SCHEDULE USERS GUIDE

J. J. Dongarra and D. C. Sorensen

Mathematics and Computer Science Division
Argonne National Laboratory
9700 South Cass Avenue
Argonne, Illinois 60439-4844

1. Introduction

Many new parallel computers are now emerging as commercial products Exploitation of the parallel capabilities requires either extensions to an existing language such as Fortran or development of an entirely new language. A number of activities are under way to develop new languages that promise to provide the ability to exploit parallelism without the considerable effort that may be required in using an inherently serial language that has been extended for parallelism. We applaud such activities and expect they will offer a true solution to the software dilemma in the future. However, in the short term we feel there is a need to confront some of the software issues, with particular emphasis placed on transportability and use of existing software.

Our interests lie mainly with mathematical software typically associated with scientific computations. Therefore, we concentrate here on using the Fortran language. Each vendor of a parallel machine designed primarily for numerical calculations has provided its own parallel extensions to Fortran. These extensions have taken many forms already and are usually dictated by the underlying hardware and by the capabilities that the vendor wishes to supply the user. This has led to widely different extensions ranging from the ability to synchronize on every assignment of a variable with a full empty property to attempts at automatically detecting loop-based parallelism with a preprocessing compiler aided by user directives The act of getting a parallel process executing on a physical processor ranges from a simple "create" statement which imposes the overhead of a subroutine call, to "tskstart" which imposes an overhead on the order of 10^6 machine cycles, to no formal mechanism whatsoever These different approaches reflect characteristics of underlying hardware and operating systems and to a large extent are dictated by the vendors view of which aspects of parallelism are marketable. It is too early to impose a standard on these vendors, yet it is disconcerting that there is no agreement among any of them on which extensions should be included. There is not even an agreed-upon naming convention for extensions that have identical functionality. Program developers interested in producing implementations of parallel algorithms that will run on a number of different parallel machines are therefore faced with an overwhelming task. The process of developing portable parallel packages is complicated by additional factors that lie beyond each computer manufacturer supplying its own, very different mechanism for parallel processing. A given implementation may require several different communicating parallel processes, perhaps

* Work supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contracts W-31-109-Eng-38, DE-AC05-840R21400.

with different levels of granularity. An efficient implementation may require the ability to dynamically start processes, perhaps many more than the number of physical processors in the system. This feature is either lacking or prohibitively expensive on most commercially available parallel computers. Instead, many of the manufacturers have limited themselves to providing one-level loop-based parallelism.

This paper describes an environment for the transportable implementation of parallel algorithms in a Fortran setting. By this we mean that a user's code is virtually identical for each machine. The main tool in this environment is a package called SCHEDULE which has been designed to aid a programmer familiar with a Fortran programming environment to implement a parallel algorithm in a manner that will lend itself to transporting the resulting program across a wide variety of parallel machines. The package is designed to allow existing Fortran subroutines to be called through SCHEDULE, without modification, thereby permitting users access to a wide body of existing library software in a parallel setting. Machine intrinsics are invoked within the SCHEDULE package, and considerable effort may be required on our part to move SCHEDULE from one machine to another. On the other hand, the user of SCHEDULE is relieved of the burden of modifying each code he disires to transport from one machine to another.

Our work has primarily been influenced by the work of R. Babb J.C. Browne and E. Lusk and R. Overbeek . We present here our approach, which aids in the programming of explicitly parallel algorithms in Fortran and which allows one to make use of existing Fortran libraries in the parallel setting. The approach taken here should be regarded as minimalist: it has a very limited scope. There are two reasons for this. First, the goal of portability of user code will be less difficult to achieve. Second, the real hope for a solution to the software problems associated with parallel programming lies with new programming languages or perhaps with the "right" extension to Fortran. Our approach is expected to have a limited lifetime. Its purpose is to allow us to exploit existing hardware immediately.

2. Terminology

Within the science of parallel computation there seems to be no standard definition of terms. A certain terminology will be adopted here for the sake of dialogue. It will not be "standard" and is intended only to apply within the scope of this document.

Process - A unit of computation, an independently executable Fortran subroutine together with calling sequence parameters, common data, and externals.

Task - A main program, processes, and a virtual processor.

Virtual Processor - A process designed to assume the identity of every process within a given task (through an appropriate subroutine call).

Processor - A physical device capable of executing a main program or a virtual processor.

Shared Data - Variables that are read and/or written by more than one process (including copies of processes).

Data Dependency - A situation wherein one process (A) reads any shared data that another process (B) writes. This data dependency is satisfied when B has written the shared data.

Schedulable Process - A process whose data dependencies have all been satisfied.

3. Parallel Programming Ideas

When designing a parallel algorithm one is required to describe the data dependencies, parallel structures, and shared variables involved in the solution. Typically, such algorithms are first designed at a conceptual level and later implemented in Fortran and its extensions. Each manufacturer provides a different set of extensions and targets these extensions at different implementation levels. For example, some manufacturers allow only test-and-set along with spawn-a-process, while others allow concurrent execution of different loop iterations.

Our attempt here is to allow the user to define the data dependencies, parallel structures, and shared variables in his application and then to implement these ideas in a Fortran program written in terms of subroutine calls to SCHEDULE. Each set of subroutine calls specifies a unit of computation or process which consists of a subroutine name along with the calling parameters and the data dependencies necessary to coordinate the parallel execution.

The basic philosophy here is that Fortran programs are naturally broken into subroutines that identify self-contained units of computation which operate on shared data structures. This allows one to call on existing library subroutines in a parallel setting without modification, and without having to write an envelope around the library subroutine call in order to conform to some unusual data-passing conventions imposed by a given parallel programming environment.

A parallel(izable) program is written in terms of calls to subroutines which, in principle, may be performed either independently or according to data dependency requirements that the user is responsible for defining. The result is a serial program that can run in parallel given a way to schedule the units of computation on a system of parallel processors while obeying the data dependencies.

4. Parallel Programming Using SCHEDULE

The package SCHEDULE requires a user to specify the subroutine calls along with the execution dependencies in order to carry out a parallel computation. Each of these calls represents a process, and the user must take the responsibility of ensuring that the data dependencies represented by the graph are valid. This concept is perhaps difficult to grasp without some experience with writing parallel programs. We shall try to explain it in this section by example; in the following section we shall describe the underlying concepts and the SCHEDULE mechanism.

To use SCHEDULE, one must be able to express (i.e., program) an algorithm in terms of processes and execution dependencies among the processes. A convenient way to view this is through a computational graph. For example, the following graph

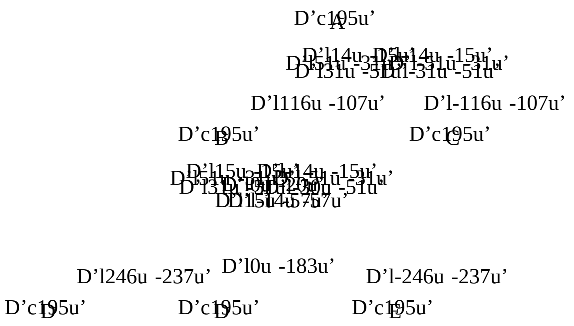

Figure 4.1.
Data Dependency Graph

denotes five subroutines A, B, C, D, and E (here with two "copies" of subroutine D operating on different data). We intend the execution to start simultaneously on subroutines C,D,D, and E since they appear as leaves in the data dependency graph (D will be initiated twice with different data). Once D,D,and E have completed then B may execute. When B and C have completed execution then A may start and the entire computation is finished when A has completed. To use SCHEDULE, one is required to specify the subroutine calling sequence of each of the six units of computation, along with a representation of this dependency graph.

For each node in the graph, SCHEDULE requires two subroutine calls. One contains information about the user's routine to be called, such as the name of the routine, calling sequence parameters, and a simple tag to identify the process. The second subroutine call defines the dependency in the graph to nodes above and below the one being specified, and specifies the tag to identify the process. In this example, after an initial call to set up the environment for SCHEDULE, six pairs of calls would be made to define the relationships and data in the computational graph.

These concepts are perhaps more easily grasped through an actual Fortran example. A very simple example is a parallel algorithm for computing the inner product of two vectors. The intention here is to illustrate the mechanics of using SCHEDULE. This algorithm and the use of SCHEDULE on a problem of such small granularity are not necessarily recommended.

Problem: Given real vectors a and b, each of length n, compute $\sigma = a^T b$.

Parallel Algorithm:

Let $a^T = (a_1^T, a_2^T, ..., a_k^T)$ and $b^T = (b_1^T, b_2^T, ..., b_k^T)$
be a partitioning of the vectors a and b into smaller vectors a_i and b_i.

Compute (in parallel)

$$\sigma_j = a_j^T b_j \ , \ j = 1,k.$$

When all done

$$\sigma = \sigma_1 + \sigma_2 + \cdots + \sigma_k.$$

Each of the parallel processes will execute code of the following form:

```
      subroutine inprod(m,a,b,sig)
      integer m
      real a(*),b(*),sig
      sig = 0.0
      do 100 j = 1,m
         sig = sig + a(j)*b(j)
  100 continue
      return
      end
```

The following routine is used to accumulate the result:

```
      subroutine addup(k,sigma,temp)
      integer k
      real sigma,temp(*)
      sigma = 0.0
      do 100 j = 1,k
         sigma = sigma + temp(j)
  100 continue
      return
      end
```

The first step in constructing a code is to understand the parallel algorithm in terms of schedulable processes and a data dependency graph. Then the algorithm is expressed in a standard (serial) Fortran code. This code consists of a main program which initializes the shared data and a "parallel" subroutine **parprd** to compute the inner product by invoking the parallel processes **inprd** and **addup**. The program and associated data dependency graph are shown below.

Serial Code:

```
      program main
      integer n,k
      real a(1000),b(1000),temp(50),sigma
      read (5,*) n,k
      do 100 j = 1,n
         a(j) = j
         b(j) = 1
  100 continue
c
      call parprd(n,k,a,b,temp,sigma)
c
      write(6,*) ' sigma = ',sigma
      stop
      end


      subroutine parprd(n,k,a,b,temp,sigma)
c
c     declare shared variables
c
      integer n,k
      real a(*),b(*),temp(*),sigma
c
c     declare local variables
c
      integer m,indx,j
c
      m = n/k
      indx = 1
      do 200 j = 1,k
c
         call inprod(m,a(indx),b(indx),temp(j))
c
         indx = indx + m
         if (j .eq. k-1) m = n - indx + 1
  200 continue
c
      call addup(k,sigma,temp)
c
      return
      end
```

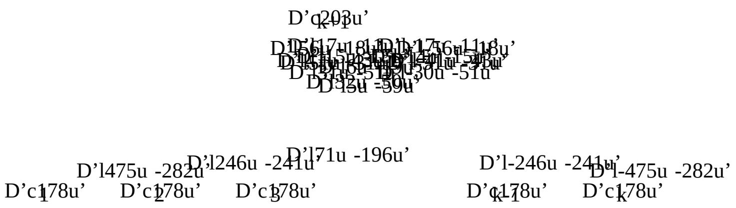

Figure 4.2.
Depedency Graph for Parallel Inner Product

In this data dependency graph we have identified *k* processes *inprod*(*m*, *a*(*indx*), *b*(*indx*), *temp*(*j*)) , $j = 1, 2, ..., k$, $indx = 1 + (j-1)*m$

which are not data dependent. Each of them reads a segment of the shared data *a* , *b* and writes on its own entry of the array *temp*, but none of them needs to read data that some other process will write. This fact is evident in the graphical representation shown in Figure 4.2 where they are *leaves*. One process,

addup(*k*,*sigma*,*temp*),

labeled $k+1$ is data dependent on each of the processes $1,2,...,k$. This is because **addup** needs to read each entry of the array *temp* in order to compute the sum and place it into σ.

From this data dependency graph we may proceed to write the parallel program. Once we have understood the computation well enough to have carried out these two steps, the invocation of SCHEDULE to provide for the parallel execution of schedulable processes is straightforward. Calls to **parprd**, **inprod**, and **addup** are replaced by calls to SCHEDULE to identify the routines to be executed as well as the information relating to the dependency graph. The modified code follows.

Parallel Main:

```
      program main
      integer n,k
c
      EXTERNAL PARPRD
c
      real a(1000),b(1000),temp(50),sigma
      read (5,*) n,k,NPROCS
      do 100 j = 1,n
         a(j) = j
         b(j) = 1
  100 continue
c
      CALL SCHED(nprocs,PARPRD,n,k,a,b,temp,sigma)
c
      write(6,*) ' sigma = ',sigma
      stop
      end


      subroutine parprd(n,k,a,b,temp,sigma)
c
c     declare shared variables
c
      integer n,k
      real a(*),b(*),temp(*),sigma
c
c     declare local variables
c
      integer m1,m2,indx,j,jobtag,icango,ncheks,mychkn(2)
c
      EXTERNAL INPROD,ADDUP
      save m1,m2
c
      m1   = n/k
```

```
      indx = 1
      do 200 j = 1,k-1
c
c         express data dependencies
c
          JOBTAG = j
          ICANGO = 0
          NCHEKS = 1
          MYCHKN(1) = k+1
c
          CALL  DEP(jobtag,icango,ncheks,mychkn)
          CALL  PUTQ(jobtag,INPROD,m1,a(indx),b(indx),temp(j))
c
          indx = indx + m1
  200 continue
      m2 = n - indx + 1
c
c     express data dependencies for clean up step
c
          JOBTAG = k
          ICANGO = 0
          NCHEKS = 1
          MYCHKN(1) = k+1
c
          CALL  DEP(jobtag,icango,ncheks,mychkn)
          CALL  PUTQ(jobtag,INPROD,m2,a(indx),b(indx),temp(k))
c
          indx = indx + m1
c
          JOBTAG = k+1
          ICANGO = k
          NCHEKS = 0
c
      CALL  DEP(jobtag,icango,ncheks,mychkn)
      CALL  PUTQ(jobtag,ADDUP,k,sigma,temp)
c
      return
      end
```

The code that will execute in parallel has been derived from the serial code by replacing calls to **parprd**, **inprd**, **addup** with calls to SCHEDULE routines that invoke these routines. The modifications are signified by putting calls to SCHEDULE routines in capital letters. Let us now describe the purpose of each of these calls.

```
      CALL SCHED(nprocs,PARPRD,n,k,a,b,temp,sigma)
```

This replaces the call to **parprd** in the serial code. The effect is to devote *nprocs* virtual processors to the parallel subroutine **parprd**. The parameter list following the subroutine name consist of the calling sequence one would use to make a normal call to **parprd**. Each of these parameters must be called by reference and not by value. No constants or arithmetic expressions should be passed as parameters through a call to **sched**. This call to **sched** will activate *nprocs* copies of a virtual processor **work**. This virtual processor is a SCHEDULE procedure (written in C) that is internal to the package and not explicitly available to the user.

```
          JOBTAG = j
          ICANGO = 0
          NCHEKS = 1
```

```
        MYCHKN(1) = k+1
c
        CALL   DEP(jogtag,icango,ncheks,mychkn)
        CALL   PUTQ(jobtag,INPROD,m,a(indx),b(indx),temp(j))
```

This code fragment shows the *j–th* instance of the process **inprod** being placed on a queue. The information needed to schedule this process is contained in the data dependency graph. In this case, the *j–th* instance of a call to **inprod** is being placed on the queue, so *jobtag* is set to *j*. The value zero is placed in *icango*, indicating that this process does not depend on any others. If this process were dependent on *p*, other processes then *icango* would be set to *p*.

The mechanism just described allows static scheduling of parallel processes. In this program the partitioning and data dependencies are known in advance even though they are parameterized. It is possible to dynamically allocate processes; this capability will be explained later. It might be worthwhile at this point to discuss the mechanism that this package relies on.

5. The SCHEDULE Mechanism

The call to the SCHEDULE routines **dep** and **putq**, respectively, places process dependencies and process descriptors on a queue. A unique user supplied identifier *jobtag* is associated with each node of the dependency graph. This identifier is a positive integer. Internally it represents a pointer to a process. The items needed to specify a data dependency are non-negative integers *icango* and *ncheks* and an integer array *mychkn*. The *icango* specifies the number of processes that process *jobtag* depends on. The *ncheks* specifies the number of processes that depend on process *jobtag*. The *mychkn* is an integer array whose first *ncheks* entries contain the identifiers (i.e., *jobtag*s) of the processes that depend on process *jobtag*.

icango = 2

ncheks = 3

D'˜ 60u -116u 297u 0u'

D'c325u'

D'c195u'

D'l159u -239u'   D'l-75u -220u'

Figure 5.1.

A Node in a Dependency Graph

In Figure 5.1 a typical node of a data dependency graph is shown. This node has two incoming arcs and three outgoing arcs. As shown to the left of the node one would set *icango* = 2, *ncheks* = 3, and the first three entries of *mychkn* to the identifiers of the processes pointed to by the outgoing arcs.

The initial call to *sched*(nprocs,subname,<parms>) results in *nprocs* virtual processors called **work** to begin executing on *nprocs* separate physical processors. Typically *nprocs* should be set to a value that is less than or equal to the number of physical processors available on the given system. These **work** routines access a ready queue of *jobtag*s for schedulable processes. Recall that a schedulable process is one whose data dependencies have been satisfied. After a **work** routine has been successful in obtaining the *jobtag* of a schedulable process, it makes the subroutine call associated with that *jobtag* during the call to **putq**. When this subroutine executes a *return*, control is returned to **work**, and a SCHEDULE routine **chekin** is called which decrements the *icango* counter of each of the *ncheks* processes that depend on process *jobtag*. If any of these *icango* values has been decremented to zero, the identifier of that process is placed on the ready queue immediately.

We depict this mechanism in Figure 5.2 . The array labeled **parmq** holds a process descriptor for each *jobtag*. A process descriptor consists of data dependency information and a subroutine name together with a calling sequence for that subroutine. This information is placed on **parmq** through the two calls

```
CALL  DEP(jobtag,icango,ncheks,mychkn)
CALL  PUTQ(jobtag,subname,<parms>).
```

When making these two calls the user has assured that a call to *subname* with the argument list <*parms*> is valid in a data dependency sense whenever the counter *icango* has been decremented to the value zero. When a **work** routine has finished a call to **chekin** , it gets the *jobtag* of the next available schedulable process off the *readyq* and then assumes the identity of the appropriate subroutine by making a call to *subname* with the argument list <*parms*>.

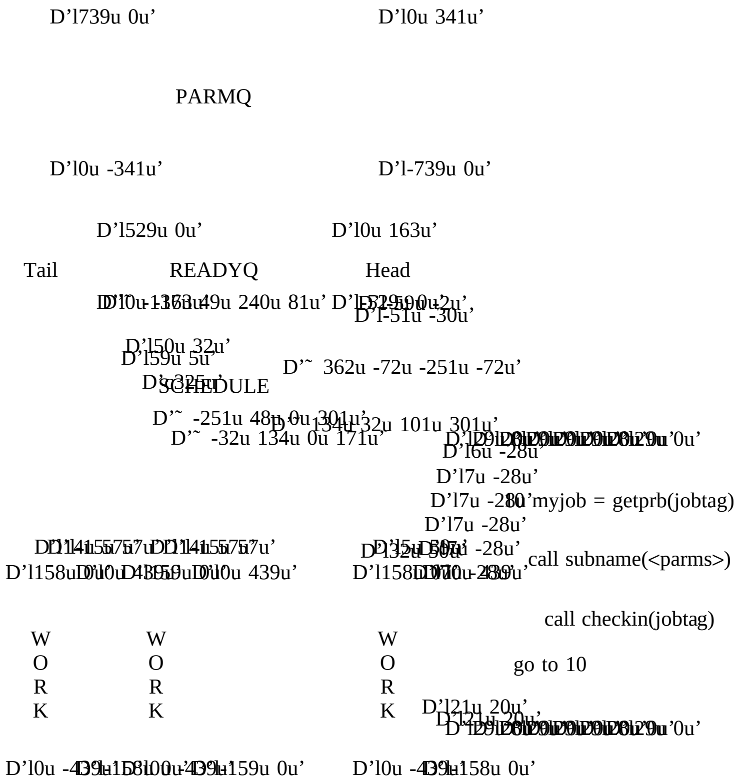

Figure 5.2.
The SCHEDULE Mechanism

6. How to Use Schedule

As explained in the previous sections the basic idea in writing a Schedule program is to partition a computation into units of computation and data dependencies between these units. In this section we assume familiarity with these concepts. We also assume familiarity with the notion of a data dependence graph and the association of nodes in this graph with units of computation. Data dependencies are really assertions made by the user and the responsibility for the correctness of these assertions rests with the user. Schedule provides a mechanism to record the units of computation, express the dependencies between them, and then execute the computation described by the Schedule program. It is very important to clearly define and understand the shared data of a problem and how it is to be partitioned for parallel processing.

Partitioning a Problem

The first thing that must be understood is the distinction between local and global variables. Global variables are shared amongst all the processes which reference them. Local variables are defined within a subroutine and a local version of these variables is obtained each time a unit of computation based upon this subroutine is executed.

All shared data should be grouped in named COMMON. This is not a strict requirement but it is a good programming practice and can increase performance on some machines. Before discussing how data is partitioned it is usefule to have an idea of what a Schedule program will look like. The structure of a Schedule program generally takes the following form:

Main Program

```
      common /prbdef/<array declarations >
      external paralg

      initialize variables
              .
              .
              .
         call sched(nprocs,paralg,<parms>)
              .
              .
              .
      output result

      stop
      end
```

Static Data Dependency Graph

```
      subroutine paralg(<parms>)
      declare global variables

      declare local variables
      external subname

      do 100 j = 1,nunits
                .
                .
                .
c
         jobtag = the identifier of this unit of computation
```

```
          icango  =  number of nodes jobtag depends on
          ncheks  =  number of nodes which depend on jobtag
          list    =  list of identifiers of these ncheks dependents
c
          call  dep(jobtag,icango,ncheks,list)
          call  putq(jobtag,subname,<parms>)
                 .
                 .
                 .
  100  continue
       return
       end
```

Unit of Computation with Dynamic Spawning

```
       subroutine  subname(<parms>)
       declare global variables

       declare local variables
       external  sublow

       do  200  j  =  1,nkids
           .
           .
           .
          call  nxtag(mytag, jdummy)
          call  spawn(mytag, jdummy, sublow, <parms>)
           .
           .
           .
  200  continue
       return
       end
```

In this generic example *nunits* units of computation were defined in the the 100 loop of the subroutine *paralg*. Here, all of the units of computation involve the subroutine *subname* operating on different data. There may be several different subroutines associated with various units of computation. The initial call to *sched* acquires *nprocs* physical processors devoted to the execution of the computation defined by subroutine *paralg*. There can be several calls to *sched* in the main program. Each call involves the overhead of acquiring *nprocs* physical processors on the given system. Care must be exercised on some systems because the expense of acquiring a number of physical processors may be substantial.

This example shows the three levels of parallelism typically available in a Schedule program. The subroutine *paralg* defines the static data dependency graph which is to be obeyed throughout the course of the computation. The subroutine *subname* exploits the lowest level of parallelism through dynamic spawning of processes involving *sublow*. This subroutine was constructed to dynamically spawn processes at run time using the calls to *nxtag* and *spawn* in the 200 loop.

More will be said about the details of this mechanism below. At this point it should be stressed that the subroutine *paralg* only records the data dependencies and the unit of computation associated with each node in the data dependency graph. None of these will execute until *paralg* completes. When *paralg* does complete (ie. executes a return statement) then the computation begins by executing those nodes which have *icango* = 0. During the execution, these units of computation will execute as their data dependencies are satisfied. Additional units of computation may be placed on the pool of work to be done during execution through the *spawn* mechanism.

Static Partition

In the generic example above, the subroutine *paralg* defines the static data dependency graph. It is important to understand that none of the units of computation defined through *dep* and *putq* will execute until a return has been executed by *paralg*. Thus the parameters passed through the calling sequence to *subname* must be global variables. That is, every variable passed through the calling sequence to *subname* with a call to *putq* must either reference a variable passed through the calling sequence of *paralg* or must be in shared common.

To define the static data dependency graph one must write a program that includes a call to *dep* followed by a call to *putq* corresponding to every node on the graph. Each node of the graph should be assigned a *jobtag*. These jobtags must be successive positive integers beginning at 1 and not exceeding 1000. In the program defining the static data dependency graph there must be the statements

call **dep**(jobtag,icango,ncheks,list)
call **putq**(jobtag,**subname**,<parms>)

corresponding to each node. The call to *dep* records the data dependencies for the node that has been labeled *jobtag*. The dependencies are defined through specification of

icango = *an integer specifying the number of nodes jobtag depends on*

ncheks = *an integer specifying the number of nodes which depend on jobtag*

list = *an integer array containing list of identifiers (ie. jobtags) of the ncheks processes dependent upon completion of jobtag .*

It is essential that at least one node has *icango* = 0 set. Without this condition the program cannot start. There can be several nodes with *icango* = 0. Exactly one node must have the condition *ncheks*=0 set. The program cannot finish without this. Error messages will be output and the Schedule program will execute a Fortran stop if either of these conditions are not met. A unit of computation is "scheduled" for execution when its *icango* has been set to 0 indicating that it is not waiting upon the completion of any other unit of computation. The *icango* counter of each of the *ncheks* dependents will be decremented automatically when *jobtag* completes. The unit of computation labeled *jobtag* is recorded through the call to *putq*. Once the data dependencies for *jobtag* have been satisfied a

call **subname** (<parms>)

will be executed where <parms> represents the list of parameters to be passed to the subroutine *subname*. The call to *putq* records the entry point to *subname* and the addresses of each parameter in the calling sequence <parms>. Execution of this unit of computation only occurs after the data dependencies have been satisfied and a physical processor has become available. The order of execution is dictated by the order in which schedulable processes get placed on a "ready to execute" queue. When the number of physical processors is 1 then this order is predetermined, but when more than one processor is active the order is somewhat nondeterministic. This queue is served by *nprocs* workers on a self scheduled.

Simple Spawn

If one wishes to do dynamic spawning within one of the executing processes that has been defined statically, there are three options. The simplest situation is when the spawning process does not do any additional work which depends upon the results of the spawned processes. In this case the spawning process has the form

subroutine subname(<parms>)
declare global variables

declare local variables

external **sublow**

```
      do 200 j = 1,nkids
         .
         .
         .
         call nxtag(mytag, jdummy)
         call spawn(mytag, jdummy, sublow, <parms>)
         .
         .
         .
  200 continue
      return
      end
```

Let us call the executing process defined by *subname* together with its arguments the parent. In the computation above, the parent process will spawn *nkids* child processes. The parameter *mytag* which appears in the calling sequence of both *nxtag* and *spawn* must contain the *jobtag* of the parent process. The parameter *mytag* must be stored in a global variable which has been passed to the parent *subname* either through the calling sequence or through named common. The calls to *nxtag* and *spawn* must be done together. The statement

call **nxtag**(mytag,jdummy)

returns a *jobtag* in the parameter *jdummy* which has been assigned by Schedule internally. The data dependencies are automatically defined: the each spawned process will be forced to complete before the parent can finish. The statement

call **spawn**(mytag,jdummy,**sublow**,<parms>)

is similar in nature to a call to *putq* in the sense that will invoke a call to subroutine *sublow*(<parms>).

If this mechanism for dynamic spawning is used then the parent *subname* must not do any computations dependent upon completion of any of the *nkids* invocations of the subroutine *sublow* associated with the spawning within the 200 loop. Any processes dependent upon completion of *subname* will be assured that each of these *nkids* spawned processes have completed when *subname* has completed.

Spawning with Multiple Entry Points .

On many occasions one wishes return execution control to parent process after the child processes have completed and resume computations which are dependent upon the completion of these child processes. This may be accomplished in the following way.

```
      subroutine subname(<parms>)
      declare global variables

      declare local variables

      external sublow
c
c     jump to entry point L000 if integer function ientry
c     returns the value L
c
      go to (1000,2000,...,N000),ientry(mytag,N)
 1000 continue
         .
         .
         .
      nkids = the number of child process to be spawned
              this number may be determined at run time
      do 200 j = 1,nkids
         .
         .
         .
         call nxtag(mytag,jdummy)
         call spawn(mytag,jdummy,sublow,<parms>)
         .
         .
         .
  200 continue

      return
 2000 continue
         .
         .
         .
      return
 N000 continue
         .
         .
         .
```

```
      return
      end
```

In this subroutine the labels 1000,2000,...,N000 should be considered as entry points. The entry points are taken in succession beginning with label 1000 and continuing through label N000. A *return* statement must precede each of the labels in this list. The subroutine *ientry*(*myid*,*N*) returns the number of times process *myid* has been entered so that the label L000 will be the entry point on the L -th time this process is entered. The integer N must correspond to the total number of entry points on this list. It is important to understand that this process will be exited at the *return* just before each entry point. Each time the subroutine is reentered execution begins at the first executable statement just as a normal subroutine, but when the computed *go to* statement is reached the function *ientry* will force the computation to execute at the label following the last return. Upon completion of the *nkids* spawned processes in the 200 loop computation will resume at the entry point 2000 just as in the simple case. The user is responsible for restoring the state of any local variables defined in subname at this point. In this method the subroutine *subname* is exited after each round of spawning. Another mechanism may be used to avoid this forced exit and also to specify a specific return label. This mechanism uses the logical function *wait* .

```
      subroutine subname(<parms>)
      declare global variables

      declare local variables

      logical wait
      external sublow
      go to (1000,2000,...,N000),ientry(mytag,N)
 1000 continue
         .
         .
         .
      do 200 j = 1,nkids
         .
         .
         .
         call nxtag(mytag,jdummy)
         call spawn(mytag,jdummy,sublow,<parms>)
         .
         .
```



```
              .
  200  continue

       if (wait(mytag,2)) return
 2000  continue
              .
              .
              .
       return
 N000  continue
              .
              .
              .
       return
       end
```

Of course *wait* must be declared logical in *subname* . The arguments to *wait* are *mytag* which has been set to the *jobtag* of the particular instance of *subname* followed by an integer which has been set to the number that ientry will return when *subname* is reentered. This mechanism sets a logical barrier which is not a busy wait. The functionality provides for exiting *subname* only if a copy of *sublow* has not yet completed.

Recursive Spawn To be supplied later.

7. Low-Level Synchronization

Ideally, the mechanism we have just described will relieve the user of explicitly invoking any synchronization primitives. Unfortunately, some powerful parallel constructs are not so easily described by this mechanism. It may be desirable to have two processes executing simultaneously that are not truly data independent of each other. A typical example is in pipelining a computation, that is, when several parallel processes are writing on the same data in a specified order which is coordinated through explicit synchronization. To provide this capability, two low-level synchronization primitives have been made available within SCHEDULE. They are **lockon** and **lockoff**. Each takes an integer argument. An example of usage is

```
call lockon(ilock)
     ilocal = indx
     indx = indx + 5
call unlock(ilock)
```

In this example a critical section has been placed around the act of getting a local copy of the shared variable *indx* and updating the value of *indx*. If several concurrent processes are executing this code, then only one of them will be able to occupy this critical section at any given time. The variable *ilock* must be a globally shared variable and it must be initialized before any usage as an argument to **lockon** or **unlock** by calling the routine **lckasn**. In the above example the statement

```
call lckasn(ilock)
```

must execute exactly once and before any of the calls to **lockon** are made. After execution of this statement the lock variable *ilock* has been declared as a lock variable and has been initiated to be *off*. If there are low-level data dependencies among any of the processes that will be scheduled, then it will be necessary to enforce those data dependencies using locks. It is preferable to avoid using locks if possible. However, in certain cases such as pipelining, locks will be required. mmmmm

8. Standard Tricks

In this section we discuss some standard graphs, performance issues and frequent errors.

A. Typical Graphs

d and c
fork join
tsolve

Common Errors and How to Avoid Them

Certainly the most common error is failing to declare a subroutine external before referencing it in a call to putq or spawn. This will cause unusual behavior which usually results in a bus error. It is the first thing one should check when such unexplained behaviour occurs.

Perhaps the most serious error is to pass an argument to static or spawned subroutine which is not global. These errors are difficult to trace and find. The first clue of this type of error is to have a correctly running program when only one physical processor is active which fails when more than one processor is active.

Another error which gives similar behavior is to povide incorrect analysis of data dependencies. This is when the user describes the data dependency graph incorrectly in the sense that two processes which read and write the same data item are allowed to execute simultaneously. Again the symptom is a correctly running program on one processor that fails on more than one processor.

A related error which is more easily found is the improper specification of the graph through gross programmingerror. This will usually result in a deadlock situation or in incorrect results even with only one processor active. An easy way to check for such error is to use the graphics facility described later.

Dynamic spawning is usually straightforward unless there are reentry points. When there are multiple entry points a typical error is failing to restore state after spawning processes. For example

```
         .
         .
         .
         call spawn(...)
         k = 5
      return
 2000 continue

      m = k*(k+1)
         .
         .
         .
```

will give an incorrect result for *m* if *k* is a local variable in the subroutine that is spawning. To make this correct the user must re-initialize *k* after entry point 2000.

Transporting Users Code

To be supplied later

9. GRAPHICS TRACE FACILITY

Performance issues are vital to the effective use of computers. Yet the current status of software environments for implementing parallel algorithms is in a primitive state. Consequently, we felt compelled to develop a system (described in the previous section) to allow transportable implementation of parallel algorithms. We feel equally compelled to develop tools to aid in understanding how our parallel implementation performs when run on a parallel processor.

Our goals here are simple: to be able to trace the flow of execution within our application and to understand where bottlenecks occur. Monitoring the run time by using a timing routine has been employed in the past. This approach, however, has a number of limitations in the parallel setting. We want an animation of run-time behavior, visualizing the parallel parts in execution as the application is running. We would like to understand what performance issues arise during execution and what bottlenecks develop, and to view where programming errors cause a parallel programming to "hang."

To achieve these goals, we have developed graphics tools that allow us to trace the flow of execution. In addition, we have devised tools to aid in constructiing programs using our SCHEDULE package.

The graphics interface that provides trace information has been designed to use a Sun-3 workstation. Our plan in the near term is to convert the trace program to use X-windows, a portable windowing system.

We hope these tools can be used to influence the design of an algorithm by providing insight into how the program behaves when run in parallel, thus allowing use to tune the algorithm. Additionally, we hope that these tools will further our understanding of the nature of the parallel algorithm and of parallel processing in general.

10. Tracing an Execution

To trace the flow of execution, an algorithm is implemented using the SCHEDULE package, linked with the trace version of the Schedule library, and run on a parallel machine.

The execution produces an output file of trace events in a file named *trace.graph*. The information in the file is a trace of the user specification of the computational graph and the execution of the graph by the computer. Each node in the graph represents a subroutine, and the arcs represent the dependencies between the codes. The trace file records the definition of the graph along with the execution and completion of the subroutine. When execution has completed, this file is then moved to a Sun workstation where the trace facility has been installed. (It is assumed that suntools are used.)

The information in the trace files has the following meaning from the Scheduler.

Trace File Items

0	id	# child	# parents	parent ids
1	id			
2	id	timing info		
3	owner id	id		
4	owner id	id		
5	owner id	id	timing info	

Rather than write each event out as it occurs, the Scheduler buffers up a set of trace events. When a limit is reached, the next set of trace events is written to the trace file.

Trace events begin with a digit that defines the operation the Scheduler has performed.

Definition of Events

0	create a static node
1	start a static node executing
2	stop a static node executing
3	create a dynamic node
4	start a dynamic node executing
5	stop a dynamic node executing

11. Replaying a Trace

After the program has completed execution, the trace file can be viewed on the Sun workstation. The trace file is moved to the Sun file system and given a name of the form *trace.xxxxx.* (If NFS is used between the Sun and the parallel machine, the file can be read directly.) To start the trace program, one types *sched.trace*. Figure 1 shows what will appear on the canvas on the Sun screen.

Figure 1: Information on the Sun Screen

A number of buttons and slide bars help in controlling the information to be viewed. After the canvas is on the screen, a trace file must be loaded. The file that is currently referenced is displayed on the second line next to the words **Trace file:.** The file name can be changed by moving the mouse cursor to point to **Trace file:** and clicking the left mouse button through all the trace files available for viewing. Alternatively, one can move the mouse cursor to **Trace file:** and hold down the right button; this will result in the display of a popup menu showing all the trace files to choose from. To select a file, one moves the mouse cursor into the popup box while still holding down the left button, and releases the button on the desired file.

After the desired file is selected, it must be loaded into the system. This is accomplished by clicking the **Load** button. The count of number of events loaded is displayed at the end of the slider bar marked **Events.**

To start the execution, one clicks on the **Go** button. When the **Go** button is clicked initially, the events are processed internally until the first executable event (one of the events that has as its leading digit 1-5) is reached; then the data dependency graph is displayed on the screen. The nodes in the dependency graph will change their state as the trace proceeds. There are four types of nodes in the graph: clear, lined, hashed, and black. The *clear* nodes represent subroutines waiting to have their data dependencies satisified before starting execution. The *lined* nodes represent subroutines whose data dependencies are satisfied, but whose execution has not started. The *hashed* nodes represent subroutines that are currently executing. The *black* nodes represent subroutines that have finished execution. The arcs represent the data dependenciess between subroutines. The numbers associated with each node in the graph represent the *jobtag* for that node as defined through the user calls to *dep* and *putq.*

The viewing of events can be stopped or restarted by clicking on the **Stop** or the **Go** buttons at any time. Execution begins at the bottom of the graph and works its way to the top, with the top node the last thing to execute.

The **Quit** button is used to terminate the trace facility.

The **Full** button is used in conjunction with the **Redraw** button to expand the canvas by a factor of four. First the **Full** button is clicked; then the **Redraw** button clicked to redisplay the expanded canvas. The **Full** button is really a toggle. Thus, to get the "normal" canvas size displayed, one clicks the **Full** button, then the **Redraw** button.

The next two buttons, **Path** and **Cpath,** are used together after the execution has completed. They are intended to display the critical path of execution in the graph (i.e., the path that is taking the most time to execute). The critical path determines the running time of the entire application. To use this feature, one clicks on the **Cpath** button and then the **Path** button. **Cpath** is used to compute the critical paths in the graph and **Path** to display the path. *

The **Subtree** button is used to display a subtree of the graph. To use this feature, one places the mouse cursor on a node. The node number clicked on will be displayed in bottom right line of the top panel. To see the subtree, one clicks the **Subtree** button. This step can be repeated to zoom in on a particular part of the graph. To redisplay the full graph, one clicks the **Redraw** button.

The **Step** button is used to single-step through the execution of the trace file. A way to use this is by loading a trace file, starting execution, and stopping execution at the desired point. Then the **Step** button is used to single-step through the execution. The activity can be resumed at normal speeds by clicking the **Go** button.

The last button is **Histogram.** When clicked, this will display a histogram of the events run versus the number of processors active.

---

* In the current release of the graphics program, the critical path is chosen by the maximum number of nodes from the start to finish in the graph. This should really be based on time, not number of nodes.

12. Building a Schedule Program

13. Futures

Use sockets to pass information between processor and graphics.

Use X-windows to provide a portable environment.

14. Conclusions

Proving useful in understanding run-time behavior of our parallel programs. Allows us to see into the machine. Simplifies the task, no longer required examine output.

15.
Schedule Subroutines

This section contains a list of Schedule subroutines and a brief description of their functionality.

PUTQ

Used to place descriptor of a statically defined unit of computation
on an internal problem list

Syntax

putq(jobtag,subname,<parameter list>)

Comments

This subroutine is used to place the descriptor of the unit of computation indexed by *jobtag* on the problem list. Every Schedule program must have at least one call to *putq*. Moreover, a call to *putq* must be preceded by a call to *dep*:

```
call dep(jobtag,icango,nchks,list)
call putq(jobtag,subname,<parameter list>)
```

The parameter list to the right of *subname* should be a valid set of parameters to make the call:

```
call subname(<parameter list>) .
```

This call will be made when the data dependencies associated with *jobtag* have been satisfied. The program calling putq must have *subname* declared external.

Calling Sequence

jobtag - an integer which labels the unit of computation this integer may be a local variable in the subroutine calling *putq*

subname - the name of the Fortran subroutine to be called when data dependencies for the process indexed by *jobtag* have been satisfied.

<parameter list> - a list of parameters for a call to *subname*. they must be global variables and there should not be more than 20 parameters.

Restrictions

The number of parameters in <parameter list> must not exceed 20.

Each parameter passed in <parameter list > must be a location in shared memory.

Constants and expressions should not be passed.

Variables declared locally within the subroutine making the call to *putq* should not be passed in <parameter list>.

DEP

defines the data dependencies for a unit of computation

Syntax

dep(jobtag,icango,ncheks,list)

Comments This subroutine puts data dependencies for process *jobtag* on the problem list. A call to *dep* is always associated with a call to *putq* :

call dep(jobtag,icango,nchks,list)
call putq(jobtag,yyy,<parameter list>)

Calling Sequence

jobtag - a positive integer indexing a unique unit of computa

icango - a positive integer specifying how many processes must check in to this process before it can be placed on the ready

nchks - the number of processes that depend upon the completion of this process.

list - an integer array specifying the jobtags of the proce which depend upon completion of this process.

Restrictions

The integer *jobtag* must be between 1 and 1000.

The integer *nchks* must be between 0 and 25.

The array *list* must have *nchks* entries which are the *jobtag*s of dependent processes.

NXTAG

Claims the next available *jobtag* for dynamic spawning

Syntax

nxtag(mypar,jobtag)

Comments This subroutine is used to claim the next available *jobtag* for dynamic spawning. The parent process *mypar* makes a call to *nxtag* and receives a valid *jobtag* for a child process when *nxtag* returns. A call to *nxtag* is always followed by a call to *spawn*:

```
call nxtag(mypar,jobtag)
call spawn(mypar,jobtag,subname,<parameter list>)
```

Data dependencies are set automatically so that *mypar* depends upon the completion of the child process. The child *jobtag* will have its *icango* counter set to 0.

Calling Sequence

mypar - an integer containing the index of the parent process that will spawn a child process with index *jobtag*.

jobtag - an integer labeling a dynamically spawned unit of pn which will be given the data dependency of a child of *mypar*.

Restrictions

The number of calls to *nxtag* plus the number of calls to *dep* may not exceed 1000.

WAIT

Used to set a logical barrier after dynamic spawning

Syntax

wait(mytag,lablno)

Comments

This logical function will be used to set a logical barrier in process *mytag* after the dynamic spawning of a number of child processes. The process will be allowed to continue execution immediately after this barrier once all of the spawned child processes have completed. It should only be used if child processes have been spawned by *mytag* through the use of the subroutine *spawn*. This routine must be used in conjunction with integer function *ientry*. The required usage is

```
      go to (1000,...,L000,...,N000), ientry(mytag,N)
 1000 continue
        .
        .
        .
      do 100 j = 1,nproc
             .
             . (set parameters to define spawned process)
             .
         call nxtag(mytag,jobtag)
         call spawn(mytag,jobtag,subname,<parms>)
  100 continue
      label = L
      if (wait(mytag,label)) return
 L000 continue
        .
        .
        .
```

If this function returns a value of .true. then the calling process *mytag* should issue a return. A return value .true. indicates that some spawned process has not yet completed and checked in. When this happens the parent process *mytag* must be exited to avoid busy waiting and also to allow this code to execute on a single physical processor. When all of the spawned processes have completed then the parent process *mytag* will be reentered, execute the code before the computed *go to* referencing *ientry*, and then resume execution at the statement immediately following the reference to *wait* (ie. at L000 in the example above). A return value .false. indicates all spawned processes have checked in. If a value of .false. is returned then the parent process simply continues to execute without exiting (ie. at L000 in the example above).

Calling Sequence

mytag - an integer containing the index of the parent proces that has spawned at least one child process

label - an integer indicating where computation is to resume if all spawned child processes have completed. If *wait* returns .true. then a return should be made and the next reentry of process *mytag* the routine *ientry* will return the value *label*.

Restrictions

There must be an invocation of *ientry* in the process *mytag*.

SPAWN

Used to dynamically spawn processes during execution

Syntax

spawn(mypar,jobtag,subname,<parameter list>)

Comments

This subroutine puts the descriptor of a unit of computation labeled by *jobtag* onto the problem list. This process will be scheduled for execution immediately since it is a leaf to the parent node *mypar* in the dynamic data dependency graph. Moreover, a call to *spawn* must be preceded by a call to *nxtag*:

```
call nxtag(mytag,jobtag)
call spawn(mytag,jobtag,subname,<parmeter list>)
```

The parameter list to the right of *subname* should be a valid set of parameters to make the call:

```
call subname(<parameter list>) .
```

This call will be made when the data dependencies associated with *jobtag* have been satisfied. The subroutine calling *spawn* must have *subname* declared external. The action of this procedure differs from *putq* in that the user does not assign jobtags or data dependencies. A parent may spawn any number of children (within limits given below) but these child processes only report to the parent.

Calling Sequence

mypar - an integer which labels the parent process which is spawning *jobtag* with a call to *spawn*.

jobtag - an integer which labels the unit of computation this integer may be a local variable in the subroutine calling *spawn*. Its value must have been assigned by Schedule through a call to *nxtag*.

subname - the name of the Fortran subroutine to be called whe data dependencies for the process labeled by *jobtag* have been satisfied.

<parameter list> - a list of parameters for a call to *subname*.
they must be global variables and there should
not be more than 20 parameters.

Restrictions

The number of parameters in <parameter list> must not exceed 20.

Each parameter passed in <parameter list > must be a location in shared memory.

Constants and expressions should not be passed

Variables declared locally within the subroutine making the call to *putq* should not be passed in <parameter list>.

SCHED

Acquires physical processors and executes the Schedule program

---

Syntax

sched(nprocs,static,<parameter list>)

Comments

This subroutine initializes locks and queues internal to Schedule, acquires *nprocs* physical processors, executes *static*(<*parameterlist*>) and then executes the Schedule program defined by *static*. It is very important to understand that the execution of *static* merely describes a Schedule program by placing units of computation their data dependencies on a problem list. This computation proceeds only after *static* has completed. The parameter list to the right of *static* should be a valid set of parameters to make the call:

call static(<parameter list>) .

The program calling *sched* must have *static* declared external. Control is returned to the calling program from *sched* once the entire computation defined by *static* has been completed and a single thread of execution is resumed within the calling program. There may be any number of calls to *sched* within the calling program but the user must be aware that the cost of obtaining *nprocs* physical processors is incurred with each call to *sched*. Note it is useful to do an initial debugging phase with *nprocs* set to 1. Once the correctness of the program is established with *nprocs*=1 the remaining bugs are generally associated with incorrect partitioning of shared data and incorrect specification of data dependencies.

Calling Sequence

nprocs - an integer which specifies the number of physical p[illegible]
which are to be devoted to the execution of the Schedule
program defined by *static*.

static - the name of the Fortran subroutine to be called to
build the static data dependency graph and record [illegible]
of the units of computation associated with each node of the graph.

<parameter list> - a list of parameters for a call to *static*.
they must be global variables and there should not be

more than 20 parameters.

Restrictions

The number of parameters in <parameter list> must not exceed 20.

Each parameter passed in <parameter list > must be a location in shared memory.

Constants and expressions should not be passed

Variables declared locally within the subroutine making the call to *putq* should not be passed in <parameter list>.

lockon
lockoff

LOCKON, LOCKOFF

Low level synchronization primitives

Syntax

lockon(lock) lockoff(lock)

Comments

Subroutine lockon takes an integer argument *lock*. the process which calls *lockon* spin-waits until the value of *lock* is has the value 0. this is read and set to 1 in an autonomous operation. Subroutine lockoff sets the value of *lock* to 0. Caution must be exercised in using locks. In particular one must be careful to initialize properly.