View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

NASA Contractor Report 177961

ICASE REPORT NO. 85-35

NASA-CR-177961
19850026209

ICASE

REORDERING COMPUTATIONS FOR PARALLEL EXECUTION

Loyce Adams

AFOSR 85-1089 and NAS1-17070

July 1985

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

LIBRARY COPY

SEP 25 1985

LANGLEY RESEARCH CENTER
LIBRARY, NASA
HAMPTON, VIRGINIA

NASA
National Aeronautics and Space Administration

Langley Research Center
Hampton Virginia 23665

1 Report No NASA CR-177961
ICASE Report No. 85-35

2 Government Accession No

3 Recipient's Catalog No

4 Title and Subtitle

REORDERING COMPUTATIONS FOR PARALLEL EXECUTION

5 Report Date
July 1985

6 Performing Organization Code

7 Author(s)
Loyce Adams

8 Performing Organization Report No
85-35

10 Work Unit No

9 Performing Organization Name and Address
Institute for Computer Applications in Science
and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, VA 23665

11 Contract or Grant No
AFOSR 85-1089
NAS1-17070

12 Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, D.C. 20546

13 Type of Report and Period Covered
Contractor Report

14 Sponsoring Agency Code
505-31-83-01

15 Supplementary Notes
Langley Technical Monitor:
J. C. South Jr.
Final Report

Submitted to Communications for
Applied Numerical Methods

16 Abstract

In this paper we show how to reorder the computations in the SOR algorithm to maintain the same asymptotic rate of convergence as the rowwise ordering and to obtain parallelism at different levels. A parallel program is written to illustrate these ideas and actual machines for implementation of this program are discussed.

17 Key Words (Suggested by Author(s))
successive overrelaxation, parallel algorithms, parallel programming, multicolor algorithms, vector computers

18 Distribution Statement
61 - Computer Programming & Software
64 - Numerical Analysis

Unclassified - Unlimited

19 Security Classif (of this report)	20 Security Classif (of this page)	21 No of Pages	22 Price
Unclassified	Unclassified	19	A02

For sale by the National Technical Information Service, Springfield, Virginia 22161

REORDERING COMPUTATIONS FOR PARALLEL EXECUTION

Loyce Adams

University of California, Los Angeles

ABSTRACT

In this paper we show how to reorder the computations in the SOR algorithm to maintain the same asymptotic rate of convergence as the rowwise ordering and to obtain parallelism at different levels. A parallel program is written to illustrate these ideas and actual machines for implementation of this program are discussed.

Research was supported in part by the Air Force Office of Scientific Research under Contract No. AFOSR 85-1089 and in part by the National Aeronautics and Space Administration under NASA Contract No. NAS1-17070 while the author was in residence at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665.

N85-34522#

1. Introduction

The effective implementation of a numerical method on a vector computer requires the computations to be ordered to allow vector operations. On computers with multiple processing elements, or with multiple instruction streams, the operations must be ordered in a way that permits them to be divided into groups that can execute in parallel with minimal communication. Since the ordering that results in either case is usually different from an ordering that would be used when implementing the method on sequential computers, we say that the operations have been reordered to achieve parallelism. (See Voigt[1985] for a description of four paradigms for achieving parallelism).

Using reordering of operations as a paradigm for obtaining parallelism leads to two major problems if not properly controlled. Firstly, we are unlikely to develop an ordering that is best for every new parallel computer that is built or porposed. Instead, we would like to find an ordering that permits us to *express* through the semantics of a parallel programming language the levels of parallelism we wish to obtain. Then, if this language can be effectively implemented on the actual architecture, our algorithms will be portable. Secondly, ordering the computations in different ways can change the mathematical properties of the algorithm. For example, convergence rates of iterative methods or roundoff error of gaussian elimination may be different when the computations are done in different orders. Hence, we would like orderings that satisfy some mathematical rule as well as provide parallelism.

In section 2, a procedure is demonstrated for finding an ordering for the SOR (Successive Overrelaxation) method that yields the same asymptotic convergence rate as an ordering that is often used on sequential computers. In section 3, we show how to group the operations and how to store the data within groups to achieve parallelism at two different levels. In section 4, we use PISCES Fortran, see Pratt[1985], to code the algorithm and hence express the parallelism that was described in section 3. This PISCES program is used to show the parallelism that may be obtained in the communication as well as the arithmetic in the algorithm. But just as important, this program points out the operations that must be or are currently being performed in sequence, thereby indicating the overhead costs or areas for improvement in the algorithm. In section 5, we indicate what architectures our

algorithm can be implemented on by simply interpreting what the PISCES semantics would mean on these architectures The hope is to convince the reader by writing only one program that the ordering, the grouping of operations, and the storage strategy within each group that was given in section 3 can be effectively implemented on a variety of architectures.

2. A Parallel Ordering for SOR

Suppose the system of linear equations,

$$A\underline{u}=\underline{b} \tag{1}$$

with A symmetric and positive definite, has arisen from a discretization of an elliptic partial differential equation on a rectangular domain with the 9-point stencil as shown in Figure 1.

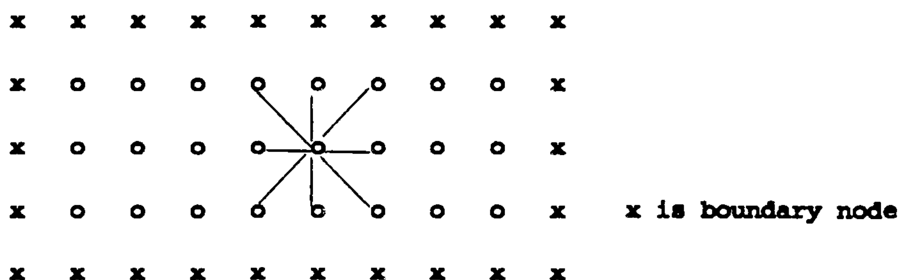

Figure 1. Discretized Problem Domain

For simplicity assume there is one unknown per grid node and that Dirichlet boundary conditions are imposed. We wish to solve system (1) by the SOR method.

The first job is to order the unknowns at the nodes in Figure 1 to indicate which nodes must be updated before others. An "ordering" implies that the nodes must update sequentially. So instead, we would like to color the nodes, see Adams and Ortega[1982], so that nodes of the same color may update simultaneously. Then, the "ordering" is by colors with nodes of a given color ordered arbitrarily. This implies that two nodes can not be the same color if they are on the same 9-point stencil. It is easy to show that for this stencil, only three distinct four-color topologies can be used to color the nodes. These are shown in Figures 2a, 2b, and 2c below (the color pattern repeats beyond the region shown)

```
G O R B      O B O B      G O G O
R B G O      G R G R      R B R B
G O R B      B O B O      G O G O
R B G O      R G R G      R B R B

Figure 2a.   Figure 2b.   Figure 2c.
```

For each figure above, we can order the colors in 4!=24 ways. Hence, there are 72 different orderings we could choose, and the asymptotic convergence rates may be different for some of these orderings.

To aid in choosing an ordering, we require our ordering to have the same asymptotic convergence rate as a baseline ordering; for example, the rowwise ordering of the domain -- bottom to top, left to right. This rowwise ordering is depicted by the stencil rule in Figure 3.

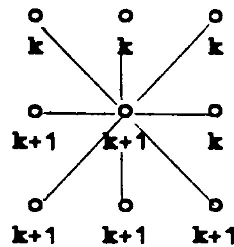

Figure 3. Stencil Rule for Rowwise Ordering

This figure indicates that a node may not be updated on iteration $k+1$ until all nodes on the same stencil to the left and below have been updated on iteration $k+1$ and that values from iteration k must be used for nodes on the same stencil to the right and above.

Adams and Jordan[1984] give a systematic procedure for finding a particular 4-color ordering for this stencil that gives the same asymptotic convergence rate as the rowwise ordering. This strategy was also shown to produce multicolor orderings for many other stencils as well. The idea is to apply the stencil rule of Figure 3 to the grid in Figure 1 but permit nodes to update on subsequent iterations as soon as the appropriate data is available. This strategy leads to a sequence of update times for each node as shown in Figure 4.

R	B	G	O	R	B	G	O
5,9	6,10	7,11	8,12	9	10	11	12
G	O	R	B	G	O	R	B
3,7,11	4,8,12	5,9	6,10	7,11	8,12	9	10
R	B	G	O	R	B	G	O
1,5,9	2,6,10	3,7,11	4,8,12	5,9	6,10	7,11	8,12

Figure 4. R/B/G/O Coloring and Ordering

Figure 4 consists of three distinct sets of nodes. Those in the first set are on the third iteration while those in the second set are on the second iteration while those in the third set are on the first iteration. We simply color all nodes with the same sequence of times the same color, and order the colors by the update times in the first set; that is, R/B/G/O are colors 1/2/3/4 respectively. On parallel computers, we will not process by the update times in Figure 4, but instead, the R nodes will be updated across the entire domain, followed by the B nodes, the G nodes, and finally the O nodes This ordering is one of the 24 orderings depicted in Figure 2a. In the next section, we show that this ordering permits us to group the operations and store the data within group to achieve parallelism at two different levels.

3. Grouping Nodes to Tasks and Task Data Storage

The nodes with the coloring shown in Figure 4 must be grouped into tasks that can run simultaneously. It is desirable to have an equal number of each color of nodes in each task so that all tasks can update nodes of the same color simultaneously. It is also desirable for programming considerations to have the same color pattern in each task. Figure 5 shows the unknowns grouped into 3 rows and 3 columns of tasks. Each task has a unique number (it,jt) as shown to indicate it is in the *it−th* task row and the *jt−th* task column.

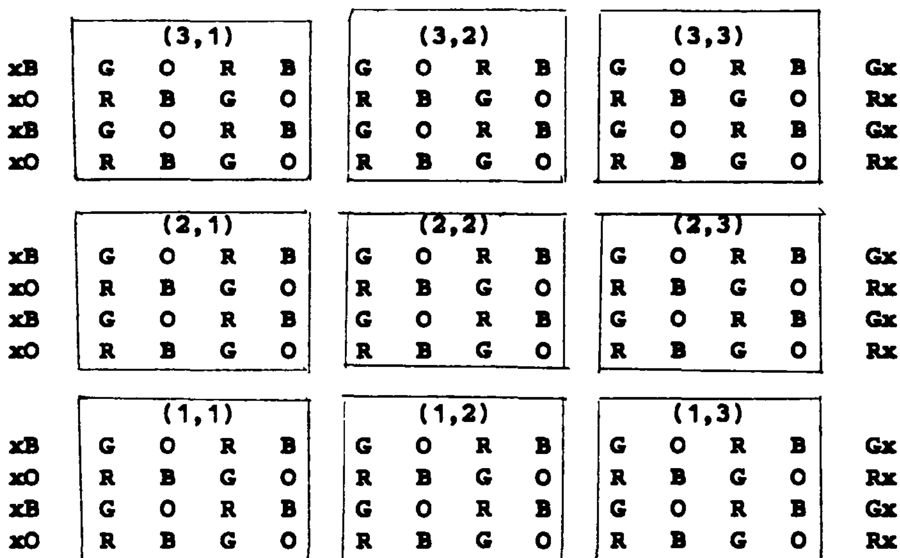

Figure 5. Grouping of Nodes to Update Tasks

Next, we must decide upon a storage scheme for the unknowns, $\underline{u}$, the right hand side $\underline{b}$, and hence the coefficient matrix A, within each update task. We want to insure that a task can run on a vector computer as well as a sequential one because in the near future we are likely to have multiple processor systems for which each processor is a vector computer. Figures 6a, and 6b show two possible storage schemes for the unknowns in the (2,2) task of Figure 5 that permit vector operations within a task.

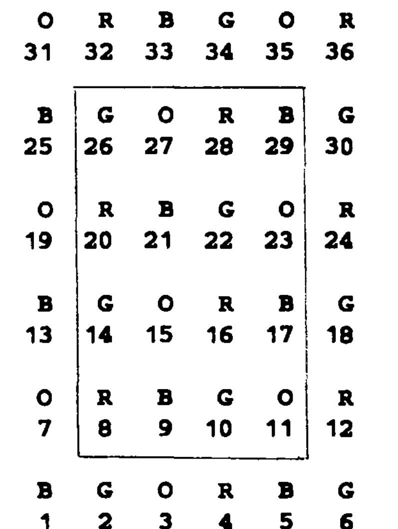

Figure 6a. Constant Stride

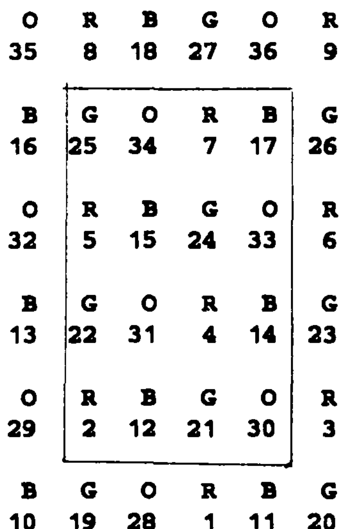

Figure 6b. Contiguous

The nodes within the box are unknowns the (2,2) task is to compute. The nodes around the box are neighbor task unknowns that must be known at particular times in the iteration; hence, they must be stored

In Figure 6a, we number all nodes consecutively from 1 to 36, wrapping around from row to row In general, if we have tr rows and tc columns of nodes (including neighbor border nodes) we will store $\underline{u}$ as a $tr*tc$ linear vector. Notice that by including the neighbor task's nodes in this storage scheme, nodes of the same color are a constant stride apart in memory (the stride is nc the number of colors). This means on a vector computer that allows a vector to be storage locations a constant stride apart, we may vectorize the updating of nodes of a given color within a task This would require the coefficients of A for this task to be stored as a two dimensional array, $A(tr*tc,ns+1)$ where $ns+1$ represents the 9 stencil locations, the last one being for the diagonal element Of course, at the end of updating a given color, we must reset the border nodes back to their original value since they were updated only to permit the computation to be vectorized. A control feature like that of the CYBER 205 can be used to prohibit these nodes from actually being updated. Notice that the nodes of the same color are also a constant stride apart in the vertical direction, namely, a stride of $2*tr$. Therefore, if a control feature is not available, the resetting of values can be done by vector operations, since the nodes that require resetting form a vector Because nodes of the same color are a constant stride apart in both directions, updated values may be packaged as a vector to be sent to appropriate neighbor tasks and in turn the neighbor tasks can store the values as vectors

Figure 6b shows the storage of unknowns for vector computers that require a vector to be a continuous set of storage locations Notice all R's are numbered first, followed by all B's, etc, since unknowns must be processed the same way they are stored The matrix A must be similarly organized Note that in the vertical direction, nodes of the same color are not contiguous Therefore, we would expect to have more overhead in packaging values to be communicated to neighbors

At this point is it worth commenting that the coloring in Figure 2c does not permit vectorization at the task level for either constant stride or contiguous vectors So, if some mathematical rule had selected the coloring of Figure 2c, we might backtrack and settle for other mathematical properties in order to achieve more parallelism There is always this tradeoff.

4. A PISCES Program for the R/B/G/O SOR Algorithm

PISCES (Parallel Implementation of Scientific Computing Environments), see Pratt[1985], is a high-level programming environment for a carefully specified *virtual parallel computer*. It may be thought of as a parallel programming language (in the semantic sense) for the PISCES *virtual computer*. Programs written in PISCES FORTRAN may be run efficiently on various actual parallel computers if there is an efficient match between the virtual and real computers. This is analogous to saying that programs written in FORTRAN may be run efficiently on many sequential computers since the FORTRAN *virtual computer* may be efficiently mapped onto many actual sequential machines. For our purposes, the PISCES virtual computer views parallel computing the following way (For a precise definition, see Pratt[1985]).

(1) Tasks are complete programs There is no shared data between tasks.

(2) Tasks can initiate other tasks. The initiator is the parent. Tasks communicate to other tasks by message passing. Only a task knows its own id number. A task can only communicate to another task if the id number of the other task is known; hence, a task must give its id number to other tasks if messages are to be received from these tasks

(3) Tasks accept messages by the *message-type,* not by any properties of the message sender A task may only accept a message of a given type if the task has a *handler* (a special subroutine) capable of processing messages of this type.

(4) Tasks and their handlers can run on different processors, but they must communicate through shared memory via the FORTRAN Common block

(5) Finer grains of parallelism can be obtained within a task Statement level parallelism is obtained via the *parbegin-parend* construct; parallelism across instances of a *do* loop is obtained with the *pardo-continue* construct, and parallelism at the operation level is obtained via vector operations.

We now describe the PISCES program of Figure 7 for the SOR algorithm that uses the ordering of Figure 4, the grouping of Figure 5, and the storage scheme of Figure 6a. The program has two types of tasks. a *parent* task

and an *update* task for each group of nodes in Figure 5.

The first job of the parent task is to initiate all the *ntr*ntc* update tasks, each with its own data, its logical identification number (i,j) and its starting and ending rows (*is* and *ie*), and its starting and ending columns (*js* and *je*) of nodes. These rows and columns include the border nodes from neighbor update tasks as shown in Figure 6. These initiations may be done in parallel if enough processors are available on the actual hardware; otherwise the *pardo*1 and *pardo*2 loops can be changed or interpreted as ordinary sequential *do* loops

Since the true physical identification numbers of the update tasks are not known until these tasks begin execution, and since these tasks will need to communicate with each other, they must exchange ids. An easy way to accomplish this is to send their ids to the parent task and in turn the parent task collects the ids and then broadcasts the result back to all the update tasks. Now, the parent *accepts* the ids as *ntr*ntc* messages of type *identity* The handler *identity* reads the data that is sent into the parameter list like a read statement. Thus, the ids sent are stored by this handler in the array *tidset* (of type *taskid*) which is in memory common with the parent task The parent waits until all ids have been stored in the array *tidset* and then *sends* a message of type *tid* to all the update tasks for their future use

The parent task then enters the SOR iteration loop and waits for *ntr*ntc* total local convergence messages, either type *converge* or type *noconv*, which are sent each iteration by the update tasks. to be accepted by the handler *noconv* or to be counted as a *signal* if the type is *converge* (*Signals* are special messages that require no processing by a handler.) These two types of messages can be accepted simultaneously since they are within the same *accept* statement, that is, the *noconv* handler could be running on a different processor than the task that is accepting *signals* If the variable *global* is true at the end of an iteration, the parent task exits the iteration loop and waits for the answers that are sent by each update task to be accepted by the *answers* handler The answers which are now stored in the *u* array are written to a file and the parent task terminates

We note that all instances of a given handler can execute in parallel For example, all $ntr \times ntc$ *identity* messages can be received at once, provided enough processors are available There is always the cost consideration of whether obtaining all the parallelism inherent in the algorithm actually saves time The point here is that this

parallelism is *expressable* and parameter studies could be done to determine for what machines it is worthwhile and the implementation of PISCES adjusted accordingly.

Next, we turn to the description of an update task. This task first sends its physical id (pppslf) back to the parent task in a message of type *identity* and then waits until handler *tid* has accepted from the parent and stored the task ids of all other update tasks. An update task only needs to communicate to *nhbs* other update tasks whose logical ids are stored in *neigh*(*nhbs*,2). Notice that the task ids of all the update tasks are stored in *taskid* even though not all will be used. This is because it is probably less time consuming for the parent task to send all the ids rather than to package and send individually to each task just what is needed.

The update task now begins the SOR iteration loop 100, and the loop 200 over the number of colors, *nc* Each time through the 200 loop, the task processes all nodes of the same color assuming the storage arrangement in Figure 6a. The notation *temp*(*st*;*sp*;*nc*)=0 means that locations of the linear array *temp* beginning in location *st*, ending in location *sp* with stride *nc* are set to zero as a vector operation. Likewise, *A*(*st*;*sp*;*nc*,*nst*) means all rows starting with row *st* ending with row *sp*, incrementing by stride *nc*, with column *nst* fixed are processed as a vector Note that the arithmetic in the algorithm within a task vectorizes in statements 201-203, and a vector dot product is done in statement 204

After the values of a given color have been updated, the border values of this color are sent to appropriate neighbors This is done for each neighbor in parallel with the *pardo* 300 statement. The update task retrieves the package to be sent as a vector with starting location *st*(*ne*), stopping location *sp*(*ne*), and stride *inc*(*ne*) and the neighbor's handler stores the package as a vector with starting location *nest*(*ne*), stopping location *nsp*(*ne*), and stride *ninc*(*ne*) Of particular note is the type of message sent; namely, *ncolors.uvals*. There are *nc* different u value handlers, one to deal with messages of each color from all the update tasks. Only instances of handlers for a given color can work in parallel since a task must accept all R's before the B's can be accepted for the next iteration The reason for having *nc* different handlers to accept the u values instead of only one is not for parallelism, but rather to ensure *correctness* and *simplicity* of the program. That is, if all messages were of type *uvals* and a neighbor update task operates faster and sends a message with B values, this message could be counted in the

total. In this case, all R values would not be accepted before the update task begins to compute B values. That is, we can not be guaranteed that messages arrive in the same order they are sent, but by accepting only messages of the type *ncolors uvals* we are guaranteed that all our *total* messages are R ones on the first pass through the 200 loop. Of course, we could add a data field to indicate color to messages of type *uvals* This would require more processing to examine the messages and more storage to save messages of the *wrong* color until they are needed, thereby adding to the complexity of the program.

After all colors have been processed, the local error is determined as a dot product $er^T er$ in statement 204 The task then sends a message of type *converge* or *noconv* to the parent and awaits the decision. The parent sends this decision as a message of type *convflag* which handler *convflag* records in the variable *global*. If converged, the update task sends its *tr*tc* $\underline{u}$ values back to the parent with a message of type *answers*; otherwise, the next iteration begins

These programs show the levels of parallelism that are achievable with the R/B/G/O ordering, the grouping of unknowns, and the data storage scheme within a task that were given in Figures 4, 5, and 6a respectively. This parallelism can be broken into parallel arithmetic and parallel communication as summarized below

Parallel Arithmetic

(1) Nodes in different tasks update simultaneously

(2) Nodes of the same color within a task can execute simultaneously, probably as vector operations

Parallel Communication

(1) All update tasks can be initiated simultaneously

(2) The parent accepts the task ids in parallel from the update tasks. The update tasks can then accept the broadcasted result simultaneously

(3) The parent accepts convergence information from the update tasks in parallel. The broadcasted convergence decision may be accepted in parallel by the tasks.

(4) Updated values can be sent to neighbors simultaneously.

(5) Values of the same color may be accepted from all neighbors simultaneously.

(6) Answers send back in parallel from the update tasks can be accepted in parallel by the parent.

However, the algorithm points out where sequential operation seems to prevail.

Sequential Arithmetic

(1) The dot product is partially sequential and can be more costly than vector multiplication, for example, on vector machines. The dot products are done in parallel across tasks.

(2) Starting and stopping indices must be computed within each task, but are done in parallel across tasks.

5. Possible Architectures for Implementation

The PISCES program that was described in the last section shows that the R/B/G/O SOR algorithm can be run on different parallel computer architectures. We conclude by mentioning four types of architectures and the proper interpretation of the tasks and handlers for these architectures.

(1) If only one task is defined and the vector operations are converted to *do* loops, no handlers are needed and the result is the sequential machine algorithm.

(2) If only one task is defined and the vector operations remain, no handlers are needed and the result is a constant stride vector program that could run on the CRAY, for example. (Syntax would need repairing, but the semantics are the same)

(3) Assume a task and its handlers run on the same processor and that we have many tasks and many processors. Also assume that each processor is a scalar processor and that the processors do not have shared memory, but are connected by some interconnection network. Then, if the interconnection network is that of the hypercube, for example, our algorithm will run on Caltech's Hypercube. If instead, the interconnection network is that of a flat 8 nearest neighbor connection, the algorithm will run on NASA Langley's Finite Element Machine.

(4) If we relax the requirement that tasks can not have shared memory (deviating from PISCES), and let several tasks run on the same processor in multiple instruction stream fashion (pipelined MIMD), we see that our algorithms can be run on the Denelcor HEP with one or multiple PEMs The *accept* statement would be analogous to synchronization through a shared variable; since, for example, a task would only need to wait until R values were updated by all tasks (HEP proceses) before the updating of B values is begun

(5) To actually realize all the levels of parallelism in our algorithm, we will have to wait until novel architectures comprised of many processors, are available Some of the processors would be vector processors for the *update* tasks while others would be scalar processors for the handlers and the parent task Since the processor for an update task and the processors for the the update task's handlers require shared data they could be connected to a shared memory. A communication network might be used to connect different update processor/handler processor clusters

```
       tasktype parent
       parameter (ntr= ,ntc= ,n= ,maxiter= ,<other parameters>)
       common /withidentity/ tidset
       common /withnoconv/ global
       common /withanswers/ u
       real u(n,n),<other real declarations>
       integer <integer declarations>
       taskid tidset(ntr,ntc)
       handler noconv,answers,identity
       signal converge
       end declarations

*initiate the ntr*ntc update tasks
       pardo 1 i=1,ntr
             pardo 2 j=1,ntc
                   is(i,j)=  , ie(i,j)=
                   js(i,j)=  , je(i,j)=
                   on same initiate update(i,j,is(i,j),ie(i,j),js(i,j),je(i,j),<other data>)
2            continue
1      continue

*accept the task ids of the tasks and send results back to all update tasks
       accept (ntr*ntc)
         identity
       end
       to all send tid(((tidset(i,j),i=1,ntr),j=1,ntc))

*begin the iteration
       do 100 iter=1,maxiter
          global=.true
          accept (ntr*ntc) of
             converge
             noconv
          end accept
          to all send convflag(global)
          if (global) then go to 3
100    continue

*accept the answers from all the tasks
3      accept (ntr*ntc)
          answers
       end accept

*write the answers to file 6
       pardo 4 i=1,n
          write(6,*)(u(i,j),j=1,n)
4      continue

       stop
       end
```

Figure 7. PISCES Program (continued on next 3 pages)

```
handler noconv
common /withnoconv/ global
logical global
end declarations
global=.false
return
end

handler identity (it,jt,tidset(it,jt))
parameter (ntr= ,ntc= )
common /withidentity/ tidset
integer it,jt
taskid tidset(ntr,ntc)
end declarations
return
end

handler answers (is,ie,js,je,((u(i,j),i=is,ie),j=js,je))
parameter (n= )
common /withanswers/ u
integer is,ie,js,je
real u(n,n)
end declarations
return
end
```

```
tasktype update(it,jt,is,ie,js,je,<task's other data>)
parameter (ntr= ,ntc= ,tr= ,tc= ,nc= ,w= ,ns= ,nbhs= ,maxiter= ,<task's other parameters>)
common /withtid/ tidset
common /withconvflag/ global
common /withuvals/ u
integer it,jt,is,js <other integer declarations>
real    u(tr*tc), b(tr*tc), A(tr*tc,ns+1),stencil(ns),neigh(nhbs,2),<others>
        logical  global
        taskid   tidset(ntr,ntc)
        signal   done,converge
        handler  tid,convflag,(k.uvals,k=1,nc)
        end declarations

* retrieve task ids of other update tasks
        to parent send identity(it,jt,pppslf)
        accept
          tid
        end accept

*begin the iteration
        do 100 iter=1,maxiter
           do 200 ncolors=1,nc
              st= , sp=
201           temp(st;sp;nc)=0
              do 202 nst=1,ns
                 temp(st;sp;nc)=temp(st;sp;nc)+A(st;sp;nc,nst)*u(st+stencil(nst);sp+stencil(nst);nc
202           continue
              <reset temp to be u at the boundary nodes>
              temp(st;sp;nc)=(b(st;sp;nc)-temp(st;sp;nc))/A(st;sp;nc,ns+1)
              er(st;sp;nc)=temp(st;sp;nc)-u(st;sp;nc)
203           u(st;sp;nc)=w*temp(st;sp;nc)+(1-w)*temp(st;sp;nc)

*send appropriate u values of this color to all neighbors
              pardo 300 ne=1,nbhs
                    num(ne)=
                    if(num(ne).ne.0)then
                      st(ne)= ,sp(ne)= ,inc(ne)=
                      nest(ne)= ,nsp(ne), ninc(ne)=
                      to tidset(neigh(ne,1)+it,neigh(ne,2)+jt) send ncolors.uvals
                       (nest(ne),nsp(ne),ninc(ne),(u(i),i=st(ne),sp(ne),inc(ne)))
                    endif
300           continue

*accept (total) messages of type ncolors.uvals from any neighbor
              total=
              accept (total)
                ncolors.uvals
              end accept

200        continue
```

```
*check for local and then global convergence
        e=dot(er,er)
        if (e.lt.eps) then
                to parent send converge
         else
                to parent send noconv
         endif

 *accept convergence decision from parent
         accept
            convflag
         end accept

*if converged then exit iteration loop
         if (global) then go to 5
100   continue

*send answers back to parent
5     to parent send answers (is,ie,js,je,(u(i),i=1,tr*tc))
      stop
      end

      handler tid (((tidset(i,j),i=1,ntr),j=1,ntc))
      parameter (ntr= ,ntr= )
      common /withtid/ tidset
      taskid tidset(ntr,ntc)
      end declarations
      return
      end

      handler convflag(global)
      common /withconvflag/ global
      logical global
      end declarations
      return
      end

      handler k.uvals (st,sp,inc,(u(i),i=st,sp,inc))
      parameter (tr= ,tc=, nc= )
      common /withuvals/ u
      integer st,sp,inc
      real u(tr,tc)
      return
      end
```

REFERENCES

Adams, L.M., Ortega, J.M. [1982]. "A Multi-Color SOR Method for Parallel Computation," *Proc of the 1982 Intl. Conference on Parallel Processing*, IEEE Catalog No. 82CH1794-7, August, pp. 53-56.

Adams, Loyce M., Jordan, Harry F. [1984]. "Is SOR Color-blind?", ICASE Report No. 84-14. Accepted, *Siam Journal on Scientific and Statistical Computing*.

Pratt, Terrence W. [1985] "PISCES: An Environment for Parallel Scientific Computation," ICASE Report No. 85-12, accepted *IEEE Software*.

Voigt, Robert G. [1985]. "Where Are the Parallel Algorithms?", ICASE Report No. 85-2, accepted *NCC'85 Conference Proceedings*.

End of Document