A Graphics Tool to Aid in the Generation of Parallel FORTRAN Programs

CONF-8909231--1

DE90 001429

Orlie Brewer, Jack Dongarra, and Danny Sorensen

Mathematics and Computer Science Division

Argonne National Laboratory

Argonne, Illinois 60439-4801

ABSTRACT

This paper describes a graphics tool called BUILD that can be used to help automate the process of writing parallel FORTRAN programs for the SCHEDULE package. The user can interactively build an execution graph that describes his algorithm and then have the tool generate the necessary calls to the SCHEDULE package. We describe the tool and its use and then we present some examples that have been built using the tool.

1. Introduction

In developing software, the initial definitions and specifications are often done graphically, using such things as flow charts or dependency graphs. Usually one can grasp the overall structure of the problem far more easily from these graphical representations than from words and numbers. Unfortunately, of course, the computer cannot. These charts and graphs must eventually be translated to a computer language in order to implement the particular algorithm on a computer.

†Work supported in part by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U. S. Department of Energy, under Contract W-31-109-Eng-38. Typeset on January 31, 1989.

However, in some cases, the translation is straightforward and can easily be automated, allowing the computer to generate the necessary code from the graphical description of the problem. This is the case with the SCHEDULE package. The SCHEDULE package is a library of FORTRAN subroutines, which have been implemented on various parallel processors, that allow the application programmer to write parallel programs in FORTRAN using subroutine calls to the SCHEDULE library to define the parallelism [3]. The programmer usually starts with a directed acyclic dependency graph, which describes the order of execution and defines the parallelism. He then generates the subroutine calls to the SCHEDULE library using the information from the graph. This large grain control flow graph differs from large grain data flow in that the arcs do not represent explicit data items. Instead they represent an assertion by the programmer of a legitimate partial order on the units of computation represented by the nodes of the graph.

In the SCHEDULE environment, the parallelism is at the subroutine level. Each node in the execution graph represents a self-contained unit of computation which is defined by a FORTRAN subroutine name and its calling parameters. The arcs represent execution dependencies between the units of computation. There must be at least one node with no incoming dependencies and one and only one node with no outgoing dependencies. These conditions are necessary for program initiation and termination. Each node requires two subroutine calls to SCHEDULE. One subroutine call defines the node's execution dependencies in the graph, and the other defines the unit of computation. During the execution of the program, a unit of computation is scheduled for execution whenever its execution dependencies are satisfied. Thus, there are as many processes executing as allowed by the capability of the machine and the parallelism in the execution graph.

We have developed a graphics tool called BUILD that will allow a user to interactively draw an execution graph on a workstation screen, specify the user-supplied subroutines for each node in the graph, and have the workstation generate the FORTRAN code with the necessary calls to the SCHEDULE library. This code can then be compiled, linked with the user-supplied subroutines and the SCHEDULE library. The code is independent of any particular machine and can be run on any one of the existing parallel processors [3,4] which will now run the SCHEDULE package. Moreover, one can obtain a post-processing animated graphics display and analysis of the execution history. This feature is described in [3,4].

2. Motivation

When trying to implement parallel algorithms on parallel computers, one of the most frustrating tasks is mapping a particular algorithm onto a particular parallel computer. Even when the parallel aspects of the algorithm are completely understood, a great deal of time can be spent determining the correct use of the particular parallel constructs of the target machine, generating code, and debugging. Unfortunately, due to the lack of standards, this tedious and time consuming process must be repeated for every parallel computer one would like to use. At the moment, each manufacturer has defined its own constructs for parallel programming. With so many different parallel computers on today's market, the problem of producing portable codes has become an increasingly important issue. The SCHEDULE package was an attempt to address this issue by providing a common environment for parallel programming [5].

The SCHEDULE programming environment provides a mechanism to construct an explicitly parallel program. This is done using calls to routines in the SCHEDULE library. Specifically, the programmer draws or imagines the execution graph and then gathers from the graph the necessary information for the calls to SCHEDULE. Translating the visual execution graph into the symbolic calls to SCHEDULE is tedious and prone to both typographic errors and misinterpretation of the graph. However, it is a straightforward procedure and is easily automated.

With BUILD, we are trying to eliminate the need for the programmer to translate from the execution graph to the SCHEDULE library subroutine calls. The user can design an algorithm graphically, define its units of computation, and let BUILD tie them all together.

A number of systems have been developed that use a type of graphical description as the initial step in developing parallel programs. Through a series of steps the graphical description is converted into a parallel program for a specific implementation. Babb [1] uses large-grain data flow (LGDF) graphs, the Poker Programming Environment of Snyder [6] uses a interactive graphics interface to embed a graph in a lattice, and the Computationally-Oriented Display Environment (CODE) by Browne et al. [2] uses a version of generalized dependency graphs as its programming language. We have learned a great deal from their efforts.

3. Goals

The long term goal of this work is to create a tool that will generate a parallel FORTRAN code from a graphical description that will run using the SCHEDULE package. Here, we

describe a preliminary version of the tool which demonstrates the feasibility of our approach. It has been used by us and by others to generate parallel programs which have then executed successfully on a parallel computer. At this point, it can generate any program that can be described using a static graph with at most one level of dynamic spawning of processes. However, it cannot yet be used to generate the most general SCHEDULE program. In particular, any of the examples described in [3,4] could be generated using BUILD. Some user intervention is required to produce the final program. However, this is typically minor and does not involve the specification of the dependencies, which is fully automated.

We wanted the tool to be written under a widely used windowing system so as to be available to a large number of users. The tool was originally written in SunView, and is currently being ported to the X Window System. NeWS and Display Postscript are under consideration. Since BUILD is closely related to the SCHEDULE trace facility that we developed, it was desirable to produce the same graphical representation in both tools. A user should be able to work with the same representation in both the development tool and the analysis tool.

4. Description of the Tool

The graphics display of BUILD has two parts, the user control interface and the drawing surface (see Figure 1). An execution graph can be built, displayed, and manipulated on the drawing surface using the user control interface and certain mouse button sequences. The definition of the graph can be saved and later retrieved. At any time in the construction of an execution graph, the FORTRAN code with the appropriate SCHEDULE calls representing that execution graph

can be generated.

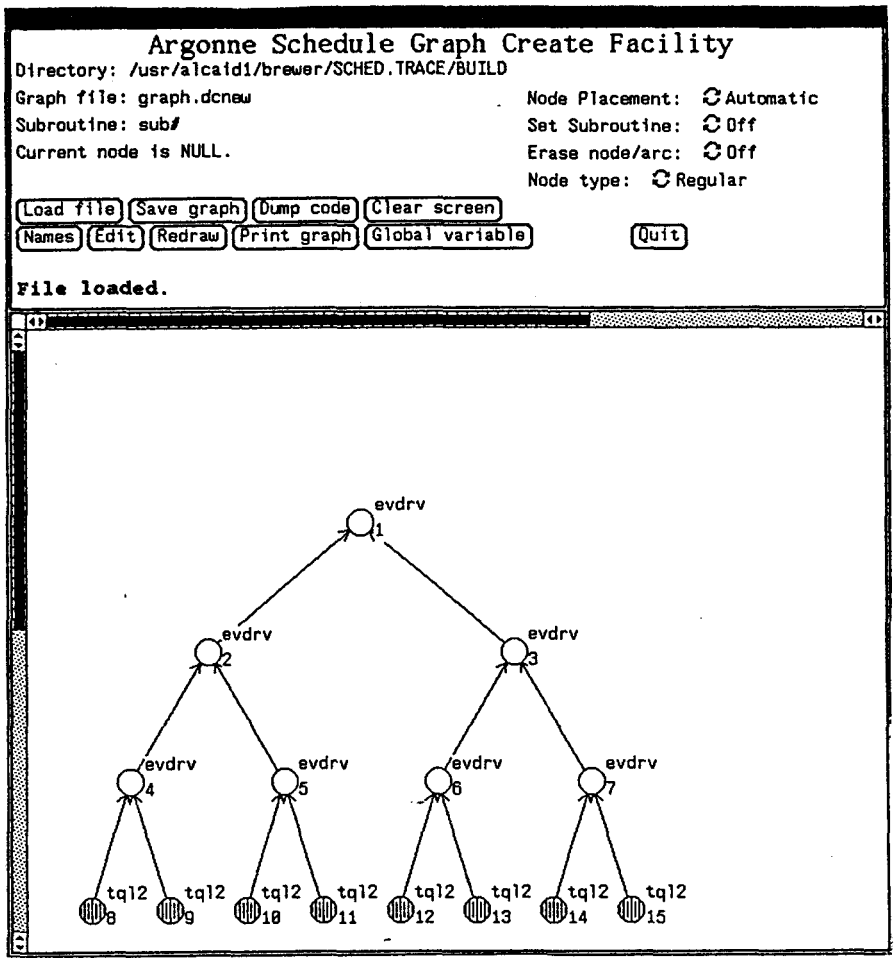

Figure 1. The graphics display of BUILD

By default, BUILD is in *insert* mode. In this mode, the mouse buttons perform the following functions whenever depressed and released in the drawing surface:

• Left mouse button: Depressing and releasing this button where no other node is located will add a new node to the graph. Depressing this button on an existing node, moving the mouse cursor to another location, and releasing the button will move the selected node to the new location (when the node placement option is user, as explained below).

• Middle mouse button: Depressing this button in one node and releasing it in another will add an arc to the graph.

• Right mouse button: Depressing and releasing this button within a node will display the subroutine name assigned to that node. Depressing and releasing this button where no node is located will erase any such display.

The panel subwindow (see Figure 2) is BUILD's main user control interface and contains several features:

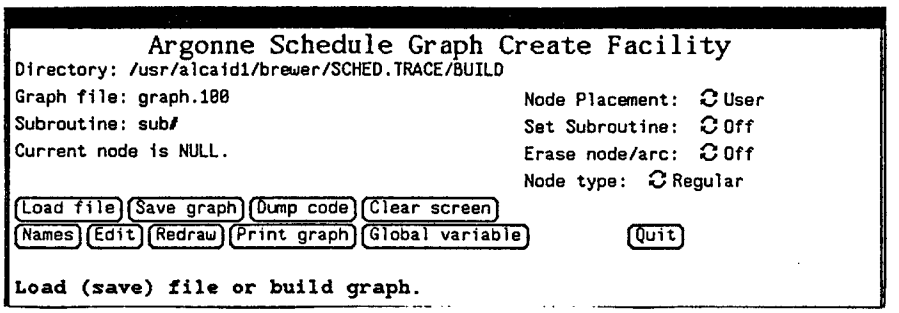

Figure 2. BUILD control panel

• Directory: The user can step through various directories to locate a desired graph file. If the cursor is placed over the end of the directory string and the right mouse button is pressed, a menu listing of other directories will appear. To change to one of these directories, the user simply uses the cursor to highlight the directory and releases the right mouse button. One of the directories in the listing will have a check next to it (most probably the "." directory). Depressing and releasing the left mouse button while the cursor is positioned on the directory string will cause a change to the directory listed after the checked directory (most probably the ".." directory).

• Graph file: When files exist in the current directory that begin with the letters *graph*, the first of these will appear in the graph file string. Positioning the mouse cursor on the string and depressing the right mouse button will cause a menu of graph files to appear. Files can be selected from the menu by highlighting the file name and releasing the right mouse button. Depressing and releasing the left mouse button will cause the next file in the menu to be selected.

• Subroutine: When the file *sub.list* exists in the current directory, BUILD assumes it contains names of FORTRAN subroutines and will read the names into a subroutine list. Positioning the mouse cursor on the word *Subroutine:* and depressing the right mouse button will cause a menu of the current subroutine names to appear. Names can be selected from the menu by highlighting the subroutine name and releasing the right mouse button. Depressing and releasing the left mouse button will cause the next subroutine name in the menu to be selected. If any graph files are subsequently loaded and the node definitions contain subroutine names, these names will be added to the subroutine list.

• Current node: This line will list the last node on which any mouse button was clicked.

• Node placement: There are two options for placement of the nodes. The *user* option allows the user to control the placement of the node, while the *automatic* option allows the program to position the nodes. The program uses a simple algorithm for positioning the nodes so that there is no noticeable delay in drawing the graph.

• Subroutine: This feature modifies the behavior of the right mouse button while in the drawing

surface. Depressing and releasing the right mouse button while the mouse cursor is in a node will assign the currently selected subroutine name to that node. Depressing and releasing the right mouse button while not in any node has the same effect as explained before. Subroutine mode will remain in effect until switched off.

• Erase node/arc: This feature allows for the deletion of one node or of one arc. To delete a node after turning the feature on, one simply moves the mouse cursor to the desired node and depresses and releases the left mouse button. To delete an arc, one moves the mouse cursor into one node, depresses the middle mouse button, and then moves the mouse cursor into the second node and releases the middle mouse button. Erase mode is active for one deletion only. It automatically turns itself off after each deletion.

• Node type: There are two node types; *regular* and *loop*. The regular node type represents one unit of computation as defined by the node's subroutine name and its calling parameters. The loop node type represents a FORTRAN DO loop where the body of the loop is a unit of computation as defined by the node's subroutine name and its calling parameters. Thus, this node defines n units of computation where n is the number of iterations of the loop. The SPAWN mechanism of SCHEDULE[3,4] is used here to dynamically spawn these units of computation. The value of n may be specified at run time.

The ovals in the panel act as buttons. Any button in the panel may be activated by clicking the left mouse button while the mouse cursor is within the boundaries of that panel button. The

button will remain gray as long as the action started by the button continues.

- Load file: This action will reset BUILD and load the currently listed graph file. Once a graph file has been chosen, it must be loaded before before one performs any other functions on it. If any other graph is currently displayed on the drawing surface, it will be erased (and all information lost unless it was saved; see below) and the new graph will be displayed. If the node definitions contain subroutine names, these names will be added to the subroutine list (if not already present). If there are X and Y coordinates for each node, the nodes will be positioned at those coordinates, and the node placement option will be set to *user*. If not, they will be positioned by the program, and the node placement option will be set to *automatic*.

- Save graph: This button will bring up a subwindow that allows the user the option of saving the current graph definition in the currently listed graph file or in another file.

- Dump code: This button will bring up a subwindow that allows the user the option of dumping the generated SCHEDULE calls to the standard output or writing them to a specified file.

- Clear screen: This button will clear the drawing surface and delete all information on the current graph.

- Names: This button will display next to each node its currently assigned subroutine name. For large graphs, this will produce a very busy picture and may blot out most of the graph. The right mouse button feature described above may be more useful in this case.

- Edit: This button will allow the user to select a node and edit its subroutine. The program assumes that the file to be edited will be *name.f* where *name* is the subroutine name assigned to the selected node. After selecting *Edit*, the user selects the desired node by moving the mouse cursor into a node and pressing and releasing the left mouse button. This will bring up a shelltool in which the *vi* editor will be invoked. After editing the file, the user may quit the editor normally and exit the shell. To exit the edit mode without editing a file, one simple selects the *Edit* button again.

- Redraw: This button will redraw the graph.

- Quit: This will completely exit the BUILD tool, first asking for confirmation.

The line below the buttons provides information for the user during the use of BUILD.

5. Examples

Below we describe some simple examples using BUILD. As mentioned earlier, a complete program is not generated from the graph at this time. One must make some minor adjustments to the file before it can be compiled and linked to produce an executable image. These adjustments

mainly have to do with variable declarations and initializations, and with parameter lists to the units of computation. Of course, our goal is to have BUILD produce a program that requires no additional manipulation and we intend on adding these enhancements in the future.

In each example, a file containing the FORTRAN code with the calls to the SCHEDULE library was generated by BUILD. We then added the variable declarations and initializations, and adjusted the parameter lists to the units of computation. We compiled the file on the Alliant FX-8 and linked with the object file containing the units of computation and with the SCHEDULE library to produce an executable image.

5.1. Dot product

Let us begin with a very simple example. Consider the problem of computing the dot product of two vectors of length n. If we partition each vector into k smaller vectors, each of length n/k, then the dot product of each pair of smaller vectors can be computed in parallel. After these k dot products have been computed, they can be added together to produce the final dot product. Please note that we are using this problem for an example only; we do not recommend the use of SCHEDULE on a problem of such small granularity.

The execution graph (see Figure 3) representing this problem was constructed with BUILD. In our case $k=10$. The ten nodes along the bottom of the graph represent the calculation of the dot products of the k smaller vectors. These nodes are leaves of our execution graph and so can begin execution immediately. The subroutine *inprod* has been selected for each of the ten nodes. There is an arc from each one of these nodes to the node on the top. This node represents the

addition of k dot products. The subroutine *addup* has been selected for that node. The arcs indicate that the node may not execute until each of the k leaf nodes have completed.

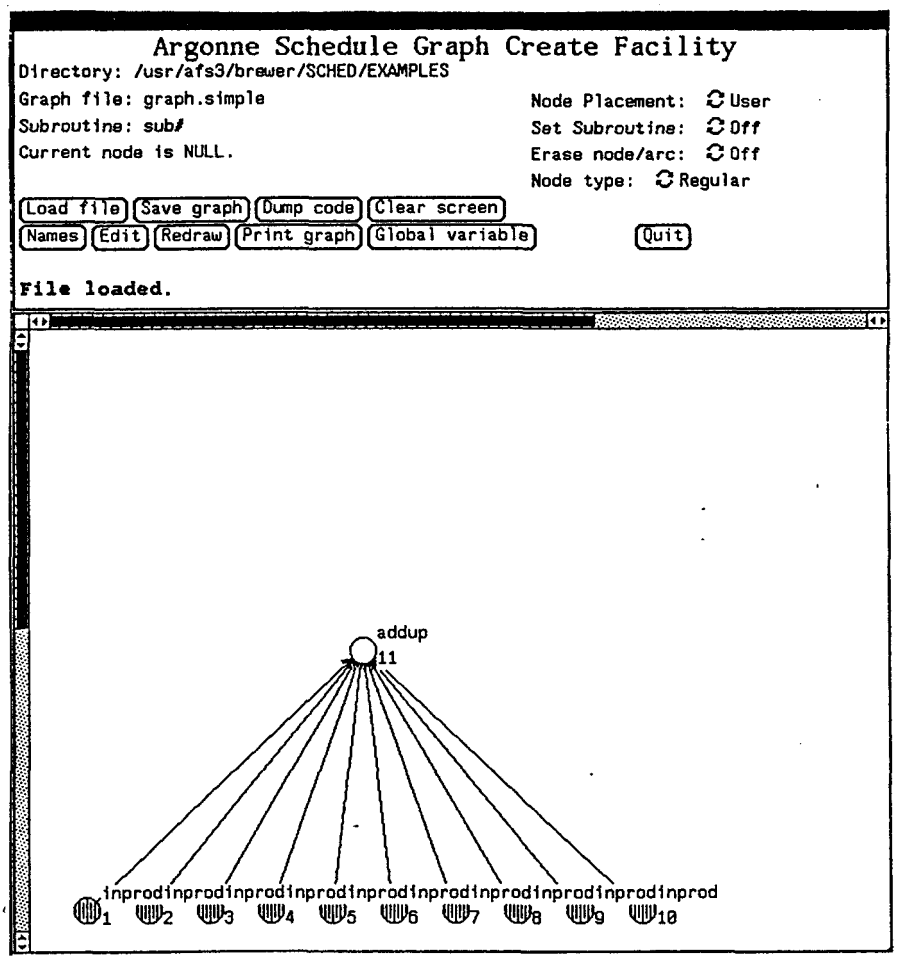

Figure 3. The execution graph of the dot product

5.2. Using the loop construct

An alternative to the above method is to use the loop construct. This allows the value of k to be determined at run time and the appropriate number of processes initiated. Figure 4 shows the graph representing this method. The code from BUILD (see below) was compiled and then linked with a special SCHEDULE library. This special SCHEDULE library will produce a trace

file describing the order of the execution when the program is run. Figure 4 also shows the play-back of this trace file using our analysis tool.

The following listing shows the actual code generated by BUILD for this example. The lower-case indicates the actual code from BUILD while the upper-case indicates adjustments made for variable declarations and parameter lists. The three dots at the beginning indicate code initialization. The variable K is the same as the *k* described above and M is *n/k*. The variables A and B are the vectors, TEMP accumulates the *k* dot products, and SIGMA is the final dot product.

```
      PROGRAM MAIN
      COMMON /BUILD/ ID, NUMITS
      external paralg
      REAL A(1000), B(1000), TEMP(50), SIGMA
        .
        .
        .
      call sched(nproc,paralg,M,K,A,B,TEMP,SIGMA)
      stop
      end
c
      subroutine paralg(M,K,A,B,TEMP,SIGMA)
      INTEGER M,K
```

```
      REAL A(*), B(*), TEMP(*), SIGMA
      COMMON /BUILD/ ID, NUMITS
      external addup
      external loop1
      integer jobtag, icango, nchks, mychks(1)
      INTEGER ID, NUMITS
c
       jobtag = 1
       icango = 1
       nchks  = 0
       call dep(jobtag, icango, nchks, mychks)
       call putq(jobtag, addup,K,SIGMA,TEMP)
       jobtag = 2
       icango = 0
       nchks  = 1
       mychks(1) = 1
       ID = JOBTAG
       NUMITS = K
       call dep(jobtag, icango, nchks, mychks)
       call putq(jobtag, loop1, id, numits,M,A,B,TEMP)
      return
```

```
      end
c
      subroutine loop1(myid,n,M,A,B,TEMP)
      REAL A(*), B(*), TEMP(*)
      integer myid, n
      logical wait
      external inprod
c
      go to (1100,1200), ientry(myid,2)
 1100 continue
       do 1150 i = 2, n
        call nxtag(myid, jdummy)
        INDEX = N*(I-1)+1
        call spawn(myid, jdummy, inprod,M,A(INDEX),B(INDEX),TEMP(I))
 1150   continue
       call inprod(M,A(1),B(1),TEMP(1))
       if (wait(myid,2)) return
 1200 continue
      return
      end
```

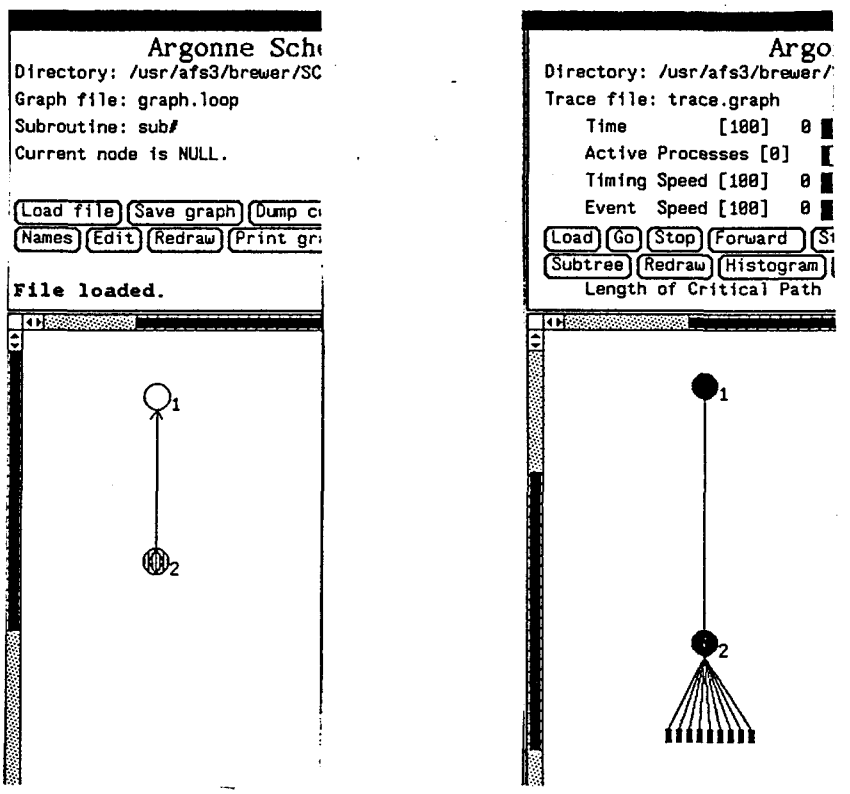

Figure 4. The graph from BUILD (left) and the trace produced by SCHEDULE (right)

5.3. Solution of a triangular system of equations

Next, we give an example which requires a more intricate synchronization mechanism. Consider the solution to the matrix equation $Ax=b$, where A is an $n \times n$ lower triangular matrix, and x and b are both $n \times 1$ column vectors. The solution of x_1 is trivial. It is simply

$$x_1 = \frac{b_1}{a_{11}}.$$

The remaining rows can then be simplified through the process of forward substitution. Row i,

$$a_{i1}x_1 + \ldots + a_{ii}x_i = b_i,$$

becomes

$$a_{i2}x_2 + \ldots + a_{ii}x_i = b_i - a_{i1}x_1.$$

Note that these updates can be done in parallel.

After the update to the second row, the solution to x_2 can be computed. Moreover, note that once x_2 has been computed then each of the updates $a_{i2}x_2$ can be computed and then rows 3 to n can be updated, again in parallel. This continues until the solution to x_n becomes possible. Figure 5 shows the computation graph for this algorithm for an 8×8 matrix.

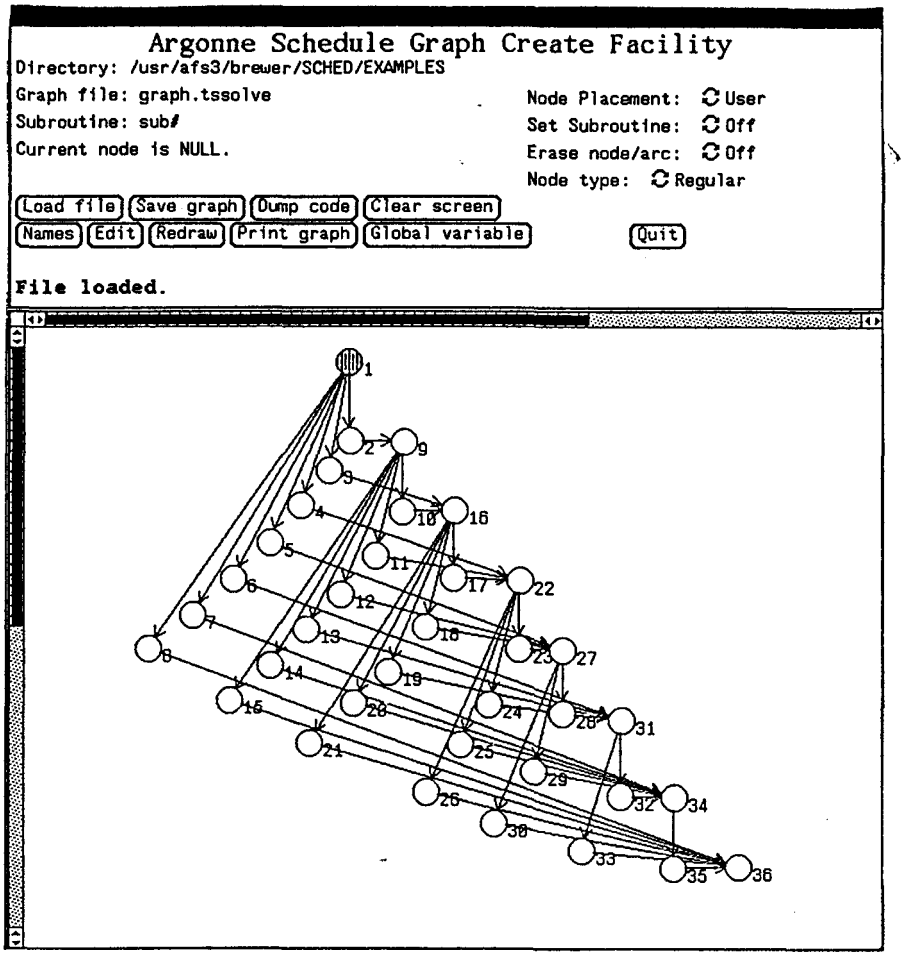

Figure 5. The execution graph of the triangular solve

The node at the top has no input dependencies and can begin the execution. It represents the calculation of the solution for x_1. The arcs pointing down to nodes 2 to 8 indicate that once the solution for x_1 has been calculated, rows 2 through 8 can be updated with that solution. The arc from node 2 to node 9 indicates that once row 2 has been updated with the solution for x_1, the solution for x_2 can be calculated. Thus, the nodes on the right diagonal represent the calculations for the solutions of the x_i and the nodes below the diagonal represent the calculations for updating the rows with previous solutions.

In reality one would not partition the solution of a triangular system at such a fine grain level for use with SCHEDULE. One would instead use the same synchronization mechanism applied to a block partitioning of the matrix. In this case elements a_{ii} would be replaced by lower triangular blocks, and elements a_{ij} , $i>j$ would be replaced by rectangular blocks. Moreover, the scalar operations b_j/a_{jj} and $a_{ij}x_j$ would be replaced by triangular solution and matrix vector product respectively. Strictly speaking, the synchronization must include coordination of the updates to the right hand side b. This may be accomplished either by use of an explicit locking mechanism or by providing work space to accumulate the updates and then waiting until a triangular unit of computation is ready to execute and accumulating all of the updates at once just prior to the triangular solve step. The latter is preferable in our view.

6. Availability of the Tools

The software described in this report is available electronically via *netlib*. To retrieve a copy, one should send electronic mail to netlib@mcs.anl.gov. In the mail message type:

send build from sched

A UNIX *shar* file containing the software will be sent back to the originating mail address automatically. At present the software runs on SUN 3/60 as well as the higher model numbers. There is a color version that is also available. It is recommended that one request the SCHEDULE and SCHEDULE/trace facility at the same time. A version is available for the SUN so that the resulting parallel program may be run with a single processor on the SUN. To

get these again send mail to netlib@mcs.anl.gov and in the mail message type:

send sun from sched

send trace from sched

to get a version for other machines substitute the machine name in place of the word *sun*. The message "send index from sched " will give a list of available machines.

7. Summary

We feel that graphics tools for the development and analysis of parallel programs will play an important part in the future of parallel processing. With the proliferation of bit-mapped displays and network transparent windowing systems, tools of this type will be available to most programmers and scientists. We also feel that they are necessary. Jumping from sequential programming to parallel programming is like jumping from two-dimensional geometry to three-dimensional geometry. Trying to understand what is going on is a lot easier when one can see it.

References

1. R.G. Babb, "Parallel Processing with Large Grain Data Flow Techniques," *IEEE Computer*, vol. 17, no. 7, pp. 55-61, July 1984.

2. J. C. Browne, M. Azam, and S. Sobek, "Architectural and Language Independent Parallel Programming: A Feasibility Demonstration," *Tech. Report, Department of Computer Science, University of Texas, Austin*, February 15, 1988.

3. J. J. Dongarra and D. C. Sorensen, "SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs," in *The Characteristics of Parallel Algorithms*, ed. L. H. Jamieson, D. B. Gannon, and R. J. Douglass, The MIT Press, Cambridge, Mass., 1987.

4. J. J. Dongarra, D. C. Sorensen, and O. Brewer, "Tools and Methodology for Programming Parallel Processors," *Proceedings IFIPS Working Group 2.5 Conference*, Stanford University, Aug 1988.

5. J. J. Dongarra, D. C. Sorensen, K. Connolly, and J. Patterson, "Programming Methodology and Performance Issues for Advanced Computer Architectures," *Parallel Computing*, vol. 8, no. 1-3, pp. 41-58, Oct. 1988.

6. L. Snyder, "Parallel Programming and the Poker Programming Environment," *IEEE Computer*, vol. 17, no. 7, pp. 27-36, July 1984.