Design of software for distributed/multiprocessor systems

by TERRENCE R. MCKELVEY and DHARMA P. AGRAWAL
Wayne State University
Detroit, Michigan

ABSTRACT

Software design methodologies for distributed/multiprocessor systems are investigated. Parallelism and multitasking are considered as key issues in the design process. Petri-nets and precedence graphs are presented as techniques for the modeling of a problem for implementation on a computer system. Techniques using the Petri-net and precedence graph to decompose the problem model into subsets that may be executed on a distributed/multiprocessor system are presented. These techniques offer a systematic design methodology for the design of distributed/multiprocessor system software.
INTRODUCTION

Since the advent of the digital computer, the need for faster, larger, more reliable, and expandable systems has existed. Distributed/multiprocessor computer systems have resulted from this need. Though hardware has been developed to allow the exploitation of concurrent computation, the application of this hardware to real-world problems, and the development of software to solve these problems is developing at a slower pace. This paper presents some of the current thinking on the subject of software design methodologies for distributed/multiprocessor systems. The following sections discuss some basic concepts and definitions, parallelism at the various levels of a computer system, multitasking as a design approach for multiprocessor systems, graphical techniques for the representation of an application, and finally some techniques for using these graphical representations to decompose the application into sections that can be run concurrently on a multiprocessor system.

For the purposes of this paper, the concept of a distributed/multiprocessor system may be defined as follows: “Distributed computing refers to the use of multiple, quasi-independent processing modules, whose actions are coordinated to accomplish a large task or to implement a large system.” Though the system need not be large, the need for multiple, fairly independent processing modules tied together into a system is key to the concept of distributed/multiprocessor computation. Another key point is this one: “In a distributed computing system, the fact that multiple processing modules exist is visible to the user of the system, and therefore meant to be exploited in the design of applications.” Though much work is being done to develop tools that hide this visibility from the user, optimal use of distributed/multiprocessor systems will result when the designer detects and exploits parallelism existing in the application, as examined in the rest of this paper.

PARALLELISM IN COMPUTER SYSTEMS

Parallelism may be introduced into a system at various levels. Computation that can be done in parallel may be done within separate processor modules, thus obtaining the speed and reliability advantages offered by distributed systems. The design of the software will ultimately determine the success or failure of a computer system to solve a given real-world problem. “Matching a program representation to the underlying hardware or interpretive resources in a computer system is a key problem in computer system design. Failure to accurately and completely represent the computation significantly degrades the performance of the resulting execution.” The translation of a problem into a computer implementation may be represented as a hierarchical structure:

1. An algorithm (or solution of the problem) specifies
2. A set of tasks (functions to be performed), composed of
3. Higher-level-language statements, represented by
4. Computer instructions, which cause
5. State transitions in the computer hardware.

This hierarchy forms a pyramid, where each level of the hierarchy is composed of a number of elements of the next level. (An algorithm is composed of tasks, for example.) Parallelism may be detected at any of these levels. Of these levels, the following three are the important levels at which parallelism may be detected:

1. Algorithm level
2. Source language level (i.e., the higher language level)
3. Machine language level (i.e., instruction and state transitions)

Detection and exploitation of parallelism is the key to the effective design of distributed/multiprocessor systems. Studies in the dynamic detection of parallelism at the machine language level have been performed. These studies have led to the conclusion that an overall net parallelism detection of less than a fivefold factor over the strictly sequential execution of machine language instructions is theoretically possible. This would probably lead at the practical level to a twofold improvement over the sequential approach. This level manifests itself in the detection and parallel execution of independent machine language instructions. Pipelining at the instruction level is a similar technique to exploit parallelism at the machine instruction level.

Detection and exploitation of parallelism at the source language level is currently an area of active research. Program analyzers have been written to detect inherent parallelism in programs written in higher-level languages. It has been empirically observed via such an analyzer that a speedup, SP, on P processors would be possible:

\[ SP = \frac{P}{10 \log_{10} P} \]

This result is based on the analysis of FORTRAN programs and may be found to be better or worse for other higher-level languages or implementations of FORTRAN on other computers.

The detection of parallelism at the algorithm level is, of course, very dependent upon the problem to be solved. Various approaches to the parallel execution of sorting and searching algorithms appear in the literature. However, very little
has been written concerning a design methodology to detect and exploit parallelism at the algorithm level.

Very few real-world applications are as specific as searching and sorting algorithms. Many real-world applications consist of several algorithms, and one would most probably consider the algorithm level of the previous discussion to be the system design level as practiced by the computing community. This level may produce a system made up of hundreds of individual programs to perform the intended function. The process of breaking up the system into a number of tasks (to be defined shortly) and determining which of these tasks may be executed in parallel offers a mechanism to detect and exploit parallelism at the algorithm level. This method, known as multitasking, has been used on uniprocessor systems for years. The subject of multitasking will be discussed in the following section. As an aside, however, it is worth noting that the methodologies presented offer no panacea for the design of distributed systems. The designer must intelligently, and often iteratively, apply these techniques in order to find the optimal design for his or her application.

MULTITASKING AS A DESIGN APPROACH FOR DISTRIBUTED SYSTEMS

As discussed in the previous section, multitasking offers a design methodology for the detection and exploitation of parallelism for distributed/multiprocessor systems. This section will define tasks and multitasking and present some examples of the multitasking approach to system design.

A task may be defined as a unit of computational activity. When the computer first became available, a programmer would code an application as one large program or algorithm. The computer would load and execute this and any other tasks, one at a time, from start to completion of the algorithm. As time went on, it was found that the central processing unit (CPU) would be idle during certain activities—I/O, for example. During this time, it was proposed, another task could be executed until CPU had to wait. Thus was born the concept of multitasking, the capability of executing more than one task concurrently. This capability, extended to multiple-processor computer systems, is known as multiprocessing. The capability of multitasking is also known as multiprogramming.

The concept of multitasking, as stated above, was initially conceived to take advantage of expensive CPU idle time. However, it is a valid method for the design of software systems. There is no reason why a problem must be programmed and executed one step at a time from start to finish. In fact, for many problems, this approach becomes extremely awkward, especially in many real-time applications where data must be collected, displayed, and analyzed concurrently.

Designers often shy away from the multitasking approach, since people generally tend to think sequentially. However, the multitasking approach offers advantages in terms of efficiency of resource use; improvement in the overall speed of execution; and, most important, a natural design methodology. An example will make the above statement clearer as well as allow a comparison of the sequential and multitasking approaches to design.

Suppose a designer is asked to design a system that will read a record of data from a data collection subsystem, display the data on a CRT, and save the data on a disk. Assume the following additional requirements:

1. Data records must be averaged across ten readings.
2. The display must be updated at least once every five seconds.

A sequential design, presented in a pseudo-high-level language, might be that shown in Figure 1. Though the solution is relatively straightforward, it could easily not fulfill a requirement of the system: it may very well take longer than five seconds to reach the statement to display the results on the CRT. In fact, one may find that records from the data collector might be missed if it takes too long to write the record to disk or to update the CRT display. Admittedly, this example ignores the fact that direct memory access and interrupt processing capabilities could solve some of these problems, but the main point remains that the sequential approach may take too long and could potentially fail to do the required job.

The multitasking approach asks the question, "Is there any way to break up this system into a number of cooperating tasks which could run concurrently?" The answer is "yes," and a solution is shown in Figure 2. The above solution assumes the existence of three record buffers for use by the three tasks. These three tasks can operate in a pipelined manner: that is, Task 1 collects a record bufferful and passes it to Task 2, which averages and saves the buffer on disk while Task 1 is collecting the next bufferful. Task 2 gets the results of Task 1; and while Task 3 displays the results, Task 2 processes the second bufferful while Task 1 is collecting the next bufferful. Once the pipeline is running, all three major functions (collection, averaging and disk storage, and display) are proceeding concurrently. On a uniprocessor any one task could run while the others are in wait states. To implement this any-one-of-three

```
DO FOREVER:
  CLEAR RECORD_BUFFER
  COUNT = 1
  DO WHILE COUNT < 10:
    WAIT FOR RECORD
    RECORD_BUFFER = RECORD
    COUNT = COUNT + 1
  END

  COUNT = 1
  CLEAR RESULT_BUFFER
  DO WHILE COUNT < 10:
    RESULT_BUFFER = RESULT_BUFFER + RECORD_BUFFER(COUNT)
    COUNT = COUNT + 1
  END

  RESULT_BUFFER = RESULT_BUFFER / 10
  WRITE RESULT_BUFFER TO DISK
  DISPLAY RESULT_BUFFER ON CRT
END
```

Figure 1—Sequential design presented in pseudo high-level language
task system $C$, made up of tasks $T_1$, $T_2$, ..., $T_n$, Tasks $T$ and $T'$ of $C$ are noninterfering if either of the following conditions is true:

1. $T$ is a successor or predecessor of $T'$. That is, $T$ runs to completion before (predecessor) or after (successor) $T'$ runs to completion. In other words, $T$ and $T'$ run in a strictly sequential relationship to one another. 
2. The intersection of the following sets is the null set:
   a. Ranges of $T$ and $T'$
   b. Range of $T$ and domain of $T'$
   c. Domain of $T$ and range of $T'$

That is, \( R_T \cap R_{T'} = R_T \cap D_{T'} = D_T \cap R_{T'} = \emptyset \)

For \( C = \{T_1, T_2, \ldots, T_n\} \), \( T_i \) and \( T_j \) are mutually noninterfering if \( T_i \) and \( T_j \) are noninterfering for all \( i, j \) where \( i \neq j \). Task systems made up of mutually noninterfering tasks are determinate.\(^5\)

What the above discussion indicates is that one establishes a precedence relationship to assure the determinacy of a task system, thus assuring consistent results when executing the task system. A method will be presented under the discussion of precedence graphs to find the set of tasks that can be executed in parallel, given a determinate task system.

A key question one might ask is, "How does one go about detecting possible tasks?" The answer lies in the concept of stepwise decomposition of the application into major functions to be performed and the major functions into subfunctions until one reaches a level of detail sufficient for understanding how the application will be implemented. This list of functions and subfunctions defines a potential list of tasks and steps within tasks. Using the previously presented example, the application was a monitoring system. This system was decomposed into three major functions: read input, average and store on disk, and display on a CRT. For this particular system, this level of decomposition defines the application enough to make software implementation possible. Of course, on a larger system, more functions and even subfunctions could be defined. The relationships between tasks are established, and it becomes possible to model the system by means of one of a number of graphical techniques.

Various existing graphical techniques may be used to design the algorithms and tasks required for a specific application. Methods exist for partitioning these graphs into segments that may be executed in parallel. These techniques will now be discussed.

**GRAPHICAL TECHNIQUES FOR THE REPRESENTATION OF SYSTEMS**

As stated previously, this section will discuss graphical techniques for the representation of application systems. These techniques may be used to partition an application into tasks and tasks into programs. Additionally, techniques exist for the partitioning of these graphs into segments that may be
executed in parallel. Two graphical techniques will be presented:

1. Petri nets
2. Precedence graphs

These are commonly used techniques in the computer engineering and computer science disciplines, respectively.

PETRI NETS

The Petri net has been discussed extensively in current literature; therefore only a brief introduction will be presented here. The major emphasis will be upon partitioning techniques. A Petri net is a graph model for "modelling the flow of information and control in systems, especially those which exhibit asynchronous and concurrent properties." Two types of nodes exist in Petri nets: circles (called places) that represent conditions and bars (called transitions) that represent events. Black dots (called tokens) appear in circles to represent the holding of a condition at that place. The distribution of tokens throughout the graph represents the state of the system. The behavior of the system can be determined by tracing the flow of tokens through the system. Tokens move from one place to another when a transition fires. The following rules define the conditions under which transition firing may occur:

1. A transition is enabled if and only if each of its input places has at least one token.
2. A transition can fire only if it is enabled.
3. When a transition fires:
   a. A token is removed from each of its input places.
   b. A token is deposited into each of its output places.

Figure 3 depicts a simple computer system modeled via a Petri net. The concurrency between I/O on tape and the execution of computing on the CPU is apparent: when the tape is ready, the processor is ready, and an input queue entry exists, the transition fires, I/O on tape and computing run concurrently, and the results go to the output queue after the appropriate I/O and computing have completed. One could "walk through" the system in Figure 3 by moving the tokens from place to place.

As shown in Figure 3, concurrent operations fork at the transition at the top of the figure and join at the transition at bottom of the figure. In a Petri net of more considerable size there may be many such forks and joins in the net, which represent sites of concurrent activities. By partitioning the Petri net along these forks (called distribution AND nodes) and joins (called synchronization AND nodes) the net can be broken up into subnets having an initiation point and a termination point with no forks or joins between nodes. In other words, a subnet of strictly sequential places and transitions can be produced by breaking up a Petri net at each of its fork and join nodes. However, these subnets, while representing the maximum level of parallelism, cannot all execute in parallel at the same time. This is due to the fact that the transitions firing resulting in forks happens at different times. Therefore, although one has all the subnets that could potentially run concurrently, one does not have subnets of operations that can all run concurrently.

Since one does not have the set of all subnets that can all be executed concurrently, one would not be efficiently using processors if one loaded each subnet into a separate processor and triggered each processor at the appropriate transition point. Some processors would be executing in parallel, it is true; but many would be idle during the course of execution of each subnet. What is desired is not to split the Petri net into subnets at each fork and join point, but rather, after the first fork, where one does do a split, to follow sequential chains through each fork and join. In other words, one breaks the Petri net at the first fork and proceeds down the places and transitions until either a fork or a join is encountered. In the case of a fork, one of the possible paths is chosen and the subnet chain continues along that pathway. The other possible pathway at the fork becomes the beginning of another chain. If a join is encountered, again a subnet chain continues; however, the other arrows entering that join transition become the termination points for the other subnet chains followed up to that point along other pathways. By following various pathways through the Petri net in this way, one produces subnets of sequential places and transitions that are longer than the approach of splitting the Petri net at each fork and join. If the paths are carefully chosen, it is possible to produce subnets that can be executed concurrently, although not necessarily all at the same time. To summarize, what is desired is to produce a set of subnets from an initial Petri net so that the following conditions are met:

1. A minimal number of subnets are produced, all of which are sequential chains of places and transitions.
2. The subnets are chosen to allow maximal concurrent execution of each subnet.

Toulotte and Parsy\(^7\) present an algorithm for this decomposition that would satisfy Condition 1. This algorithm produces a set of subnets based on the idea that the optimal set of such subnets is the set having the least number of subnets, where each subnet is a sequential chain. The algorithm may be summarized as follows:

1. Define the initial place.
2. Trace down the chain of places and transitions until a fork or join transition is encountered (called an AND node).
3. If the AND node is a fork, determine which output path will result in the smallest number of additional subnets.
4. If the AND node is a join, determine which input path, if continued, would result in the smallest number of additional subnets.

Though this algorithm would produce a minimal set of subnets (see Toulotte and Parsy\(^7\) for more details on the algorithm itself), this minimal set may not be the optimal set for maximally concurrent execution. That is, Condition 2 is not covered by this algorithm. With some alteration, the algorithm could probably be modified to find the set of such subnets such that both Conditions 1 and 2 above would be met. This would result in an algorithm that would allow one to determine the maximum number of subnets that could concurrently execute on a set of processors, requiring one processor per subnet. This algorithm could be automated and done on a computer once a Petri net of the application was produced. Some guidelines for producing the initial Petri net will be made after a specific example of the subnet splitting technique is presented.

Figure 4 presents a Petri net having several AND nodes. The places are labeled P1 through P6 and the transitions t1 through t6. If one were to split up this net at each fork and join, one would produce the following subnets:

\[
\begin{align*}
S1 &= P1, t1 \\
S2 &= P2, t2 \\
S3 &= P3, t5, P7, t3 \\
S4 &= P4, t6, P8, t4 \\
S5 &= P5, t3 \\
S6 &= P6, t4 \\
\end{align*}
\]

The above subnets represent the maximally parallel set of subnets. However, these subnets cannot all be run concurrently. Applying the above algorithm, one produces the subnets illustrated in Figure 5. These three subnets happen also to fulfill both the conditions listed above. For this example, the optimal number of processors would be three, where the second processor begins executing at t1 and the third processor at t2, with all three running until t3, at which point the second processor stops and the first and third processors continue until t4. While satisfying Condition 2 above was fairly obvious for this example, in a larger Petri net various alternative chains might have to be tried to find the optimal set of subnets. This, like the discussion on Petri net generation that follows, may require an iterative process to obtain the optimal results.

As stated above, several guidelines may be presented on the
generation of the Petri net model for an application. The guidelines may be summarized as follows:

1. Break up the application into major tasks to be performed. These become the places in the Petri net.
2. Define the precedence relationships between the major tasks (i.e., which tasks depend on results from other tasks). These precedence relationships define the transitions between the tasks. Tasks that produce results needed by more than one other task are connected to those other tasks via a fork transition and have a predecessor relationship to tasks needing the results of that task. Tasks that need results produced by more than one other task are connected to those other tasks via a join transition and have a successor relationship to those other tasks.
3. Apply the splitting technique, based on the two conditions listed above.
4. Having found the major concurrent task subnets, further decompose each task, represented by a place, into subtasks; and repeat Steps 1 and 2 above to decompose task subnets into subtask subnets that can run concurrently.

In other words, the decomposition technique presented above is used on Petri nets to find the set of subnets that can be executed on a distributed/multiprocessor system.

PRECEDENCE GRAPHS

As previously stated, precedence graphs may be used to show the relationships of tasks within task systems. The method of decomposition of an application into a set of subfunctions, and subfunctions into tasks is used as the first step in the creation of a precedence graph. One then defines the precedence of the tasks based on their required order of execution to assure that a determinate task system results. Predecessor tasks trigger successor tasks, which is indicated by a directed arc in Figure 6. As stated earlier, a task takes inputs, performs some transformation function upon the inputs, and produces outputs. Predecessor tasks produce outputs, which become the inputs to successor tasks. Figure 6 presents a simple precedence graph. Task T1 is the initiator task to the entire task system. It is the immediate predecessor of tasks T2 and T3, which are T1's immediate successors. T3 is the immediate predecessor of T4 and T5, and T6 is the terminator task for the task system. Once one has established a determinate task system, it becomes possible to apply a theorem to find the maximally parallel graph of the task system. Given a maximally parallel graph, one can visually ascertain the maximum number of tasks that may execute in parallel at any given time.

Theorem given previously states the following:

From a given determinate task system C, construct a new system C' that is the transitive closure of the relation

\[ X = (R_T \cap R_T') \cup (R_T \cap D_{T'}) \cup (D_T \cap R_{T'}) = \emptyset \]

then C' is the unique maximally parallel system equivalent to C. In other words, one performs the following steps to find the maximally parallel system equivalent of a task system:

1. Calculate the relation X:
   a. Ranges of T and T'
   b. Range of T and domain of T'
   c. Domain of T and range of T'
2. Take the transitive closure of X by drawing the precedence graph of the relation, X', and eliminating redundant arcs.

The basic idea of this theorem is to take a determinate system and "relax" the determinacy to the point where any further "relaxation" would result in the system's becoming nondeterminate. Therefore if one starts by defining the task system as being entirely sequential—i.e., Task 1 is followed by Task 2, etc., one has defined a nonparallel, determinate task system. One then applies the procedure to find the maximally parallel task system resulting from relaxing the determinacy applied to the system by defining a strictly sequential precedence relationship among the tasks making up the original task system.

This procedure is best understood by example. Assume that a task system is given whose input and output values are represented by the set M, where

\[ M = \{M_1, M_2, M_3, M_4, M_5\}. \]

These five values lie in the various domains and ranges of eight tasks that make up the task system. Table I summarizes which values lie in the domains and ranges of each task.

<table>
<thead>
<tr>
<th>Value</th>
<th>In domain of tasks</th>
<th>In range of tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>1, 2, 7, 8</td>
<td>3</td>
</tr>
<tr>
<td>M2</td>
<td>1, 7</td>
<td>5</td>
</tr>
<tr>
<td>M3</td>
<td>3, 4, 8</td>
<td>1</td>
</tr>
<tr>
<td>M4</td>
<td>3, 4, 5, 7</td>
<td>2, 7</td>
</tr>
<tr>
<td>M5</td>
<td>6</td>
<td>4, 6, 8</td>
</tr>
</tbody>
</table>

Table I—Values in relation to domains and ranges of tasks
The relation, \( X \), is then calculated. As an example, \( M_1 \) lies in the domain of Task 1 and in the range of Task 3. This defines the ordered pair \((1, 3)\). One proceeds to find all the ordered pairs resulting from comparing the domains and ranges of the tasks as defined by Relation \( X \). This results in the set \( X \):

\[
(1, 3), (1, 4), (1, 5), (1, 8) \\
(2, 3), (2, 4), (2, 5), (2, 7) \\
(3, 7), (3, 8) \\
(4, 6), (4, 7), (4, 8) \\
(5, 7) \\
(6, 8)
\]

One then draws a precedence graph, \( G \), of the relation, \( X \), as shown in Figure 7. The transitive closure of \( X \) can be found by by eliminating all redundant arcs in \( G \). For example, Task 1 has an arc to Task 3 and to Task 8. Task 3 has an arc to Task 8. Therefore, the arc from Task 1 to Task 8 is redundant and can be eliminated. Having done this for all redundant arcs, and redrawing \( G \) to produce \( G' \), one has the maximally parallel graph of the task system originally defined to be the strictly sequential task system executing from Task 1 through Task 8. This maximally parallel graph is shown in Figure 8. From Figure 8 one can see that Tasks 1 and 2 can run concurrently and that when they are done, Tasks 3, 4, and 5 can then run concurrently. When Task 4 is completed, Task 6 may start; when Tasks 4 and 5 are completed, Task 7 may run. Finally, Task 8 runs when both Tasks 3 and 6 are completed. Therefore, one could use three processors effectively to implement the example task system on a distributed/parallel system.

**COMPARISON OF TECHNIQUES**

As a final example, to illustrate the previously presented techniques as applied to a real problem, an automotive trip computer/speed control is to be designed. It will monitor mileage, fuel use, and time and maintain speed. The required system can be broken down into 10 tasks that perform the following functions:

<table>
<thead>
<tr>
<th>Task</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Read fuel flow, odometer, time, desired speed</td>
</tr>
<tr>
<td>2</td>
<td>Calculate delta time: current time minus old time</td>
</tr>
</tbody>
</table>
| 3    | Calculate miles per gallon: 
\[
mpg = \frac{(\text{new odometer} - \text{old odometer})}{(\text{new flow} + \text{old flow})/2}
\] |
| 4    | Calculate speed: 
\[
\text{speed} = \frac{(\text{new odometer} - \text{old odometer})}{\text{delta time}}
\] |
| 5    | Calculate throttle value: 
If speed < desired speed minus 2, then increment throttle; otherwise, 
If speed > desired speed plus 2, then decrement throttle |
| 6    | Output throttle value to throttle control |
| 7    | Calculate fuel left: 
\[
\text{fuel left} = \text{fuel left} - (\text{new flow} + \text{old flow}) \times \text{delta time}/2
\] |
| 8    | Read selected function button |
| 9    | Update time, flow, odometer: 
old time = new time 
old flow = new flow 
old odometer = new odometer |
| 10   | Display selected function: odometer, miles per gallon, time, or fuel left |

**Figure 7**—Graph of relation, \( X \)

**Figure 8**—Maximally parallel graph
TABLE II—Task list variables in relation to domains and ranges

<table>
<thead>
<tr>
<th>Variable</th>
<th>Domain of task</th>
<th>Range of task</th>
</tr>
</thead>
<tbody>
<tr>
<td>New odometer</td>
<td>3, 4, 9, 10</td>
<td>1</td>
</tr>
<tr>
<td>New time</td>
<td>2, 10</td>
<td>1</td>
</tr>
<tr>
<td>Desired speed</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Old time</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Delta time</td>
<td>4, 7</td>
<td>2</td>
</tr>
<tr>
<td>Mpg</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>Old odometer</td>
<td>3, 4</td>
<td>9</td>
</tr>
<tr>
<td>New flow</td>
<td>3, 7, 9</td>
<td>1</td>
</tr>
<tr>
<td>Old flow</td>
<td>3, 7</td>
<td>9</td>
</tr>
<tr>
<td>Speed</td>
<td>4, 5</td>
<td>4</td>
</tr>
<tr>
<td>Throttle</td>
<td>5, 6</td>
<td>5</td>
</tr>
<tr>
<td>Throttle control</td>
<td>—</td>
<td>6</td>
</tr>
<tr>
<td>Fuel left</td>
<td>7, 10</td>
<td>7</td>
</tr>
<tr>
<td>Function index</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>New time</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Display output</td>
<td>—</td>
<td>10</td>
</tr>
<tr>
<td>Read input</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

From this task list a table of variables (Table II) is created that specifies the variables themselves and whether they lie in the domain and range for each of the tasks. From this table it becomes possible to construct either a Petri net or a maximally parallel precedence graph. If the latter approach is taken, determinacy is established by requiring the strictly sequential execution of Tasks 1 through 10 in that order. Figure 9 presents the Petri net constructed from the above table, and the subnets broken out from it. As can be seen, four subnets are possible, the fourth being one task. This implies that the optimal number of concurrently running processors is three. Figure 10 presents the calculation of the $X$ relation and the resulting precedence graph. Figure 11 presents the result of taking the transitive closure of the graph in Figure 10—i.e., elimination of all redundant arcs—to produce the maximally parallel precedence graph for the above task system. Depend-
ing on the speed of execution of Tasks 2 and 3, the task system could function on at most three processors concurrently.

One assumption made here is that the dependencies of some of the variables upon being updated by Task 9 before any of the tasks could run is eliminated by an initialization step in Task 1. What this means is that the first execution of the task system starts with Task 1 as noted and that after this the results of Task 9 are used, since the task system is basically completed at Task 10 and then recycles back to Task 1. This recycling can be represented, or left out, without affecting the way the task system would function. In this case it was left out for clarity.

Having performed the above exercise, the authors found the precedence graph approach to be easier to apply, since from Table II the calculation of the X relation is straightforward. From the X relation a set of ordered pairs could be defined which in turn provides the transitive closure and hence, could easily produce a precedence graph. The production of the Petri net proved to be more difficult and time-consuming, since no calculation could be performed to create the ordering for the graph. However, both procedures could be computerized to perform the work after the initial task specification and identification of variables required had been performed.

The Petri net, in the authors' opinion, presents more of the details of the intertask relationships, since the transitions indicate that variables are input and output to cause the transitions to fire. However, as stated above, the approach using the production of a maximally parallel precedence graph was easier to implement. Perhaps a method of combining the two techniques would be possible; however, that lies beyond the scope of this paper.

CONCLUSION

This paper has presented the concept of multitasking as a design methodology for the production of software to execute on a distributed/multiprocessor processing system. Two graphical techniques were presented: the Petri net and the precedence graph. These techniques offer a means of visualizing the flow of control in such a software system. Both Petri nets and precedence graphs have been analyzed and methods found to find concurrent segments of these graphs. However, in the final analysis, all these methods still require the designer to be cognizant of potential concurrency in the decomposition of his or her application into tasks to which the above techniques may be applied.

REFERENCES
