maintain proper synchronization between the processes. It must be invoked before more processes are created (i.e., before the other processes that will utilize the monitor begin to try acquiring subscript values from it).

4. Lines 58-62 represent the code that reads in the number of processes desired and prints it back out. It is quite useful to make the number of processes an input value. It is highly desirable that the code run properly with a value of 1, which allows the code to be (more or less) debugged on a uniprocessor.

5. Lines 77-80 create the extra processes that have been requested. Line 80 illustrates the use of the `create` macro. Execution of the `create` will cause a new process to be created on the processor designated in the second parameter. The first parameter gives the name of the file containing the compiled code to be loaded onto the processor.
6. The master process creates the slave processes and then invokes the subroutine `work`. The slave processes all just invoke the subroutine `work`. If you examine the subroutine `work`, you will see the code that all of the processes are simultaneously executing. The "heart" of the self-scheduling FOR-loop is given in lines 40-47. Each process executes the macro command `getsub(GS,i,n,nprocs)`. Think of this command as executing instantaneously. This is, of course, not true; but because it is a monitor command it is indivisible. Only one process will be allowed to execute the command at any point in time. The result of the command is to return in \( i \) the next available subscript in the range 1-\( n \). The first parameter is just the name of the monitor (to connect the `getsub` with the uses of the `gsdec` and `gsinit` macros), and the last parameter is the number of processes that will be competing for the subscripts. Each process claims a subscript (lines 41 and 45). If there are no more, \( i \) will be returned as 0. Otherwise, the subscript is processed (line 44), and the process claims another available subscript (line 45). This continues until the processes exhaust the subscripts in the range 1-\( n \).

There are a few fine points worth noting. First, no process will be returned a value of 0 before all processes have reached the point where they are requesting a subscript and there is none available (that is why \( nprocs \) must be a parameter to the request -- it allows the monitor command `getsub` to be able to determine when all of the processes have completed their last tasks). This ensures that no process will actually return before the last subscript has been processed.

Second, it is assumed that when the slave processes exit from `slave` (after returning from `work`) that they will "die"; furthermore, it is assumed that the slaves may exit either before or after the master returns (to line 88) to complete its functions. This assumption may, in fact, not be valid for some implementations (which would make the code fail to port properly). Please note that our development of these tools is still at a preliminary state. Once we have run experiments on several more machines, we are confident that a (more or less) complete set of macros will evolve leading to a high probability of portability.

Let us now pause and consider why this pattern of synchronization is called a "self-scheduling FOR-loop". Note that the illustrated logic is used to replace a loop of the following form (which would occur in the uniprocessing version of the code):

```
for i := 1 to n do
    begin
        <body of the loop>
```
The pattern will work only when executions of the body of the loop can be done in parallel. As an exercise to see whether or not you fully grasp this pattern, we suggest that you consider the case in which you have nested loops that can all be run in parallel. That is, consider the following pattern:

```
for i := 1 to n do
  for j := 1 to m do
    begin
      <body of the loop>
    end;
```

Assume that the different instances of the loop body can all compute in parallel. Try to construct the appropriate logic using two monitors named G1 and G2 to implement such logic for parallel execution.

3. Barrier Synchronization

Now let us introduce a second pattern of synchronization called barrier synchronization. This type of synchronization is quite simple: the execution of

```
barrier(B1, n)
```

will cause a process to "hang" in the monitor command until \( n \) processes are all "hanging". Then they are all released from the barrier and allowed to continue.

One common use of barrier synchronization occurs when multiple processes must be coordinated to perform a sequence of tasks, where the individual tasks must be executed strictly in order. To illustrate, suppose that we wished to perform a two-stage computation. The first stage is just as in our previous example, where \( c \) is set to \( a + b \). The second stage involves creating a new vector \( d \), where each value in \( d \) is a 0 or 1. The values in \( d \) are 0 exactly when the corresponding value in \( c \) is identical to the value that precedes it in \( c \). That is, if \( a = (1,4,2,5,6) \) and \( b = (3,2,4,2,1) \), then \( c \) would be set to \( (4,6,6,7,7) \) and \( d \) would then be set to \( (1,1,0,1,0) \). This example is obviously contrived. Clearly, both \( c \) and \( d \) could be created in a single loop. But suppose that the creation of \( c \) is a complex task calling for parallelism and that the creation of \( d \) could then be done most efficiently after creating \( c \). In this case, the following code might be used:
program getdups(input,output);
{
This program reads in two vectors a and b. It then creates
c = a + b. Then d is created as a vector in which each element
is set as follows: \( d[i] = 0 \) iff \( c[i] = c[i-1] \). Those elements
of d which are not set to 0 are set to 1. D[1] is always set
to 1.
}


type
dectype;
{
set up variables in shared memory
}


globmem = record;
case char of
  'l' : (a : array [1..1000] of integer;
    b : array [1..1000] of integer;
    c : array [1..1000] of integer;
    d : array [1..1000] of integer;
    n : integer;
    nprocs : integer;
    gsecd(G1);
    gsecd(G2);
    bardec(BA));
end;


var

import large : globmem;
i : integer;

lemirenv;

procedure work;
begin
  with large do
  begin
    getsub(G1,i,n,nprocs);
    while i <> 0 do
    begin

\( c[i] := a[i] + b[i]; \)
\( \text{getsub}(G1, i, n, nprocs); \)
\( \text{end;} \)
\( \text{barrier}(BA, nprocs); \)
\( \text{getsub}(G2, i, n, nprocs); \)
\( \text{while } i <> 0 \text{ do} \)
\( \text{begin} \)
\( \quad \text{if } i = 1 \text{ then} \)
\( \quad \quad d[i] := 1 \)
\( \quad \text{else} \)
\( \quad \quad \text{if } c[i] = c[i-1] \text{ then} \)
\( \quad \quad \quad d[i] := 0 \)
\( \quad \quad \text{else} \)
\( \quad \quad \quad d[i] := 1; \)
\( \quad \text{getsub}(G2, i, n, nprocs); \)
\( \text{end;} \)
\( \text{end;} \)
\( \text{end;} \)
\( \text{begin} \)
\( \quad \text{with large do} \)
\( \quad \text{begin} \)
\( \quad \quad \{ \text{initialize the monitors } \} \)
\( \quad \quad \text{gsinit}(G1); \)
\( \quad \quad \text{gsinit}(G2); \)
\( \quad \quad \text{barinit}(BA); \)
\( \quad \quad \text{write('Input number of processes : ');} \)
\( \quad \quad \text{readln(nprocs);} \)
\( \quad \quad \text{writeln('nprocs = ',nprocs:0);} \)
\( \quad \text{begin} \)
\( \quad \quad \{ \text{read in the two input vectors } \} \)
\( \quad \quad \text{write('Input vector length : ');} \)
\( \quad \quad \text{readln(n);} \)
\( \quad \quad \text{writeln('Input values for vector a');} \)
\( \quad \quad \text{for } i := 1 \text{ to } n \text{ do} \)
\( \quad \quad \quad \text{readln}(a[i]); \)
\( \quad \quad \text{writeln('Input values for vector b');} \)
\( \quad \quad \text{for } i := 1 \text{ to } n \text{ do} \)
\( \quad \quad \quad \text{readln}(b[i]); \)
for i := 1 to nprocs-1 do
    create(slave.exe,i);
    work;

writeln('The values in the c vector are as follows:');
for i := 1 to n do
    writeln(c[i]:0);
writeln('The values in the d vector are as follows:');
for i := 1 to n do
    writeln(d[i]:0);
end;
end.

------------ slave program -------------

program slave1;
{

This program reads in two vectors a and b. It then creates c = a + b. Then d is created as a vector in which each element is set as follows: d[i] = 0 iff (c[i] = c[i-1]). Those elements of d which are not set to 0 are set to 1. D[1] is always set to 1.

}

type
{ declare the monitor type } dtype;

{ set up variables in shared memory }
globmem = record;
    case char of
        'i' : (a : array [1..1000] of integer;
            b : array [1..1000] of integer;
            c : array [1..1000] of integer;
            d : array [1..1000] of integer;
            n : integer;
            nprocs : integer;
            gsdec(G1);
Again, several points should be mentioned:
1. Two self-scheduling FOR-loops are used in work. We used two monitors G1 and G2. Actually, since the getsub command reinitializes itself before releasing all of the processes at the end, a single monitor would suffice. We used two to illustrate the use of similar monitors with distinct names.

2. The barrier monitor is declared with bardec(BA) in a way completely analogous to the declaration for self-scheduling loops. Similarly, barinit(BA) initializes the barrier.

3. The barrier(BA,nprocs) in line 49 sets a barrier between the two stages of computation.

4. Since self-scheduling FOR-loops do not release any process with a subscript of 0, this barrier is actually unnecessary.

In some sense, the above example is awful. It uses two monitors (G1 and G2) where one would suffice, and it uses an unnecessary barrier. However, it should help you to become more familiar with the statements required to declare, initialize, and use monitors. Furthermore, it does illustrate in a fairly simple manner how a barrier can be set.

4. A More Challenging Example

Let us proceed to a more challenging example. The next program constructs vectors in descending order and then sorts them into ascending order using a Shell sort algorithm. It acquires timings for sorting vectors of sizes 100, 1000, and 10000. The Shell sort is interesting in its own right and requires some explanation.

To see how a Shell sort works, one should first understand how a simple insertion sort works. Consider the vector (2,-1,1,6,10,-8,18,0). To sort this vector into ascending order, one could proceed from left to right starting with the second element and inserting each element into the sorted list to its left. Thus, -1 would be examined first and inserted into the list to its left (which contains only the value 2); the result would be (-1,2,1,6,10,-8,18,0). Then the third element would be examined and inserted into the sorted list to its left, producing (-1,1,2,6,10,-8,18,0). The fourth element would be left unchanged, as would the fifth; they are already in correct order. Processing the sixth element results in (-8,-1,1,2,6,10,18,0). The seventh element would be left unchanged. Finally, processing the last element would give (-8,-1,0,1,2,6,10,18). This process of inserting each new element into the sorted list to its left is the essential idea behind insertion sorting.

The problem with simple insertion sorting is that a large number of comparisons may be required to move a number that occurs late in the list into its
correct location when that is toward the front of the list. The Shell sort corrects this with a rather ingenious idea. Again, consider the vector (2,-1,1,6,10,-8,18,0). Now suppose that the sort were to take place in two stages. In the first stage, the original list of 8 elements is viewed as four interleaved lists. The first list is composed of the first element and the fifth (i.e., 2 and 10); the second list is (-1,-8); the third is (1,18); and the fourth is (6,0). During the first stage, each of these lists is ordered using a simple insertion sort, giving (2,-8,1,0,10,-1,18,6). Clearly, fairly long moves can occur during the first stage. Then the resulting list can be sorted with a simple insertion sort.

This notion becomes quite significant when the original vector is long. Of course, we can carry the idea further and use more stages with gradually diminishing numbers of lists, as long as the last stage treats the whole vector as a single list.

This basic notion led to the following algorithm (written in pseudocode) for sorting a vector V containing N elements:

```plaintext
procedure shellsort(V,N)

compute h(1), h(2),...h(i+2) such that h(1)=1,
h(k) = (3*h(k-1)) + 1, h(i+1) < N, and h(i+2) > N.

set gap = h(i)

while (gap > 0) do

    set nextel = gap + 1

    while (nextel <= N) do

        sortel = nextel - gap

        while (sortel > 0) and (V(sortel) > V(sortel+gap)) do

            exchange V(sortel) and V(sortel+gap)

            set sortel = sortel - gap

        enddo

    set nextel = nextel + 1

dode
```


The sort algorithm is somewhat complex, but it does represent a fairly good sort algorithm that is easily implemented.

The above algorithm can clearly take advantage of multiprocessing on all but the last pass (i.e., on all executions of the main loop with a "gap" size greater than 1). This is achieved by allowing separate processes to sort the interleaved lists. The program that achieves this result using self-scheduling loops and barrier synchronization is as follows:

```
----------------------- master program -----------------------

1 defines(mode, real)
2
3 program srtpgm(input, output);
4 |
5 |
6 This program demonstrates the "BARRIER" and "self-scheduling FOR-loop"
7 synchronization primitives. It fills in a vector (a) with values in
8 descending order. Then it uses a shell sort (see Knuth's 3rd volume
9 on Sorting and Searching Algorithms) to sort the values into
10 ascending order.
11 |
12 |
13 type
14 { declare type for monitors }  
15 decType;
16 |
17 |
18 { set up variables in shared memory }
19 globalMN = record
20 case char of
21   'L' : (a : array [1..10000] of mode;
22     nprocs, n, gap : integer;
23     pgdone : boolean;
24     gsdec(GS);
25     bardec(B1));
```
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 FOR-loop is used to allocate subscripts. Note that the master participates in this logic, so the program can be run with nproc set to 1.
procedure sort;
{
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 $h_1, \ldots, h_2, h_1$: $h_1$ is 1, $h_i$ is $(3^{h_{i-1}} + 1)$;
$h_{i+2} \geq n$. See Knuth for arguments in favor of these values.
}

var
i1, i2, i3 : integer;

begin
with large do
begin
i1 := 1;
i2 := (i1 * 3) + 1;
i3 := (i2 * 3) + 1;
while i3 < n do
begin
i1 := i2;
i2 := i3;
i3 := (i2 * 3) + 1;
end;
gap := i1;
while gap > 0 do
begin
loop;
gap := (gap - 1) div 3;
end;
procedure sort;

procedure fill;

The following little routine just fills the vector with values in descending order.

```pascal
var
  i : integer;

begin
  with large do
  begin
    for i := 1 to n do
      a[i] := (n - i) + 1.0;
  end;
end;
```

```
begin
  with large do
  begin
    initialize the BARRIER and self-scheduling FOR-loop monitors
    gsinit(GS);
    barinit(B1);
    pgdone := false;
    write('Enter number of processes: ');
    readln(nprocs);
    writeln('nprocs = ',nprocs:0);
    for i := 1 to nprocs-1 do
      create(slave.exe,i);
  end;
end;
```

The main logic just fills in the table and sorts it.
Timings are taken for tables of 100, 1000, and 10000.
define (rrnde, real)

program save2;
{
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.
}
type
dectype;

set up variables in shared memory

var
airport large : globrn;

set up local variables

procedure loop;

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 nproc set to 1

var
done : boolean;
t : mode;
k1,k : integer;

begin
with large do
begin

barrier(B1,nprocs);
while not pgdone do
begin
getsub(GS,j,gap,nprocs);
while j <> 0 do


begin
  k := j + gap;
  while k <= n do
  begin
    k1 := k - gap;
    done := false;
    while not(done) and (k1 >= 1) do
      begin
        if a[k1] <= a[k1+gap] then
          done := true
        else
          begin
            t := a[k1];
            a[k1] := a[k1+gap];
            a[k1+gap] := t;
            k1 := k1 - gap;
          end;
      end;
      k := k + gap;
  end;
  getsub(GS,j,gap,nprocs);
end;

Again, several features of this program merit discussion:

1. First, line 1 is an example of a very simple macro. It states that every occurrence of the word "mode" will be replaced by the word "real". This macro allows a programmer to easily alter features of a program. We use the UNIX macro processor \texttt{m4}. Any macro processor will work equally well, but \texttt{m4} is a standard UNIX utility and will become commonly available. Because our approach to coding requires you to code simple macros, we encourage you to gain familiarity with its use. We hope that all of the required details will be adequately covered in this document, but for more details you should consult the UNIX documentation.
2. The variable \texttt{pgdone} is set to true when there are no more vectors to sort. It is referenced by the slave processes when they fail through the barrier in the procedure \texttt{loop}. It is used as a signal to the slaves to destroy themselves.

3. The \texttt{mLin} routine just constructs vectors to sort (by calling the procedure \texttt{fill}) and accumulates the time required to sort each table. Note the use of the \texttt{clock} macro to read the system clock (lines 175 and 180). In environments that do not support a system clock, the macro generates code to return a meaningless value (as is done, for example, with the macros designed for use on UNIX in a uniprocessing environment).

4. In this code the routine \texttt{loop} is multiprocessed logic. Since the master process prepares computational tasks (i.e., sets up vectors to sort, and creates the parameters for each stage of a sort), it will exit \texttt{loop} after each stage of a sort. The slaves, on the other hand, always wait in \texttt{loop} for the next stage of a sort (they hang on the barrier in \texttt{loop}, waiting for the master to set up the problem and join them at the barrier).

5. The slaves are created and wait in \texttt{loop} for the master to join them. Before joining them, the master process (in the routine \texttt{sort}) calculates the "gap" required on the next stage of a sort. The master calls \texttt{loop} with decreasing "gap" lengths until the a vector is completely sorted (i.e., until after calling \texttt{loop} with \texttt{gap} set to 1).

6. The logic of \texttt{loop} is based on the idea that each process selects one of the "interleaved lists" and sorts it into order. For each stage of a sort, there will be \texttt{gap} lists to sort.

This sort certainly does not fully utilize all available parallelism. In fact, the last stage of each sort (with \texttt{gap} = 1) is handled by a single process. However, on large vectors it does give a speedup roughly equal to half the number of available processors.

5. The Concept of Monitors

We have deferred discussion of exactly what we mean by a monitor to allow you to become familiar with a number of the details involved in our basic approach to implementing multiprocessing algorithms. It is now time to analyze the concept of monitors in some detail.

In our examples of self-scheduling FOR-loops and barriers, it was very important that each process be able to view execution of \texttt{getsub} and \texttt{barrier} commands as indivisible operations. The implementation of these commands involves data structures (generated by \texttt{gsdec} and \texttt{bardec} macros) that are
shared among a number of processes. The essential problem here is to make sure that access and updates are carefully synchronized to prevent destructive interference among the processes. Sections of code that can safely be executed by only a single process at a time are called critical sections.

For each set of shared data structures, there is a set of operations that represent critical sections. In the case of a self-scheduling FOR-loop, there is a single critical section (claiming the next subscript) associated with the data structures. In more complex cases, there may be a number of critical sections.

A monitor is a conceptual abstraction composed of three distinct parts:

1. The data that are shared,
2. The operations that represent critical sections associated with the shared data, and
3. The code required to initialize the shared structures.

Thus, when we speak of using a monitor to coordinate activity based on some shared data, we mean that the programmer carefully defines the shared data, the operations on the shared data, and the operations required to initialize the data. The details of how monitor operations are implemented to prevent destructive interference represent the central difficulty. When a process has begun to execute a critical section (i.e., when it enters a monitor operation), we refer to this as “having entered the monitor”. Only one process may be “in a monitor” at any given point. This is achieved conceptually by setting a lock at the point where the operation is entered and releasing the lock at the point where the operation is exited. Thus, in a self-scheduling FOR-loop, the operation getsub is basically

set a lock
claim the next subscript (which is a shared variable)
release the lock.

In the case of the self-scheduling FOR-loop, the gsdec(GS) defines the shared data in the global memory. The gsinit(GS) initializes the lock to “not set” and the subscript to 1. However, in the case in which there are no more available subscripts, we decided to cause the process issuing the getsub to “hang” until all of the remaining processes were also requesting a subscript. Only then will the getsub release them all (with a value of 0 to indicate that they were not successful in claiming a subscript). Conceptually, we “hang” a process as follows.

When a process is in a monitor and cannot complete the operation until a later point, it is delayed. This means that the process relinquishes the monitor
A delay queue can contain any number of "hung" processes, in arbitrary order. Each delayed process can be reactivated only by another process issuing a \textit{continue}. It is important that only one process ever be "in a monitor". Thus, the process that issues the \textit{continue} (which can be done only when it is in the monitor) exits from the monitor at the same instant that the (arbitrarily chosen) delayed process is reactivated (and regains control of the monitor).

Summarizing, we have identified four basic monitor operations:

- \textit{wenter(<monitor-name>)}: this operation is the first instruction in a monitor operation. It sets a lock, which ensures that the process can \textit{enter} the monitor operation only in the case in which no other process \textit{is active} in any operation associated with \textit{<monitor-name>}.

- \textit{mexit(<monitor-name>)}: this operation is the last instruction in a monitor. It releases the lock set by \textit{wenter(<monitor-name>)}.

- \textit{delay(<monitor-name>,<queue-number>)}: this operation causes the process that is in the monitor to be delayed in the queue designated by \textit{<queue-number>}. The process loses control of the monitor.

- \textit{continue(<monitor-name>,<queue-number>)}: this operation causes the active process to immediately exit from the monitor. If there are any delayed processes in the designated queue, one of them will be activated (thus, inheriting the lock set by the process leaving the monitor). Otherwise, the lock will be released by the exiting process.

With these definitions in hand, we can now give the exact logic of the \textit{getsub} monitor:
getsub(<monitor-name>,<returned-subscript>,<max-val>,<num-proc>)

mentor(<monitor-name>)

if (the shared subscript is less than or equal to <max-val>) then

set <returned-subscript> to the value of the shared subscript

increment the shared subscript

else

set <returned-subscript> to 0

if (fewer than (<num-proc> - 1) processes are in delay-queue-1)

delay(<monitor-name>,1)

endif

reset the shared subscript to 1

continue(<monitor-name>,1)

endif

mexit(<monitor-name>)
Once this logic is understood, the logic for barrier synchronization becomes trivial:

\[
\text{barrier(<monitor-name>,<num-proc>)}
\]
\[
\text{menter(<monitor-name>)}
\]
\[
\text{if (fewer than (<num-proc> - 1) processes are in delay-queue1)}
\]
\[
\text{delay(<monitor-name>,1)}
\]
\[
\text{endif}
\]
\[
\text{continue(<monitor-name>,1)}
\]
\[
\text{mexit(<monitor-name>)}
\]

We do not intend to introduce you to all of the features available in \texttt{m4} for implementing such monitors as macro invocation. However, you will need to understand the essential features of \texttt{m4} in order to work with our more complex monitors (and to code your own monitor operations, when necessary). Therefore, let us introduce you to \texttt{m4} by presenting a macro definition of the \texttt{barrier} operation:

\[
\text{define(barrier,\texttt{'\{bar }$1$\texttt{ for }$2$\texttt{ processes\}})}
\]
\[
\text{begin}
\]
\[
\text{menter($1$);}
\]
\[
\text{if ($1$.count < ($2$ - 1)) then}
\]
\[
\text{delay($1$);}
\]
\[
\text{continue($1$);}
\]
\[
\text{end\texttt{')}}
\]

The \texttt{define} command is used to create new macros. It contains two arguments – the name of the macro and what should be generated by the macro. In this simple example, "barrier" is the name, and the text surrounded by the apostrophes is the replacement text. \texttt{m4} can be used to search for strings of the form "barrier($a_1,a_2$)", which represent invocations of the macro. Whenever such a string is found, the replacement text will replace it. The only complexity involves the arguments of the invocation: wherever $1$ occurs in the replacement text, the first argument of the macro invocation will be used, and similarly for any number of arguments. Thus,
\texttt{barrier(BA,nprocs)}

would generate

\begin{verbatim}
{bar BA for nprocs processes}
begin
  menter(BA);
  if (BA.count < (nprocs - 1)) then
    delay(BA);
  continue(BA);
end
\end{verbatim}

This code would then be expanded to actual F\textsc{ascal} using definitions of the basic monitor operations that we supply for different machines.

Before leaving this example, one more comment is required. The variable \texttt{BA.count} occurs in the macro definition (and gets expanded in our example to \texttt{BA.count}). This variable, by definition, contains the number of processes delayed in the first delay queue for the monitor.

6. An Introduction to the Askfor Monitor

In our studies of implementing algorithms for multiprocessors, we have become increasingly aware of the significance of a single synchronization pattern. It appears with surprising frequency in both numeric and non-numeric applications. The pattern involves the concept of "a pool of tasks", where processes "ask for a task to work on" and "add tasks to the pool". In some sense, this is exactly what a self-scheduling FOR-loop does, except that for the loop processes never add new tasks to the pool. In the case of the loop, the pool of problems is represented by a subscript; if the shared subscript is \(I\) and the maximum value is \(N\), then the pool of remaining tasks to be parcelled out is the subscript values from the current value of \(I\) to the maximum value \(N\). This fairly simple special case, however, is not sufficient for many applications. Frequently, the pool of remaining problems cannot be represented by a single subscript; in fact, the structure required to represent the pool is often quite complex.

Let us first give a more precise description of the general pattern:

1) A sequence of computational tasks (i.e., problems) must be solved. We shall refer to these as the "major" tasks \(T_1, T_2, \ldots\).
2) Each major task $T_i$ may be decomposed into one or more minor tasks $t_{i1}, t_{i2}, \ldots$.
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 by exhaustion.

Let us illustrate one instance of this pattern that we found in writing a routine to perform QR-factorization of a sequence of matrices. In this case, each major task amounts to computing the desired factorization. We use Householder's algorithm for performing the factorization. It is not important that you understand what this algorithm does; it is necessary only to understand the following synchronization requirements:

1. The first step in factoring a matrix is to create the reflection for column 1. This reflection can then be applied to all remaining columns. In general, you create the reflection for column $k$ after the previous $k-1$ columns have all had their reflections created and applied to column $k$.
2. A reflection for column $k$ can be applied to all columns to the right of $k$. However, the reflections must be applied in order. That is, the reflection created from column 1 must be applied to column 3 before the reflection created from column 2 is applied to column 3.

In this case, the pool of remaining tasks includes two types of minor tasks: create a reflection for column $i$ and apply the reflection for column $i$ to column $j$. Initially, the pool will contain only the minor task "create the reflection for column 1". The solution of the first minor task will cause a number of new minor tasks (to apply the reflection to each of the remaining columns) to be added to the pool. Processes claim minor tasks and add them to the pool until the pool is exhausted, representing completion of the major task. In this particular case, all major tasks are solved by exhaustion (and are solved "successfully"). This is frequently not the case. The challenge involved in setting up the problem is how to represent the "pool of problems" in a reasonably efficient manner. This is a problem-dependent aspect of the synchronization: different problems utilize widely different representations of the problem pool. Thus, special-purpose logic will be needed to "claim a problem from the pool" or to "insert a problem into the pool".

We have found this pattern of synchronization, which we will call the askfor monitor (since processes "ask for" the next available task to perform), to be
quite difficult to implement properly. The fact that problem-dependent code is required to manage the pool of outstanding tasks makes the use of our askfor monitor more complex than using the barrier or getsub monitors. We begin, therefore, with the simplest of settings -- our first example that adds two vectors to produce a third. In this case, since it is a self-scheduling FOR-loop, the pool of problems can be managed by a single shared subscript. The code is as follows:

--- master program ---

```fortran
1 define(getproh $1 := sub <= $2 then
2        begin
3            $1 := sub;
4            sub := sub + 1;
5            $3 := 0;
6        end;
7
8 program asktwo(input,output);
9 {
10    This program demonstrates the "ASKFOR" synchronization primitive.
11    It reads in two vectors, adds them, and prints the results.
12 }
13
type
dectype;
14
globmem = record
15    case char of
16        '1' : (a : array [1..1000] of integer;
17            b : array [1..1000] of integer;
18            n : integer;
19            nprocs, a[1..1000] of integer;
20            m : integer;
21            adecmp(MO);
22            sub : integer);
23    end;
24
25 var
26    import large : globmem;
27    i : integer;
28    {
29    set up local variables
30    }
31
32
33
34
35
36
```
procedure work;
{

The work subroutine just causes a process to grab available
subscripts until all of the work has been completed. At that
point the subroutine exits. Note that if there is a single
process (i.e. no slaves), the algorithm still works just
fine.

}

var
i, rc : integer;

begin
  with large do
  begin
    askfor(MO,rc,nprocs,getprob(i,n,rc),);
    while rc = 0 do
        begin
          c[i] := a[i] + b[i];
          askfor(MO,rc,nprocs,getprob(i,n,rc),);
        end;
      end;
  end;

begin
  with large do
  begin
    { initialize the monitor }
    ainit(MO);
    sub := 1;
    { read in the number of processes to run in parallel }
    write('Enter the number of processes : ');
    readln(nprocs);
    writeln(nprocs = ',nprocs:4);
    { read in the two input vectors }
    write('Enter vector length : ');
    readln(n);
writeln('Enter values for vector a');
for i := 1 to n do
  readln(a[i]);
writeln('Enter values for vector b');
for i := 1 to n do
  readln(b[i]);

{ Create the slave processes }

for i := 1 to nprocs-1 do
  create(slave.exe,i);

work;

writeln('The resulting vector c : ');
for i := 1 to n do
  writeln(c[i]);
end;
end.

------------------ slave program -----------------------

define(getprob,'if sub <= $2 then
begin
$1 := sub;
sub := sub + 1;
$3 := 0;
end;')

program slave3;
{
This program demonstrates the "ASKFOR" synchronization primitive.
It reads in two vectors, adds them, and prints the results.
}

type
{
declare monitor type
}
dectype;

{
set up variables in shared memory
}
globmem = record
case char of
  'i' : (a : array [1..1000] of integer;
b : array [1..1000] of integer;
c : array [0..999] of integer;
nprocs, n : integer;
adeq(MO);
sub : integer;

end;

var
import large : glohmen;

procedure work;

The work subroutine just causes a process to grab available
subscripts until all of the work has been completed. At that
point the subroutine exits. Note that if there is a single
process (i.e., no slaves), the algorithm still works just
fine.

begin

work;
To understand this program, the reader should compare it to the version using self-scheduling FOR-loops. Several points are worth mentioning:

1. First, note that the programmer explicitly gives the shared subscript used to manage the pool of processes (line 29). He must also initialize his representation of the pool (line 72).

2. The `getsub` operation has been replaced with invocations of the `askfor` monitor operation (lines 57 and 61). For example,

```c
57  askfor(MO,rc,nprocs,getprob(i,n,rc));
```

Here, the first argument (MO) is the name of the monitor. The second argument (rc) is a variable set by the monitor operation to indicate whether a problem could be taken from the pool (set to 0 on successful acquisition of a problem; otherwise it will get set to 1). The third argument (nprocs) gives the number of processes sharing the pool. The fifth argument gives a macro invocation that can be used to generate the problem-dependent logic required to try to claim a problem from the pool. There is a missing sixth argument which is used to generate the code required to reinitialize the pool, since we are only adding one pair of vectors, there is no need to reinitialize the pool.

The intrinsic complexity involved in the `askfor` monitor is based on the fact that the user has to provide a macro to generate the logic to claim tasks (and the logic to reinitialize the pool, if a sequence of major tasks are to be processed). The logic to claim a problem is generated by the macro `getprob`, which is defined in lines 1-6 of the program. Note that `getprob` does not generate a monitor operation; rather, it generates a block of code in the middle of the `askfor` monitor operation. This is why it does not begin with an `menuer`.

7. A More Complex Example of the Askfor Monitor

Many applications of the `askfor` monitor are substantially more complex than our last example. Frequently, the complexity originates in the representation of the pool of problems. However, there can be other sources of complexity:

1. Some problems can be "solved" by any arbitrary process working on a minor task. All processes must (on their next request for a task) be notified of "problem end" by a positive value in the return code. For example, suppose that the object of our previous problem were to
determine whether the sum of any two corresponding elements in
\(a\) and \(b\) exceeded 100. In this case, a solution by exhaustion (return code of 1) would indicate that no values did exceed 100. On the other hand, any process that found a sum that exceeded 100 could "post" end-of-problem, using the actual sum as the return code (actually any value greater than 1 would be adequate). Posting an end-of-problem condition is accomplished with the \textit{probend} macro.

2. Some programs must compute solutions to a sequence of major tasks. This requires the inclusion of an extra parameter on the \textit{askfor} invocation giving a macro that can be used to reinitialize the monitor. Furthermore, each process should receive the end-of-problem return code exactly. Once a process has received such a return code, it can ask for another task (which is what slave processes normally do) or itself go and acquire another major task to solve (which is what the master process would normally do).

3. In the cases where a sequence of major tasks are to be solved, some means must exist for signalling "end of program" to processes. This is done by invoking the \textit{progend} macro. In this case, every process that has been delayed or requests another task will receive a return code value of -1.

To see exactly how these concepts come into play, let us consider a modified version of our earlier example. In this example, a sequence of pairs of input vectors are processed. For each pair, the question "Does the sum of any corresponding elements exceed 100?" is answered. Between problems, the reset logic just resets the shared subscript to 0. The subscript is set to 1 when a new pair of vectors are ready to be processed.

\begin{verbatim}
----------------------- master program -----------------------
1 define(getprob,
2   'if sub > 0 then
3     begin
4       if sub <= #2 then
5         begin
6           $1 := sub;
7           sub := sub + 1;
8           $3 := 0;
9         end;
10       end;
11     end;
\end{verbatim}
This program demonstrates the "ASKFOR" synchronization primitive. It reads in pairs of vectors. For each pair, it determines whether or not the sum of any two corresponding elements exceeds 100.

type
dectype; 

globrem = record
case caar of
 'L' : (a : array [1..1000] of integer;
    n, npocs : integer;
    sub : integer);
end;

var
import large : globrem;

procedure work;
var
i,j,rc,sum : integer;

begin
with large do
begin
askfor(MO,rc,nprocs.getprob(i,n,rc),reset);
while rc = 0 do
begin
sum := a[i] + b[i];
if sum > 100 then
probend(MO,2);
askfor(MO,rc,nprocs.getprob(i,n,rc),reset);
end; [while]
writeln('The return code = ',rc:4);
end;
end;

begin
with large do
begin
  [initialize the ASKFOR monitor ]
ainit(MO);
sub := 0;
{ read in the number of processes to run in parallel } 
write('Enter number of processes : ');
readln(nprocs);
writeln('nprocs = ',nprocs:4);
{ create the slave processes } 
for i := 1 to nprocs-1 do
  create(slave.exe,i);
{ read in the number of pairs to process } 
write('Enter number of problems : ');
readln(nprobs);
for j := 1 to nprobs do
begin
write('Enter vector length : ');
readln(n);
writeln('Enter values for vector a');
for i := 1 to n do 
  readln(a[i]);
writeln('Enter values for vector b');
for i := 1 to n do 
  readln(b[i]);
probstart;
work;
end; [for]
progend(MO);
end;

------------------ slave program ------------------
define(getprob,
  'if sub > 0 then
    begin
      if sub <= $2 then
        begin
          $1 := sub;
          sub := sub + 1;
          $3 := 0;
        end;
      end,')
define(reset,
  'sub := 0;'
)
program slave4;
{
This program demonstrates the "ASKFOR" synchronization primitive. It reads in pairs of vectors. For each pair, it determines whether or not the sum of any two corresponding elements exceeds 100.
}
type
{| declare monitor type |}
dectype;
{| set up variables for shared memory |}
globmem = record
{| case char of |}
'L' : (a : array [1..1000] of integer;
  b : array [1..1000] of integer;
  n, nprocs : integer;
  adec(MO);
  sub : integer);
end;

var
{| import large : globmem; |}
i,j,nprocs : integer;
{| set up local variables |}
lemarenv;
procedure work;
{| var |}
i,rc,sum : integer;
begi
{| with large do |}
rc := 0;
{| while rc <> -1 do |}
begin
{| askfor(MO,rc,nprocs,getprob(i,n,rc),reset); |}
while rc = 0 do
begin
{| sum := a[i] + b[i]; |}
if sum > 100 then
{| probend(MO,2); |}
askfor(MO,rc,nprocs,getprob(i,n,rc),reset);
end; |{while|
end;
end;
end;
Note the following points:

1. **The getprob** logic must now check to make sure that there is an active problem. If the logic does not alter rc, then the requesting process will be delayed until **probstart** is executed. Here **probstart** is an actual monitor operation. It issues a **continue** to activate any processes that might be waiting for work to arrive.

2. If a process recognizes an end-of-problem condition, it still must go through the **askfor** requesting a task. Our logic insists on returning the end-of-job return code to each of the cooperating processes (and waits for them all to be delayed, so that it can release them all with the appropriate return code).

3. The second operand on the **probend** monitor operation is the return code to be passed back to each process. It should always be greater than 1.

The reader should note that the logic to support the full **askfor** monitor is somewhat complex. For a detailed description, you should consult [9].

### B. Locks

Before ending this discussion of basic synchronization primitives, some discussion of "locks" should be included. Consider the problem of computing a dot product of two vectors **avec** and **bvec**, storing the result in the integer **idot**. If a self-scheduling for-loop is used, processes are allocated distinct subscripts. However, the problem remains of exactly how to synchronize the addition of values into the integer **idot**. The basic idea behind the use of monitors would suggest that a monitor be established for the variable **idot**, with the single operation of adding a value to the variable. This is a perfectly reasonable way to handle the problem.

However, in some problems it becomes necessary to selectively gain access to a variety of data items and then to update them in fairly complex patterns. This can be done most easily using "locks". A lock amounts to a "monitor enter" operation, while an "unlock" is the corresponding "monitor exit" operation. When utilizing locks, there is no notion of delaying the process. The process simply gains control of the lock, performs the critical section, and releases the
lock.

To allow this style of programming, we have include for macros in our standard package:

\[ \text{lockdec}(N) \] declares the variables required to support \( N \) locks. These locks are based on an array (named \( \text{LO} \)) of monitors.

\[ \text{lockinit}(N) \] initializes the \( N \) locks.

\[ \text{lock}(j) \] acquires the \( j \)th lock. This amounts to an \( \text{menter}(\text{LOj}) \) operation, which acquires entry to the \( j \)th monitor of the array named \( \text{LO} \).

\[ \text{unlock}(j) \] releases the \( j \)th lock.

To update the shared variable \( \text{idot} \), one would then use

\[
\text{lock}(1) \\
\text{idot} = \text{idot} + \text{some value} \\
\text{unlock}(1)
\]

where the single lock would be declared with

\[ \text{lockdef}(1) \]

and initialization would be accomplished with

\[ \text{lockinit}(1) \]

In those cases where a number of locks must be held before updating a set of values, the programmer should always acquire the locks in some fixed order (e.g., increasing order of lock number). This technique is called a **standard allocation pattern** and will guarantee that a deadlock will not occur between two processes attempting to claim overlapping sets of locks. Thus, if process 1 requires locks 1 and 2, while process 2 requires locks 2, 3, and 1, a deadlock could occur unless the fixed order of locking is observed (forcing process 2 to claim lock 1 before lock 2, and lock 2 before lock 3).

9. **The Fetch-and-Add**

There has been a great deal of interest in the use of the fetch-and-add as a primitive operation in a multiprocessing environment[2]. For many applications, it can be used to eliminate critical sections. It seems quite likely that, as people contemplate the potential advantages of machines with thousands of processors, the significance of the fetch-and-add operation will turn out to be substantial.
To allow researchers to develop codes for machines that support fetch-and-add on multiprocessors that do not (or to allow portability from systems that do to those that do not), we have written a set of macros that allow one to code using the fetch-and-add. On systems that do not support the operation as a primitive, the macros generate the code to support it as a monitor operation (with the implied overhead associated with the critical section).

The macros to support fetch-and-add are as follows:

\[ fadec(<\text{monitor-name}>,<\text{type}>) \] is used to generate the variables required to support the monitor. Here \(<\text{type}>\) should be \text{integer} or \text{real}, giving the type associated with the value maintained by the monitor.

\[ fainit(<\text{monitor-name}>) \] initializes the monitor, setting the associated value to 0.

\[ fadd(<\text{monitor-name}>,<\text{old}>,<\text{new}>,<\text{incr}>) \] returns the "old" value in \(<\text{old}>\), increments it with \(<\text{incr}>\), and returns the "new" value in \(<\text{new}>\). The value associated with the monitor takes on the "new" value.

Thus, the macros might be used as follows:

\[
\begin{align*}
&fadec(F1, \text{integer}) \\
&fainit(F1) \\
&fadd(F1, oldi, newi, j)
\end{align*}
\]

Here the integer value associated with \(F1\) is incremented by the fetch-and-add. The original value is returned in \(oldi\), and the new value in \(newi\).

This method really does not convey the full power of fetch-and-add, since the operation is designed to be applicable on any word in memory. If you think of its use in manipulating linked lists, then the above version of the macros is really quite inadequate. To somewhat improve the situation, we allow you to declare and manipulate an array of monitors, rather than a single monitor. When this is done, the subscript of the desired monitor (which can be thought of as a "cell" address) is used as the last argument. For example,

\[
fadec(F1, \text{integer}, 100)
\]
\texttt{fainint(F1, 100)}

\texttt{fadd(F1, oldi, newi, j, cell)}

declares and utilizes 100 monitors. Here the \texttt{fadd} includes an extra argument, which is the subscript of the monitor for which the operation applies.

10. Running the Package

To prepare a program for execution on some particular machine, you must expand the macro definitions into versions for the desired machine. Typically, you would keep one file of macro definitions for all monitors. This file would not include the basic macro package (which includes \texttt{menter} and \texttt{mexit}). The basic macro package is machine-specific. User-defined macros specific to a given program are normally included at the front of the program. Following these conventions,

\texttt{m4 <basic-macro-library> <standard-monitors> <user-source> >! <dest>}

can be used to create the desired source in \texttt{<dest>}. We can supply versions of the \texttt{<basic-macro-library>} for both UNIX uniprocessors (for debugging on a uniprocessor system) and Lemur Pascal. The \texttt{<standard-monitors>} code at this point includes the monitors that we have discussed in this document, as well as a number of other standard monitors (which we hope to discuss in future reports).

11. Conclusion

This tutorial focuses on the use of three of our macro packages -- those for the self-scheduling FOR-loop, barrier, and askfor monitors. We have not discussed all features of these macros, but we have covered their most common uses. We believe, however, that the monitors have wide applicability in more demanding contexts.

If you have any comments or suggestions, we would appreciate hearing from you.
References


Distribution for ANL-84-63

Internal:

J. A. Clausing (5)
R. Hagstrom (40)
K. L. Kliewer
A. B. Krisciunas
E. L. Lusk (19)
P. C. Messina
R. A. Overbeek (18)
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, Tynsboro, MA
G. H. Golub, Stanford U.
W. C. Lynch, Xerox Corp., Palo Alto
M. F. Wheeler, Ricc U.
L. P. Kadanoff, U. of Chicago (Board of Governors liaison)
D. Austin, Office of Basic Energy Sciences, DOE
G. Michael, LLL