162
1/12/86
RB

(1)

(39)

I-24543

ANL-85-70

ANL-85-70

DE-1489.3

QUASI-AUTOMATIC PARALLELIZATION: A SIMPLIFIED APPROACH TO MULTIPROCESSING

by

B. W. Glickfeld and R. A. Overbeek

MASTER

ARGONNE NATIONAL LABORATORY, ARGONNE, ILLINOIS

Operated by THE UNIVERSITY OF CHICAGO
for the U. S. DEPARTMENT OF ENERGY
under Contract W-31-109-Eng-38

Distribution Category:
Mathematics and Computers
(UC-32)

ANL-85-70

ANL--85-70

DE86 005120

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

Quasi-Automatic Parallelization: A Simplified Approach to Multiprocessing

B. W. Glickfeld
Northern Illinois University

and

R. A. Overbeek
Argonne National Laboratory

October 1985

Table of Contents

Quasi-Automatic Parallelization: A Simplified Approach to Multiprocessing

B. W. Glickfeld

Northern Illinois University
DeKalb, Illinois 60115

R. A. Overbeek

Argonne National Laboratory
Argonne, Illinois 60439

ABSTRACT

As multiprocessors become commercially available, a great deal of concern is being focused on the problems involved in writing and debugging software for such machines. Earlier work described the use of monitors implemented by macro processors to attain portable code. This work formulates a general-purpose monitor which simplifies the programming of a wide class of numeric algorithms. We believe that the approach of describing a set of schedulable units of computation advocated by Brown offers a real simplification for the applications programmer. In this paper, we propose a straightforward programming paradigm for describing schedulable units of computation that allows the description of many algorithms with very little effort.

1. Introduction

Early work on implementing portable code for multiprocessors [4, 5, 3] resulted in the definition of a general-purpose monitor that dispatched units of work to processes. It was found that this monitor, the ***askfor*** monitor, could be effectively used to write portable implementations of numeric algorithms for multiprocessors having a globally shared memory. However, substantial effort was needed to implement the logic for managing the pool of available units of work.

As an alternative, the self-scheduling DO-loop [2] was defined, which offers a highly desirable simplicity lacking in the ***askfor*** monitor. Until recently, however, the use of this self-scheduling DO-loop has been limited to only special classes of numeric algorithms.

This paper describes efforts to extend the outlook utilized in the self-scheduling DO-loop to a much wider class of synchronization patterns. By analyzing the synchronization required by a number of algorithms, we developed an abstraction of the basic patterns relating schedulable units of computation. We discovered that this abstraction could be visualized naturally, using geometric diagrams, and that the actual code for dispatching units of computation could be reduced (for a wide category of algorithms) to simply specifying the parameters that characterized the geometric pattern required by any particular application.

This paper presents a basic structure for a substantial class of parallel algorithms, a macro package based on that structure, and a series of geometric synchronization patterns that depict the parallelization structure. The macro package, which consists of one central macro *gs2* (so called because a unit of computation is characterized by two subscripts that specify its location in a 2-dimensional depiction of the synchronization dependencies) and some supporting macros, enables the quasi-automatic parallelization of many algorithms.

By quasi-automatic parallelization of an algorithm we mean a cooperative venture between user and macro package where the work of parallelization is done by the macro monitors in the package, with minimal input from the programmer. The user must perform the simple mechanical tasks setting up the interaction between user application code and the macro monitors. In particular, he must supply certain values to the macros, i.e., see that certain values are properly defined before the relevant macro call. Underscoring the simplicity of these efforts is the fact that the user need not do any macroprocessor language coding (as contrasted, say, with the *askfor* monitor [5]). In addition, the common synchronization errors associated with programming multiprocessors can be dramatically reduced.

The methods used here differ from previous methods of quasi-automatic parallelization. For example, they have a far wider range of applicability than the self-scheduling DO-loop, although they fall short of the broad range of the *askfor* monitor. Furthermore, they are simpler to use than the *askfor* monitor; like the self-scheduling DO-loop, our methods do not require the user to create problem-dependent data structures and problem-dependent macro definitions.

To further clarify the relationship between the self-scheduling DO-loop and the *gs2* monitor, the user should consider algorithms in which the relationship between schedulable units of work can be visualized in a two-dimensional Cartesian plane. In the case of the self-scheduling DO-loop, the units of work simply form a horizontal line. No interdependencies exist between the units of work; they can all be done in parallel. The *gs2* monitor allows more complex interdependencies between the units of work. Essentially, it allows the units of work to be represented by either one or two regions in the Cartesian plane. In the case in which a single region suffices, the units of work (characterized by unique coordinates) may still have interdependencies. For example, it is possible to implement the case in which all of the units in one row must complete before those in the next row begin. Alternatively, one might weaken the constraint to the case in which a unit of work cannot be scheduled until the unit directly above it (i.e., in the previous row) has completed. Or, one might wish to introduce a dependency upon units of work in the previous row, but have the dependency include a skew factor. For example, the unit of work may become schedulable when the unit in the previous row, but one column to the right, has completed. In the case in which the units of work can be viewed as a single region in the plane, the region will be called a *simplex*.

While many algorithms can easily be described within the intellectual framework of a simplex, some algorithms conceptually require two disjoint plane regions. We then refer to the two regions as a *2-complex*. In this case, the units within each region may have all of the dependencies introduced in the case of a simplex. In addition, there may exist scheduling constraints between the boundary elements in each region.

Examples presented in the following sections will illustrate the power and range of the structure. In particular, a discrete grid version of the Dirichlet problem for a cube will display the force of the simplex, while the Householder algorithm used in the QR factorization of a matrix will display the force of the 2-complex.

The examples will include all those presented by Overbeek and Lusk [5], along with the above-mentioned grid problem (used in [1] to illustrate the range of the *askfor* monitor,) a variant of the getdups problem presented in [3] and a parallel version of matrix multiplication. Both general and detailed analyses of all examples will be presented.

The theory and methods presented here provide portable, reliable, and efficient code which may be implemented on most multiprocessors featuring a number of processes acting on a globally shared memory. Examples of such multiprocessors include the CRAY X-MP and the Denelcor HEP. Further research is needed to deal with the applicability of gsn macros for n > 2, as well as other 2-subscript monitor structures besides the simplex and 2-complex presented here.

2. The Basic Idea and Its Implementation

In this section we will offer an informal description of the basic ideas employed in the *gs2* monitor. A formal description will be given in a later section. We will follow our informal discussion with a description of the macro package that implements these concepts. We will illustrate its use on a straightforward example – a program to multiply two matrices. Then we will analyze a number of examples that illustrate the basic notions. We believe that these examples offer evidence of both the power and the simplicity of the approach advocated in this paper.

2.1. Summary of the Parallelization Structure

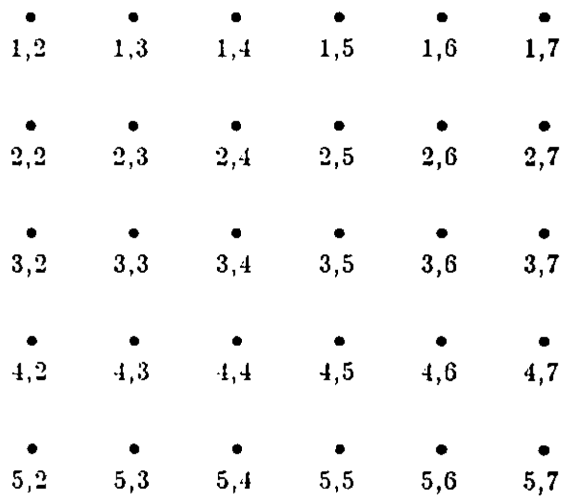
By a parallelization simplex we mean an ordered triple (S,T,U), where S is a regular region in the Cartesian integer plane, T is a function whose domain is S and whose range is the set of computational tasks (i.e., for each (i,j) in S, T(i,j) is a computational task), and U is a finite (perhaps empty) set of synchronization constraints.

The Cartesian integer plane Z_2 is the set of all ordered pairs of integers. By a region we mean a subset of Z_2. We call it regular if it is both

i) the set of all ordered pairs of integers lying between and including an upper boundary and a lower boundary (either of which might be jagged), and

ii) the set of all ordered pairs of integers lying between and including a left boundary and a right boundary.

For example, the rectangle below depicts a regular region.



To picture ordered pairs, we choose conventions different from the usual ones. The first coordinate is represented on the vertical axis, with larger values corresponding to lower points than do smaller values. The second coordinate is represented on the horizontal axis, with larger values corresponding to points to the right of those representing smaller values. These conventions are made because of the way a matrix is usually depicted. A matrix is indexed by ordered pairs of integers. The first index (the row index) is usually considered as a vertical index; matrix elements with larger row indices appear below matrix elements with smaller row indices. Similarly, the second index (the column index) is usually considered as a horizontal index; matrix elements with larger column indices appear to the right of matrix elements with smaller column indices. Thus, our convention enables us to depict, in the Cartesian integer plane, the set of index pairs of a matrix so that they appear as they normally do.

There are two possible synchronization constraints on a simplex:

i) the column constraint C, which requires that no task in a column begin until all earlier tasks (i.e., those from a row with a smaller subscript) in that column have completed, and

ii) the row constraint R, which requires that no task in a given row begin until all tasks from the preceding row (i.e., the row whose subscript is one less than that of the row under consideration) have completed.

C may include a skew factor k -- a non-negative integer such that for all j, no task in the jth column can begin until all earlier tasks in the $(j+k)$th column, as well as all earlier tasks in the jth column, have ended.

An example of a simplex occurs when the underlying region S is an integer rectangle; such a simplex is simply a computational task matrix.

The simplex structure implies that tasks begin in lexicographic order; that is, within a given row, tasks begin in order of column subscripts, and tasks in rows with larger row subscripts begin after those in rows with smaller subscripts. Although some efficiency is lost because of this restriction, the loss is not significant for the algorithms examined in this paper. It is counterbalanced by the power and simplicity of the structure, the ease of use of the macro package, and the user's ability to avoid programming macros for a macroprocessor.

During the execution of a program using the *gs2* macro and a simplex structure, at any given time either all tasks in the simplex have begun or some particular point (i_0,j_0) in the underlying region S represents the current task (i.e., the task next to begin). Implying a lexicographic order may at times result in such phenomena as the following:

a) the current task is not yet ready to begin (e.g., because of the failure of the C constraint to be satisfied) while some other task is ready to begin, or

b) although the current task is ready to begin, other tasks also are ready to begin, whose beginning would in some sense be more optimal.

The disadvantages attendant to the use of the implied order are, as we have stated above, compensated for in all the examples in this paper, as well as in a substantial class outside this paper. We use the analogy from mathematics of approximating general functions by smooth functions; what we lose in exactness we often more than recover in ease of manipulation. Here we are approximating a general sequence of points in the underlying region by one in which there is a smooth progression of subscript pairs, with analogous loss of exactness and gains in manipulability.

Although the simplex structure is adequate for most of our problems, certain of them require more regions. Thus we introduce the notion of parallelization complex or, more briefly, complex. There are two kinds of complexes: 1-complexes and 2-complexes. A 1-complex is a simplex. A 2-complex is an ordered triple $((S^1,T^1,U^1),(S^2,T^2,U^2),\mathrm{V})$ where the first two elements of the triple are the simplexes (S^1,T^1,U^1) and (S^2,T^2,U^2), respectively, and the third element is a non-empty finite set V of cross-constraints, each of which requires that certain tasks in one of the underlying simplexes can begin only after certain other tasks in the other underlying simplex have ended. It is also required that the underlying regions S^1 and S^2 be disjoint, i.e., have no points in common.

Given a 2-complex, we will call (S^1,T^1,U^1) simplex 1, and (S^2,T^2,U^2) simplex 2. Permissible cross-constraints include XC12 (the cross-column constraint from simplex 1 to simplex 2), XC21 (the cross-column constraint from simplex 2 to simplex 1), XR12 (the cross-row constraint from simplex 1 to simplex 2) and XR21 (the cross-row constraint from simplex 2 to simplex 1).

XC12 XC12 means that whenever j is an integer such that S^1 and S^2 both have a jth column (i.e., they both contain points whose second coordinate is j), then the computational task $T^1(i,j)$ with the largest possible row subscript i must end before the computational task $T^2(k,j)$ with the smallest possible row subscript k can begin.

XC21 XC21 is analogously defined, with the roles of simplex 1 and simplex 2 reversed; for the largest possible k, $T^2(k,j)$ must end before $T^1(i,j)$, for the smallest possible i, can begin.

XR12 XR12 means that whenever i is an integer such that S^1 and S^2 both have an ith row (i.e., they both contain points whose first coordinate is i), then all the T^1 tasks in that row must end before the computational task $T^2(i,j)$ with the smallest possible j can begin.

XR21 XR21 is analogously defined, with the roles of simplex 1 and simplex 2 reversed; here all the T^2 tasks in the ith row must end before $T^1(i,j)$ with the smallest possible j can begin.

In our formulation, XC12 (XC21) can apply only when S^1 borders S^2 from above (S^2 borders S^1 from above). Similarly, XR12 (XR21) can apply only when S^1 borders S^2 from the left (S^2 borders S^1 from the left). All these concepts will be presented in detail further on.

Thus, in a 2-complex there are two disjoint simplexes, simplex 1 and simplex 2, with disjoint underlying regions S^1 and S^2, respectively, such that at least one of the simplexes in some sense borders and constrains the other (it is possible for both simplexes to border and constrain each other, as will be demonstrated in a later example).

As mentioned earlier, a 1-complex is merely a simplex, which will be referred to as simplex 1. The dimension of a 1-complex is defined to be 1, and the dimension of a 2-complex is defined to be 2. During the execution of a program using the *gs2* macro and a 2-complex structure, at any given time either

i) all tasks in the complex have begun, or

ii) there is either a point (i_1,j_1) in S^1 that represents the current task in simplex 1, or a point (i_2,j_2) in S^2 that represents the current task in simplex 2, or both.

If ii) holds, the next task to begin will be either $T^1(i_1,j_1)$ or $T^2(i_2,j_2)$; by convention, priority is tilted in favor of simplex 1. Within the complex structure, each simplex has its underlying region traversed lexicographically during program execution, subject to internal constraints from among C1, R1, C2, and R2, external cross-constraints from among XC12, XC21, XR12, and XR21, and the above priority convention. (Note that C1 refers to the column constraint C for simplex 1, and R1 to the row constraint R for simplex 1. C2 and R2 refer to the analogous constraints for simplex 2.) It can easily be shown that the resulting parallelization is free from deadlock.

2.2. Description of the Implementation via Macros

The macro package will now be discussed, along with an elaboration of the program structure we employ to use the macros effectively.

We note that the dimension of the complex under consideration remains fixed throughout the program. However, a given program may execute many (i.e., more than 1) complexes. Each complex, of course, represents a set of computational tasks structured in a certain manner, along with certain constraints. When we say that the program executes a complex, we mean that during its execution it will perform all, or a number, of those tasks in accordance with the structure and the constraints.

Two complexes are equal if their dimensions are equal and their underlying simplexes are respectively equal. Two simplexes are equal if the underlying regions are equal as sets, the underlying task functions are identical, and the imposed constraints are identical.

When a program executes more than one complex, the complexes may differ either because of differences in the underlying regions of corresponding simplexes, or because of differences in the underlying task functions in corresponding simplexes. However, with the program structure used in this paper, the underlying constraints in corresponding simplexes must be identical, except that they may have differing corresponding skew factors.

The formal notion of executing a complex corresponds to the intuitive notion of executing a problem. Thus the formal statement that a given program may execute many complexes is equivalent to the intuitive one that one execution of a program may solve many problems.

To illustrate the actual placement and coding of the macro statements, we include a straightforward program that computes the product of two matrices. We will reference the actual lines of code throughout the remainder of this section.

```
1     define(RB11,1)
2     define(RB12,21)
3     define(CB11,1)
4     define(CB12,21)
5     define(RB21,1)
6     define(RB22,2)
7     define(CB21,1)
8     define(CB22,1)
9
10    ************************************************************************
11    *
12    * THIS PROGRAM READS IN TWO MATRICES AND COMPUTES THEIR PRODUCT.
13    *
14    ************************************************************************
15
16          PROGRAM MATMUL
17          newproc(SLAVE)
18    *
19    * COMMON AREA VARIABLES
20    *
21          INTEGER A(20,20), B(20,20), C(20,20)
22          INTEGER NPROCS, AI, AJ, BJ
23          COMMON /MAINC/ A, B, C, NPROCS, AI, AJ, BJ
24    *
25          gs2var
26    *
27    *
28    ************************************************************************
```

```
*
*   INITIALIZE THE MONITOR
*
************************************************************************
*
      gs2init1(1,00,00,00,00)
*
      READ (5,10) NPROCS
10    FORMAT(I4)
      WRITE(6,20) NPROCS
20    FORMAT(' NPROCS = ',I4)
*
************************************************************************
*
*   READ IN THE TWO INPUT MATRICES
*
************************************************************************
*
      READ (5,10) AI
      READ (5,10) AJ
      READ (5,10) BJ

      DO 2 I = 1,AI
          DO 1 J = 1,AJ
              READ (5,10) A(I,J)
1         CONTINUE
2     CONTINUE

      DO 4 I = 1,AJ
          DO 3 J = 1,BJ
              READ (5,10) B(I,J)
3         CONTINUE
4     CONTINUE

*
      MNRV1 = 1
      MXRV1 = AI
      MNCV1 = 1
      MXCV1 = BJ

      DO 5 I = 1,AI
```

```
          LFBDY1(I) = 1
5     CONTINUE

      DO 6 I = 1,AI
          RTBDY1(I) = BJ
6     CONTINUE

      DO 7 I = 1,BJ
          UPBDY1(I) = 1
7     CONTINUE

      DO 8 I = 1,BJ
          LWBDY1(I) = AI
8     CONTINUE

      gs2init2(1,00,00,00,00)

      DO 30 I=1,NPROCS-1
          create(SLAVE)
30    CONTINUE
*
      CALL WORK
*
      WRITE (6,40)
40    FORMAT(' THE VALUES IN C ARE AS FOLLOWS:')
      DO 41 I = 1,AI
          DO 42 J = 1,BJ
              WRITE(6,43) I,J,C(I,J)
42        CONTINUE
41    CONTINUE
43    FORMAT('     ',I2,' ',I2,' ',I8)
      STOP
      END
*
*******************************************************************
*
*   THE SLAVE PROCESSES JUST CALL THE WORK SUBROUTINE
*   WHERE THEY CLAIM TASKS TO WORK ON.
*
*******************************************************************
*
```

```
      SUBROUTINE SLAVE
*
      CALL WORK
      RETURN
      END
*
*********************************************************************
*
* THE WORK SUBROUTINE CONTAINS THE CODE TO CLAIM A TASK,
* PERFORM THE TASK, AND GO BACK TO GET ANOTHER TASK TO WORK ON.
*
*********************************************************************
*
      SUBROUTINE WORK
*
* COMMON AREA VARIABLES
*
      INTEGER A(20,20), B(20,20), C(20,20)
      INTEGER NPROCS, AI, AJ, BJ
      COMMON /MAINC/ A, B, C, NPROCS, AI, AJ, BJ
*
      gs2var
*
      INTEGER K
*
10    CONTINUE

      gs2(1,00,00,00,00)
1000  CONTINUE

*
      C(I,J) = 0
      DO 1001 K = 1,AJ
          C(I,J) = C(I,J) + (A(I,K) * B(K,J))
1001  CONTINUE

      end1(1,00,00,00,00)
      GO TO 10
*
3000  CONTINUE
*
```

```
152          RETURN
153          END
```

2.2.1. The gs2var Macro

The code begins with the main program. At the end of the type declarations, the first macro call is inserted; it is a call to *gs2var*. This macro declares the variables and common blocks to be used by the monitor package; it takes no parameters.

Prior to the call to *gs2var*, eight define statements should appear in the program (e.g., see lines 1-8 in MATMUL). These statements enable certain arrays within *gs2var* to be properly defined.

The constant RB11 should be some integer constant less than or equal to the smallest row value of all simplex 1's to be executed by the program. By a simplex 1 to be executed by a program we mean a simplex 1 (S^1, T^1, U^1) belonging to a complex to be executed by that program. A row value of a simplex (S,T,U) is an integer *i* such that some *(i,j)* is in S, the underlying region of the simplex. If the smallest row value of all simplex 1's to be executed by the program can be obtained without much difficulty, it should be used for the constant. Otherwise, the largest integer that satisfies the condition and can be obtained without much difficulty should be used.

The constant RB12 should be some integer constant greater than (but not equal to) the largest row value of all simplex 1's to be executed by the program. If that largest row value can be obtained without much difficulty, that value + 1 should be used for the constant. Otherwise, the smallest integer that satisfies the condition and can be obtained without much difficulty should be used.

The constant CB11 should be some integer constant less than or equal to the smallest column value of all simplex 1's to be executed by the program. A column value of a simplex (S,T,U) is an integer *j* such that some *(i,j)* is in S, the underlying region of the simplex. If this smallest column value can be easily obtained, it should be used. Otherwise, the largest integer that satisfies the condition and can be obtained without much difficulty should be used.

The constant CB12 should be some integer constant greater than or equal to the sum of the largest column value of all simplex 1's to be executed by the program and the largest skew factor of all simplex 1's to be executed by the program. (If a skew factor is not explicitly given by the user, it defaults to 0.) If this sum can be easily obtained, it should be used. Otherwise, the smallest integer that satisfies the condition and can be easily obtained should be used.

The remaining integer constants (RB21, RB22, CB21, and CB22) are defined in much the same way as those just described. Here, however, simplex 2's rather than simplex 1's are used in the definition of the constants. If a program will execute one (or many) 1-dimensional complexes (so that no simplex 2's are relevant), the user may set RB21 to 1, RB22 to 2, CB21 to 1, and CB22 to 1.

Of course, the variables declared in *gs2var* must not be used elsewhere in the program by the user in such a way as to conflict with the use dictated by the macro package. For this reason, a list of the variables declared in *gs2var* is appended to this paper.

2.2.2. The gs2init1 and gs2init2 Macros

The next macro call (after *gs2var*) in the main program is to *gs2init1*. This macro initializes certain of the variables in *gs2var*. During execution of the program, *gs2init1* is called only once, regardless of how many different complexes are executed by the program. Therefore *gs2init1* may be termed a program initialization.

The macro *gs2init1* takes 5 parameters; this parameter structure will be used repeatedly in most of our remaining macro calls (except for one call to the *barrier* macro, to be discussed later). The 5 parameters, denoted by \$1, \$2, \$3, \$4, and \$5, are described as follows:

1. \$1 is one character. This character is 1 if the complexes to be executed by the program (all of which must have the same dimension) are 1-dimensional, i.e., if each of them consists of one simplex (and perhaps constraints). The character is 2 if the complexes to be executed by the program are 2-dimensional, i.e., if each of them consists of 2 simplexes (and perhaps constraints).
2. All the remaining parameters have exactly two characters. The first character of \$2 is C if C1 holds, i.e., if the column constraint holds in simplex 1; otherwise it is 0. The second character of \$2 is X if XC21 holds, i.e., if the cross-column constraint from simplex 2 to simplex 1 holds; otherwise it is 0.
3. The first character of \$3 is R if R1 holds, i.e., if the row constraint holds in simplex 1; otherwise it is 0. The second character of \$3 is X if XR21 holds, i.e., if the cross-column constraint from simplex 2 to simplex 1 holds; otherwise it is 0.
4. \$4 and \$5 are analogous to \$2 and \$3, respectively, with the roles of simplex 1 and simplex 2 reversed. In detail, the first character of \$4 is C if C2 holds, i.e., if the column constraint holds in simplex 2; otherwise it is 0. The second character of \$4 is X if XC12 holds, i.e., if the cross-

column constraint from simplex 1 to simplex 2 holds; otherwise it is 0.

5. The first character of $5 is R if R2 holds, i.e., if the row constraint holds in simplex 2; otherwise it is 0. The second character of $5 is X if XR12 holds, i.e., if the cross-column constraint from simplex 1 to simplex 2 holds; otherwise it is 0.

We define the dimension of a program (within the structure that we use) to be the (common) dimension of all complexes that it executes. As indicated earlier, this dimension is equal to $1. If it is 1, then each complex to be executed consists only of its simplex 1. We then set both $4 and $5 equal to 00, set the second character of $3 equal to 0, and set the second character of $4 equal to 0.

We define the multiplicity m of our program to be the total number of times it will execute some complex. (Thus, repeated executions of the same complex, as well as executions of different complexes, all count towards this multiplicity.) Our exposition now temporarily diverges between programs of muliplicity m = 1 (where one complex is executed once) and programs with m > 1. We will return in a later section to the case in which m > 1.

Consider the case when m = 1. Then the next macro call after *gs2init1* will be to *gs2init2*. It will be executed exactly once during the program, since we are solving one problem once. Therefore, *gs2init2* is called the problem initialization, in contrast to the program initialization *gs2init1*.

The macro *gs2init2* takes 5 parameters; they are exactly the same as those in *gs2init1*. Before *gs2init2* is called, a certain number of Fortran INTEGER variables must be assigned values, which will be called user-supplied values to the macro. The supplying of these values is another part of the user's contribution to the quasi-automatic parallelization -- to the cooperative venture between user and macro package. Since m = 1, there is only one complex to consider, namely, the unique complex to be executed by the program. These variables comprise the following:

MNRV1 (the minimum row value, or first coordinate, of simplex 1).

MXRV1 (the maximum row value of simplex 1).

MNCV1 (the maximum column value, or second coordinate, of simplex 1).

MXCV2 (the maximum column value of simplex 1).

MNRV2, MXRV2, MNCV2, and MXCV2 (the analogous values for simplex 2). If the dimension of the complex is 1, then these need not be specified.

SKW1 (the skew factor for simplex 1). This is specified only when the column constraint C1 holds and has a non-zero skew factor.

SKW2 (the analogous value for simplex 2). This is specified only when the complex dimension is 2, the column constraint C2 holds, and C2 has a non-zero skew factor.

LFBDY1(JJJ), where JJJ is an integer variable (the user is free to substitute a different, conflict-free integer variable for JJJ) ranging from MNRV1 to MXRV1 (these are the left boundary values of simplex 1, whose indices vary from the minimum row value to the maximum row value).

RTBDY1(JJJ), where JJJ is an integer variable (the user is free to substitute a different, conflict-free integer variable for JJJ) ranging from MNRV1 to MXRV1 (these are the right boundary values of simplex 1, whose indices vary from the minimum row value to the maximum row value).

UPBDY1(JJJ), where JJJ is an integer variable (the user is free to substitute a different, conflict-free integer variable for JJJ) ranging from MNCV1 to MXCV1 (these are the upper boundary values of simplex 1, whose indices vary from the minimum column value to the maximum column value).

Furthermore, if simplex 1 cross-column constrains simplex 2 (i.e., if XC12 holds), then the user must also supply

LWBDY1(JJJ), where JJJ is again an integer variable (the user is free to substitute a different, conflict-free integer variable for JJJ) ranging from MNCV1 to MXCV1 (these are the lower boundary values of simplex 1, whose indices vary from the minimum column value to the maximum column value).

It is important to note that these user-supplied values to the macro are reserved variables; the user must provide the appropriate values for precisely those named variables given above; otherwise *gs2init2* will not be able to perform properly. Of course, the dummy variable JJJ may be replaced, as indicated above.

If the program dimension is 2, then the user must also supply to the macro the analogous values for simplex 2, namely, LFBDY2(JJJ), for JJJ ranging from MNRV2 to MXRV2 (these are the left boundary values for simplex 2),

RTBDY2(JJJ), for JJJ ranging from MNRV2 to MXRV2 (these are the right boundary values for simplex 2), and

UPBDY2(JJJ), for JJJ ranging from MNCV2 to MXCV2 (these are the upper boundary values for simplex 2).

Furthermore, if simplex 2 cross-constrains simplex 1 (i.e., if XC21 holds), then the user must also specify

LWBDY2(JJJ), for JJJ ranging from MNCV2 to MXCV2 (these are the lower boundary values for simplex 2).

2.2.3. Creation of SLAVE Processes

After the call to *gs2init2*, while still in the main program, a certain number NPROCS - 1 of copies of a subroutine named SLAVE are created with the create macro, which is called in the form create(SLAVE). NPROCS is a Fortran INTEGER variable, declared by the user, whose value (set by the user) is the number of processes that the program will employ.

The subroutine SLAVE is quite simple; all it does is CALL the subroutine WORK. WORK is also CALLed by the main program, at some time after the SLAVEs are created.

Once the SLAVEs are created, the program is executed by NPROCS processes; one of them will be executing either the main program or a copy of WORK called from that program, while each of the other NPROCS - 1 processes will be executing either its copy of SLAVE or a copy of WORK called from that copy of SLAVE.

No parameters are passed from the main program to WORK or from a SLAVE to WORK. (Thus all communication between routines is done by shared variables in COMMON.) Furthermore, no macros from the package are used in SLAVE. A RETURN statement executed by a SLAVE has the effect of a total self-annihilation; that SLAVE simply vanishes. On the other hand, a RETURN statement executed by a copy of WORK simply transfers control back to the calling program, which is either the main program or a SLAVE.

2.2.4. The WORK Routine

A macro call to *gs2var* also appears in subroutine WORK. There it again declares the variables and common blocks used by the monitor. Again no parameters are used in the call to *gs2var*.

Except for a numbered Fortran CONTINUE statement, which provides a target statement to transfer control to, the first executable statements of subroutine WORK are provided by the *gs2* macro. The parameters used in the *gs2* call are exactly the same as those used in the *gs2init1* call.

The following unifying notation will be helpful. If we have a 1-complex consisting of simplex 1 (S^1, T^1, U^1), then we define the underlying region of the complex S to be S^1 and the underlying computational task function T to be T^1. If, however, we have a 2-complex $((S^1, T^1, U^1), (S^2, T^2, U^2), V)$, where simplex 1 is (S^1, T^1, U^1) and simplex 2 is (S^2, T^2, U^2), then we define the underlying region of the complex S to be the union of S^1 and S^2, and we define the underlying computational task function T to have the domain S and to be given by the rule that T = T^1 on S^1 and T = T^2 on S^2.

2.2.4.1. The gs2 Macro

Subroutine WORK performs the computational tasks T(i,j) of the complex to be executed by the program. When a process begins executing the statements in WORK, the instructions in the *gs2* macro are executed.

The function of the *gs2* macro is to hand out a unique pair of subscripts (i,j) from the underlying region of the complex S, so long as there remain subscript pairs from S to be handed out. Thus every time *gs2* is executed, until it has handed out every subscript pair in S, it hands out a different subscript pair (i,j).

A process executing the *gs2* macro monitor is searching for a computational task T(i,j) to do. More precisely, it is searching for a subscript pair (i,j) such that the corresponding task T(i,j) is ready, i.e., all the constraints on T(i,j)'s beginning have been satisfied. The search focuses solely on the current tasks of underlying simplexes of the complex to be executed. (As is consistent with other usage in this paper, we will simply say that (i,j) is ready as a short form in place of the full T(i,j) is ready.)

The search will be carried out among all (either 1 or 2) simplexes of the complex being executed whose subscript pairs have not yet all been handed out to searching processes executing *gs2*. If simplex 1 is such a simplex, the searching process will determine if the current task of simplex 1 is ready to begin. If the task is ready, then the process will acquire the current subscript pair, exit *gs2*, and unlock the monitor so that a subsequent process may enter.

If simplex 1 is not such a simplex, or if the current task in simplex 1 is not yet ready to begin (i.e., some constraint is not yet satisfied), then the search will either turn to simplex 2 (if we are executing a 2-complex and there are still subscript pairs in simplex 2 to be handed out) or loop repeatedly, determining whether the current task in simplex 1 is ready until it is; once it is ready, the process acquires the current subscript pair and exits and unlocks the monitor, just as above.

If the search turns to simplex 2, either because simplex 1 has no more subscript pairs to hand out or because the current task there is not ready, then the above description still holds (with the roles of simplex 2 and simplex 1 reversed).

If any subscript pairs from the complex remain to be handed out when the searching process begins executing the *gs2* macro, the process will eventually grab a pair and exit and unlock the monitor. This pair will be either the current pair in simplex 1 or the current pair in simplex 2. If the current pair in simplex 1 (simplex 2) is obtained, the process will, upon exiting the monitor, go to the statement with (reserved) label 1000 (2000). (The monitor will update its internal variables after the grab so that the same subscript pair is not handed out twice.) However, if no such pairs are left to be handed out (i.e., all the computational tasks T(i,j) have been already handed out to some process), then the searching process will begin windup processing. This means that the process will enter a delay queue and unlock the monitor so that other processes can enter and begin searching for (the no longer available) subscript pairs. Once all NPROCS processes have entered the monitor (i.e., begun executing the *gs2* macro) to perform the (futile) search for subscript pairs, then they all exit the monitor. Upon exiting the monitor, each process will go to the statement with (reserved) label 3000. The last one out resets certain of the monitor's internal variables used during the execution of the complex (although this reset is useful only when the program will execute several complexes, i.e., when the multiplicity m is greater than 1).

Note that the subscript pairs in a simplex of the complex to be executed will be handed out in lexicographic order: (i_1,j_1) precedes (i_2,j_2) if either $i_1 < i_2$ or both $i_1 = i_2$ and $j_1 < j_2$. However, we cannot know in advance the manner in which subscript pairs from the different simplexes will be interleaved, except in that applicable cross-constraints from XC12, XC21, XR12, and XR21 must be satisfied.

2.2.4.2. The end1 and end2 Macros

Once a process is given a subscript pair and exits the monitor, it undertakes the computation of the task corresponding to that pair. If the completion of this task is a prerequisite to the beginning of some other, by means of one of the complex constraints, the *gs2* macro monitor (which dispatches ready subscript pairs) must be notified when the computation of the task has ended.

This notification is accomplished with the *end1* macro for tasks in simplex 1 and with the *end2* macro for tasks in simplex 2. Both macros take the same five parameters as does the *gs2init1* macro.

The macro *end1* appears at most once in subroutine WORK. It should be inserted into WORK if there is some internal constraint C1 or R1 on simplex 1, or if simplex 1 cross-constrains simplex 2 externally via either XC12 or XR12; otherwise it should not appear. The macro should be inserted into the code just after the code to perform all the computational tasks T^1 of simplex 1 has ended.

The rules governing the use of *end2* are the same, with the roles of simplex 1 and simplex 2 reversed (for example, XC21 or XR21 rather than XC12 or XR12 would mandate the use of *end2*). Of course, if the dimension of our complex is 1, then *end2* should not appear in the program.

2.2.4.3. The cmplxend Macro

Recall that we are still discussing the case m = 1, i.e., the case when the program executes only one complex. To complete our discussion of the macros used in this case (all macros used in the case of multiplicity 1 will also be used for higher multiplicities), we introduce the *cmplxend* macro.

This macro is used when some task T(i,j) computes a result that solves the problem under consideration, so that no new computational tasks should begin and those in progress should perform an orderly windup. It notifies the *gs2* macro to hand out no more subscript pairs; thus, all processes that subsequently execute *gs2* after completing their present computational task will begin windup processing there. It also sets the reserved Fortran variable EXHST to 0. This indicates that the execution of the complex was terminated by the occurrence of a solution, rather than by the exhaustion of all the computational tasks without a solution being found.

The *cmplxend* macro will be used in a small number of examples. For instance, if any duplicate is found in the program GETADUP, *cmplxend* is used to terminate processing.

2.2.5. Complications Introduced by Multiplicity Greater Than 1

Having discussed the use of our macros and our program structure when the program multiplicity m is 1, we now look at the modifications needed when the multiplicity is greater than 1.

First of all, the calls to WORK in this case use a parameter. The call from the main program passes a 0 to WORK, while the calls from the SLAVES created by NPROCS - 1 all pass a 1 to WORK. WORK has the one argument WHO. Thus a process executing WORK can determine its origins. If WHO is 0, then it comes directly from the main program; if WHO is 1, then it comes from a SLAVE.

In the main program, a DO-loop must be set up to handle the m problems. One iteration of the loop will be performed for each problem. The body of the loop contains the call to WORK.

Just as in the case m = 1, the main program creates NPROCS - 1 processes, each of which begins executing a copy of subroutine SLAVE. This takes place once for the entire program. When m > 1, we will place the call to *gs2init2* after the creation of the SLAVEs, reversing the order for m = 1. The code determining and supplying the user-supplied values for the macros will appear after the creation of the SLAVEs and before the macro call for *gs2init2*.

If we wish, we can place *gs2init2*, together with the code providing the user-supplied values, within the above-mentioned DO-loop. This placement provides a capability to reconfigure the underlying simplex regions before each complex execution. Thus, for example, one program may perform an identical matrix analysis on successive complex executions, but for matrices of different sizes. Another might perform the same kind of sort on successive complex executions, but for arrays of different sizes. We can even, within the framework introduced here, perform a matrix analysis on one complex execution and a sort on the next, but our problems below do not utilize this capability.

When m > 1, the non-trivial executable statements (i.e., those that are not no-ops) of WORK begin with a call to the *barrier* macro. In this call *barrier* takes just one parameter; it appears in the form *barrier*(1). The *barrier* macro has the effect of delaying any process reaching it until all NPROCS processes are so delayed; then they are all released (in some order). The parameter 1 used here in *barrier* simply provides a label.

The Fortran INTEGER COMMON variable NDONE is used in a conditional branch statement to determine whether a process executing WORK should execute the *gs2* macro or bypass it so as to perform windup processing for the entire program. (Windup processing for individual problems within the program is done by the *gs2* macro.) NDONE is initialized to 0 in the main program, and set to 1 in just after the last iteration of the m problem DO-loop is completed.

After NDONE is set to 1, the main program calls WORK one last time for windup processing. The conditional test of NDONE is performed in WORK, immediately following the macro call to *barrier* and immediately preceding the macro call to *gs2*.

After each execution of a complex, the processes that come from a SLAVE hang on the barrier in WORK, while the process that comes directly from the main program returns to the main program. This divergence of behavior for different processes is accomplished by the WHO variable described above. The subsequent call to WORK from the main program in the next iteration places this last process also at the barrier, so that now all processes can penetrate the barrier, and either the concurrent executions of copies of WORK relevant to the new problem may begin, or the final windup processing may begin.

The main program can communicate to the processes executing WORK which complex they are executing (i.e., which problem they are working on) by the use of COMMON variables.

Other than the modifications given here, the case when $m > 1$ is dealt with just as the case when $m = 1$.

3. A Geometric Representation of Synchronization Dependencies

We now discuss the geometric "synchronization patterns" (which provide a concise pictorial description of a given parallelization structure) and their construction.

To construct the synchronization pattern, we must draw the points of the underlying region of the complex. If there are two simplexes in the complex, then the respective underlying regions are separated by a line. Thus, for example, if the underlying region of simplex 1 of a 2-complex is

$$S^1 = \{ (1,1), (1,2), (1,3) \}$$

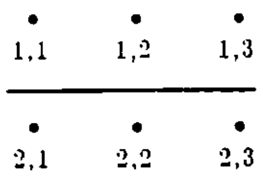
and the underlying region of simplex 2 is

$$S^2 = \{ (2,1), (2,2), (2,3) \}$$

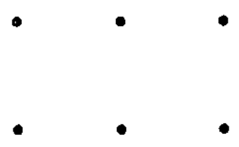
the underlying region S of the entire complex is

$$S = \{ (1,1), (1,2), (1,3), (2,1), (2,2), (2,3) \}$$

and may be drawn as:



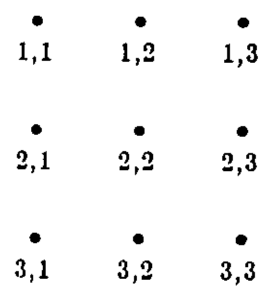
The two underlying simplex regions above are separated by the line. Assuming no constraints, the parallelization structure, drawn below, would appear exactly as S above, except that for simplicity we usually omit coordinate labels when drawing the synchronization pattern, as well as the line separating the simplexes.



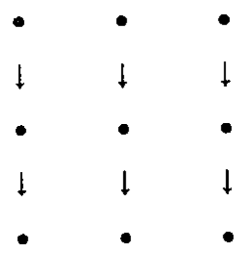
In general, constraints are indicated on the synchronization pattern. The C constraint (internal column constraint) within a simplex is indicated by a vertical arrow connecting two neighboring points in a column, drawn from the one above to the one below. Thus, for example, suppose that the underlying region of a 1-complex consists of the points

$S = \{ (1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3) \}$.

Then S may be drawn as follows:

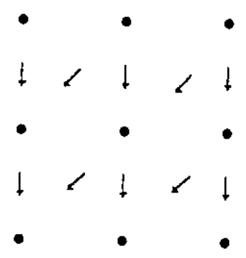


If the C constraint holds in this complex, then the synchronization pattern could be drawn as

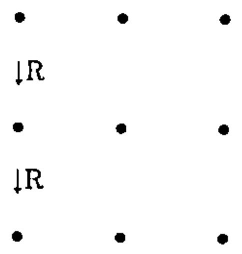


If, furthermore, this column constraint includes a positive skew factor K, that skew factor must also be indicated on the synchronization pattern by additional arrows that connect each point pair of the form (I-1,J+K) and (I,J). Each additional arrow is drawn from some point (I-1,J+K) to the corresponding point (I,J).

Thus if this column constraint includes a skew factor of 1, the synchronization pattern would now be drawn as



A row constraint is indicated by an arrow (with an "R" next to it) pointing from the leftmost point in each (but the last) row to the leftmost point in the row below. Thus, if the row constraint holds in the above underlying region S, the synchronization pattern would be depicted as



Of course, once we know that R holds, we dispense with C, since it is implied by R.

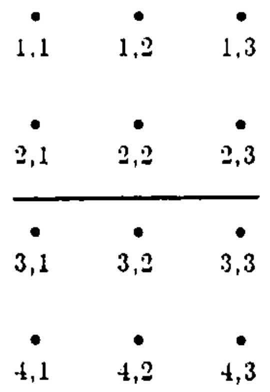
A cross-column constraint XC is indicated by drawing, for each point P on the lower boundary of the constraining simplex that lies directly above some point Q in the constrained simplex, a vertical arrow (with an "X" next to it) pointing downward from P to Q. Thus, for example, if we have a 2-complex with

$S^1 = \{ (1,1), (1,2), (1,3), (2,1), (2,2), (2,3) \}$

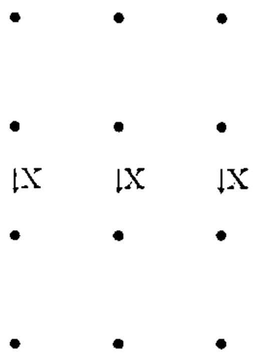
and

$S^2 = \{ (3,1), (3,2), (3,3), (4,1), (4,2), (4,3) \}$,

and the constraint XC12 holds, then S would be drawn as



and the synchronization pattern would be drawn as



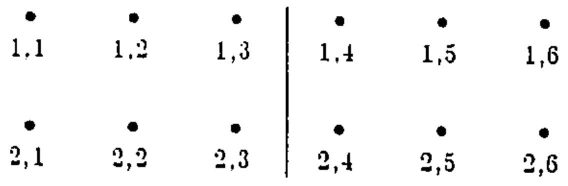
Similarly, a cross-row constraint XR is indicated by drawing, for each point P on the right boundary of the constraining simplex that lies immediately to the left of some point Q in the constrained simplex, a horizontal arrow pointing rightward from P to Q. Thus, for example, if we have a 2-complex with

$S^1 = \{ (1,4), (1,5), (1,6), (2,4), (2,5), (2,6) \}$

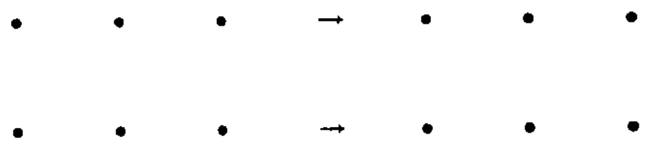
and

$$S^2 = \{ (1,1), (1,2), (1,3), (2,1), (2,2), (2,3) \}$$

and the constraint XR21 holds, the underlying region S would be drawn as



and the synchronization pattern would be



Of course, all constraints that hold must be indicated on a synchronization pattern; if more than one holds, they are each indicated on the pattern.

4. A Formal Description of the Structure

In the previous sections, we have attempted to motivate our approach to scheduling units of computation based on a geometric representation in a two-dimensional plane. We now give a detailed, formal presentation of the structure.

We start with some basic definitions.

Let M_1 and M_2 be integers, with $M_1 < M_2$. The closed integer interval $[M_1,M_2]$ is defined to be the set of all integers from M_1 to M_2. All intervals used in this paper will be integer intervals, i.e., will contain only integers.

Let f be an integer-valued function whose domain is the interval $[M_1,M_2]$. The graph of f, written **graph** f, is the subset of the Cartesian integer plane Z_2 consisting of those points $(M,f(M))$, where M is in the domain $[M_1,M_2]$. The reversed graph of f (so-called because the roles of first and second coordinate are reversed), written **rev** f, is the subset of the Cartesian integer plane Z_2 consisting of those points $(f(M),M)$, where M is in the domain $[M_1,M_2]$.

As mentioned previously, our conventions for representing ordered pairs follow the conventions for representing elements in a matrix, rather than the more common graphical representation of Z_2.

Note that a matrix is actually a function of two integer variables and that its set of index pairs is the domain of that function. Thus it is natural, in looking for a way to depict the domain of a computational task function T(I,J) which is also a function of two integer variables, to take as a starting point the usual picture for the set of index pairs of a matrix. (However, while the set of index pairs of a matrix is an integer rectangle, the domain of a computational task function is more general. For example, it may have jagged boundaries, or be an "integer line.")

We now use the notions of graph and reversed graph as building blocks to define regions and regular regions in the Cartesian integer plane Z_2. Suppose that we have two functions f and g with the same domain $[M_1,M_2]$ and that $f(M) < g(M)$ for all M in $[M_1,M_2]$. We define the region D = D(f,g) to be the set of all points in Z_2 whose first coordinate M is in $[M_1,M_2]$ and whose second coordinate lies in the interval $[f(M),g(M)]$. Thus D(f,g) can be thought of as the set of all points in Z_2 that lie between (or on) the graph of f and the graph of g. Similarly, under the above conditions on f and g we define the region **rev** D = **rev** D(f,g) to be the set of all points in Z_2 whose second coordinate L is in $[M_1,M_2]$ and whose first coordinate K lies in the interval $[f(L),g(L)]$. Thus **rev** D(f,g) can be thought of as the set of all points in Z_2 that lie between (or on) the reversed graph of f and the reversed graph of g.

Note that if S is a subset of Z_2 which is of the form D(f,g), then f, g and the common domain of f and g are all uniquely determined; similarly when S is of the form **rev** D(f,g).

A subset S of Z_2 is called a regular region if there are functions f_1, f_2, g_1 and g_2 such that f_1 and g_1 have common domains, f_2 and g_2 have common domains, S = $D(f_1,g_1)$, and S = **rev** $D(f_2,g_2)$. (In other words, to be regular S must be both a region of the form $D(f_1,g_1)$ and a region of the form **rev** $D(f_2,g_2)$.) If S is a regular region (or, in brief, regular), f_1 is called the left boundary function of S, g_1 the right boundary function, f_2 the upper boundary function, and g_2 the lower boundary function.

Now that we have the notion of "regular region" defined, we can introduce computational task functions. A computational task function (or, more briefly, task function) is a function T^1 whose domain is a regular region S in Z_2 and which assigns to each (I,J) in S some computational task T^1(I,J). (For convenience, we will also use the alternate notation T^1_{IJ} for T^1(I,J).) For purposes of the parallelization structure developed here, we are concerned not with the details of the particular computer instructions required by the T^1(I,J), or the mathematical algorithm that the T^1(I,J) represent, but rather with the times when the T^1(I,J) are able to begin during the execution of a program and the times at which the T^1(I,J) end.

Within a parallelization structure a certain number (perhaps zero) of constraints will be placed on the T^1(I,J). No T^1(I,J) may begin until all the constraints on its beginning have been satisfied; such constraints will either be internal and require that certain other T^1(I,J) have ended, or be external and require that certain T^2(K,L) have ended, where T^2 is a different computational task function within the same parallelization structure.

For now we will focus on internal constraints. Suppose that we have a computational task function T defined on a regular region S. For (I,J) in S, the begin time of the task T(I,J) (i.e., the time at which that task begins during the execution of a program) may be denoted by b(T(I,J)). Since our discussion henceforth will be with reference to a particular task function which is known from the context, for ease of notation we will suppress the explicit reference to T and instead denote the begin time of T(I,J) simply by b(I,J). (We will sometimes use the alternate notation b_{IJ} for b(I,J).) Similarly, we will denote the end time of the task T(I,J) (i.e., the time at which that task ends during the execution of a program) by e(I,J). (As above, we will sometimes use the alternate notation e_{IJ} for e(i,j).) In general, when it is clear from the context what is meant, we will often not distinguish between the computational task T(I,J) and the point (I,J) in S. Thus, for example, we might speak of the time at which a "point begins" to mean the time at which the computational task assigned to that point begins.

Let T be a computational task function whose domain is the regular region S. The basic internal constraints on T(I,J) are of two kinds, begin constraints and end constraints. Begin constraints are of the form

i) $b(I,J) > b(K,L)$;

i.e., the (I,J)'th task T(I,J) can begin only after the (K,L)'th task T(K,L) has begun ((I,J) and (K,L) are points in S). End constraints are of the form

ii) $b(I,J) > e(K,L)$;

i.e., the (I,J)'th task T(I,J) can begin only after the (K,L)'th task T(K,L) has ended ((I,J) and (K,L) are points in S).

Throughout this paper, whenever (I,J) and (I,K) are both in S, and $J < K$, the constraint

$b(I,K) > b(I,J)$

will be understood to be imposed. The imposition of this constraint requires that in terms of begin times, each row in the regular region S will be traversed in the natural order (from lower column subscripts to higher column subscripts). (A row in S is defined to be the (non-empty) set of all points in S with a particular first coordinate; that common first coordinate is called the row subscript of the row. Similarly, a column in S is defined to be the (non-empty) set of all points in S with a particular second coordinate; that common second coordinate is called the column subscript of the column.) Also, throughout this paper, whenever (I,J) and (K,L) are both in S, and $I < K$, the constraint

$b(K,L) > b(I,J)$

will be understood to be imposed. The imposition of this constraint requires that in terms of begin times, each point in a lower row (i.e., one with a larger row subscript) will begin after each point in a higher row (i.e., one with a smaller row subscript.) This implies that a task in a row may begin only after all tasks in rows with smaller row subscripts have begun.

Together, the above two constraints imply that whenever we have a regular region S and a computational task function T defined on S, the points of S are traversed lexicographically in terms of begin times. The tasks in each row begin in the same order as that of the column subscripts, and the tasks in rows with smaller row subscripts begin before the tasks in rows with larger subscripts.

An equivalent way of stating this is as follows: given distinct points (I,J) and (K,L) in S, (I,J) begins before (K,L) if and only if either $I < K$ or both $I = K$ and $J < L$.

Thus, a program executing the computational task function T defined on the regular region S (in terms of begin times) traverses S row by row (going from lower row subscripts to higher row subscripts) and traverses each row going from lower column subscripts to higher column subscripts.

We now consider the relevant internal end constraints on a computational task function T defined on a regular region S. Unlike the two internal begin constraints given above which are always implied and which are the only begin constraints that hold in the parallelization structures used here, the relevant internal end constraints must always be explicitly given; furthermore they may or may not apply.

We can organize the internal end constraints that apply here via the notions of row constraint and column constraint.

Definition: Let T be a computational task function defined on the regular region S. (S,T) is row-constrained if the first (leftmost) task in a row can begin only after all tasks in the previous row have completed. (Given a row R with row subscript I, the previous row is the row with row subscript I - 1.)

Note that it follows from the first begin constraint given above that each task in a row can begin no sooner than the first (leftmost) task in that row does.

Alternative ways of stating that (S,T) is row-constrained include "the row constraint holds in (S,T)," "the row-constraint R holds in (S,T)," or simply "R holds in (S,T)." Usually the mention of (S,T) will be suppressed when no confusion can result.

To express the notion "R holds in (S,T)" in formula, we write $S = D(f_1,g_1)$, where f_1 and g_1 are the left and right boundary functions of S, respectively, and $[M_1,M_2]$ is the common domain of f_1 and g_1. Then we can write "R holds" as

$$b(M,f_1(M)) > e(M\text{-}1,K)$$

whenever M-1 and M are in $[M_1,M_2]$, and (M-1,K) is in S. Equivalently, we could write

$$b(M,f_1(M)) > e(M\text{-}1,K)$$

whenever M-1 and M are in $[M_1,M_2]$, and

$$f_1(M\text{-}1) < K < g_1(M\text{-}1).$$

Having defined "row constraint," we now define the other internal end constraint to be used in this paper, the "column constraint."

Definition: Let T be a computational task function defined on the regular region S. (S,T) is column-constrained, with the non-negative integer L as skew factor, if for all J, a task in the J'th column can begin only after all tasks in the J'th column from earlier rows and all tasks in the (J+L)'th column from earlier rows have completed. (Given as row R with row subscript I, an earlier row is a row whose row subscript is smaller than I. We know, from the implied begin constraints, that tasks in an earlier row than R will begin before any of the tasks in R do.)

Alternative ways of stating that (S,T) is column-constrained, with skew-factor L," include "the column constraint, with skew factor L, holds in (S,T)," "the column constraint C, with skew factor L holds in (S,T)," or simply "C, with skew factor L, holds in (S,T)." Usually, the mention of (S,T) will be suppressed when no confusion can result. Also, often the mention of the skew-factor L will be suppressed; either it may be supplied later or, if omitted, will be understood to be 0.

To express the notion "C, with skew factor L, holds in (S,T)" in formula, we write S = **rev** $D(f_2,g_2)$, where f_2 and g_2 are the upper and lower boundary functions of S, respectively, and $[M_1,M_2]$ is the common domain of f_2 and f_2. Then "C holds, with skew factor L," is equivalent to the two conditions

i) $b(I,J) > e(K,J)$

whenever (I,J) and (K,J) are in S and $I > K$

and

ii) $b(I,J) > e(K,J+L)$

whenever (I,J) and (K,J+L) are in S and $I > K$.

Another equivalent formulation of "C holds, with skew factor L," is given by the two conditions

i) $b(I,J) > e\ (K,J)$

whenever J is in $[M_1,M_2]$, both I and K are in $[f_2(J),g_2(J)]$, and $I > K$,

and

ii) $b(I,J) > e(K,J+L)$

whenever J and J+L are in $[M_1,M_2]$, I is in $[f_2(J),g_2(J)]$, K is in $[f_2(J+L),g_2(J+L)]$, and I > K.

Clearly a row-constrained simplex is always column constrained (with any non-negative skew factor.) Thus we need never consider the situation where both R and C hold. It suffices to consider (S,T) where R holds, or C holds with some skew factor L, or no constraints hold.

Note that when R holds, we have a degenerate case of two-dimensional parallelization (the two dimensions come from the two-dimensionality of the Cartesian integer plane Z_2 and from S being a subset of Z_2.) Since when R holds, each row of computational tasks must end before any tasks in the next row can begin, each row may be thought of as a separate "one-dimensional" parallelization unit. Thus the structure when R holds effectively becomes that of a sequence of one-dimensional parallelization units.

We are now ready to define the basic parallelization structure, the simplex.

Definition: A parallelization simplex (or more briefly, a simplex) is a triple (S,T,U) where S is a regular region in Z_2, T is a computational task function defined on S, and U is a pair (C,L), where L is a non-negative integer and C is a character, or the character R, or empty.

U merely tells us whether the simplex is column constrained with skew factor L, row constrained, or unconstrained.

The simplex may be thought of as the basic "atom" of two-dimensional parallelization structure. The begin constraints described earlier are internalized within it. Note that no end constraint of the form b(I,J) < b(I,K) within a given row can occur. Thus a task, when waiting to begin, may be waiting for the completion of tasks from earlier rows, but is never waiting on the completion of tasks from the same or from later rows. If we are in a computational situation where a task in a row cannot begin until another task in that row has ended, we need a parallelization structure more complicated than the simplex, i.e., one in which constraints can operate across simplex boundaries. There will also be instances where it will be useful for constraints operating within a column, as well as constraints operating within a row, to be able to operate across simplex boundaries. We now set about defining a parallelization structure (the complex) that will allow us to deal with certain of these situations.

Definition: Let (S^1,T^1,U^1) and (S^2,T^2,U^2) be parallelization simplexes such that the underlying regions S^1 and S^2 are disjoint, i.e., have no points in common.

Write $S^1 = \mathrm{D}(f_{11}, g_{11})$ and $S^2 = \mathrm{D}(f_{21}, g_{21})$, where f_{11} and g_{11} are the left and right boundary functions, respectively, of S^1; f_{21} and g_{21} are the left and right boundary functions, respectively, of S^2; $[M_{11}, M_{12}]$ is the common domain of f_{11} and g_{11}; and $[M_{21}, M_{22}]$ is the common domain of f_{21} and g_{21}. (S^1, T^1, U^1) will be said to "border (S^2, T^2, U^2) from the left" if

i) the intersection of the two intervals $[M_{11}, M_{12}]$ and $[M_{21}, M_{22}]$ is non-empty, i.e., the intervals have some points in common (note that this intersection must also be an interval), and

ii) if M is in the intersection of the intervals $[M_{11}, M_{12}]$ and $[M_{21}, M_{22}]$, then

$$g_{11}(M) + 1 = f_{21}(M),$$

i.e., the right boundary function value for the first simplex is one less than the left boundary value for the second simplex.

Definition: Let (S^1, T^1, U^1) and (S^2, T^2, U^2) be parallelization simplexes such that the underlying regions S^1 and S^2 are disjoint, i.e., have no points in common. Write $S^1 = \textbf{rev}\ \mathrm{D}(f_{12}, g_{12})$ and $S^2 = \textbf{rev}\ \mathrm{D}(f_{22}, g_{22})$, where f_{12} and g_{12} are the upper and lower boundary functions, respectively, of S^1; f_{22} and g_{22} are the upper and lower boundary functions, respectively, of S^2; $[M_{11}, M_{12}]$ is the common domain of f_{12} and g_{12}; and $[M_{21}, M_{22}]$ is the common domain of f_{22} and g_{22}. (S^1, T^1, U^1) will be said to "border (S^2, T^2, U^2) from above" if

i) the intersection of the two intervals $[M_{11}, M_{12}]$ and $[M_{21}, M_{22}]$ is non-empty, i.e., the intervals have some points in common (note that this intersection must also be an interval), and

ii) if M is in the intersection of the intervals $[M_{11}, M_{12}]$ and $[M_{21}, M_{22}]$, then

$$g_{12}(M) + 1 = f_{22}(M),$$

i.e., the lower boundary function value for the first simplex is one less than the upper boundary function value for the second simplex.

We now focus on the external end constraints imposed by one simplex on another(we make no use of external begin constraints in this paper.)

Definition: Let (S^1, T^1, U^1) and (S^2, T^2, U^2) be parallelization simplexes with disjoint underlying regions S^1 and S^2. An external end constraint from (S^1, T^1, U^1) to (S^2, T^2, U^2) is a constraint of the form

$b(I,J) > e(K,L)$,

where (I,J) is in S^2 and (K,L) is in S^1.

This says that the (I,J)'th task $T^2(I,J)$ cannot begin until the (K,L)'th task $T^1(K,L)$ has completed. Note that we can define an external end constraint from (S^2,T^2,U^2) to (S^1,T^1,U^1) simply by requiring (I,J) to be in S^1 and (K,L) to be in S^2 in the above definition.

We now may organize the external end constraints that apply within the parallelization structures used here via the notions of cross-row constraint and cross-column constraint.

Definition: Let (S^1,T^1,U^1) and (S^2,T^2,U^2) be parallelization simplexes with disjoint underlying regions S^1 and S^2. Write $S^1 = D(f_1,g_1)$ and $S^2 = D(f_2,g_2)$, and let $[M_1,M_2]$ be the non-empty interval that is the intersection of the common domain of f_1 and g_1 and the common domain of f_2 and g_2. We say that (S^1,T^1,U^1) cross-row end constrains (or sometimes, more briefly, row constrains) (S^2,T^2,U^2) if

i) (S^1,T^1,U^1) borders (S^2,T^2,U^2) from the left, and

ii) for each M in $[M_1,M_2]$, all tasks in the M'th row of (S^1,T^1,U^1) must end before any tasks in the M'th row of (S^2,T^2,U^2) may begin.

In formula, condition ii) may be written as

ii') $b(M,N) > e(M,K)$

whenever $f_2(M) < N < g_2(M)$ and $f_1(M) < K < g_1(M)$, or alternatively as

ii'') $b(M,f_2(M)) > e(M,K)$

whenever $f_1(M) < K < g_1(M)$.

Conditions ii') and ii'') are equivalent because of the implied internal begin constraints within (S^2,T^2,U^2), which require that no task in the M'th row of (S^2,T^2,U^2) may begin until the leftmost task in that row (i.e., the task with column subscript $f_2(M)$) does.

Having defined cross-row constraints, we now turn to the definition of cross-column constraints.

Definition: Let (S^1,T^1,U^1) and (S^2,T^2,U^2) be parallelization simplexes with disjoint underlying regions S^1 and S^2. Write S^1 = **rev** $D(f_1,g_1)$ and S^2 = **rev**

$D(f_2,g_2)$, and let $[M_1,M_2]$ be the non-empty interval that is the intersection of the common domain of f_1 and g_1 and the common domain of f_2 and g_2. We say that (S^1,T^1,U^1) cross column end constrains (or sometimes, more briefly, column constrains) (S^2,T^2,U^2) if

i) (S^1,T^1,U^1) borders (S^2,T^2,U^2) from above, and

ii) for each M in $[M_1,M_2]$, the last task in the M'th column of (S^1,T^1,U^1) (i.e., the one with the largest row subscript) must end before the first task in the M'th column of (S^2,T^2,U^2) (i.e., the one with the smallest row subscript) can begin.

Condition ii) may be written as

ii') $b(f_2(M),M) > e(g_1(M),M)$.

Note that by interchanging the roles of the two simplexes in each of the above two definitions, we can define "(S^2,T^2,U^2) cross-row constrains (S^1,T^1,U^1)" as well as "(S^2,T^2,U^2) cross-column constrains (S^1,T^1,U^1)."

Linguistic variants of "(S^1,T^1,U^1) cross-row constrains (S^2,T^2,U^2)" will include "(S^1,T^1,U^1) row constrains (S^2,T^2,U^2)," "(S^2,T^2,U^2) is cross-row constrained by (S^1,T^1,U^1)," and "(S^2,T^2,U^2) is row constrained by (S^1,T^1,U^1)." Similar variants will be used for "(S^1,T^1,U^1) cross-column constrains (S^2,T^2,U^2)."

We are now able to define the notion of "parallelization complex," or, more briefly, "complex." All parallelization structures dealt with in this paper fall within this category. Two kinds of complexes will be utilized here, namely, 1-complexes and 2-complexes. The dimension of an n-complex, where n is either 1 or 2, is defined to be n. Future research is needed to define and apply complexes with dimension greater than 2.

Definition: A parallelization 1-complex (or, more briefly, a 1-complex) is merely a simplex (S^1,T^1,U^1). A 2-complex is an ordered triple $((S^1,T^1,U^1),(S^2,T^2,U^2),V)$ whose first element is a simplex (S^1,T^1,U^1), whose second element is a simplex (S^2,T^2,U^2)), and whose third element V is a non-empty set of 1 to 4 character strings chosen from among the strings "XC12", "XC21", "XR12", and "XR21." Thus V indicates the cross-constraints that hold between the simplexes of the complex; if XR12 is in V, then (S^1,T^1,U^1) cross-row constrains (S^2,T^2,U^2), while if XR21 is in V, then (S^2,T^2,U^2) cross-row constrains (S^1,T^1,U^1). Similar interpretations are given to XC12 and XC21.

If (S^1,T^1,U^1) is a 1-complex or the first element of a 1-complex, it will be referred to as the simplex 1 of that complex (or, more briefly, as simplex 1).

Similarly, if (S^2, T^2, U^2) is the second element of a 2-complex, it will be referred to as simplex 2.

5. Summary of Examples

We now summarize the examples to be presented. As indicated earlier, they comprise the set of examples that appear in the two Overbeek and Lusk papers [4, 5, 3]. Added to this set are an example computing an approximate solution to a discretization of the Dirichlet problem for a three-dimensional grid, which Overbeek and Lusk use in their classes in parallel processing given at the Argonne National Laboratory, and an example "GETADUP" (a modification of the example "GETDUPS"), which provides a simple illustration of a 2-complex structure.

The first example, ADDTWO, is simply the vector addition of two vectors, with the component additions done in parallel. Here we have a 1-complex whose underlying region is simply a rectangle with one row. There are no constraints imposed on the simplex. ADDTWO illustrates how the self-scheduling DO-loop (see Overbeek and Lusk [2]) is handled as a special case of the *gs2* macro with one row in the underlying region of the relevant simplex.

The second example, CHECKTWO, is the modification of ADDTWO obtained by considering the problem of determining whether the sum of two vectors has a component greater than or equal to 100. Here we have a 1-complex whose underlying region is identical with that of ADDTWO; again there are no constraints. CHECKTWO illustrates the use of the "***cmplxend***" macro, which institutes windup processing once some component greater than or equal to 100 is found.

The third example, "GETDUPS," considers the problem of determining all the duplicates (for purposes of this problem, "duplicates" means successive components that are equal) in the sum of two vectors. The parallelization structure is a 1-complex whose underlying region has two rows. This example provides an elementary introduction to the use of the column constraint C. Alternative formulations introduce the skew factor; one formulation uses a skew factor of 0, while the other uses a skew factor of 1.

The fourth example, "GETADUP," is the modification of "GETDUPS" obtained by simply trying to determine whether the sum of two vectors has a duplicate, rather than trying to determine all the duplicates. This provides our first example of a parallelization structure that is a 2-complex; each simplex has one row. It also illustrates the cross-column constraint XC21, and incidentally provides another use of the ***cmplxend*** macro.

The fifth example, MATMULT, computes the product of two matrices, with the (independent) computations of the elements of the product matrix done in parallel. MATMULT provides an elementary example of a parallelization structure that is a simplex, has no constraints, and whose underlying region is a rectangle in the Cartesian integer plane whose sides can have any number of points and are parallel to the coordinate axes.

The sixth example, "SORT," performs a shell sort on several one-dimensional arrays of differing lengths. It gives rise to a simplex whose underlying region has a jagged right boundary (as well as a jagged lower one.) The rows of this region correspond to stages of the sort. SORT introduces the use of the row constraint R. Furthermore, it illustrates how repeated simplex executions are handled -- in particular, how the underlying region of the structure is reconfigured prior to a new complex execution. The reader is provided with a self-contained explanation of the Shell sort.

The seventh example, "GRID," provides an approximate solution to a discrete analogue of the Dirichlet problem for a cube. The parallelization structure is a 1-complex. The underlying region is an integer rectangle whose number of rows is the number of iterations to be performed and whose number of columns is two less than the number of columns of the grid imposed on the cube. The relevant constraint is the column constraint C, with skew factor 1. Two copies of the grid are used; an iteration updates one copy by producing, on the other copy, function values that are averages of the "nearest neighbor" function values on the copy to be updated. Each updating of an interior slice of the grid parallel to the yz plane provides a separate computational task.

GRID provides a non-elementary example with wide physical application which is elegantly and simply dealt with via the *gs2* macro in conjunction with the parallelization structure introduced in this paper. The resulting program is significantly simpler and shorter than that obtained via the "askfor" monitor and, of course, requires no user-coded macros.

The eighth and last example, "QR," performs the Householder algorithm used in the QR factorization of a square matrix. It provides a non-elementary example of a 2-complex structure that illustrates the power of the macro package presented here. The underlying region is a right isosceles integer triangle in the Cartesian integer plane. Simplex 1 has the hypotenuse as its underlying region, while the remaining points of the triangle form the underlying region for simplex 2. Tasks associated with simplex 1 are called "creates," while tasks associated with simplex 2 are called "applies."

This example uses the C2, XR12 and XC21 constraints. XR12 means here that each "create" must be completed before a corresponding row of "applies" can begin. XC21 and C2 together mean here that a column of "applies" must be completed before the corresponding "create" can begin, and each "apply" must be completed before the "apply" directly below it can begin.

As in the last example, the user here who only has to insert the macros with correct parameters (as well as provide the relevant reserved variables with values) is spared the considerable complexity of writing his own monitor macros (for contrast, see the treatment of this example via the askfor monitor in [1].)

6. Examples

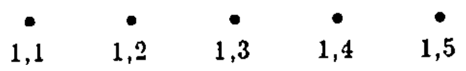
6.1. ADDTWO

A and B are two vectors with the same number N of elements. N is assumed to be less than or equal to 1000. A and B are to be added so as to produce C, i.e.,

$$C(J) = A(J)+B(J) \text{ for } 1 \le J \le N.$$

The N additions are all independent of one another.

This parallelization structure may be described by a simplex whose underlying region S has 1 row and N columns. S is the 1xN integer rectangle depicted below for the representative case N = 5:



For each (1,J), T_{1J} is the task "add A(J) + B(J) to produce C(J)."

There are no row or column constraints in this simplex. Thus the macro call parameters include no C's, R's, or X's; for example, the call to the gs2 macro is given by *gs2(1,00,00,00,00)*.

User-supplied values to the ***gs2var*** macro are as follows:

$RB11 = 1$

$RB12 = 2$

$CB11 = 1$

$CB12 = 1000$

$RB21 = 1$

$RB22 = 2$

$CB21 = 1$

$CB22 = 1$

The above values are supplied to the ***gs2var*** macro via eight define statements. The last four values (RB21, RB22, CB21 and CB22) are formal values; they are relevant only to simplex 2, while the parallelization structure here involves only simplex 1. In future examples involving only one simplex these four values will not be mentioned, but the same values should always be supplied.

User-supplied values to the *gs2init2* macro are as follows:

$MNRV1 = 1$

$MXRV1 = 1$

$MNCV1 = 1$

$MXCV1 = N$

$LFBDY1(1) = 1$

$RTBDY1(1) = N$

$UPBDY1(J) = 1$ for $1 \leq J \leq N$.

The above values are supplied to the *gs2init2* macro via Fortran assignment statements; of course, those for UPBDY1 are supplied within a DO loop.

Note: This example can be handled easily by the *getsub* macro (see the Overbeek and Lusk papers). It illustrates that the *getsub* macro can be viewed as the special case of the *gs2* macro obtained when the parallelization structure is a simplex and the underlying region S consists of exactly one row.

Since there are no constraints in this example, the synchronization pattern here is exactly the same as the picture of the underlying region S given above.

6.2. CHECKTWO

This is the last example dealt with in the Overbeek and Lusk tutorial [2]. It deals with a sequence of identical problems. The individual problem takes two vectors A and B with the same number N of elements (N is assumed to be less than or equal to 1000) and determines whether the sum C of A and B, defined just as in ADDTWO by

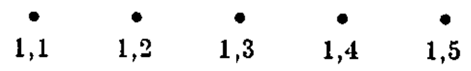
$C(J) = A(J)+B(J)$, for $1 \leq J \leq N$

has some component C(J) greater than 100.

Thus we wish to add corresponding components of A and B and keep going either until all are added, with none of the sums being greater than 100 (this is called a solution by exhaustion, since we have exhausted all the tasks to be done without satisfying the given condition), or until some one component sum exceeds 100. If this event happens, then no new component sums should begin, while all those already begun but not yet completed should be terminated in an orderly way (the way this is accomplished here is by allowing them to complete), and then "windup processing" should take place.

The parallelization structure here is precisely the same as in ADDTWO, since the possibility of terminating the computation without performing all N component sums is superimposed on the parallelization structure rather than being a part of it.

Thus once again we have a simplex whose underlying region S has 1 row and N columns. S is the 1xN integer rectangle drawn below for the representative case N = 5:



For each (1,J), T_{1J} is the task "add A(J) + B(J) to produce C(J), and terminate the problem if $C(J)>100$."

Just as with ADDTWO, there are no row or column constraints in the simplex. The macro call to $gs2$ is done via $gs2$(1,00,00,00,00).

The user-supplied values to both the *gs2var* and the *gs2init*2 macros are exactly the same as in ADDTWO. Thus the user-supplied values to the *gs2var* macro are as follows:

$RB11 = 1$

$RB12 = 2$

$CB11 = 1$

$CB12 = 1000$

$RB21 = 1$

$RB22 = 2$

$CB21 = 1$

$CB22 = 1$

Similarly, the user-supplied values to the *gs2init2* macro are

$MNRV1 = 1$

$MXRV1 = 1$

$MNCV1 = 1$

$MXCV1 = N$

$LFBDY1(1) = 1$

$RTBDY1(1) = N$

$UPBDY1(J) = 1$ for $1 \le J \le N$.

Termination in the event of the condition $C(J)>100$ being satisfied is accomplished via the "end of complex" macro ***cmplxend***. This macro sets variables that are internal to the *gs*2 macro monitor so that no new subscript pairs will be handed out . Then once a process completes a component sum and returns to the *gs*2 monitor, it enters into a windup processing stage of the problem. Furthermore, ***cmplxend*** will set the variable EXHST to 0, indicating that the possibility of exhausting all the tasks in the problem without finding one for which the sum C(J) exceeds 100 has been ruled out.

To deal with a sequence of such problems, we utilize the ***barrier*** macro in a manner virtually identical to the way the Overbeek-Lusk ***barrier*** macro is used in their tutorial (in their "Shell sort" program). This same method is used in our version of the "Shell sort"; this is one of the subsequent examples below. As noted there, when using our ***barrier*** macro to enable the repeated execution of the *gs*2 macro in a given program, it should be called by the macro call ***barrier***(1), and the statement after it should be labeled 3001.

6.3. GETDUPS

A and B are two vectors with the same number N of elements. N is assumed to be less than or equal to 1000. A and B are to be added to produce C, i.e.,

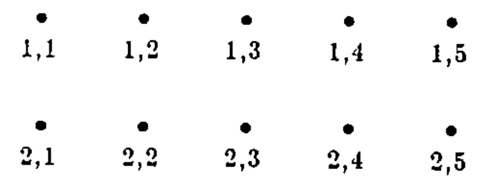
$C(J) = A(J)+B(J)$, for $1 \le J \le N$.

The N additions are all independent of each other. For each J, $2 \le J \le N$, $D(J)$ is set equal to 0 if $C(J)$ and $C(J-1)$ duplicate each other, i.e., if

$C(J) = C(J-1)$,

and is set equal to 1 otherwise. D(1) is set equal to 1.

The parallelization structure may be described as a simplex whose underlying region has 2 rows and N columns. More specifically, S is the 2xN integer rectangle depicted below for the representative case N = 5:



For each (1,J), T_{1J} is the task

add $A(J)+B(J)$ *produce* $C(J)$.

For (2,1), T_{21} is the task

set $D(1) = 1$.

For $2 \le J \le N$, T_{2J} is the task

set $D(J) = 0$ if $C(J) = C(J-1)$ and *set* $D(J) = 1$ *otherwise*.

The parallelization structure here has one constraint, i.e., the column constraint C with skew factor 0. This constraint requires that

$b_{2J} \ge e_{1J}$ for $1 \le J \le N$,

i.e., that for all J, T_{2J} cannot begin until after T_{1J} ends.

The fact that in a simplex each row is traversed from left to right, i.e., that

$b_{IJ} < b_{IK}$ *whenever* $J < K$,

enables the constraint to be formulated this easily. Let us see how the column constraint C, together with left to right row traversal work together here. Suppose, for example, that the next task to begin is T_{23}. Then we know that T_{22} has already begun; since the column constraint must have been satisfied for (2,2), T_{12} must have completed, i.e. C(2) has already been computed. Once the column constraint is satisfied for (2,3), then T_{13} has completed, i.e. C(3) has been computed. Since C(2) and C(3) have already been computed, T_{23}, which determines whether or not they are equal, may then begin.

Since there is a column constraint C in this structure, the *gs*2 macro should be called by *gs*2(1,*C*0,00,00,00).

User-supplied values to the *gs2var* macro are as follows:

$RB11 = 1$

$RB12 = 3$

$CB11 = 1$

$CB12 = 1000$.

User-supplied values to the *gs2init2* macro are as follows:

$MNRV1 = 1$

$MXRV1 = 2$

$MNCV1 = 1$

$MXCV1 = N$

$LFBDY1(1) = 1$

$LFBDY1(2) = 1$

$RTBDY1(1) = N$

$RTBDY1(2) = N$

$UPBDY1(J) = 1$ for $1 \leq J \leq N$.

A picture of the synchronization pattern for N = 5 is given by

$$\begin{tikzcd}
\bullet \arrow[d] & \bullet \arrow[d] & \bullet \arrow[d] & \bullet \arrow[d] & \bullet \arrow[d] \\
\bullet & \bullet & \bullet & \bullet & \bullet
\end{tikzcd}$$

It is instructive to examine now some variants of the parallelization structure described above in connection with the same problem.

First, suppose that all the D(J) are initialized to 1. Then task T_{21}, which is "set D(1) equal to 1," is redundant. We may then redefine T_{21} to be the "empty task" φ and write

$T_{21} = \varphi$.

Notice that the column constraint C remains in full force, so that T_{21} cannot begin execution until T_{11}, i.e.,

$C(1) = A(1)+B(1)$

completes.

Although T_{21} is the null task, we cannot completely remove it from the parallelization structure. If this were attempted by redefining LFBDY1(2) to be 2 (we assume that $N \geq 2$), then T_{22} would only be constrained to start after T_{12} ($C(2) = A(2)+B(2)$) ends. Thus it would be possible for T_{22} to start before T_{11} ended, i.e., before C(1) was computed to be A(1) + B(1). But this would thwart the purpose of T_{22}, which is to compare C(1) and C(2).

Note that with the initializaton of all D(J) to 1, we may simplify the task T_{2J} for $2 \leq J \leq N$ to

set $D(J) = 0$ if $C(J-1) = C(J)$.

There is no need to specify what happens otherwise, since D(J) is already equal to 1 when T_{2J} begins execution.

Thus, by modifying the parallelization structure used in this example, we have introduced the concept of a "null task." Further modifications will illustrate the concept of "skew factor" and of "jagged boundary."

Let us remove T_{21} (whether it be "set D(1) = 1" or the null task φ) and then shift the remaining tasks in the second row one unit to the left.

In other words, we define a new task function T^1 by the rules

$$T_{1J}^1 = T_1 \text{ for } 1 \le J \le N,$$

and

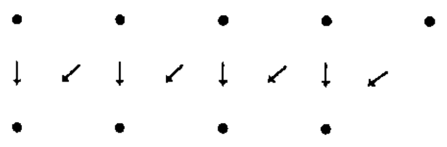
$$T_{2J}^1 = T_{2(J+1)} \text{ for } 1 \le J \le N-1.$$

Define the region S^1 to be

$$S^1 = \{ (1,J)\text{: } 1 \le J \le N \} \cup \{ (2,J)\text{: } 1 \le J \le N-1 \}.$$

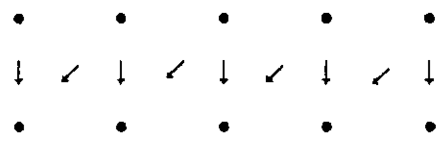
Then the simplex $(S^1, T^1 \cdot U^1)$ also models the "getdups" example, so long as U^1, which indicates the constraints on the simplex, indicates the column constraint C with skew factor 1.

S^1 now has a jagged right boundary; although RTBDY1(1) is still N, now RTBDY1(2) is $N-1$. The synchronization pattern may now be depicted (again for N = 5) as



The user-supplied values must also be modified. Now $CB12$ = 1001, since the skew factor of 1 must be taken into account.

An equivalent formulation would be to extend S^1 (restoring the straight right boundary) and to define T_{2N}^1 to be the null task φ. The skew factor is still 1, and CB12 is still N+1. But now once again the user-supplied value RFBDY1(2) is N. The synchronization pattern for N = 5 may now be drawn as



Since there is no N+1'st column in S^1, T_{2N}^1 is constrained only by T_{1N}^1.

6.4. GETADUP

This modification of example 2 provides an instructive, elementary example of a parallelization 2-complex.

As in example 2, A and B are two vectors with the same number N of elements. N is assumed to be less than or equal to 1000. A and B are to be added to produce C via

$$C(J) = A(J)+B(J), \text{ for } 1 \le J \le N.$$

The N component additions are all mutually independent.

We wish to determine if any successive C(J)'s duplicate each other, i.e., if there is some J such that

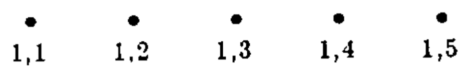
$$2 \le J \le N \text{ and } C(J-1) = C(J).$$

All we care about is whether there are any such duplicates. Therefore, if we find one duplicate pair, we can terminate the computation.

We initialize D(J) to be 1 for $1 \le J \le N$. For $2 \le J \le N$, D(J) is set equal to 0 if $C(J-1) = C(J)$. A certain bit (actually, a Fortran INTEGER variable named EXHST) is initialized to 1. If some D(J) is set to 0, then this bit is set to 0 and the computation for this problem is terminated in an orderly manner, including perhaps printing out the result that there is a duplicate. If no D(J) is set to 0, this bit remains at 1. Thus, after all the D(J) are determined, termination processing may include printing out the result that there are no duplicates.

The parallelization structure may be described as a 2-complex whose two simplexes are (S^1, T^1, U^1), which is simplex 1, and (S^2, T^2, U^2), which is simplex 2. As is always the case with our priority convention for 2-complexes, simplex 1 has higher priority than simplex 2. This implies that when a process executes the *gs*2 macro searching for a subscript pair ready to begin execution from among the current pair in simplex 1 and the current pair in simplex 2, it tilts its search in favor of simplex 1 by examining that current pair first.

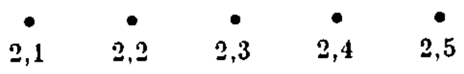
Here S^2 is the $1 \times N$ integer rectangle (drawn for N = 5)



and for each $(1,J)$ such that $1 \le J \le N$, T^2_{1J} is the task

"add A(J) to B(J) to produce C(J)."

S^1 is (for N = 5) the integer rectangle



T_{21}^{1} is the null task φ. For $2 \le J \le N$, T_{2J}^{1} is the task

if $C(J-1) = C(J)$, then set $D(J) = 0$, begin no new computational tasks and instead do windup processing, and clear a bit (EXHST) indicating that the computation has been terminated by the finding of a duplicate pair, rather than by performing all the T_{IJ}^{2} and finding no duplicates.

Since there are no internal constraints within the individual simplexes, U^1 and U^2 are empty. However (as is always the case with a 2-complex according to the definition here), there is a cross constraint. Simplex 2 column constrains simplex 1, i.e., XC21 holds. This means that

T_{2J}^{1} cannot begin until T_{1J}^{2} has ended,

when $1 \le J \le N$.

The synchronization pattern may be drawn as

$$\begin{tikzcd}
\bullet \arrow[d, "X"] & \bullet \arrow[d, "X"] & \bullet \arrow[d, "X"] & \bullet \arrow[d, "X"] & \bullet \arrow[d, "X"] \\
\bullet & \bullet & \bullet & \bullet & \bullet
\end{tikzcd}$$

Points in simplex 1 lie on the top line, while points in simplex 2 lie on the bottom line. As indicated earlier, simplex 1 points take priority over simplex 2 points.

The fact that computation can be terminated because of a condition being satisfied for some T_{2J}^{1} is not part of the synchronization pattern. It enters into the program through the use of the macro ***cmplxend***.

Note that this method of structuring the problem offers the advantage that if a duplicate pair is discovered early enough, many of the simplex 2 additions may not be done, because they are superfluous and their superfluity was discovered in time.

Since the only constraint is the cross-column constraint XC21, the *gs*2 macro call is given by *gs*2(2,0*X*,00,00,00).

User-supplied values for simplex 1 to the ***gs2var*** macro are as follows:

$RB11 = 2$

$RB12 = 3$

$CB11 = 1$

$CB12 = N$.

User-supplied values for simplex 2 to the ***gs2var*** macro are as follows:

$RB21 = 1$

$RB22 = 2$

$CB21 = 1$

$CB22 = N.$

User-supplied values for simplex 1 to the *gs2init2* macro are as follows:

$MNRV1 = 2$

$MXRV1 = 2$

$MNCV1 = 1$

$MXCV1 = N$

$LFBDY1(2) = 1$

$RTBDY1(2) = N$

$UPBDY1(J) = 2$ for $1 \leq J \leq N$

$LWBDY1(J) = 2$ for $1 \leq J \leq N.$

User-supplied values for simplex 2 to the *gs2init2* macro are as follows:

$MNRV2 = 1$

$MXRV2 = 1$

$MNCV2 = 1$

$MXCV2 = N$

$LFBDY2(1) = 1$

$RTBDY2(1) = N$

$UPBDY2(J) = 1$ for $1 \leq J \leq N$

$LWBDY2(J) = 1$ for $1 \leq J \leq N.$

The macro ***cmplxend*** is used to handle termination in the case when a duplicate is found, just as in example CHECKTWO above. In particular, it will then set the variable EXHST to 0.

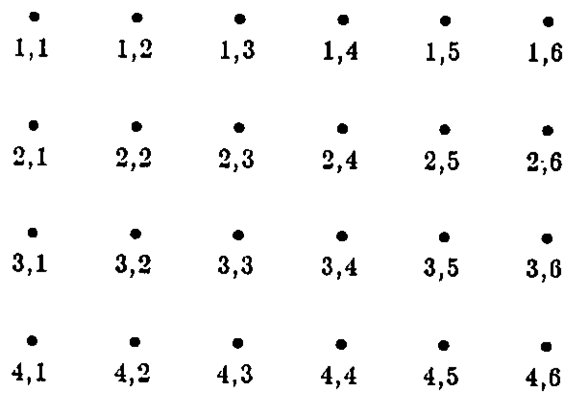
6.5. MATMULT

Let A and B be matrices, where A has M rows and N columns, and B has N rows and P columns. M, N, and P are all assumed to be less than or equal to 300. A and B are to be multiplied together by matrix multiplication to produce the MxP matrix C. If A(I,J), B(J,K) and C(I,K) denote, respectively, the (I,J)'th element of A, the (J,K)'th element of B, and the (I,K)'th element of C, then we can write the formula for matrix multiplication as

$$C(I,K) = \sum_{J=1}^{N} A(I,J) \times B(J,K)$$

for all I,K where $1 \le I \le M$ and $1 \le K \le P$.

The MP distinct computations of the C(I,K) are all independent. A substantial amount of the parallelization potential in this matrix multiplication may be obtained fairly simply. To do this, we use a simplex whose underlying region S has M rows and N columns. More specifically, S is the MxP integer rectangle depicted below for the case M = 4 and P = 6.



For each (I,K) in S, T_{IK} is the task

compute C(I,K) via the formula

$$C(I,K) = \sum_{J=1}^{N} A(I,J) \times B(J,K)$$

Since all the tasks T(I,K) are independent, this parallelization structure has no constraints; therefore, since no constraints need be indicated on the synchronization pattern, the picture of the synchronization pattern coincides with the above integer rectangle.

Also, since there are no constraints, the *gs*2 macro may be called via *gs*2(1,00,00,00,00).

User-supplied values to the ***gs2var*** macro are as follows:

$RB11 = 1$

$RB12 = 301$

$CB11 = 1$

$CB12 = 300$.

User supplied values to the *gs2init2* macro are as follows:

$MNRV1 = 1$

$MXRV1 = M$

$MNCV1 = 1$

$MXCV1 = P$

$LFBDY1(I) = 1$ for $1 \leq I \leq M$

$RTBDY1(I) = P$ for $1 \leq I \leq M$

$UPBDY1(J) = 1$ for $1 \leq J \leq P$.

This is a basic example of a two-dimensional parallelization structure whose underlying region is an integer rectangle with sides of arbitrary length and which has no constraints imposed.

6.6. SORT

Three Shell sorts are to be performed in this example: the first on a vector of 100 elements, the second on a vector of 1000 elements, and the third on a vector of 10000 elements. We are going to set, in the *gs2* context, the exact same Shell sort example dealt with in Overbeek and Lusk [2], on pages 13-20.

This example will illustrate, among other things, how in the *gs2* setting regions with jagged boundaries occur, how the internal row constraint R arises, how the (internal) row-constraint synchronization pattern is depicted, and how multiple instances of the same problem are handled within one program (e.g., three Shell sorts on arrays of different lengths within one program).

In connection with the last question, recall that the program structure utilized here will always have a main program, a subroutine SLAVE, and a subroutine WORK. The program will employ NPROCS processes. One of these processes will begin executing the main program. At some time during the execution of the main program, it will create the remaining NPROCS-1 processes, each of which will begin executing a copy of SLAVE. All subroutine SLAVE does is to call subroutine WORK. The main program will also call WORK. No parameters will be passed when the main program creates a copy of SLAVE.

If our program is to deal only with one instance of one problem, then no parameters are passed when WORK is called, either by the main program or by a copy of SLAVE.

However, if multiple instances of one problem are contemplated, then one parameter will be passed whenever WORK is called; it will be received in WORK by the dummy integer variable WHO. When the main program calls WORK, it will pass the value 0; but when a SLAVE calls WORK, it will pass the value 1. Thus a copy of WORK will be able to use the variable WHO to determine whether its immediate ancestor is the main program or a SLAVE and thus to where, after a particular instance of the problem has been completed, control should be transferred. The use of the variable WHO in this way is illustrated in the Overbeek-Lusk tutorial (in particular, see p. 20, note 4, as well as the preceding code). It should also be clear from an examination of the (more structured) version of the Shell sort presented here.

We will return later in this example to the question of dealing with the multiple instances of a Shell sort. But for now, let us turn to the question of dealing with one Shell sort.

We begin by giving, for completeness, a somewhat formal summary presentation of the Shell sort to be used here. If the reader would like a more intuitive exposition, he should consult the Overbeek-Lusk tutorial at pages 13-20.

First an overview of our presentation. To sort a sequence

$$x_{i_1}, x_{i_2}, \cdots, x_{i_K}$$

means to rearrange the values x_{i_k} while keeping the subscripts

$$i_1, i_2, \cdots, i_K$$

so that the rearranged sequence

$$y_{i_1}, y_{i_2}, \cdots, y_{i_K}$$

is monotone increasing, i.e.,

$$y_{i_1} \leq y_{i_2} \leq \cdots \leq y_{i_K}$$

We assume here, of course, that the sequence of integer indices is strictly monotone increasing, i.e., that

$$i_1 < i_2 < \cdots < i_K.$$

The "insertion to the left" (IL) sort sorts the x_{i_k} by first sorting the first (leftmost) two x_{i_k}, then the first three, then the first four, and so on, until all the x_{i_k} are sorted.

IL is the function composition of the IL_J (insert the J'th element to the left) sorts. For $2 \leq J \leq K$ IL_J sorts the first J x_{i_k} if the first $J-1$ x_{i_k} are already sorted.

In formula,

$$IL = IL_K \cdot IL_{K-1} \cdot \cdots \cdot IL_3 \cdot IL_2$$

where $\cdot$ is used to denote function composition. Thus to apply IL to a sequence, first apply IL_2, then IL_3, and so on until IL_K.

The action of the IL_J may be defined recursively. IL_2 sorts a sequence y_{i_1}, y_{i_2} by doing nothing if the two values are in the right order and otherwise interchanging the two values, producing a sorted sequence z_{i_1}, z_{i_2}.

IL_3 sorts a sequence

$$y_{i_1}, y_{i_2}, y_{i_3},$$

where it is assumed that the sequence y_{i_1}, y_{i_2} is already sorted, as follows: if y_{i_2} and y_{i_3} are in the right order, then IL_3 does nothing. Otherwise it interchanges the values of y_{i_2} and y_{i_3}, producing the output sequence

$$z_{i_1}, z_{i_2}, z_{i_3}.$$

Now IL_2 is applied to the sequence z_{i_1}, z_{i_2} to produce a sorted result

$$w_{i_1}, w_{i_2}, w_{i_3}$$

In general, IL_J sorts the sequence

$$y_{i_1}, y_{i_2}, \cdots, y_{i_J},$$

where it is assumed that the sequence

$$y_{i_1}, y_{i_2}, \cdots, y_{i_{J-1}}$$

is already sorted, as follows: if $y_{i_{J-1}}$ and y_{i_J} are in the right order, then IL_J does nothing. Otherwise it interchanges the values of $y_{i_{J-1}}$ and y_{i_J}, producing the output sequence

$$z_{i_1}, z_{i_2}, \cdots, z_{i_J}.$$

Now IL_{J-1} is applied to the subsequence

$$z_{i_1}, z_{i_2}, \cdots, z_{i_{J-1}}$$

to produce the sorted result

$$w_{i_1}, w_{i_2}, \cdots, w_{i_J}.$$

The Shell sort sorts a sequence of distinct integers

$$x_1, x_2, \cdots, x_N.$$

It takes as given a suitable decreasing sequence of integers

$$G_1 > G_2 > \cdots > G_M ,$$

where $1 = G_M$ and $G_1 < \frac{N}{3}$. The G_j are called "gaps," and G_j is called the j'th gap.

Within the framework of this example, the gaps G_j, together with the number of gaps M, will be defined by the relations

$$G_M = 1,$$

$$G_I = 3G_{I+1} + 1 \text{ for } 1 \le I \le M-1,$$

$$3G_1 + 1 < N,$$

and

$$3(3G_1 + 1) + 1 \ge N.$$

The Shell sort is done in M iterations, one for each gap G_j. The j'th iteration begins by decomposing its input sequence

$$z_1, z_2, \cdots , z_N$$

into G_j separate, disjoint subsequences, each of which is of the form

$$z_q, z_{q+G_j}, z_{q+2G_j}, \cdots , z_{q+HG_j},$$

where

$$1 \le q \le G_j,$$

and H is the largest integer such that

$$q + HG_j \le N.$$

(Note that successive indices in the above sequence of z's differ by G_j; thus the term "gap" for the G_j.)

The j'th iteration continues by applying IL to sort each of the G_j separate, disjoint subsequences. Then these sorted subsequences are reassembled into a sequence

$$w_1, w_2, \cdots , w_N.$$

This sequence is the output (result) of the j'th iteration and becomes the input sequence to the (j+1)'st iteration if $j \le M$. The result of the M'th iteration is the result (output) of the Shell sort and is completely sorted.

Now that the summary presentation of the Shell sort is done, we turn to setting up the relevant Fortran program and parallelization structure. Let A be an array with N elements, where N is assumed to be less than or equal to 10000.

$$A(1), A(2), \cdots, A(N)$$

is the sequence to be sorted by the Shell sort. The sort will be done in M iterations, with gaps

$$GAP(1), GAP(2), \cdots, GAP(M).$$

The gaps and M will be obtained by first constructing a sequence

$$XGAP(1), XGAP(2), \cdots, XGAP(K)$$

by setting

$$XGAP(1) = 1,$$

$$XGAP(L) = 3XGAP(L-1) + 1 \text{ for } L \geq 2$$

and letting K be the smallest integer such that

$$XGAP(K) \geq N$$

where N is the number of integers to be sorted. Then we set

$$M = K-2,$$

and

$$GAP(I) = XGAP(M+1-I) \text{ for } 1 \leq I \leq M.$$

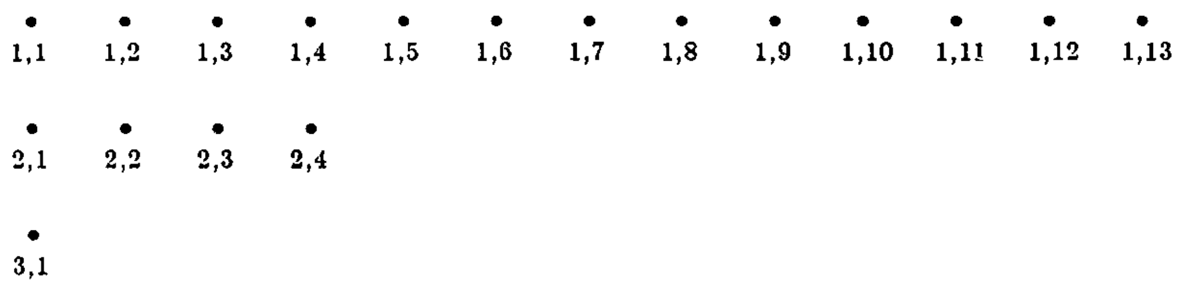
Thus the sequence of GAP's from 1 to M is simply the sequence of XGAP's from 1 to M, but taken in reverse order.

The parallelization structure is a simplex whose underlying region has M rows and N columns, where N is the number of elements to be sorted and M is the number of iterations to be performed. More specifically, S is the region in the Cartesian integer plane consisting of all integer pairs (I,J) such that

$$1 \leq I \leq M, \quad \text{and} \quad 1 \leq J \leq GAP(I).$$

A representative picture of S for the case when M = 3 is presented below. The relevant gaps then are

$$GAP(1) = 13, \quad GAP(2) = 4, \quad \text{and} \quad GAP(1) = 1.$$



Thus, the Shell sort exhibits a simplex with a jagged right boundary.

For each (I,J) in S, T_{IJ} is the task

apply the "insertion to the left sort" IL to the subsequence of A obtained by considering only those values A(I) whose indices I belong to the sequence

$$J,\ J+GAP(I),\ J+2GAP(I),\ \cdots,\ J+QGAP(I),$$

where Q is the largest integer such that

$$J + QGAP(I) \leq N.$$

For a fixed I, where $1 \leq I \leq M$, the computational tasks in the I'th row, i.e.

$$\{T_{IJ} : 1 \leq J \leq GAP(I)\},$$

provide the tasks for the I'th iteration of the Shell sort. The set of all computational tasks, i.e.,

$$\{T_{IJ} : 1 \leq I \leq M,\ 1 < J = \leq GAP(I)\},$$

provide the tasks for one full Shell sort.

It is natural here to impose the constraint that the (I-1)'st iteration be completed before the I'th iteration begins, i.e., to impose the row constraint R on our simplex. When R is the internal constraint of a simplex, all the parallelism to be obtained is contained within the individual rows of tasks; we are effectively doing a serial DO loop of individual rows. (Within each row all the tasks may be performed in parallel, subject only to the implied begin constraint that the tasks within a row begin in order of their column subscripts.)

The row constraint R is represented on the picture of the underlying region S by an arrow (with an "R") pointing down from the leftmost element of each (but the last) row to the leftmost point in the next row.

Thus the synchronization pattern for this example when M = 3 may be drawn as

$$\begin{tikzcd}
\bullet \arrow[d, "R"] & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\
\bullet \arrow[d, "R"] & \bullet & \bullet & \bullet \\
\bullet
\end{tikzcd}$$

Since the parallelization structure is that of a simplex with the row constraint R, the call to the gs2 macro is given by *gs2(1,00,R0,00,00)*.

User-supplied values to the *gs2var* macro are as follows:

$RB11 = 1$

$RB12 = 20$

$CB11 = 1$

$CB12 = 10000.$

User-supplied values to the *gs2init2* macro are as follows:

$MNRV1 = 1$

$MXRV1 = M$

$MNCV1 = 1$

$MXCV1 = GAP(1)$

$LFBDY1(I) = 1$ for $1 \le I \le M$

$RTBDY1(I) = GAP(I)$ for $1 \le I \le M$

$UPBDY1(J) = 1$ for $1 \le J \le GAP(I)$.

To supply these values, the user must first compute M and $GAP(1), \cdots, GAP(M)$.

As we remarked earlier, in this example we will do not only one Shell sort but three. Performing more than one Shell sort, i.e., executing the simplex more than once, involves a use of the ***barrier*** macro virtually identical to the use Overbeek and Lusk give their ***barrier*** macro (at p.19, line 189, of the tutorial) in their Shell sort program. Although there are certain differences between our ***barrier*** macro and the one used by Overbeek and Lusk, for purposes of this paper the reader need only note that the call to our ***barrier*** macro when it is used to support multiple complex executions is given by "*barrier(1)*" and that the Fortran statement which follows the call "*barrier(1)*" should be labeled 3000.

Each time a new Shell sort is done, the user-supplied values to the *gs2init2* macro that have changed need to be recomputed. In this example, where three Shell sorts are done on vectors with differing numbers of elements, we will find it simplest to recompute before each Shell sort all the user-supplied values to the *gs2init2* macro.

To accomplish this, we must also recompute M and $GAP(1), \cdots, GAP(M)$. (This recomputation could be simplified considerably if the order of the vectors to be sorted was reversed, i.e., if the 10000 element vector was sorted first, then the 1000 element vector, and last the 100 element vector. But it is more instructive to consider the case that requires more recomputation.)

Once the user-supplied variables to *gs2init2* are recomputed, then a call to that macro (in the form "*gs2init2(1,00,R0,00,00)*") will provide certain appropriate initializations needed for the particular problem (i.e., Shell sort) under consideration. Immediately following the "problem initialization" performed by *gs2init2* should be the call to subroutine WORK.

6.7. GRID

We now consider an example that is a discrete model of the Dirichlet problem for a cube in Euclidean three-dimensional space. In that problem, one is given a continuous function f defined on the boundary of the cube and must extend f to a continuous function g defined on the whole cube such that g is harmonic on the interior of the cube, i.e., g satisfies the Laplace equation

$$g_{xx} + g_{yy} = 0.$$

Here we are given a "discrete cube" (i.e., a three-dimensional cubic grid) and a function f defined on the boundary of the grid. We are to produce a three-dimensional Fortran array whose indices are the coordinates of the grid, whose values at grid boundary points are the same as the values of f, and which is in some sense a discrete analogue of a harmonic function.

A harmonic function satisfies the property that its value at a point is the average of its values in a neighborhood of that point. We obtain our array by using a discrete analogue of this averaging property.

We employ two 3-dimensional Fortran arrays A and B. The indices of A, as well as the indices of B consist of the points of the cubic grid under consideration. The values of A and B for indices which correspond to grid boundary points are always identical to the function values of f at those points. The values of A at interior grid points are initially set to 0.

A fixed number of iterations is set in advance. In an odd iteration, the interior values of B are computed by averaging neighboring values of A. In an even iteration, the roles of A and B are reversed. A column constraint, with skew factor 1, provides for the necessary synchronization.

After the given number of iterations has been completed, the last array to be computed (A if the number of iterations is even, B if it is odd) provides the desired discrete analogue of an approximate solution of the Dirichlet problem.

Now we focus in more sharply on the details. Let DIM be a positive integer such that $2 \leq DIM \leq 20$. Let A and B be 20x20x20 three-dimensional real arrays. For purposes of this problem we will focus on those elements A(I,J,K) of A and on those elements B(I,J,K) of B whose coordinates I, J, and K all are less than or equal to DIM. With this restriction in focus, A and B both represent real-valued functions defined on the three-dimensional integer cubic grid

$$D = [1,DIM]\times[1,DIM]\times[1,DIM].$$

The boundary of D, denoted by dD, is defined as the union of the 6 sets

$$[1]\times[1,DIM]\times[1,DIM]$$
$$[DIM]\times[1.YDIM]\times[1,ZDIM]$$
$$[1,DIM]\times[1]\times[1,DIM]$$
$$[1,DIM]\times[DIM]x[1,DIM]$$
$$[1,DIM]\times[1,DIM]x[1]$$
$$[1,DIM]\times[1,DIM]x[DIM].$$

The interior of D, denoted by **int** D , is defined as $D-dD$, i.e., the set of points of D that are not on the boundary of D. It can be represented as a Cartesian product by

$$\mathbf{int}\,D = [2,DIM-1]\times[2,DIM-1]\times[2,DIM-1].$$

We are given a function f defined on dD. We will exhibit a method of extending it to a function g defined on all of D.

Define, for all (I,J,K) in dD,

$$A(I,J,K) = f(I,J,K) \quad \text{and} \quad B(I,J,K) = f(I,J,K).$$

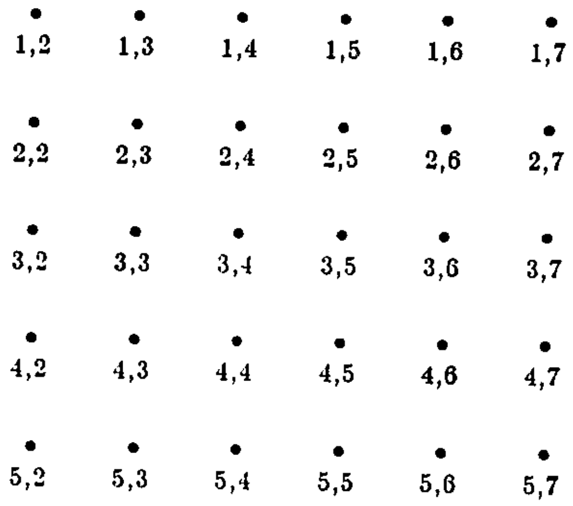
Throughout the computation the values of A(I,J,K) and B(I,J,K) for (I,J,K) in dD will remain unchanged.

Initialize

$$A(I,J,K) = 0 \; for \; all \; (I,J,K) \; in \; \mathbf{int}\,D.$$

A specific positive number M of iterations is given. Assume M to be less than or equal to 1000.

The parallelization structure may be described by a simplex whose underlying region S has M rows and DIM−2 columns. More specifically, S is the $M\times(DIM-2)$ integer rectangle drawn for the representative case M = 5 and DIM = 8 below.



T_{12} is the task

> compute all the B(2,J,K), where (2,J,K) is in the interior of D, by setting B(2,J,K) equal to the average of the A values at the six neighboring points of (2,J,K).

Those six neighboring points are

$$(1,J,K),\ (3,J,K),\ (2,J-1,K),\ (2,J+1,K),\ (2,J,K-1) \text{ and } (2,J,K+1).$$

T_{12} may be thought of as updating the values of the A function on the interior grid slice $I = 2$ by computing B values there, each of which is the average of the six neighboring A values.

More generally, for $2 \leq L \leq DIM-1$, T_{1L} is the task

> compute all the B(L,J,K), where (L,J,K) is in the interior of D, by setting B(L,J,K) equal to the average of the A values at the six neighboring points of (L,J,K).

T_{1L} can be thought of as updating the A values on the interior grid slice $I = L$ (here, I denotes an integer variable ranging over the first coordinate in three-dimensional space) by averaging, just as in the particular case L=2 described above.

We have now described the first row of tasks T_{1L} of the parallelization simplex. We will now define the remaining rows of tasks. If I is an odd integer, and $1 \leq I \leq M$, then whenever $2 \leq L \leq M-1$ T_{IL} has exactly the same definition as does T_{1L}. T_{IL} is the task compute all B(L,J,K), where (L,J,K) is in the interior of D, by setting B(L,J,K) equal to the average of the A values at the six neighboring points of (L,J,K).

T_{IL} can be thought of as updating the values of the A function on the interior grid slice $I = L$ by averaging, just the same as T_{1L}. These updated values are used as the new values of the B function.

If I is even, T_{IL} has the same definition as T_{1L}, except that the roles of A and B are reversed. Now T_{IL} is the task

compute all A(L,J,K), where (L,J,K) is in the interior of D, by setting A(L,J,K) equal to the average of the B values at the six neighboring points of (L,J,K).

For I even, T_{IL} can be thought of as updating the values of the B function on the interior grid slice $I = L$ by averaging. These updated values are used as the new values of the A function.

The simplex here has one constraint, namely, the column constraint C with skew factor 1. This constraint says that

$$b_{IJ} \geq e_{(I-1)J} \text{ whenever } 2 \leq I \leq M \text{ and } 2 \leq J \leq DIM-1,$$

and

$$b_{IJ} \geq e_{(I-1)(J+1)} \text{ whenever } 2 \leq I \leq M \text{ and } 2 \leq J \leq DIM-2.$$

To summarize, A starts off with the given values f on dD and the value 0 on int D. During the first iteration B (as it always does) assumes the given values f on dD. However, at each point of int D, B takes on the value obtained by averaging the six values of A at "neighbors" of that point. This iteration "updates" the values of A on D; the updated values are those of B on D.

During the second iteration the roles of A and B are reversed. As always, A assumes the values of f on dD. However, on int D, at each point A takes on the average of the six neighboring values of B. (These values of B are those obtained during the first iteration.) The column constraint C, with skew factor 1, ensures that a second iteration task T_{2J} will not begin until the first iteration tasks $T_{1(J-1)}$, T_{1J} and $T_{1(J+1)}$ have all ended. Once these tasks have ended, B has all those needed updated values produced by the first iteration which serve as inputs to T_{2J}.

Note that if T_{2J} is ready to begin, i.e., T_{2J} is the current task in the simplex and the constraints imposed on T_{2J} by the column constraint with skew factor 1 are satisfied, then T_{1J} and $T_{1(J+1)}$ have completed, by the definition of the column constraint with skew factor 1. But if $J>2$, then $T_{2(J-1)}$ must have already begun, since T_{2J} is the current task, and a row in a simplex is traversed in order of column subscripts. Thus $T_{1(J-1)}$ must also have ended, since its end was a prerequisite (via the column constraint) to the beginning of $T_{2(J-1)}$.

The odd iterations all are identical to the first, except that each time through the values of B are computed using the values of A obtained during the previous iteration. Similarly the even iterations are all identical to the second, with the roles of A and B interchanged from what they were with regard to odd iterations.

The L'th task of the N'th iteration computes the updated values on the slice of int D consisting of all points of int D whose first coordinate is L.

The values computed during the last (M'th) iteration are taken to be the final result. Thus, these are the values of A on D if M is even, and the values of B on D if M is odd.

The column constraint C, with skew factor 1, always ensures that the correct inputs to the task T_{IJ} are in place before that task can begin. Since the simplex has the column constraint C, the macro call to $gs2$ is exactly the same as that used in the *GETDUPS* example, namely, $gs2(1,C0,00,00,00)$. Note that the skew factor does not affect the parameters in the macro call.

User-supplied values to the *gs2var* macro are as follows:

$RB11 = 1$

$RB12 = 1001$

$CB11 = 2$

$CB12 = 20.$

User-supplied values to the *gs2init2* macro are as follows:

$MNRV1 = 1$

$MXRV1 = M$

$MNCV1 = 2$

$MXCV1 = DIM-1$

$LFBDY1(I) = 2$ for $1 \leq I \leq M$

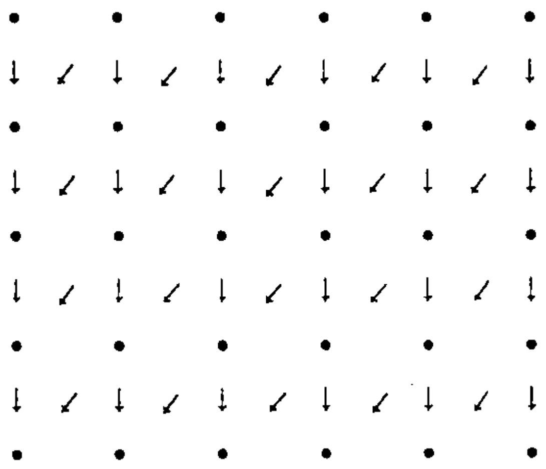
$RTBDY1(I) = DIM-1$ for $1 \leq I \leq M$

$UPBDY1(J) = 1$ for $2 \leq J \leq DIM-1$

$SKW1 = 1.$

Note that the skew factor 1 is introduced here as a user-supplied value via the variable SKW1.

A picture of the synchronization pattern is given by



6.8. QR

The formulation of this problem is essentially taken from Overbeek and Lusk [1, p. 26].

We study a parallelization structure involved in performing the QR factorization of a matrix, more specifically, in performing Householder's algorithm. It is not necessary to understand what the algorithm does; it is necessary to understand the following synchronization requirements.

1. The first step in performing Householder's algorithm on an NxN matrix is to "create the reflection for column 1." This reflection can then be applied to all remaining columns.
2. A reflection will be created for each column J. The reflection for column J can be created only after
   a. all reflections for columns L, where $L<J$, have been created, and
   b. for all $L<J$, the reflection for column L has been applied to the J'th column.
3. The reflection for column K can be applied to all columns J, where $K<J$. However, for a given column K, these "applies" must take place in order. More specifically, the application of the reflection for column K to column J, if $J>1$, cannot begin until the reflection for column K has been applied to column $J-1$.

We can reformulate these synchronization requirements as follows:

(A) For each J, $1 \le J \le N$, there is a task $C(J)$ "create the reflection for column J."

(B) For each J,K, $1 \leq J < K \leq N$, there is a task $A(J,K)$ "apply the reflection for column J to column K."

(C) $C(J)$ cannot begin until all $A(L,J)$ have ended, where $1 \leq L < J$.

(D) $A(J,K)$ cannot begin until $A(J-1,K)$ has ended, if $J \geq 2$.

And of course, since a reflection cannot be applied until it has been created, we have

(E) Whenever $1 \leq J < K \leq N$, $A(J,K)$ cannot begin until $C(J)$ has ended.

Note that the requirement

"$C(J)$ cannot begin until $C(L)$ has ended, for $L < J$"

need not be included, since it follows from (C) and (E).

Note further that (C) may be replaced by the weaker condition

(C') Whenever $J \geq 2$, $C(J)$ cannot begin until $A(J-1,J)$ has ended.

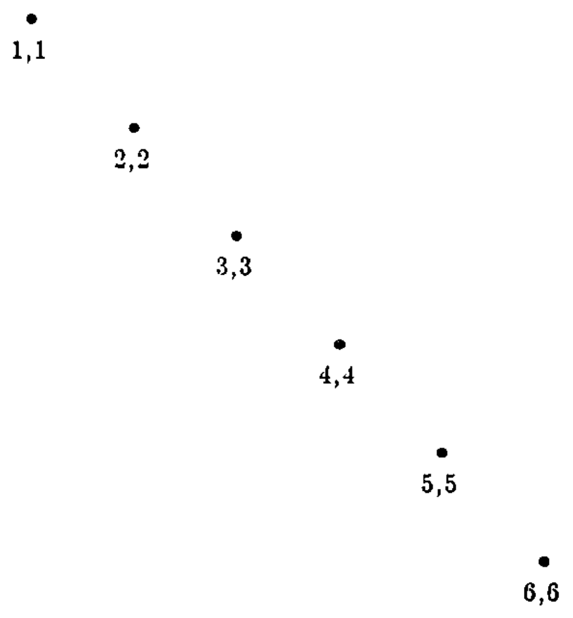
This is because (C) follows from (C') and (D).

Thus we have a "minimal" set of synchronization requirements (A), (B), (C'), (D), and (E) upon which we can base our parallelization structure.

The parallelization structure may be described by a 2-complex

$((S^1, T^1, U^1), (S^2, T^2, U^2), V)$.

The underlying region S^1 of simplex 1 is the set of all points in the integer Cartesian plane that have coordinates (J,J), where $1 \leq J \leq N$. In other words, S^1 is the integer line segment from (1,1) to (N,N). A picture of S^1 when $N = 6$ is drawn below.



For each (J,J) in S^1, T^1_{JJ} is the task $C(J)$, "create the reflection for column J."

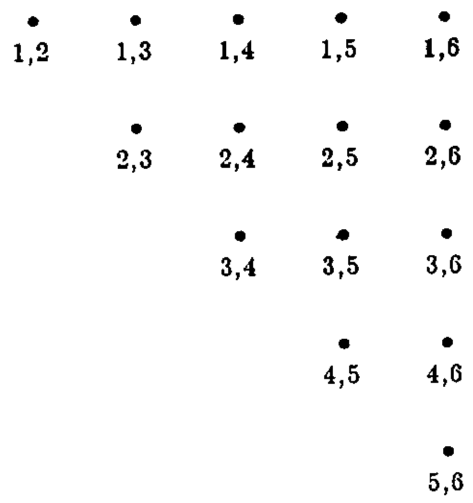
The underlying region S^2 of simplex 2 is the set of all points (J,K) in the integer Cartesian plane such that $1 \leq J < K \leq N$. In other words, S^2 is the integer triangle whose "boundaries" are the three integer line segments

$J+1 = K$, for $1 \leq J \leq N-1$,

$J = 1$, for $2 \leq K \leq N$,

$K = N$, for $1 \leq J \leq N-1$.

S^2 (for $N = 6$) is the integer triangle drawn below.



In accordance with the convention in force here, the J coordinate (the first coordinate) will be on the vertical axis, with lower points corresponding to larger values of J, and the K coordinate (the second coordinate) will be on the horizontal axis, with points to the right corresponding to larger values of K.

For each (J,K) in S^2, T^2_{JK} is the task $A(J,K)$ "apply the reflection for column J to column K.

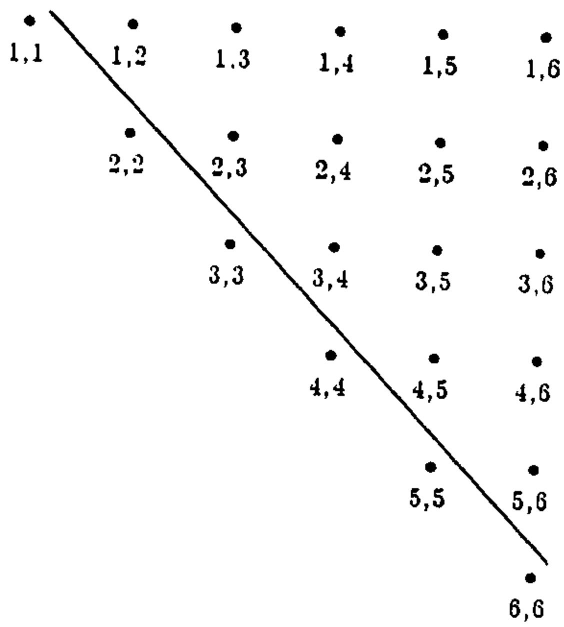
The underlying region S of the 2-complex, which is the union of S^1 and S^2, is therefore the integer triangle whose "boundaries" are the three integer line segments

$J = K$, for $1 \leq J \leq N$

$J = 1$, for $1 \leq K \leq N$

$K = N$, for $1 \leq J \leq N$.

S (for $N = 6$) is the integer triangle drawn below.



The line separates the two underlying regions S^1 and S^2. Points in S^1 lie below the line, while points in S^2 lie above. .

To find the constraints associated with the 2-complex, we interpret the synchronization requirements (C'), (D) and (E) given above.

Condition (C') requires that for $J \geq 2$, $C(J)$ cannot begin until $A(J-1,J)$ has ended. This translates into requiring that T^1_{JJ} cannot begin until $T^2_{(J-1)J}$ has ended, i.e., that

$$b_{JJ} \geq e_{(J-1)J} \text{ for } J \geq 2.$$

But this states that simplex 2 constrains simplex 1, i.e., that the cross-column constraint XC21 holds.

Condition (D) requires that for $J \geq 2$, $A(J,K)$ cannot begin until $A(J,K-1)$ has ended, i.e., that T^2_{JK} cannot begin until $T^2_{J(K-1)}$ has ended. This is precisely the condition

$$b_{JK} \geq e_{J(K-1)}$$

when $J \geq 2$, and (J,K) and $(J,K-1)$ are in S^2. But this states that simplex 2 is column constrained, i.e., that the constraint C2 (with skew factor 0) holds.

Finally, Condition (E) requires that whenever $1 \le J < K \le N$, $A(J,K)$ cannot begin until $C(J)$ has ended, i.e. that T^2_{JK} cannot begin until T^1_{JJ} has ended. This is precisely the condition

$$b_{JK} \ge e_{JJ},$$

when $1 \le J < K \le N$.

But this is equivalent to the condition that simplex 1 cross row constrains simplex 2, i.e., that the constraint XR12 holds.

Therefore, the parallel structure used here is the 2-complex

$$((S^1, T^1, U^1), (S^2, T^2, U^2), V),$$

where S^1, S^2, T^1, and T^2 are as described above. U^1, the set of internal constraints on simplex 1, is empty. U^2, the set of internal constraints on simplex 2, consists of the column constraint C2 with skew factor 0. V, the set of cross constraints, contains the cross-column constraint XC21 and the cross-row constraint XR12.

Therefore, for this example, the *gs*2 macro call is *gs*2(2,0X,00,C0,0X).

User-supplied values for simplex 1 to the *gs2var* macro are as follows (we assume that N, the size of the square matrix, is less than or equal to 300):

$RB11 = 1$

$RB12 = 301$

$CB11 = 1$

$CB12 = 300$.

User-supplied values for simplex 2 to the *gs2var* macro are as follows:

$RB21 = 1$

$RB22 = 300$

$CB21 = 2$

$CB22 = 300$.

User-supplied values for simplex 1 to the *gs2init2* macro are as follows:

$MNRV1 = 1$

$MXRV1 = N$

$MNCV1 = 1$

$MXCV1 = N$

$LFBDY1(I) = I$ for $1 \le I \le N$

$RTBDY1(I) = I$ for $1 \le I \le N$

$UPBDY1(J) = J$ for $1 \le J \le N$

$LWBDY1(J) = J$ for $1 \le J \le N$.

User-supplied values for simplex 2 to the *gs2init2* macro are as follows:

$MNRV2 = 1$

$MXRV2 = N-1$

$MNCV2 = 2$

$MXCV2 = N$

$LFBDY2(I) = I+1$ for $1 \le I \le N-1$

$RTBDY2(I) = N$ for $1 \le I \le N-1$

$UPBDY2(J) = 1$ for $2 \le J \le N$.

$LWBDY2(J) = J-1$ for $2 \le J \le N$.

Note: The formulation of this problem in Overbeek and Lusk [1, p. 26] deals with the performance of the Householder algorithm for several matrices, rather than for just one. This use of the *gs*2 macro for multiple problems is handled by the use of the ***barrier*** macro, called in the form ***barrier***(1), just as in the SORT and CHECKTWO problems dealt with earlier. Prior to the time a new Householder algorithm computation is begun, and the problem initializer *gs2init*2 is executed, the user-supplied values that have changed must be resupplied by the user. Thus, for example, if the dimension of the matrix is changed, then *MXRV*1, *MXRV*2, *MXRV*2, *MXCV*2, as well as all the underlying region boundaries (*LFBDY*1, *LFBDY*2, *RTBDY*1, *RTBDY*2, *LWBDY*1, *LWBDY*2, *UPBDY*1, and *UPBDY*2) should be recomputed before the relevant use of *gs2init*2 in the main program.

As is always the case (by convention) with a 2-complex, simplex 1 holds priority over simplex 2. This implies that if the current task in simplex 1 and the current task in simplex 2 are both ready to begin a process begins executing *gs*2 searching for a task to perform, then that process will select the current task in simplex 1.

Thus, here the "creates" take priority over the "applies." This scheme has the advantage that once a "create" is completed, it can remove a direct row end constraint on a whole row of "applies" (the XR12 constraint), while a completed "apply" can remove a direct column end contraint only on the task directly beneath it (the C2 or XC21 constraint). (Strictly speaking, one might say that a completed "apply" removes, on the average, a constraint on half a row of "applies," with reference to the row below that of the completed "apply.") Of course, if one wished, one could give the "applies" priority by switching the indices on the two simplexes of the complex.

Let us examine another perspective on the advantage obtained by giving "creates" priority over "applies." As the applies T^2_{JK} complete for K suitably near N, they free up processes that cannot begin a new task until the "create" $T^1_{(J+1)(J+1)}$ completes. To avoid having these processes have significant periods of unnecessary inactivity, the "create" $T^1_{(J+1)(J+1)}$ should begin before tasks T^2_{JK} for K near N whenever possible. Assigning priority to "creates" accomplishes this.

The synchronization pattern given by this complex is drawn below (for the representative case $N = 6$).

$$\begin{tikzcd}
\bullet \arrow[r] & \bullet \arrow[d, "X"] & \bullet \arrow[d] & \bullet \arrow[d] & \bullet \arrow[d] & \bullet \arrow[d] \\
 & \bullet \arrow[r] & \bullet \arrow[d, "X"] & \bullet \arrow[d] & \bullet \arrow[d] & \bullet \arrow[d] \\
 & & \bullet \arrow[r] & \bullet \arrow[d, "X"] & \bullet \arrow[d] & \bullet \arrow[d] \\
 & & & \bullet \arrow[r] & \bullet \arrow[d, "X"] & \bullet \arrow[d] \\
 & & & & \bullet \arrow[r] & \bullet \arrow[d, "X"] \\
 & & & & & \bullet
\end{tikzcd}$$

The horizontal arrows represent the XR12 constraint. The (plain) vertical arrows represent the C2 constraint. The vertical arrows with an "X" represent the XC21 constraint.

References

1. J. Clausing, R. Hagstrom, E. Lusk, and R. A. Overbeek, "A Technique for Achieving Portability Among Multiprocessors: Implementation on the Lemur," *Parallel Computing*, 1985. (to appear)
2. Harry F. Jordan, *Parallel Programming on the HEP Multiple Instruction Stream Computer*, August 20, 1981.
3. E. Lusk and R. A. Overbeek, "Use of monitors in FORTRAN: a tutorial on the barrier, self-scheduling DO-loop, and askfor monitors," ANL-84-51, Argonne National Laboratory, July 1984.
4. Ewing L. Lusk and Ross A. Overbeek, "Implementation of Monitors with Macros: A Programming Aid for the HEP and Other Parallel Processors," Technical Report ANL-83-97, Argonne National Laboratory, Argonne, Illinois, December 1983.
5. E. L. Lusk and R. A. Overbeek, "Use of Monitors in FORTRAN: a Tutorial on the Barrier, Self-Scheduling Do-Loop, and Askfor Monitors," in *Parallel MIMD Computation: The HEP Supercomputer and its Applications*, ed. J. S. Kowalik, The MIT Press, 1985.

Appendix A
The ADDTWO Example

```
define(RB11,1)
define(RB12,2)
define(CB11,1)
define(CB12,1000)
define(RB21,1)
define(RB22,2)
define(CB21,1)
define(CB22,1)

      PROGRAM ADDTWO
      newproc(SLAVE)
*
* COMMON AREA VARIABLES
*
      INTEGER A(1000), B(1000), C(1000)
      INTEGER NPROCS, N
      COMMON /MAINC/ A, B, C, N, NPROCS
*
************************************************************************
*
*   DECLARE THE VARIABLES AND COMMON TO SUPPORT THE MONITOR
*
************************************************************************
*
      gs2var
*
      INTEGER I,TS,TE
*
************************************************************************
*
*   INITIALIZE THE SELF-SCHEDULING DO-LOOP MONITOR
*
************************************************************************
*
      gs2init1(1,00,00,00,00)
*
************************************************************************
*
*   READ IN THE NUMBER OF PROCESSES TO RUN IN PARALLEL
*
************************************************************************
*
      READ (5,10) NPROCS
10    FORMAT(I4)
      WRITE(6,20) NPROCS
20    FORMAT(' NPROCS = ',I4)
*
************************************************************************
```

```
*
*   READ IN THE TWO INPUT VECTORS
*
***************************************************************************
*
      READ (5,10) N
      READ (5,10) (A(I), I = 1,N)
      READ (5,10) (B(I), I = 1,N)
*
      MNRV1 = 1
      MXRV1 = 1
      MNCV1 = 1
      MXCV1 = N
      LFBDY1(1) = 1
      RTBDY1(1) = N
      DO 17 I=1,N
      UPBDY1(I) = 1
17    CONTINUE
      gs2init2(1,00,00,00,00)
*
***************************************************************************
*
*   CREATE THE SLAVE PROCESSES
*
***************************************************************************
*
      clock(TS)
*
      DO 30 I=1,NPROCS-1
          create(SLAVE)
30    CONTINUE
*
      CALL WORK
*
      clock(TE)
      TE = TE - TS
      WRITE(6,40) TE
40    FORMAT(' TOTAL TIME = ',I6)
*
      WRITE (6,10) (C(I), I = 1,N)
      STOP
      END
*
***************************************************************************
*
*   THE SLAVE PROCESSES JUST CALL THE WORK SUBROUTINE TO ADD UP
*   ELEMENTS UNTIL THE END OF THE VECTOR IS REACHED.  THE PROCESSES
*   THEN EXIT (WHICH IS ASSUMED TO DESTROY THEM).
*
***************************************************************************
*
      SUBROUTINE SLAVE
*
      CALL WORK
      RETURN
      END
*
**********************************************************************
*
* THE WORK SUBROUTINE JUST CAUSES A PROCESS TO GRAB AVAILABLE
* SUBSCRIPTS UNTIL ALL OF THE WORK HAS BEEN COMPLETED.  AT THAT
* POINT THE SUBROUTINE EXITS.  NOTE THAT IF THERE IS A SINGLE
* PROCESS (I.E., NO SLAVES), THE ALGORITHM STILL WORKS JUST
* FINE.
*
**********************************************************************
*
      SUBROUTINE WORK
*
* COMMON AREA VARIABLES
*
      INTEGER A(1000), B(1000), C(1000)
      INTEGER NPROCS, N
      COMMON /MAINC/ A, B, C, N, NPROCS
*
      gs2var
*
      INTEGER I
*
10    CONTINUE
*
**********************************************************************
*
*     CLAIM THE NEXT AVAILABLE SUBSCRIPT (RETURNED IN I)
*
**********************************************************************
      gs2(1,00,00,00,00)
*
1000  CONTINUE
*
      C(J) = A(J) + B(J)
      GO TO 10
*
3000  CONTINUE
      RETURN
      END
```

Appendix B
The CHECKTWO Example

```
       define(RB11,1)
       define(RB12,2)
       define(CB11,1)
       define(CB12,1000)
       define(RB21,1)
       define(RB22,2)
       define(CB21,1)
       define(CB22,1)
******************************************************************************

      PROGRAM CHKTWO
*     newproc(SLAVE)
* COMMON AREA VARIABLES
*
*
      INTEGER A(1000), B(1000), C(1000)
      INTEGER NPROCS,N
      INTEGER I
      COMMON /MAINC/ A, B, C, N, NPROCS
*
      gs2var
*
      gs2init1(1,00,00,00,00)
*
      READ (5,10) NPROCS
  10  FORMAT(I4)
      WRITE(6,20) NPROCS
20    FORMAT('NPROCS = ',I4)
*
*
*
*****************************************************************************
*
*   READ IN THE TWO INPUT VECTORS
*
****************************************************************************
*
      READ(5,11) N
      READ(5,11)(A(I), I = 1,N)
      READ(5,11)(B(I), I = 1,N)
*
      DO 42 JJ = 1,N
      C(JJ) = -1
42    CONTINUE
*
          MNRV1 = 1
          MXRV1 = 1
          MNCV1 = 1
          MXCV1 = N
*
          DO 43 II = MNRV1,MXRV1
          LFBDY1(II) = 1
          RTBDY1(II) = N
43        CONTINUE
*
          DO 44 JJ = MNCV1,MXCV1
          UPBDY1(JJ) = 1
          LWBDY1(JJ) = 1
44        CONTINUE
*
          gs2init2(1,00,00,00,00)
*
      DO 30 I=1,NPROCS-1
          create(SLAVE)
30    CONTINUE
*
*
          CALL WORK
*
      IF(EXHST .EQ. 1) THEN
         WRITE(6,77)
      ELSE
         WRITE(6,78)
      ENDIF
77    FORMAT(' ', ' THERE IS NO VALUE GREATER THAN 100')
78    FORMAT(' ', ' THERE IS A VALUE GREATER THAN 100')
      WRITE(6,40)
40    FORMAT(' THE VALUES IN THE C VECTOR ARE AS FOLLOWS: ')
      WRITE(6,12)(C(I), I=1,N)
11    FORMAT(I4)
12    FORMAT(' ',I4)
      STOP
      END
*
************************************************************************
*
      SUBROUTINE SLAVE
*
      CALL WORK
      RETURN
      END
*
*
***********************************************************************
*
      SUBROUTINE WORK
*
      INTEGER I
      INTEGER A(1000), B(1000), C(1000)
      INTEGER NPROCS, N
      COMMON /MAINC/ A, B, C,  N, NPROCS
*
```

```
*
      gs2var
*
10    CONTINUE
*
      gs2(1,00,00,00,00)
*
1000  CONTINUE
*
      C(J) = A(J) + B(J)
      IF(C(J).GT.100) THEN
            cmplxend(1,00,00,00,00)
      ENDIF
*
      end1(1,00,00,00,00)
      GO TO 10
*
3000  CONTINUE
      RETURN
      END
```

Appendix C
The GETDUPS Example

```
define(RB11,1)
define(RB12,3)
define(CB11,1)
define(CB12,1000)
define(RB21,1)
define(RB22,2)
define(CB21,1)
define(CB22,1)

***********************************************************************
* C = A + B.  THEN D IS CREATED AS A VECTOR IN WHICH EACH ELEMENT
* IS SET AS FOLLOWS: D(I) = 0 IFF (C(I) = C(I-1).  THOSE ELEMENTS
* OF D WHICH ARE NOT SET TO 0 ARE SET TO 1.  D(1) IS ALWAYS SET
* TO 1.
*
***********************************************************************

      PROGRAM GETDUP
      newproc(SLAVE)
*
* COMMON AREA VARIABLES
*
      INTEGER A(1000), B(1000), C(1000), D(1000)
      INTEGER NPROCS, N
      COMMON /MAINC/ A, B, C, D, N, NPROCS
*
      gs2var
*
      INTEGER I
*
*
***********************************************************************
*
*   INITIALIZE THE MONITORS
*
***********************************************************************
*
      gs2init1(1,C0,00,00,00)
*
      READ (5,10) NPROCS
10    FORMAT(I4)
      WRITE(6,20) NPROCS
20    FORMAT(' NPROCS = ',I4)
*
**********************************************************************
*
*   READ IN THE TWO INPUT VECTORS
*
```

```
**********************************************************************
*
      READ (5,10) N
      READ (5,10) (A(I), I = 1,N)
      READ (5,10) (B(I), I = 1,N)
*
      MNRV1 = 1
      MXRV1 = 2
      MNCV1 = 1
      MXCV1 = N
      LFBDY1(1) = 1
      LFBDY1(2) = 1
      RTBDY1(1) = N
      RTBDY1(2) = N

      DO 25 I=1,N
      UPBDY1(I) = 1
25    CONTINUE

      gs2init2(1,C0,00,00,00)

      DO 30 I=1,NPROCS-1
          create(SLAVE)
30    CONTINUE
*
      CALL WORK
*
      WRITE (6,40)
40    FORMAT(' THE VALUES IN THE C VECTOR ARE AS FOLLOWS:')
      WRITE (6,10) (C(I), I = 1,N)
      WRITE (6,50)
50    FORMAT(' THE VALUES IN THE D VECTOR ARE AS FOLLOWS:')
      WRITE (6,10) (D(I), I = 1,N)
      STOP
      END
*
**********************************************************************
*
*   THE SLAVE PROCESSES JUST CALL THE WORK SUBROUTINE TO ADD UP
*   ELEMENTS UNTIL THE END OF THE VECTOR IS REACHED.  THE PROCESSES
*   THEN EXIT (WHICH IS ASSUMED TO DESTROY THEM).
*
**********************************************************************
*
      SUBROUTINE SLAVE
*
      CALL WORK
      RETURN
      END
*
**********************************************************************
*
* THE WORK SUBROUTINE PERFORMS A 2-STAGE COMPUTATION.  FIRST,
* C = A + B IS COMPUTED. THE SECOND STAGE CALCULATES D.
```

```
*
**************************************************************************
*
      SUBROUTINE WORK
*
* COMMON AREA VARIABLES
*
      INTEGER A(1000), B(1000), C(1000), D(1000)
      INTEGER NPROCS, N
      COMMON /MAINC/ A, B, C, D, N, NPROCS
*
      gs2var
*
      INTEGER I
*
10    CONTINUE

      gs2(1,C0,00,00,00)
1000  CONTINUE
*
      IF (I .EQ. 1) THEN
      C(J) = A(J) + B(J)
      ELSE
      IF (J .GE. 2) THEN
          IF (C(J) .EQ. C(J-1)) THEN
          D(J) = 0
          ELSE
          D(J) = 1
          ENDIF
      ELSE
          D(1) = 1
      ENDIF
      ENDIF

      end1(1,C0,00,00,00)
      GO TO 10
*
3000  CONTINUE
*
      RETURN
      END
```

Appendix D
The GETADUP Example

```
       define(RB11,2)
       define(RB12,3)
       define(CB11,1)
       define(CB12,1000)
       define(RB21,1)
       define(RB22,2)
       define(CB21,1)
       define(CB22,1000)
*************************************************************************

      PROGRAM GETADP
*     newproc(SLAVE)
* COMMON AREA VARIABLES
*
*
      INTEGER A(1000), B(1000), C(1000), D(1000)
      INTEGER NPROCS,N
      INTEGER I
      COMMON /MAINC/ A, B, C,D, N, NPROCS
*
      gs2var
*
      gs2init1(2,0X,00,00,00)
*
      READ (5,10) NPROCS
  10  FORMAT(I4)
      WRITE(6,20) NPROCS
20    FORMAT('NPROCS = ',I4)
*
*
*
*************************************************************************
*
*   READ IN THE TWO INPUT VECTORS
*
*************************************************************************
*
      READ(5,11) N
      READ(5,11)(A(I), I = 1,N)
      READ(5,11)(B(I), I = 1,N)
*
      DO 42 JJ = 1,N
      C(JJ) = -1
      D(JJ) = -1
42    CONTINUE
*
          MNRV1 = 2
          MXRV1 = 2
          MNCV1 = 1
          MXCV1 = N
*
          DO 43 II = MNRV1,MXRV1
          LFBDY1(II) = 1
          RTBDY1(II) = N
43        CONTINUE
*
          DO 44 JJ = MNCV1,MXCV1
          UPBDY1(JJ) = 2
          LWBDY1(JJ) = 2
44        CONTINUE
*
          MNRV2 = 1
          MXRV2 = 1
          MNCV2 = 1
          MXCV2 = N
*
          DO 45 II = MNRV2,MXRV2
          LFBDY2(II) = 1
          RTBDY2(II) = N
45        CONTINUE
*
          DO 46 JJ = MNCV2,MXCV2
          UPBDY2(JJ) = 1
          LWBDY2(JJ) = 1
46        CONTINUE
*
          gs2init2(2,0X,00,00,00)
*
      DO 30 I=1,NPROCS-1
          create(SLAVE)
30    CONTINUE
*
*
          CALL WORK
*
      IF(EXHST .EQ. 1) THEN
         WRITE(6,77)
      ELSE
         WRITE(6,78)
      ENDIF
77    FORMAT(' ', ' THERE ARE NO DUPLICATES')
78    FORMAT(' ', ' THERE IS A DUPLICATE')
      WRITE(6,40)
40    FORMAT(' THE VALUES IN THE C VECTOR ARE AS FOLLOWS: ')
      WRITE(6,12)(C(I), I=1,N)
      WRITE(6,50)
50    FORMAT(' THE VALUES IN THE D VECTOR ARE AS FOLLOWS: ')
      WRITE(6,12)(D(I), I=1,N)
11    FORMAT(I4)
12    FORMAT(' ',I4)
      STOP
      END
```

```
*
**********************************************************************
*
      SUBROUTINE SLAVE
*
      CALL WORK
      RETURN
      END
*
*
**********************************************************************
*
      SUBROUTINE WORK
*
      INTEGER I
      INTEGER A(1000), B(1000), C(1000), D(1000)
      INTEGER NPROCS, N
      COMMON /MAINC/ A, B, C, D, N, NPROCS
*
*
      gs2var
*
10    CONTINUE
*
      gs2(2,0X,00,00,00)
*
1000  CONTINUE
*
      IF(J.GT.1) THEN
         IF(C(J).EQ.C(J-1)) THEN
            D(J) = 0
            cmplxend(2,0X,00,00,00)
         ENDIF
      ENDIF
*
      end1(2,0X,00,00,00)
      GO TO 10
*
2000  CONTINUE
      C(J) = A(J) + B(J)
      end2(2,0X,00,00,00)
      GO TO 10
3000  CONTINUE
      RETURN
      END
```

Appendix E
The MATMULT Example

```
define(RB11,1)
define(RB12,301)
define(CB11,1)
define(CB12,300)
define(RB21,1)
define(RB22,2)
define(CB21,1)
define(CB22,1)

**********************************************************************
*
* THIS PROGRAM READS IN TWO MATRICIES AND COMPUTES THEIR PRODUCT.
*
**********************************************************************

      PROGRAM MATMUL
      newproc(SLAVE)
*
* COMMON AREA VARIABLES
*
      INTEGER A(20,20), B(20,20), C(20,20)
      INTEGER NPROCS, AI, AJ, BJ
      COMMON /MAINC/ A, B, C, NPROCS, AI, AJ, BJ
*
      gs2var
*
*
**********************************************************************
*
*   INITIALIZE THE MONITOR
*
**********************************************************************
*
      gs2init1(1,00,00,00,00)
*
      READ (5,10) NPROCS
10    FORMAT(I4)
      WRITE(6,20) NPROCS
20    FORMAT(' NPROCS = ',I4)
*
*********************************************************************
*
*   READ IN THE TWO INPUT MATRICIES
*
**********************************************************************
*
      READ (5,10) AI
      READ (5,10) AJ
      READ (5,10) BJ

      DO 2 I = 1,AI
      DO 1 J = 1,AJ
          READ (5,10) A(I,J)
1         CONTINUE
2     CONTINUE

      DO 4 I = 1,AJ
      DO 3 J = 1,BJ
          READ (5,10) B(I,J)
3         CONTINUE
4     CONTINUE

*
      MNRV1 = 1
      MXRV1 = AI
      MNCV1 = 1
      MXCV1 = BJ

      DO 5 I = 1,AI
      LFBDY1(I) = 1
5     CONTINUE

      DO 6 I = 1,AI
      RTBDY1(I) = BJ
6     CONTINUE

      DO 7 I = 1,BJ
      UPBDY1(I) = 1
7     CONTINUE

      DO 8 I = 1,BJ
      LWBDY1(I) = AI
8     CONTINUE

      gs2init2(1,00,00,00,00)

      DO 30 I=1,NPROCS-1
          create(SLAVE)
30    CONTINUE
*
      CALL WORK
*
      WRITE (6,40)
40    FORMAT(' THE VALUES IN C ARE AS FOLLOWS:')
      DO 41 I = 1,AI
      DO 42 J = 1,BJ
          WRITE(6,43) I,J,C(I,J)
42        CONTINUE
41    CONTINUE
43    FORMAT('     ',I2,' ',I2,' ',I8)
      STOP
      END
```

```
*
************************************************************************
*
*   THE SLAVE PROCESSES JUST CALL THE WORK SUBROUTINE
*   WHERE THEY CLAIM TASKS TO WORK ON.
*
************************************************************************
*
      SUBROUTINE SLAVE
*
      CALL WORK
      RETURN
      END
*
************************************************************************
*
* THE WORK SUBROUTINE CONTAINS THE CODE TO CLAIM A TASK,
* PERFORM THE TASK, AND GO BACK TO GET ANOTHER TASK TO WORK ON.
*
************************************************************************
*
      SUBROUTINE WORK
*
* COMMON AREA VARIABLES
*
      INTEGER A(20,20), B(20,20), C(20,20)
      INTEGER NPROCS, AI, AJ, BJ
      COMMON /MAINC/ A, B, C, NPROCS, AI, AJ, BJ
*
      gs2var
*
      INTEGER K
*
10    CONTINUE

      gs2(1,00,00,00,00)
1000  CONTINUE

*
      C(I,J) = 0
      DO 1001 K = 1,AJ
      C(I,J) = C(I,J) + (A(I,K) * B(K,J))
1001  CONTINUE

      end1(1,00,00,00,00)
      GO TO 10
*
3000  CONTINUE
*
      RETURN
      END
```

Appendix F
The SORT Example

```
       define(RB11,1)
       define(RB12,20)
       define(CB11,1)
       define(CB12,10000)
       define(RB21,1)
       define(RB22,2)
       define(CB21,1)
       define(CB22,1)
************************************************************************
*
* THIS PROGRAM DEMONSTRATES THE "BARRIER" AND "SELF-SCHEDULING DO-LOOP"
* SYNCHRONIZATION PRIMITIVES.  IT FILLS IN A VECTOR (A) WITH VALUES IN
* DESCENDING ORDER.  THEN IT USES A SHELL SORT (SEE KNUTH'S 3RD VOLUME
* ON SORTING AND SEARCHING ALGORITHMS) TO SORT THE VALUES INTO
* ASCENDING ORDER.  TIMES ARE ACQUIRED FOR TABLE SIZES OF 100, 1000, AND
* 10000.
*
************************************************************************

      PROGRAM SRTPGM
*     newproc(SLAVE)
* COMMON AREA VARIABLES
*
*
      INTEGER A(10000)
      INTEGER NPROCS, M, N, GAP(20), NDONE, XGAP(20)
      INTEGER I,J,K
      COMMON /MAINC/ GAP, A, M, N, NPROCS, NDONE
*
      gs2var
*
      gs2init1(1,00,R0,00,00)
*
*
*
************************************************************************
*
*   INITIALIZE THE BARRIER AND SELF-SCHEDULING DO-LOOP MONITORS
*
************************************************************************
*
*
*     NDONE = 0
      READ (5,10) NPROCS
  10  FORMAT(I4)
      WRITE(6,20) NPROCS
20    FORMAT('NPROCS = ',I4)
*
```

```
      DO 30 I=1,NPROCS-1
          create(SLAVE)
30    CONTINUE
*
*
*
*************************************************************************
*
*   READ IN THE NUMBER OF PROCESSES TO RUN IN PARALLEL
*
**************************************************************************
*
*
*************************************************************************
*
*   THE MAIN LOGIC JUST FILLS IN THE TABLE AND SORTS IT.
*   TIMINGS ARE TAKEN FOR TABLES OF 100, 1000, AND 10000.
*
**************************************************************************
*
      N = 10
      DO 50 I=1,3
          N = 10 * N
*
          DO 35 K = 1,N
          A(K) = (N-K) + 1
35        CONTINUE

          XGAP(1) = 1
          DO 36 K = 2,20
          XGAP(K) = 3*XGAP(K-1) + 1
          IF(XGAP(K).GE.N) GO TO 37
36        CONTINUE
37        M = K-2
*
          DO 38 K = 1,M
          GAP(K) = XGAP(M+1-K)
38        CONTINUE
*
          MNRV1 = 1
          MXRV1 = M
          MNCV1 = 1
          MXCV1 = GAP(1)
*
          DO 45 II = MNRV1,MXRV1
          LFBDY1(II) = 1
          RTBDY1(II) = GAP(II)
45        CONTINUE
*
          DO 46 JJ = MNCV1,MXCV1
          UPBDY1(JJ) = 1
46        CONTINUE
*
          gs2init2(1,00,R0,00,00)
```

```
*
      WRITE(6,776) (A(LL), LL = 1,N)
776   FORMAT(' ',10I5)
          CALL LOOP(0)
*
      WRITE(6,777) (A(KK), KK = 1,N)
777   FORMAT(' ',10I5)

50    CONTINUE
*
*********************************************************************
*
*   ONE LAST CALL TO LOOP IS REQUIRED TO FREE THE OTHER PROCESSES
*   FROM THE BARRIER (SO THEY CAN EXIT).
*
*********************************************************************
*
*     NDONE = 1
*     CALL LOOP(0)
      STOP
      END
*
*
*********************************************************************
*
*   THE SLAVE PROCESSES JUST HANG ON THE BARRIER IN THE "LOOP"
*   AND HELP WHEN A TABLE IS TO BE SORTED.
*
*********************************************************************
*
      SUBROUTINE SLAVE
*
      CALL LOOP(1)
      RETURN
      END
*
*
*********************************************************************
*
*   THE SORT ROUTINE IS EXECUTED BY THE MASTER PROCESS.  IT JUST
*   CALCULATES THE RADIX FOR EACH PASS OF THE SHELL SORT, AND JOINS
*   THE SLAVE PROCESSES WHEN WORKING ON EACH PASS.
*   THE RADIX VALUES ARE HT, ... H2, H1: H1 IS 1; HI IS (3*H(I-1) + 1);
*   H(T+2) >= N.  SEE KNUTH FOR ARGUMENTS IN FAVOR OF THESE VALUES.
*
*********************************************************************
*
*   THE LOOP ROUTINE IS THE CODE REQUIRED TO COORDINATE THE NPROCS
*   PROCESSES AS THEY EXECUTE ONE PASS OF A SHELL SORT.  NOTE THE
*   BARRIER AT THE TOP, WHICH IS USED TO CAUSE THE PROCESSES TO
*   WAIT FOR THE VECTOR TO BE SET UP AND THE INCREMENT CHOSEN.
*   THEN A SELF-SCHEDULING DO-LOOP IS USED TO ALLOCATE SUBSCRIPTS.
*   NOTE THAT THE MASTER PARTICIPATES IN THIS LOGIC, SO THE PROGRAM
*   CAN BE RUN WITH NPROCS SET TO 1.
```

```
*
************************************************************************
*
*
      SUBROUTINE LOOP(WHO)
      INTEGER WHO
*
      INTEGER I, J, K, L
      INTEGER A(10000), T
      INTEGER NPROCS,M, N, GAP(20), NDONE
      COMMON /MAINC/ GAP, A, M, N, NPROCS, NDONE
*
*
      gs2var
*
10    CONTINUE
      barrier(1)
3001  IF(NDONE.EQ.1) GO TO 4000
*
      gs2(1,00,R0,00,00)
*
1000  DO 30 K = J+GAP(I) , N, GAP(I)
      DO 40 L = K, J+GAP(I), -GAP(I)
      IF(A(L-GAP(I)).GT.A(L)) THEN
         T = A(L-GAP(I))
         A(L-GAP(I)) = A(L)
         A(L) = T
      ELSE
         GO TO 30
      ENDIF
40    CONTINUE
30    CONTINUE
*
      end1(1,00,R0,00,00)
*
      GO TO 10
*
3000  CONTINUE
      IF(WHO.EQ.1) GO TO 10
4000  CONTINUE
      RETURN
      END
```

Appendix G
The GRID Example

```
define(RB11,1)
define(RB12,1001)
define(CB11,2)
define(CB12,20)
define(RB21,1)
define(RB22,2)
define(CB21,1)
define(CB22,1)


      PROGRAM GRID
      newproc(SLAVE)
*
*
* THE FUNCTION OF THIS PROGRAM IS TO APPROXIMATE THE VALUE OF A
* FUNCTION 'PHI' SATISFYING BOUNDARY CONDITIONS
*
*            PHI(X,Y,Z) = X * X - Y * Y + X * Y * Z
*
* FOR (X,Y,Z) ON THE BOUNDARY OF THE GRID.  THE VALUE AT AN INTERIOR
* POINT IS APPROXIMATED AS THE AVERAVE VALUE OF THE NEIGHBORING
* POINTS.
*
* COMMON AREA VARIABLES
*
      REAL    PHI,A(20,20,20),B(20,20,20)
      INTEGER N,NPROCS,XDIM,YDIM,ZDIM
      COMMON /POOL/ A,B,N,NPROCS,XDIM,YDIM,ZDIM
*
* DECLARE THE VARIABLES AND COMMON TO SUPPORT THE MONITOR
*
      gs2var
*
* DECLARE THE WORKING VARIABLES
*
      INTEGER I,J,K,X,Y,Z,IS,IE,IT
*
*
* GET THE DIMENSIONS OF THE GRID
*
      READ (5,20) XDIM,YDIM,ZDIM
20    FORMAT(3I4)
      WRITE(6,21) XDIM,YDIM,ZDIM
21    FORMAT(' XDIM=',I4,'  YDIM=',I4,'  ZDIM=',I4)
*
* GET THE NUMBER OF ITERATIONS TO PERFORM
*
      READ (5,80) N
```

```
80     FORMAT(I4)
*
       READ (5,100) NPROCS
100    FORMAT(I4)
       WRITE(6,101) N,NPROCS
101    FORMAT(' N=',I4,'  NPROCS=',I4)
*
       gs2init1(1,C0,00,00,00)
       MNRV1 = 1
       MXRV1 = N
       MNCV1 = 2
       MXCV1 = XDIM - 1
       DO 102 I=1,N
       LFBDY1(I) = 2
       RTBDY1(I) = XDIM - 1
102    CONTINUE
*
       DO 103 I=2,XDIM-1
       UPBDY1(I) = 1
103    CONTINUE
*
       SKW1 = 1
*
       gs2init2(1,C0,00,00,00)
*
* INITIALIZE THE INTERIOR OF THE GRID TO ZERO
*
       DO 110 I=2,XDIM-1
           DO 120 J=2,YDIM-1
               DO 130 K=2,ZDIM-1
                   A(I,J,K) = 0
130            CONTINUE
120        CONTINUE
110    CONTINUE
*
* INITIALIZE THE BOUNDARY OF THE GRID
*
*      THE FACES X = 1 AND X = XDIM
*
       DO 140 J=1,YDIM
           DO 150 K=1,ZDIM
               A(1,J,K) = PHI(1,J,K)
           B(1,J,K) = A(1,J,K)
           A(XDIM,J,K) = PHI(XDIM,J,K)
           B(XDIM,J,K) = A(XDIM,J,K)
150        CONTINUE
140    CONTINUE
*
*      THE FACES Y = 1 AND Y = YDIM
*
       DO 160 I=1,XDIM
           DO 170 K=1,ZDIM
           A(I,1,K) = PHI(I,1,K)
           B(I,1,K) = A(I,1,K)
```

```
          A(I,YDIM,K) = PHI(I,YDIM,K)
          B(I,YDIM,K) = A(I,YDIM,K)
170       CONTINUE
160   CONTINUE
*
*     THE FACES Z = 1 AND Z = ZDIM
*
      DO 180 I=1,XDIM
          DO 190 J=1,YDIM
          A(I,J,1) = PHI(I,J,1)
          B(I,J,1) = A(I,J,1)
          A(I,J,ZDIM) = PHI(I,J,ZDIM)
          B(I,J,ZDIM) = A(I,J,ZDIM)
190       CONTINUE
180   CONTINUE
*
      clock(IS)
*
* CREATE THE SLAVE PROCESSES
*
      DO 220 I = 1,NPROCS-1
          create(SLAVE)
220   CONTINUE
      CALL WORK
      clock(IE)
      IT = IE - IS
      WRITE(6,221) IT
221   FORMAT(' TOTAL TIME = ',I12)
*
      IF (MOD(N,2) .EQ. 0) THEN
          CALL PRCUBE(A)
      ELSE
          CALL PRCUBE(B)
      ENDIF
*
      STOP
      END
*
**********************************************************************
*
*             P H I   F U N C T I O N
*
**********************************************************************
*
      FUNCTION PHI(X,Y,Z)
      INTEGER X,Y,Z

      PHI = (X * X) - (Y * Y) + (Z * Z)
*     PHI = 1
      RETURN
      END

*
**********************************************************************
```

```
*
*       .     S L A V E   P R O C E S S E S
*
**********************************************************************
*
      SUBROUTINE SLAVE
*
      CALL WORK
      RETURN
      END
*
**********************************************************************
*
*             W O R K   S U B R O U T I N E
*
**********************************************************************
*
      SUBROUTINE WORK
*
*
* COMMON AREA VARIABLES
*
      REAL    A(20,20,20),B(20,20,20)
      INTEGER N,NPROCS,XDIM,YDIM,ZDIM
      COMMON /POOL/ A,B,N,NPROCS,XDIM,YDIM,ZDIM
      gs2var
      INTEGER I,J
*
*
* DECLARE THE VARIABLES AND COMMON TO SUPPORT THE MONITOR
*
*
10    CONTINUE
      gs2(1,C0,00,00,00)
*
1000  CONTINUE
          IF (MOD(I,2) .EQ. 1) THEN
              CALL COMP(A,B,J)
          ELSE
              CALL COMP(B,A,J)
          ENDIF
          end1(1,C0,00,00,00)
*
          GO TO 10
*
3000  CONTINUE
      RETURN
      END
*
**********************************************************************
*
*             C O M P U T E   S U B O U T I N E
*
**********************************************************************
```

```
*
      SUBROUTINE COMP(P,Q,X)
*
      REAL    P(20,20,20),Q(20,20,20)
      INTEGER X
*
* COMMON AREA VARIABLES
*
      REAL    A(20,20,20),B(20,20,20)
      INTEGER N,NPROCS,XDIM,YDIM,ZDIM
      COMMON /POOL/ A,B,N,NPROCS,XDIM,YDIM,ZDIM
*
      INTEGER I,J,K
*
      DO 10 J=2,YDIM-1
          DO 20 K=2,ZDIM-1
              Q(X,J,K) = (P(X-1,J,K) + P(X+1,J,K) +
     -                    P(X,J-1,K) + P(X,J+1,K) +
     -                    P(X,J,K-1) + P(X,J,K+1)) / 6.0
20        CONTINUE
10    CONTINUE
*
      RETURN
      END
*
**********************************************************************
*
*             S U B R O U T I N E   P R C U B E
*
**********************************************************************
*
      SUBROUTINE PRCUBE(M)
      REAL M(20,20,20)
*
*
*
* COMMON AREA VARIABLES
*
      REAL    A(20,20,20),B(20,20,20)
      INTEGER N,NPROCS,XDIM,YDIM,ZDIM
      COMMON /POOL/ A,B,N,NPROCS,XDIM,YDIM,ZDIM
*
      INTEGER I,J,K
*
      DO 10 I=1,XDIM
      DO 20 J=1,YDIM
          DO 30 K=1,ZDIM
          WRITE (6,40) I,J,K,M(I,J,K)
40        FORMAT(' X= ',I4,' Y= ',I4,' Z= ',I4,' VALUE ',F10.5)
30        CONTINUE
20    CONTINUE
10    CONTINUE
      RETURN
      END
```

Appendix H
The QR-Factorization Example

```
define(RB11,1)
define(RB12,301)
define(CB11,1)
define(CB12,300)
define(RB21,1)
define(RB22,300)
define(CB21,2)
define(CB22,300)

*
***************************************************************
*
* T H E   M A I N   L O G I C
*
***************************************************************
*
    PROGRAM QRFAC
    newproc(QSLAVE)
    REAL A(301,300),AA(301,300),B(301)
    INTEGER WSIZE,NPROCS, M, N, I, NDONE
    COMMON /MAINC/ A, B, N, M,  NPROCS, NDONE
*
*
     gs2var
*
     gs2init1(2,0X,00,C0,0X)
*
     NDONE = 0
* N O W   C R E A T E   T H E   W O R K E R S
*
    READ (5,1111) NPROCS
1111  FORMAT(I4)
    WRITE(6,1112) NPROCS
1112  FORMAT(' NPROCS=',I4)

    DO 600 I = 1, NPROCS - 1
      create(QSLAVE)
600   CONTINUE
*
*
C
    WRITE(6,40)
  40 FORMAT('  QRFAX DECOMPOSITION TIMING')
    DO 200 N = 10,50,10
    MNRV1 = 1
    MXRV1 = N
    MNCV1 = 1
    MXCV1 = N
    MNRV2 = 1
    MXRV2 = N-1
    MNCV2 = 2
    MXCV2 = N
    DO 555 II = 1,N
      LFBDY1(II) = II
      RTBDY1(II) = II
555 CONTINUE

    DO 556 JJ = 1,N
      UPBDY1(JJ) = JJ
      LWBDY1(JJ) = JJ
556 CONTINUE

    DO 557 II = 1,N-1
      LFBDY2(II) = II + 1
      RTBDY2(II) = N
557 CONTINUE

    DO 558 JJ = 2,N
      UPBDY2(JJ) = 1
      LWBDY2(JJ) = JJ - 1
558 CONTINUE

    gs2init2(2,0X,00,C0,0X)

      DO  20 J = 1,N
        DO 10 I = J,N
          AA(I,J) = -I*J
          AA(J,I) = 2*AA(I,J)
 10       CONTINUE
        AA(J,J) = 0.0
 20     CONTINUE
      WRITE(6,50)N
 50   FORMAT(/' ORDER IS ',I5/)
      DO 70 J = 1,N
        DO 60 I = 1,N
          A(I,J) = AA(I,J)
 60       CONTINUE
 70     CONTINUE
*
      DO 103 J = 1,N
        DO 102 I = 1,N
          A(I,J) = AA(I,J)
102       CONTINUE
103     CONTINUE
      WSIZE = 3
      clock(I)
      T1 = I
*
      M = N
      CALL WORK(0)
*
      clock(I)
      T2 = I - T 1
      WRITE(6,110) T2
    IF( N .LE. 50 ) WRITE(6,1000) (B(I),I = 1,N)
 1000 FORMAT(5X,E12.5)
  110    FORMAT(' MONITOR VERSION   TIME = ',E12.3)
C
      DO 113 J = 1,N
        DO 112 I = 1,N
          A(I,J) = AA(I,J)
  112     CONTINUE
  113   CONTINUE
  200 CONTINUE
    NDONE = 1
    CALL WORK(0)
12350 CONTINUE
    STOP
    END
*
* T H E   W O R K   S U B R O U T I N E
*
    SUBROUTINE WORK(FLAG)
    INTEGER FLAG
*
    REAL A(301,300),B(301)
    INTEGER NPROCS,  M, N, NDONE
    COMMON /MAINC/ A, B, N, M,  NPROCS, NDONE
*
*
     gs2var

     INTEGER I

     INTEGER L
*
*
* DECLARATIONS FOR CREF AND APREF
*
    REAL ZERO,TAU
    INTEGER NK,KM1
    REAL ENORM
    REAL THETA
    DATA ZERO/0.0/
*
*
*****************************************************************
*
5     CONTINUE
     barrier(1)


3001  IF(NDONE .EQ. 1) GO TO 4000
10    CONTINUE
     gs2(2,0X,00,C0,0X)
*
```

```
* N IS THE NUMBER OF COLUMNS IN THE MATRIX
*
* K IS SET TO THE COLUMN UPON WHICH A REFLECTION IS TO BE
*   CREATED OR APPLIED
*
* L IS MEANINGFUL ONLY WITH AN RC OF 1 (APPLY A REFLECTION).
*   IT THEN GIVES THE REFLECTION NUMBER TO APPLY
*
***************************************************************
*
*          CREATE THE REFLECTOR FOR THE  K-TH COLUMN
*
***************************************************************
C
1000  CONTINUE
         KM1 = I - 1
         NK = N - I + 1
C
C          NOW COMPUTE AND STORE THE K-TH REFLECTOR
C
         TAU = ENORM(NK,A(I,I))
         TAU = SIGN(TAU,A(I,I))
         B(I) = -TAU
         A(I,I) = A(I,I) + TAU
***************************************************************
*
*          NOW SIGNAL THAT THE REFLECTION HAS BEEN CREATED
*
***************************************************************
*
         end1(2,0X,00,C0,0X)
*
*          NOW GET THE NEXT TASK
*
         GO TO 10
***************************************************************
*
*          APPLY THE NEXT REFLECTION (THE L-TH)
*          TO THE K-TH COLUMN
*
***************************************************************
2000  CONTINUE
         THETA = ZERO
         DO 50 L = I,M
           THETA = THETA + A(L,J)*A(L,I)
50        CONTINUE
         THETA = THETA/(B(I)*A(I,I))
         DO 60 L = I,M
           A(L,J) = A(L,J) + THETA*A(L,I)
60        CONTINUE
*
*
         end2(2,0X,00,C0,0X)
*
```

```
          GO TO 10
*
*
3000  CONTINUE
     IF (FLAG .EQ. 1) GO TO 5
4000  CONTINUE
     RETURN
     END
*
************************************************************
*
*  Q S L A V E
*
************************************************************
*
    SUBROUTINE QSLAVE
    REAL A(301,300),B(301)
    INTEGER NPROCS,  M, N, NDONE
    COMMON /MAINC/ A, B, N, M,  NPROCS, NDONE
*
*
*
     gs2var
*
    CALL WORK(1)
    RETURN
    END
    REAL FUNCTION ENORM(N,X)
    INTEGER N
    REAL X(N)
C     **********
C
C     FUNCTION ENORM
C
C     GIVEN AN N-VECTOR X, THIS FUNCTION CALCULATES THE
C     EUCLIDEAN NORM OF X.
C
C     THE EUCLIDEAN NORM IS COMPUTED BY ACCUMULATING THE SUM OF
C     SQUARES IN THREE DIFFERENT SUMS. THE SUMS OF SQUARES FOR THE
C     SMALL AND LARGE COMPONENTS ARE SCALED SO THAT NO OVERFLOWS
C     OCCUR. NON-DESTRUCTIVE UNDERFLOWS ARE PERMITTED. UNDERFLOWS
C     AND OVERFLOWS DO NOT OCCUR IN THE COMPUTATION OF THE UNSCALED
C     SUM OF SQUARES FOR THE INTERMEDIATE COMPONENTS.
C     THE DEFINITIONS OF SMALL, INTERMEDIATE AND LARGE COMPONENTS
C     DEPEND ON TWO CONSTANTS, RDWARF AND RGIANT. THE MAIN
C     RESTRICTIONS ON THESE CONSTANTS ARE THAT RDWARF**2 NOT
C     UNDERFLOW AND RGIANT**2 NOT OVERFLOW. THE CONSTANTS
C     GIVEN HERE ARE SUITABLE FOR EVERY KNOWN COMPUTER.
C
C     THE FUNCTION STATEMENT IS
C
C       REAL FUNCTION ENORM(N,X)
C
C     WHERE
```

```
C
C     N IS A POSITIVE INTEGER INPUT VARIABLE.
C
C     X IS AN INPUT ARRAY OF LENGTH N.
C
C   SUBPROGRAMS CALLED
C
C     FORTRAN-SUPPLIED ... ABS,SQRT
C
C   ARGONNE NATIONAL LABORATORY. MINPACK PROJECT. MARCH 1980.
C   BURTON S. GARBOW, KENNETH E. HILLSTROM, JORGE J. MORE
C
C   **********
      INTEGER I
      REAL AGIANT,FLOATN,ONE,RDWARF,RGIANT,S1,S2,S3,XABS,
     *           X1MAX,X3MAX,ZERO
      DATA ONE,ZERO,RDWARF,RGIANT /1.0E0,0.0E0,.294E-38,.17E39/
      S1 = ZERO
      S2 = ZERO
      S3 = ZERO
      X1MAX = ZERO
      X3MAX = ZERO
      FLOATN = N
      AGIANT = RGIANT/FLOATN
      DO 90 I = 1, N
        XABS = ABS(X(I))
        IF (XABS .GT. RDWARF .AND. XABS .LT. AGIANT) GO TO 70
          IF (XABS .LE. RDWARF) GO TO 30
C
C           SUM FOR LARGE COMPONENTS.
C
            IF (XABS .LE. X1MAX) GO TO 10
              S1 = ONE + S1*(X1MAX/XABS)**2
              X1MAX = XABS
              GO TO 20
   10       CONTINUE
              S1 = S1 + (XABS/X1MAX)**2
   20       CONTINUE
            GO TO 60
   30     CONTINUE
C
C           SUM FOR SMALL COMPONENTS.
C
            IF (XABS .LE. X3MAX) GO TO 40
              S3 = ONE + S3*(X3MAX/XABS)**2
              X3MAX = XABS
              GO TO 50
   40       CONTINUE
              IF (XABS .NE. ZERO) S3 = S3 + (XABS/X3MAX)**2
   50       CONTINUE
   60     CONTINUE
          GO TO 80
   70   CONTINUE
C
```

```
C         SUM FOR INTERMEDIATE COMPONENTS.
C
          S2 = S2 + XABS**2
  80   CONTINUE
  90   CONTINUE
C
C    CALCULATION OF NORM.
C
      IF (S1 .EQ. ZERO) GO TO 100
        ENORM = X1MAX*SQRT(S1+(S2/X1MAX)/X1MAX)
        GO TO 130
 100 CONTINUE
        IF (S2 .EQ. ZERO) GO TO 110
          IF (S2 .GE. X3MAX)
     *       ENORM = SQRT(S2*(ONE+(X3MAX/S2)*(X3MAX*S3)))
          IF (S2 .LT. X3MAX)
     *       ENORM = SQRT(X3MAX*((S2/X3MAX)+(X3MAX*S3)))
          GO TO 120
 110    CONTINUE
          ENORM = X3MAX*SQRT(S3)
 120    CONTINUE
 130 CONTINUE
      RETURN
C
C    LAST CARD OF FUNCTION ENORM.
C
      END
```

Distribution for ANL-85-70

Internal:

K. L. Kliewer
A. B. Krisciunas
P. C. Messina
R. A. Overbeek (40)
G. W. Pieper
D. M. Pool
T. M. Woods (2)

ANL Patent Department
ANL Contract File
ANL Libraries
TIS Files (6)

External:

DOE-TIC, for distribution per UC-32 (167)
Manager, Chicago Operations Office, DOE
Mathematics and Computer Science Division Review Committee:
  J. L. Bona, U. Chicago
  T. L. Brown, U. of Illinois, Urbana
  S. Gerhart, MCC, Austin, TX
  G. H. Golub, Stanford U.
  W. C. Lynch, Xerox Corp., Palo Alto
  J. A. Nohel, U. of Wisconsin, Madison
  M. F. Wheeler, Rice U.
D. Austin, ER-DOE
J. Greenberg, ER-DOE
G. Michael, LLL
B. W. Glickfeld, Northern Illinois University (20)