--

--

.

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs

Jack J. Dongarra and Danny C. Sorensen

Mathematics and Computer Science Division

Technical Memorandum No. 86

November, 1986

--

--

SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs *

J. J. Dongarra and D. C. Sorensen

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

and

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
305 Talbot Laboratory
104 South Wright Street
Urbana, Illinois 61801-2932

1. Introduction

Many new parallel computers are now emerging as commercial products dongarra duff 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 vanrosendale mcgraw sisal 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 jordan to attempts at automatically detecting loop-based parallelism with a preprocessing compiler aided by user directives alliant The act of getting a parallel process executing on a physical processor ranges from a simple "create" statement jordan which imposes the overhead of a subroutine call, to "tskstart" cray multitasking which imposes an overhead on the

* 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, and DE-FG02-85ER25001.

-- --

order of 10^6 machine cycles, to no formal mechanism whatsoever alliant 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 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 Babb babb Browne browne and Lusk and Overbeek lusk 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

D’c105u’

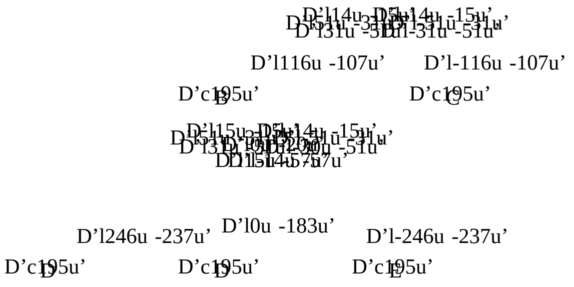

Figure 4.1.
Data Dependency Graph

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)

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

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
```

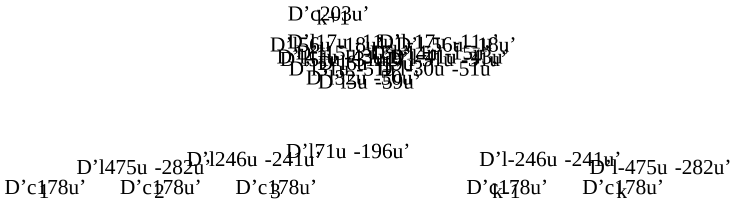

Figure 4.2.
Depedency Graph for Parallel Inner Product

In this data dependency graph we have identified k processes

$$inprod(\ m,\ a(indx),\ b(indx),\ temp(j)\ )\ ,\quad j = 1, 2,..., k\quad ,\quad indx = 1 + (j-1)*m$$

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*>.

D'325u' SCHEDULE

W
O

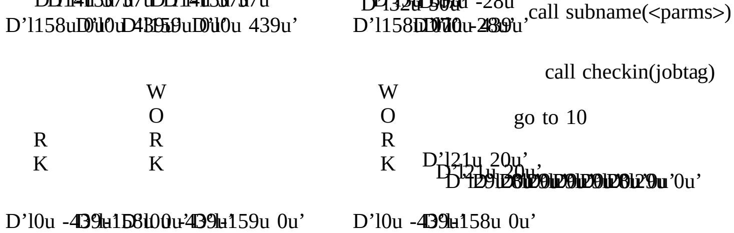

Figure 5.2.
The SCHEDULE Mechanism

6. 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 **unlock**. 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.

7. Dynamic Allocation of Processes

The scheme presented above might be considered static allocation of processes. By this we mean that the number of processes and their data dependencies were known in advance. Therefore the entire data structure (internal to SCHEDULE) representing the computational graph could be recorded in advance of the computation and is fixed throughout the computation. In many situations, however, we will not know the computational graph in advance, and we will need the ability for one process to start or spawn another depending on a computation that has taken place up to a given point in the spawning process. This dynamic allocation of processes is accomplished through the use of the SCHEDULE subroutine **spawn**. The method of specifying a process is similar to the use of **putq** described above.

We shall use the same example to illustrate this mechanism.

-- --

Processes:

subroutine inprod .... same as above

```
      subroutine addup(n,k,a,b,sigma,temp)
      integer myid,n,k
      real a(*),b(*),sigma,temp(*)
c
c     declare local variables
c
      integer j,jdummy,m1,m2
c
      EXTERNAL INPROD
c
      m1 = n/k
      indx = 1
c
      CALL PRTSPN(myid)
      do 200 j = 1,k-1
c
c        replace the call to inprod  with a call to spawn
c
         CALL SPAWN(myid,jdummy,INPROD,m1,a(indx),b(indx),temp(j))
         indx = indx + m1
  200 continue
      m2 = n - indx + 1
c
c     clean up step
c
      call inprod(m2,a(indx),b(indx),temp(k))
c
c     keep one of the inner product assignments and then call wait
c     to be sure all others have completed.
c
      CALL WAIT(myid)
c
c
c     All have checked in, now addup the results.
c
      sigma = 0.0
```

```
      do 100 j = 1,k
         sigma = sigma + temp(j)
  100 continue
      return
      end
```

The subroutine **parprd** must change somewhat.

```
      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 mychkn(1),icango,ncheks,jobtag
      EXTERNAL ADDUP
c
      JOBTAG = 1
      ICANGO = 0
      NCHEKS = 0
c
      CALL DEP(jobtag,icango,ncheks,mychkn)
      CALL PUTQ(jobtag,ADDUP,n,k,a,b,sigma,temp)
c
      return
      end
```

In this example the call to *spawn* invokes $k - 1$ instances of the subroutine *inprod*. Note that the user does not specify data dependencies. Instead, the jobtags for these processes are assigned internally and they are required to report to the spawning process. More details of this mechanism are given in the next section.

8. Syntax Summary

In this section we summarize the SCHEDULE operations and their syntax. We begin with a list of these operations and follow up with a general description of their usage. With this we hope to supplement the examples given above.

Table 1

User Calls

```
CALL SCHED ( nprocs, subname, <parms> )
CALL DEP   ( jobtag, icango, ncheks, mychkn )
CALL PUTQ  ( jobtag, subname, <parms> )
CALL PRTSPN( myid )
CALL SPAWN ( myid, jdummy, subname, <parms> )
CALL WAIT  ( myid )
```

```
CALL LCKASN( ilock )
CALL LOCKON( ilock )
CALL UNLOCK( ilock )

nprocs   - Number of virtual processors to be used on subname.
subname  - Subroutine to be executed.
<parms>  - list of parameters for subname.
jobtag   - A unique user supplied identifier.
icango   - The number of processes that must complete before
           this process starts.
ncheks   - Specifies the number of processes that depend on
           this process.
```

- - - -


```
mychkn   - An integer array whose first ncheks entries
           contain the identifiers of the processes that
           depend on this process.
myid     - An alias process id assigned by SCHEDULE to record
           completion of the spawned processes.
jdummy   - A dummy integer parameter.
ilock    - A globally shared variable used as a lock.
```

There are three basic operations in SCHEDULE other than the lowest level synchronization provided through locks. They are **call sched**, **call putq**, and **call spawn**. Various additional calls must be made in association with these three basic operations to enforce data dependencies, and orchestrate the correct execution of the parallel program.

The statement

call sched(*nprocs*,*paralg*,<*parms*>)

is used to initiate the execution of a parallel program *paralg* and will devote *nprocs* physical processors to the computation. The symbol <*parms*> stands for a set of parameters that would normally be supplied to the Fortran subroutine *paralg* through a Fortran call in a sequential program. One should think of the memory associated with the parameter list represented by <parms> as shared memory throughout the execution of *paralg*. Another way to share memory is through the use of *named common*.

The pair of statements

call dep(*jobtag*,*icango*,*ncheks*,*mychkn*)

call putq(*jobtag*,*subname*,<*parms*>)

are used to place a process in the large grained data dependency graph with *dep* used to record the data dependencies and *putq* to record the unit of computation. In this case the unit of computation will be a call to the subroutine *subname* with the calling sequence <*parms*>. When the data dependencies for this process have been satisfied the *icango* counter will have been decremented to zero and this process will be placed upon the ready queue. At some point in time a physical processor executing a *work* program will be free and will pick this process off of the ready queue and execute a *call subname*(<*parms*>). It is the user's responsibility to assure that the addresses passed in <*parms*> are addresses in shared memory. That is to say that these addresses must reference shared memory that has been declared in the program that made the call to *sched*. Hence these addresses must come from named common declared in that program and referenced by the subroutine *subname* or which have been passed as parameters in the call to *sched*.

-- --

The sequence of statements

call prtspn(*myid*)

. . .

call spawn(*myid*,*jdummy*,*subname*,<*parms*>)

. . .

call wait(*myid*)

accomplish dynamic spawning of processes. Usually calls to spawn are made in a loop but they need not be. The call to *prtspn* secures an alias process id *myid* to record completion of the spawned processes. This id must be used in the subsequent calls to *spawn* and to *wait*. There is an implied barrier at the call to *wait* and execution does not continue past this point in the spawning process until all spawned processes have been completed. The syntax of the call to *spawn* is very similar to that of *putq*. The parameter *jdummy* is a temporary location that must be supplied to *spawn* and the result is a *call subname*(<*parms*>) as soon as there is an available instantiation of *work* available to execute this call. A key difference between a call to *spawn* and a call to *putq* is the lack of user specification of jobtags and data dependencies. The only data dependency indicated here is that a spawned process must report to the parent process and no other. This is taken care of automatically with use of the above mechanism. If communication is desired at a lower level among spawned processes this must be accomplished through the use of shared data and/or locks. Local variables in the spawning routine may be passed as arguments to the spawned routines with the understanding that their scope extends to all members of the subtree rooted at the spawning process. These variables must be treated as shared data within this subtree.

The implications of the mechanism used for spawning should be discussed further. We rely upon the stack associated with the calling tree rooted at the spawning process. If one wishes to do recursion to gain parallelism then there is potential for stack overflow. This is an unfortunate consequence of an attempt to avoid the burden of placing and keeping track of entry points and saving the state of an executing process which spawns other processes. The key problem is to retain the ability to execute the parallel program on a single physical processor.

The call to *wait* does not result in a busy wait as one might suspect. Instead, a call is made to a routine that is quite similar to the *work* routines discussed earlier. This routine accesses the readyq makes a call and then "checks in" and returns to *wait*. Within the routine *wait* completion of spawned processes is monitored. If any have failed to complete another call to the special work routine is made. Obviously, this

-- --

may result in stack overflow when spawned routines spawn other routines. An alternative was used in this package previously dongarra sorensen full parallel which avoided stack overflow but was deemed overly complicated for the user.

9. An Environment for the Development of Explicitly Parallel Programs

As we mentioned at the outset, SCHEDULE is a programming aid that is intended to serve as a backbone to an environment for the development of explicitly parallel programs. Our goal is to provide a uniform interface to the parallel capabilities provided by existing and impending parallel systems. Included in this are tools for debugging and analyzing the performance of parallel programs. In this section we shall describe some of our goals in this area. Preliminary attempts at achieving these goals have been implemented or are in the design stage.

We regard SCHEDULE as a tool primarily designed as an aid to constructing an implementation of a "new" parallel algorithm. We do not think it is particularly well suited to converting an existing serial code to a parallel version, although this would be possible in some cases. The reason for this statement is that to program with SCHEDULE it is imperative that the large grain data dependencies are well understood by the programmer. This is of course required in any successful implementation of a parallel program. However, with SCHEDULE the data partitioning and construction of the data dependency graph are an explicit part of the programming effort. We expect that the designer of a parallel algorithm will have this information naturally at hand. At least this has been our experience in the design of our own parallel algorithms dongarra sorensen Linear Algebra on High-Performance Computers Enforcing this programming style goes a long ways towards avoiding bugs traditionally associated with parallel programming.

We have found it very useful to retain the capability of executing a parallel program in serial mode. A SCHEDULE program does not depend upon the number of physical processors available in a given system. It will execute with one processor active. Others have found this to be a useful property of such a programming tool lusk overbeek However, Gentleman gentleman private does not feel that this is feature is essential to the dubugging of parallel programs. We find at least two reasons to provide it. First it allows one to develop and test code on a serial machine such as a VAX 11/780. This is quite important when the parallel machine is at a remote site or if the software development environment existing on the parallel machine is inadequate or difficult to use. Second, when developing a new algorithm one would like to be assured that the numerical properties of the algorithm are correct in serial mode before entering the parallel testing regime. Operating in this sequence tends to separate ordinary programming bugs from those associated

-- --

with parallel programming. Moreover, due to the SCHEDULE mechanism one can be fairly confident that a parallel bug is due either to incorrect specification of the data dependency graph or to incorrect partitioning of data. Explicit synchronization is generally not a part of a SCHEDULE program and thus difficult synchronization bugs do not generally arise.

To aid in ascertaining a correct specification of the data dependency graph we envision a tool that will graphically represent the units of computation and the data dependency graph. A preliminary version of this tool has been implemented and experience with this tool is reported below. Similar ideas have been proposed in babb In our view the level of detail in those efforts is too fine. We expect the level of detail in such a representation of a parallel program to roughly correspond to the abstract level at which the programmer has partitioned his parallel algorithm. Let us illustrate this with a simple example. A favorite example of our's is the solution of a triangular linear system partitioned by blocks.

10. Triangular Solve Example

We can consider solving a triangular system of equations *Tx*=*b* in parallel by partitioning the matrix *T* and vectors *x* and *b* as shown in Figure 10.1.

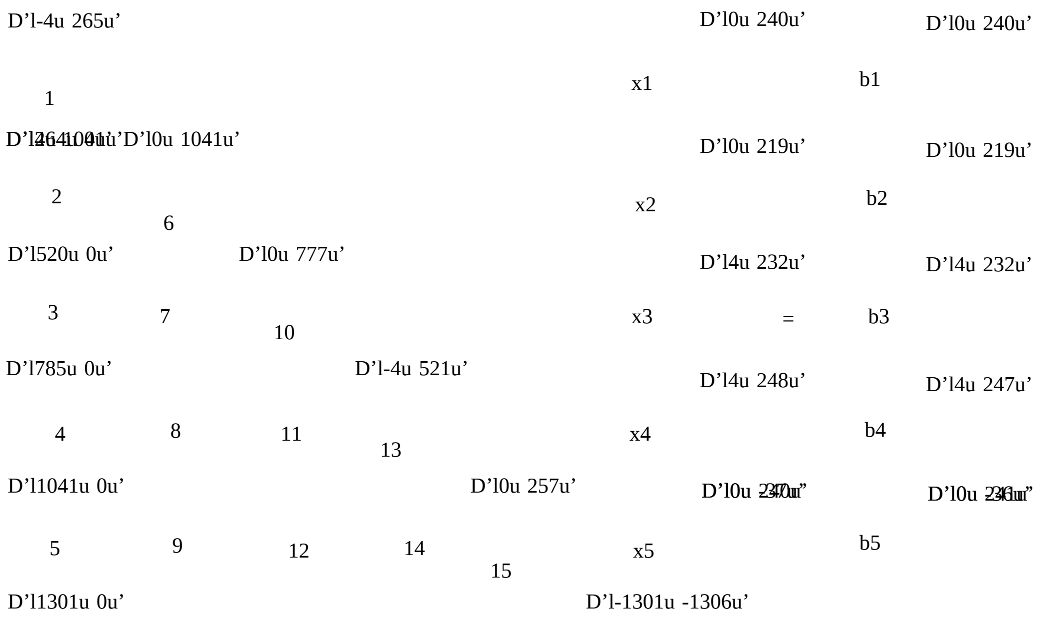

Figure 10.1
Partitioning for the Triangular Solve

The first step is to solve the system $T_1 x_1 = b_1$, this will determine the solution for that part of the vector labeled x_1. After x_1 has been computed it can be used to update the right hand side with the computations

$$b_2 = b_2 - T_2 x_1$$

$$b_3 = b_3 - T_3 x_1$$

$$b_4 = b_4 - T_4 x_1$$

$$b_5 = b_5 - T_5 x_1 \ .$$

Notice that these matrix-vector multiplications can occur in parallel, as there are no dependences. However, there may be several processes attempting to update the value of a vector b_j (for example 4,8,11 will update b_4) and this will have to be synchronized through the use of locks or the use of temporary arrays for each process. As soon as b_2 has been updated, the computation of x_2 can proceed as $x_2 = T_6^{-1} b_2$. Notice that this computation is independent of the other matrix-vector operations involving b_3, b_4, and b_5. After x_2 has been computed, it can be used to update the right hand side as follows:

$$b_3 = b_3 - T_7 x_2$$

$$b_4 = b_4 - T_8 x_2$$

$$b_5 = b_5 - T_9 x_2 \ .$$

The process is continued until the full solution is determined. The data dependency graph for this can be represented in Figure 10.2.

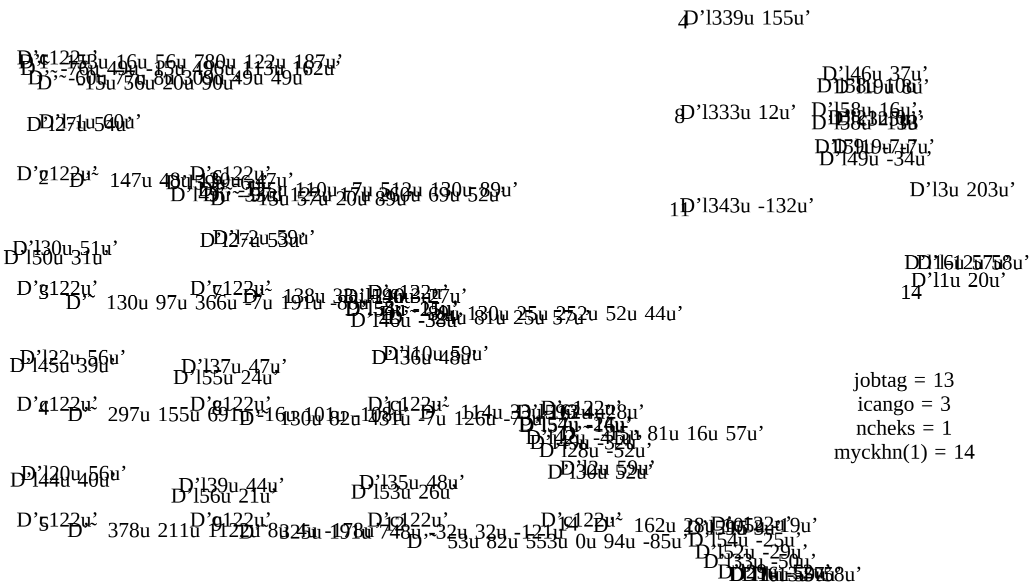

Figure 10.2

Data Dependency Graph for Triangular Example

Figure 10.3
Output from SCHEDULE for Triangular Example

In Figure 10.3 we show the output from an executing SCHEDULE implementation of this example. The program was run on an Alliant FX/8. An output file was produced as the program executed which recorded the units of computation as they were defined and executed. The file was then shipped to a SUN workstation where a graphics program interpreted this output, constructed the graph and played back the execution sequence that was run on the Alliant. In the graph shown above in Figure 10.3 the black nodes show processes which have completed execution, the hatched nodes show executing processes and the white nodes show processes waiting to execute. The program was able to produce this playback output automatically simply by linking to SCHEDULE with a request for the graph option. The user's program did not change in any way. At the moment this is the only capability we have. It would be very useful have the user construct that part of the Fortran program that corresponded to the dependence graph, i.e. calls to the SCHEDULER routines, from a "mouse driven" graphics input device.

In addition to discovering bugs in the specification of the graph, this representation is useful in exposing more subtle aspects of the executing program. For example when the graph produced by SCHEDULE is contrasted with the abstract user specified graph there are noticeable differences. The graph produced by SCHEDULE exposes inherent serial bottlenecks in the algorithm. During execution the two bar graphs at the top track the existing and maximum number of concurrently executing processes respectively. In the ideal situation, the bottom bar has a value equal to he number of physical processors available and the top bar is almost always equal to the bottom bar. In addition to discovering serial bottlenecks, one can learn about load balancing problems within an executing program. It may even be useful to put in artificial data dependencies to force certain processes to complete before others to achieve a better load balance. Another interesting feature is that this execution sequence can be accelerated if the program took a long time to execute and slowed down if it took a short time while maintaining the ratios of execution times among the parallel processes.

A final feature that is possible but not fully implemented is the possibility of observing the executing program within a node. We envision the capability of opening a workstation window corresponding to a selected node. Within this window we would see a display of selected variables being updated as execution takes place. At the same time the currently executing lines of Fortran code would be displayed. This capability currently exists for a serial Fortran program dongarra sorensen aid to fortran The output of such a display is shown in Figure 10.4

-- --

Figure 10.4
Screen Output from Debugger

The main problem to overcome here is selected display of data. A real program executing on large arrays could produce an overwhelming display of output.

11. Experience with SCHEDULE

At present the experience with using SCHEDULE is limited but encouraging. Versions are running successfully on the VAX 11/780, Alliant FX/8, and CRAY-2 computers. That is, the same user code executes without modification on all three machines. Only the SCHEDULE internals are modified, and these modifications are usually minor, but can be difficult in some cases. They involve such things as naming and parameter-passing conventions for the C - Fortran interface. They also involve coding the low-level synchronization primitives and managing to "create" the **work** processes.

On the CRAY-2 process creation is accomplished using **tskstart**, and the low-level synchronization is very similar to the low-level synchronization routines provided by the CRAY multitasking library cray multitasking For the Alliant FX/8 we coded the low-level synchronization primitives using their test-and-set instruction. To "create" the work routines, we used the CVD$L CNCALL directive before a

-- --

loop that performed *nprocs* calls to the subroutine **work**.

In addition to some toy programs used for debugging SCHEDULE, several codes have been written and executed using SCHEDULE. These codes include the algorithm TREEQL for the symmetric tridiagonal eigenvalue problem dongarra sorensen symmetric a domain decomposition code for singularly perturbed convection-diffusion PDE chin mcgraw an adaptive quadrature code comer and a block preconditioned conjugate gradient code for systems arising in reservoir simulation diaz

The graph associated with TREEQL is probably the most illustrative of the capabilities of SCHEDULE so we shall discuss it in some detail. This is a parallel algorithm for computing the eigensystem of a real symmetric tridiagonal matrix T. The problem is to compute an orthogonal matrix Q and a diagonal matrix D such that $T = QDQ^T$. The crux of the algorithm is to divide a given problem into two smaller subproblems. To do this, we consider the symmetric tridiagonal matrix

$$(11.1) \qquad T = \begin{pmatrix} T_1 & \beta e_k e_1^T \\ \beta e_1 e_k^T & T_2 \end{pmatrix}$$

$$= \begin{pmatrix} \hat{T}_1 & 0 \\ 0 & \hat{T}_2 \end{pmatrix} + \beta \begin{pmatrix} e_k \\ e_1 \end{pmatrix} (e_k^T \, , \, e_1^T)$$

where $1 \le k \le n$ and e_j represents the *j–th* unit vector of appropriate dimension. The *k–th* diagonal element of T_1 has been modified to give $\hat{T}_1$ and the first diagonal element of T_2 has been modified to give $\hat{T}_2$.

Now we have two smaller tridiagonal eigenvalue problems to solve. According to equation (11.1) we compute the two eigensystems

$$\hat{T}_1 = Q_1 D_1 Q_1^T \, , \qquad \hat{T}_2 = Q_2 D_2 Q_2^T \, .$$

This gives

$$(11.2) \qquad T = \begin{pmatrix} Q_1 D_1 Q_1^T & 0 \\ 0 & Q_2 D_2 Q_2^T \end{pmatrix} + \beta \begin{pmatrix} e_k \\ e_1 \end{pmatrix} (e_k^T \, , \, e_1^T)$$

$$= \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} \left(\begin{pmatrix} D_1 & 0 \\ 0 & D_2 \end{pmatrix} + \beta \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} (q_1^T \, , \, q_2^T) \right) \begin{pmatrix} Q_1^T & 0 \\ 0 & Q_2^T \end{pmatrix}$$

-- --

where $q_1 = Q_1^T e_k$ and $q_2 = Q_2^T e_1$. The problem at hand now is to compute the eigensystem of the interior matrix in equation (11.2). A numerical method for solving this problem has been provided in [2] and we shall discuss this method in the next section.

It should be fairly obvious how to proceed from here to exploit parallelism. One simply repeats the tearing on each of the two halves recursively until the original problem has been divided into the desired number of subproblems and then the rank one modification routine may be applied from bottom up to glue the results together again.

This *gluing* process is called updating and the general updating problem we are required to solve for each partition is that of computing the eigensystem of a matrix of the form

$$\hat{Q}\hat{D}\hat{Q}^T = D + \rho z z^T \qquad (11.3)$$

where D is a real $n \times n$ diagonal matrix, ρ is a nonzero scalar, and z is a real vector of order n .

A formula for computing an eigen-pair of the matrix on the right hand side of (11.3) is obtained by noting that if λ is a root of the equation

$$1 + \rho z^T (D - \lambda I)^{-1} z = 0 \qquad (11.4)$$

and

$$q = \theta (D - \lambda I)^{-1} z \qquad (11.5)$$

then q, λ is such an eigen-pair, i.e they satisfy the relation

$$(D + \rho z z^T) q = \lambda q.$$

In (11.5) the scalar θ may be chosen so that $\|q\| = 1$ to obtain an orthonormal eigensystem.

If we write equation (11.4) in terms of the components ζ_i of z then λ must be a root of the equation

$$f(\lambda) \equiv 1 + \rho \sum_{j=1}^{n} \frac{\zeta_j^2}{\delta_j - \lambda} = 0. \qquad (11.6)$$

Under certain assumptions this equation has precisely n roots, one in each of the open intervals $(\delta_j , \delta_{j+1})$, $j = 1,2,...n-1$ and one to the right of δ_n if $\rho > 0$ or one to the left of δ_1 if $\rho < 0$. We construct the eigenvectors corresponding to each of these roots using formula (11.5).

Due to this structure, an excellent numerical method may be devised to find the roots of the Equation (11.6) and as a by-product to compute the eigenvectors to full accuracy. It must be stressed, however, that great care must be exercised in the numerical method used to solve the secular equation and to construct the eigenvectors from formula (11.5). Theoretical results and numerical details may be found in dongarra sorensen fully parallel

The static partitioning of this algorithm results in a divide and conquer graph with a node corresponding to each split of the original matrix. There is of course a problem with such a scheme since parallelism is lost as the computation proceeds. Fortunately, in this problem the computation executing in a given node may be parallelized by splitting the root finding problems into a number of parallel parts. The location of each root is bracketed by the old eigenvalues and each of them may be computed independently. It is a simple matter, therefore, to divide the roots into groups which will be computed in parallel. An important numerical aspect of this algorithm which will not be discussed in detail here is the reduction of the number of roots to be computed in a given node through a deflation technique. This deflation is of considerable importance and has great impact both on the efficiency and numerical accuracy of the algorithm. Details are reported in dongarra sorensen fully parallel but we wish to emphasize here that this aspect of the computation is *problem dependent.* The number of roots to be found at any given step can only be known at the time of execution. Therefore, the number of groups that the roots can be reasonably divided into must be done at run time and assigning a parallel process to one of the groups requires dynamic allocation of processes. These aspects are illustrated in the following graph produced by SCHEDULE.

-- --

Figure 11.1
Output from SCHEDULE

Note that the graph produced by SCHEDULE corresponds very closely to the abstract graph a user might define for this algorithm. In Figure 11.1 we see a partially executed program. The square nodes show dynamically spawned processes, while the round nodes show the static partitioning. A particular point of interest is static node number 5 which had the potential for spawning processes just as the other nodes on that level but didn't because a computation done at run time indicated there was not enough work at this node to warrant the spawning of additional processes. This graph demonstrates the utility of such a capability because the maximum number of concurrently executing processes was met at nearly all levels of the computation. Without the potential of dynamically spawned processes the performance of the parallel algorithm would be very poor due to the loss of concurrency associated with the divide and conquer nature of the static partition. Without dynamic spawning the number of concurrent processes decreases by a factor of two at each level of the graph. This is even more disappointing when one realizes that the amount of work usually will be at a maximum in the nodes near the root of the tree. Moreover, with this particular algorithm dynamic spawning is essential because the amount of work at any given node is proportional to the

-- --

number of roots to be found at that node, but this number will be problem dependent and determinable only at the time of execution.

Another interesting example is associated with an algorithm for solving singularly perturbed convection diffusion PDE. An algorithm based upon the ideas presented in Chin Hedstrom has been implemented using SCHEDULE on the Alliant FX/8. This algorithm solves

$$u_t + c(x,t)u_x = \varepsilon \Delta u \ , \ \ 0 \le t \le \infty, \ \ 0 \le x \le 1 \ , \ \ 0 < \varepsilon < 1$$

given appropriate initial data and boundary conditions. The basis of the method is to use asymptotic analysis of the dominate scales of the equation in various subdomains as depicted in Figure 11.2. Such analysis is evident upon making a characteristics coordinate transformation $(x,t) \rightarrow (\xi,\tau)$ with

$$\frac{dx}{dt} = c(x,t) \ , \ \ x(0) = \xi \ , \ \ t \equiv \tau \ .$$

Figure 11.2

Subdomains of Convection Diffusion Equation

We refer the reader to Chin Hedstrom for details. The parallelism is obtained through solving for these characteristics in parallel block fashion with the number

-- --

of characteristics associated with each block tuned to the number of vector registers on a physical processor. For the Alliant computer this number was 32. The vector processor was used as an SIMD engine to compute these characteristics in lock step using the step size control associated with the ODE solver for a system of ODEs. The solver used in this case was the code of Shampine and Allen Shampine and Allen The domain decomposition consisted of two regions where the solution was taken to be constants on these characteristics, an internal layer where the full equation was solved, a boundary layer where two point boundary value problems were solved , and a region where the internal layer meets the boundary layer.

In the following graphics output, Figure 11.3, from SCHEDULE we show this program executing. Node 4 corresponds to gridding the internal layer with characteristics. Note that there is dynamic spawning of blocks of 32 characteristics made there. Nodes 2 and 3 correspond to convection only regions and blocks of 32 characteristics are spawned off there as well. These processes just continue to solve the boundary layer problem as soon as the characteristics meet the boundary at $x = 1$. Process 6 solves the full equation in the internal layer while process 5 grids the region where the internal and boundary layers meet. Both 5 and 6 must be completed before node 7 can execute to solve the equation where the internal and boundary layers meet. Node 1 is a dummy node used to collect all the results.

-- --

Figure 11.3
SCHEDULE Output from Domain Decomposition

12. Conclusions

The construction SCHEDULE has evolved naturally during the course of our experience with programming various parallel machines. We are primarily motivated by the lack of unformity and limited capabilities offered by vendors for explicit parallel programming. We are deeply indebted to our colleagues who are active in similar pursuits and we have used many of their ideas in constructing SCHEDULE. However, although we have been greatly influenced by their work, we feel that our experience with numerical software has guided us in subtly different directions. It is our view that the user interface at this point in time is paramount to the utility of such an effort. Our goal has been to provide a methodology that is within the grasp of a capable Fortran programmer and a to provide syntax which does not represent a radical departure in appearance from standard Fortran programming. We have already modified this interface several times based upon interaction with users. The current approach has evolved from a package written entirely in Fortran to one requiring a *C – Fortran* interface. Our goal of transportability is

-- --

made more difficult with this requirement due to the lack of standardization of such an interface. However, our goal of providing clean user interface with the parallel capabilities seems to require this mixed language approach.

Our experience with SCHEDULE has been encouraging for the most part. We do not view it as a "solution" to the software problem we face in parallel programming. However, we do think this will be useful in the short term and perhaps will have some influence on the development of a long term solution.

Acknowledgements

We would like to thank Terry Disz for introducing us to the graphic environment RT/1 and his assistance in making the graphics interface work.

References