Parallel Computing 5 (1987) 175–186
North-Holland

A portable environment for developing parallel FORTRAN programs *

J.J. DONGARRA and D.C. SORENSEN

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439-4844, and Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, Urbana, IL 61801-2932, U.S.A.

Abstract. The emergence of commercially produced parallel computers has greatly increased the problem of producing transportable mathematical software. Exploiting these new parallel capabilities has led to extensions of existing languages such as FORTRAN and to proposals for the development of entirely new parallel languages. We present an attempt at a short term solution to the transportability problem. The motivation for developing the package has been to extend capabilities beyond loop based parallelism and to provide a convenient machine independent user interface. A package called SCHEDULE is described which provides a standard user interface to several shared memory parallel machines. A user writes standard FORTRAN code and calls SCHEDULE routines which express and enforce the large grain data dependencies of his parallel algorithm. Machine dependencies are internal to SCHEDULE and change from one machine to another but the users code remains essentially the same across all such machines. The semantics and usage of SCHEDULE are described and several examples of parallel algorithms which have been implemented using SCHEDULE are presented.

Keywords. Parallel programming, portable implementation of parallel algorithms, multiprocessor systems, the package SCHEDULE.

1. Introduction

Many new parallel computers are now emerging as commercial products [7]. 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 [11,12] 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 feels appropriate for the user. This has led to widely different extensions ranging from the ability to

* 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.

synchronize on every assignment of a variable with a full empty property [9] to attempts at automatically detecting loop-based parallelism with a preprocessing compiler aided by user directives [1]. The act of getting a parallel process executing on a physical processor ranges from a simple 'create' statement [9] which imposes the overhead of a subroutine call, to 'taskstart' [5] which imposes an overhead on the order of 10^6 machine cycles, to no formal mechanism whatsoever [1]. All of these different approaches reflect characteristics of underlying hardware and operating systems. 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 package, called SCHEDULE, can help 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 desires to transport from one machine to another.

Our work has primarily been influenced by the work of Babb [2], Browne [3], and Lusk and Overbeek [10]. 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 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 processor 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 our environment. Each set of subroutine calls to the environment specifies the subroutine, or process (unit of computation), along with the calling parameters and the data dependences necessary to coordinate the parallel execution.

The basic philosophy here is that FORTRAN programs are naturally broken into subroutines that identify units of computation that are self-contained and that 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 rěsult 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 graph of Fig. 1 denotes five subroutines A,

$$\begin{tikzcd}
 & & A \arrow[dl, dash] \arrow[dr, dash] & \\
 & B \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & & C \\
D & D & E &
\end{tikzcd}$$

Fig. 1.

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 started twice with different data). Once D, D, and E have completed, B may execute. When B and C have completed execution, 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 schedulable 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 $\sigma = a^{\mathrm{T}}b$.

Parallel Algorithm. Let $a^{\mathrm{T}} = (a_1^{\mathrm{T}}, a_2^{\mathrm{T}}, \ldots, a_k^{\mathrm{T}})$ and $b^{\mathrm{T}} = (b_1^{\mathrm{T}}, b_2^{\mathrm{T}}, \ldots, b_k^{\mathrm{T}})$ be a partitioning of the vectors a and b into smaller vectors a_i and b_i.

Compute (in parallel)

$$\sigma_j = a_j^{\mathrm{T}} b_j, \quad j = 1, \ldots, k.$$

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:

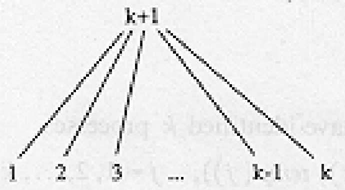

Fig. 2.

```
      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 inprod and addup. The program is shown below and the associated data dependency graph in Fig. 2.

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
```

In this data dependency graph we have identified k processes

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

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 where they are *leaves*. One process,

$$addup(k,\ sigma,\ temp),$$

labeled $k+1$ is data independent 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,nchcks,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, inprod, 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 mechanism 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*.

In Fig. 3 a typical node of a data dependency graph is shown. This node has two incoming arcs and three outgoing arcs. As shown to the right 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 situation in Fig. 4. The array labeled **parmq** holds a process descriptor for each *jobtag*. A process descriptor consists of data dependency information and a subroutine

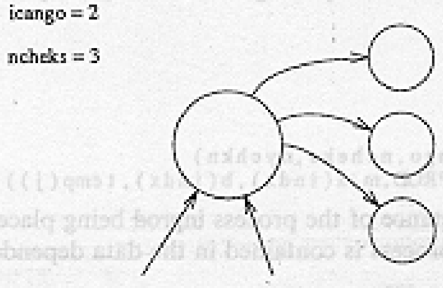

Fig. 3. A node in a dependency graph.

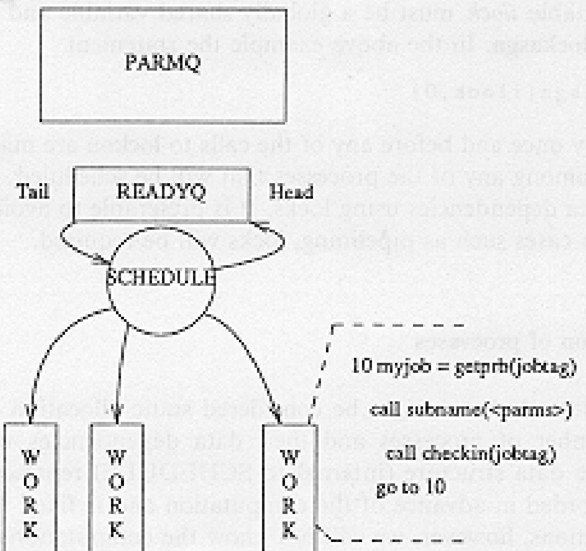

Fig. 4. The SCHEDULE mechanism.

name together with a calling sequence for that subroutine. This information is placed on parmq through the two calls

```
      CALL DEP(jobtag,icango,nchcks,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*.

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 **lockoff**. Each takes an integer argument. An example of usage is

```
      call lockon(ilock)
           ilocal = indx
           indx = indx + 5
      call lockoff(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 by calling the routine **lockasgn**. In the above example the statement

```
      call lockasgn(ilock,0)
```

must execute exactly once and before any of the calls to **lockon** are made. 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(myid,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
      LOGICAL WAIT
      EXTERNAL INPROD
      save m1,m2
c
      go to (1111,2222), IENTRY(myid)
 1111 continue
c
      m1 = n/k
      indx = 1
      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     replace the call to inprod  with a call to spawn
c
      CALL SPAWN(myid,jdummy,INPROD,m2,a(indx),b(indx),temp(k))
c
      nprocs = k
      L2222 = 2
```

```
c
c       If any of the spawned process have not completed, RETURN
c       to the scheduler and help out.  This avoids busy waiting
c       and allows this code to be executed by one processor.
c
           if (WAIT(myid,nprocs,L2222)) return
 2222      continue
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
      save jobtag
c
      JOBTAG = 1
      ICANGO = 0
      NCHEKS = 0
c
      CALL DEP(jobtag,icango,ncheks,mychkn)
      CALL PUTQ(jobtag,ADDUP,jobtag,n,k,a,b,sigma,temp)
c
      return
      end
```

8. 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 taskstart, and the low-level synchronization already matches the low-level synchronization routines provided by the CRAY multitasking library [5]. 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 [8], a domain decomposition code for singularly perturbed convection-diffusion PDE [4], and a block preconditioned conjugate gradient code for systems arising in reservoir simulation [6].

References

[1] Alliant Computer Systems Corp, Alliant FX/Fortran Programmer's Handbook, Acton, MA, 1985.

[2] R.G. Babb, Parallel processing with large grain data flow techniques, *IEEE Comput.* 17 (1984) 55–61.

[3] J.C. Browne, Framework for formulation and analysis of parallel computation structures, *Parallel Comput.* 3 (1986) 1–9.

[4] R. Chin, G. Hedstrom, F. Howes and J. McGraw, Parallel computation of multiple-scale problems, in: A. Wouk, ed., *New Computing Environments: Parallel, Vector, and Systolic* (SIAM, Philadelphia, PA, 1986) 134–151.

[5] CRAY 2 Multitasking Users Guide, Cray Research Inc., Minn, MN, 1986.

[6] J.C. Diaz, Calculating the block preconditioner on parallel multivector processors, *Proc. Workshop on Applied Computing in the Energy Field*, Stillwater, OK (1986).

[7] J.J. Dongarra and I.S. Duff, Advanced architecture computers, Argonne National Laboratory Report, ANL-MCS-TM-57 (1985).

[8] J.J. Dongarra and D.C. Sorensen, A fully parallel algorithm for the symmetric eigenvalue problem, *SIAM SISSC* 8 (2) (1987).

[9] H. Jordan, HEP architecture, programming and performance, in: J. Kowalik, ed., *Parallel MIMD Computation: HEP Supercomputer and Its Applications* (MIT Press, Cambridge, MA, 1985).

[10] E. Lusk and R. Overbeek, Implementation of monitors with macros: A programming aid for the HEP and other parallel processors, Argonne National Laboratory Report, ANL-83-97, 1983.

[11] J.R. McGraw et al., SISAL: Streams and iteration in a single assignment language, Language Reference Manual, Version 1.2, Lawrence Livermore National Laboratory.

[12] J. Van Rosendale and P. Mehrotra, The BLAZE language: A parallel language for scientific programming, ICASE Report #85-29, 1985.