# BILOGIC PRECEDENCE GRAPH MODELS

Gary J. Nutt

CU-CS-363-87 May 1987

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

# BILOGIC PRECEDENCE GRAPH MODELS

Gary J. Nutt

CU-CS-363-87 May, 1987

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

## ABSTRACT

This technical report provides a definition for a class of precedence graphs to be used to model parallel programs. The report illustrates the modeling concepts by including an example of the graphs to represent the chaotic relaxation algorithm for solving a system of linear equations.

# 1. INTRODUCTION

Constructing effective programs for distributed architectures is a difficult problem for several reasons. A program is a reflection of an algorithm to solve a problem; conventional understanding of algorithms tends toward sequential solutions (as a consequence of the mechanisms one has had available for implementing the algorithm). Also because of the types of conventional I/O devices, the physical representation of the algorithm is often expressed in some linear, textual form; visual representations tend to be better suited for human understanding.

This report defines a class of precedence graphs which can be used to model the functionality of a distributed computation. The goal of the model is to provide a visual model that is natural and intuitive for expressing abstract algorithms, yet that has a formal underpinning. This will make the mode useful as a graphic representation while it maps onto a well-defined model that can be used for analyzing the functionality of the algorithm. The intent is for the model to be useful during all phases of design of a parallel program, first by providing a flexible testbed for laying out basic ideas, then by providing a framework for refining the ideas. During the refinement, the model should be amenable to various forms of analysis, including animation and simulation. In the later stages of the development, the model may be useful as an aid in debugging the program and in measuring its performance.

The model is similar to a number of other precedence graph models in that it is composed of nodes and edges, where the edges represent processing activity and the edges represent precedence among the activities. It differs from other models in that it incorporates both disjunctive (OR) and conjunctive (AND) precedence logic. The model addresses data flow as well as control flow. It also makes provision for hierarchical decomposition of activities into functional interpretations.

The purpose of this note is to provide a definition of the model, and to illustrate its use through an extensive example.

## 1.1. Background

Precedence models are useful for planning the interactions among a set of activities that make up a system. They are particularly useful during the formative stages of the design, when the individual activities being ordered by the edges are "fuzzy" in their meaning as the designer iterates over the organization. An important property of design aids is that they allow flexibility during the early stages of planning, yet encourage consistency among refinements to early design attempts. As the model is constructed, the activities in the model become more precise, through refinement and redefinition. Precedence graphs have become a standard planning tool for organizing large sets of activities in other distributed systems, eg., a manufacturing process, an engineering plan. There is considerable evidence that they are extremely useful for organizing a set of activities in an algorithm as well.

There are many variants of precedence models, generally differing in the mechanisms allowed for representing logical combinations of partial orderings of the tasks. Our experience with these models indicates that conjunctive and disjunctive logic are useful

for representing discrete systems, and that the logic can be added to the primitives of the modeling system by defining specialized node types corresponding to the desired logical combinations of control flow. Therefore, the modeling system has evolved from a more basic modeling system through other intermediate models.

The foundation of the model for representing task precedence is Petri nets [8]. They have been used to model a wide variety of discrete systems, ranging from formal languages, to software, to computer architectures. An important successor model to Petri nets are dataflow graphs which have also been used heavily for architecture studies [1]. Petri nets were chosen as the underlying model because they are an intuitive mechanism which can represent parallel and sequential operation in the static representation, and because they are useful to represent instances and transactions in their dynamic form.

E Nets extended Petri nets by providing a mechanism for modeling deterministic behavior, and by adding interpretations to the transitions [3]. E nets were used primarily for constructing models of machine architectures. E nets are only one of many different extended Petri net-like models, and are mentioned here primarily because of their influence on the current model.

Information Control Nets (ICNs) extended E nets so that they explicitly represented data flow in addition to the control flow [2]. ICNs were focused on a particular application area -- office procedures -- rather than being applied to discrete systems in general. These models were automated by building a system to create and exercise instances of the model. *Quinault* was a graphics editor and animation system for ICNs [7], and *Strawberry* was a prototype system to distribute the animation/simulation across a network of machines [6].

## 2. MODEL DEFINITION

A *bilogic precedence graph,* (bpg), is a graph, <N, E> composed of a set of *nodes,* N, and a set of *edges,* E, where

$$N = T \cup R, T \cap R = \phi$$

and T is a finite, nonempty set of *tasks,* and R is a finite set of *data repositories.* Tasks represent units of processing activity, while data repositories are intended to represent storage media for information. The edge set is also the union of two disjoint sets, ie.,

$$E = C \cup D, C \cap D = \phi$$

The bpg can be thought of as the union of two subgraphs, where the pair <T, C> is a *control flow subgraph,* cfs, representing the precedence among the set of tasks, and <N, D> is a bipartite *data flow subgraph,* dfs, representing data flow among the tasks with intermediary storage (memory, message queue, etc.).

## 3. CONTROL FLOW SUBGRAPHS

It is required that the cfs represent sequence, conjunctive logic, and disjunctive logic for precedence relationships among tasks. For example, suppose that A, B, and C are tasks. Then the model must represent the following cases:

(1) Task A must terminate before B initiates, ie., serial control flow.

(2) Either B or C (exclusively) may be initiated after A terminates (disjunctive output logic on A, ie., control flow merge).

(3) Both B and C can be initiated after A terminates (conjunctive output logic on A, ie., control flow fork).

(4) Either A or B must terminate before C can initiate (disjunctive input logic on A, ie., control flow alternative).

(5) A and B must terminate before C can initiate (conjunctive input logic on A, ie., control flow join).

In order to represent these fundamental types of control flow, the set of tasks, T, is decomposed into smaller sets with different I/O logic. As a simplification, only three subsets are allowed (rather than five).

$$T = S \cup O \cup A,$$

where S is a finite set of *single-entry-single-exit tasks,* ie., tasks which have only a single input edge and a single output edge from the control flow edge set, C; O is a finite set of *disjunctive tasks,* ie., tasks whose input and output control flow edges are defined to use disjunctive (exclusive OR) logic. A is a finite set of *conjunctive tasks,* ie., tasks whose input and output control flow edges are defined to use conjunctive (AND) logic.

A cfs can be represented graphically by choosing distinct representations for members of S, O, and A, then constructing the graph composed of vertices interconnected with edges from C. Let members of S be represented by large, open circles, O by small open circles, and A by small, filled circles. Then Figure 1a represents sequential operation among tasks a and b, Figure 1b represents disjunctive output, Figure 1c represents conjunctive output, Figure 1d represents disjunctive input, and Figure 1e represent conjunctive input control flow.

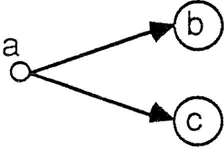
Let the cfs be <T, C> where

$$T = S \cup O \cup A,$$

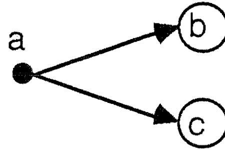
S = {a, b, c, d, e, f, g},
O = {h},
A = {i, j},
C = {(a, i), (i, b), (i, c), (b, j), (c, d), (d, j),(j, e), (e, h), (h, f), (h, g)}.

The the graph shown in Figure 2 represents the following control flow among the tasks: After a terminates, b and c are initiated. When c terminates, d is enabled. When both b and d terminate, then e can initiate. When e terminates, one of f or g will be initiated.

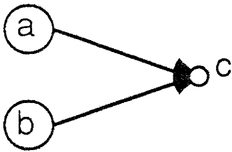
1a



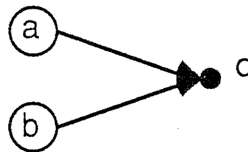

1b




1c




1d




1e

Figure 1

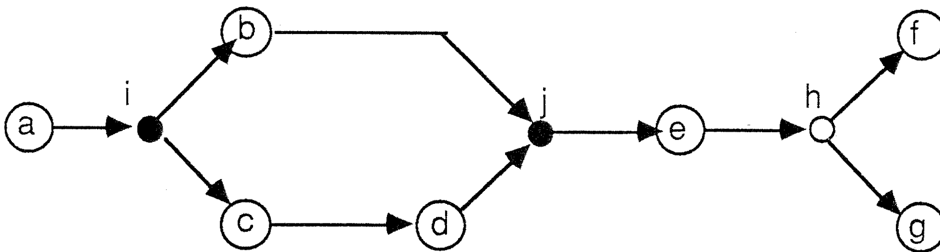



Figure 2

### 3.1. Task Refinement

A cfs is one model of a system at a level of detail defined by the number of nodes in the graph and their interconnectivity. The level of detail can be increased by refining the definition of a node, or by providing an interpretation for the nodes. Node interpretations are discussed in the next section, while we focus on task refinement in this subsection. A task is refined by defining another control flow subgraph to replace the original task. The new sub-cfs must be a set of (sub)tasks that, together, represent a structure corresponding to the original task in terms of its interactions with the other tasks in the original cfs. Within the sub-cfs, the precedence among subtasks is represented exactly like it is among tasks. However, since the sub-cfs represent a refinement of a task, and since the task meets certain precedence constraints, it is necessary for the sub-cfs to have boundary conditions that match the precedence constraints of the original task. For sequential tasks, this means that the subtask graph is a single-entry-single-exit graph. For conjunctive and disjunctive tasks, the subtask graph must not only match the number of incoming and outgoing edges, the logic of the subtask graph must match that of the parent task.

If one were to provide a refinement for task h in Figure 2, then the sub-cfs must have a single task node which has no predecessor, and two task nodes which have no successor. Furthermore, the sub-cfs must be constructed so that whenever control flows into the input task, then it flows out only one of the two output tasks (matching the semantics of the original task h). Figure 3 is an example refinement of task h. Control flows into task h[1], then may flow out of the sub-cfs to the original task g, or flow to task h[2]. After h[2], then control moves to h[3] and then on to the original task f.

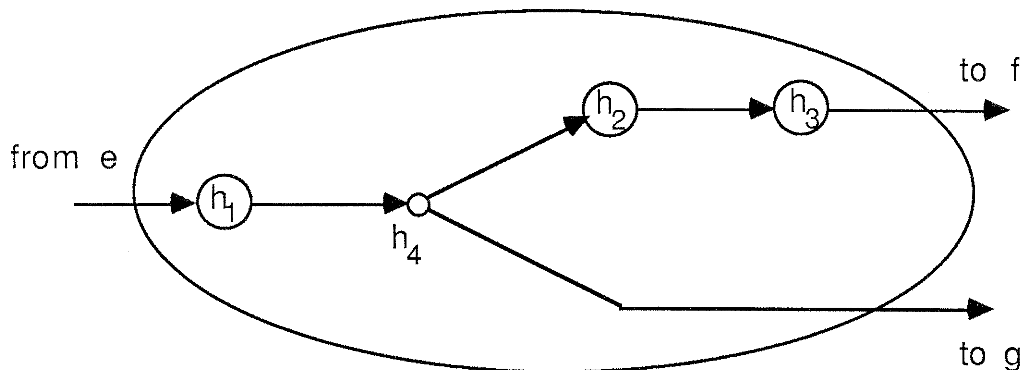



Figure 3

## 4. DATA FLOW SUBGRAPHS

The overall task model requires that one be able to represent information sharing among tasks. In some cases, the intercommunication is accomplished by sharing memory between two tasks, and at other times it might be accomplished through message passing. In order to make that sharing explicit, data flow edges are added to the precedence graph. And to take into account that the sharing may not occur at the same instant in time, the edges have an interposed data repository.

The graphical interpretation of a dfs is defined by representing members of R by rectangles, and the edges interconnecting tasks and data repositories by thick lines. If task a and task b share information -- through procedure call, message passing, shared memory, etc. -- an edge should be placed in the dfs between a and x, and between x and b, where x is a member of the set of data repositories, R. If a produces the information and b consumes it, then the edge should be directed from a to x and from x to b. If either may produce the information, then four edges should be included, one from a to x, x to b, b to x and x to a. For the dfs shown in Figure 4, and corresponding to the cfs from the previous section, R = {w, x, y, z}. Figure 4 shows that tasks b and e share information, and that b produces the information for e's consumption; c produces information for d, and d produces information for f and g.

The dfs for the example contains isolated nodes. Generally, it is more useful to consider the cfs or the bpg for a model than to look only at the dfs. Figure 5 is the bpg for the example shown in Figure 4.

### 4.1. Data Repository Refinement

In the example above, we have not supplied an interpretation for the tasks, nor for the data repositories. Therefore, the data repositories characteristics may be of no interest in the case that the intercommunication is interpreted as parameter passing. If the two tasks are interpreted as procedures, then the shared data repository may be shared memory or a parameter on a stack. If the two tasks are interpreted to be processes, then the data repository may represent a message queue or some form of backing store.

Data repository nodes may be refined, just as task nodes can be refined. It is only necessary to maintain the semantics of the edges in a manner analogous to that shown for task node refinements. Thus, a refinement of a data repository has no incoming nor outgoing control flow edges, yet the refined data flow subgraph may contain subtasks and subrepositories. The extended example, later in the technical report, provides an example of a data repository refinement.

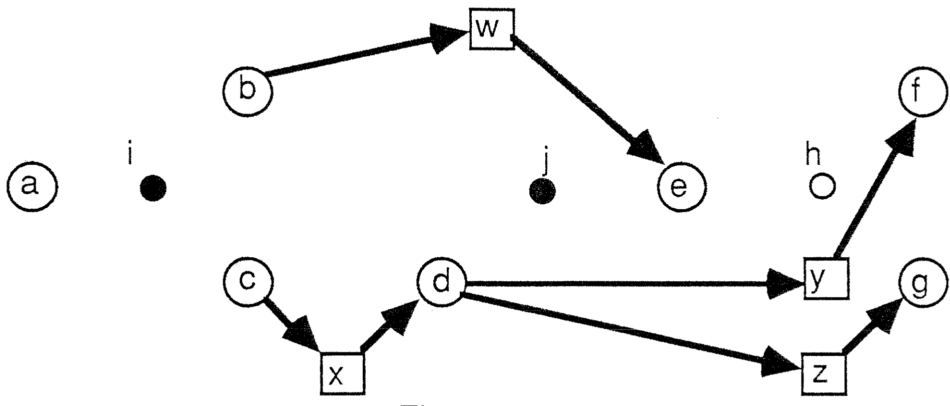


Figure 4

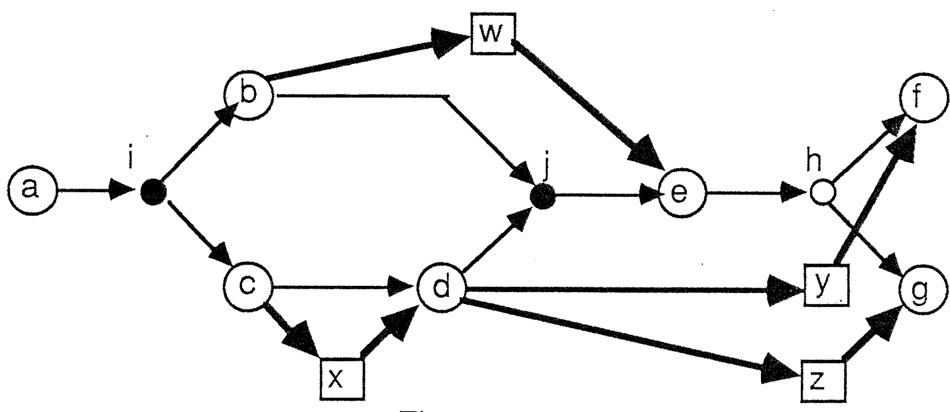


Figure 5

## 5. MODEL INTERPRETATIONS

Task and data repository refinement are useful to describe increasing levels of detail about the interaction among nodes. However, none of these refinements would provide a functional interpretation to the model. While the refinement satisfies the need to investigate structural relationship among nodes, it is not useful for predicting the performance of the approach. Functional interpretations are required in order to consider performance.

As mentioned above, nodes can also be modeled by blocks of code. For any interpretation that is a block of code, there is a higher level model which describes the precedence among the blocks of code. If the description is a block of code, then it must behave in a manner dictated by the corresponding activity or task, ie., the code must recognize a set of entry points that correspond to the input edges of the node, and a set of exits which correspond with the output edges of the parent node.

It is reasonable to use an object paradigm for describing the blocks of codes. In this paradigm, each incoming arc of the parent node maps to an incoming message for the object. Similarly, each outgoing arc should have a counterpart in the code which emits a message. The objects message patterns are more constrained than in conventional object-oriented programs, since these objects must accurately simulate the logic of the parent nodes.

### 5.1. Interpreting a Task Precedence Graph

Because of the hierarchy of descriptions, one views a precedence graph at some level of refinement. We define the most abstract task level of description to be Level 0, with successive refinements being numbered by succeedingly higher integers.

Suppose that we are interested in inspecting the interpretation of each node in a model at Level i. Then, the following rules hold for finding the interpretation:

(1) If Level i is a precedence graph, then the interpretation is supplied by Level i+1.

(2) If Level i is a procedure, then the interpretation is supplied by the procedure declaration.

(3) If Level i is undefined, then Level i is interpreted to be the same as for the corresponding node at Level i-1.

Therefore, a model can only be interpreted whenever each task ultimately has some procedural definition refinement.

### 5.2. Simulation Interpretations

Suppose that our purpose were to model the performance of a system for which we had constructed a hierarchical bpg. Then the basis of the system performance would be some combination of the performance of the individual parts, as dictated by the components of the bpg. The bpg nodes define the order in which tasks are executed and data repositories

are used. The interpretation should provide performance descriptions for the individual parts. Thus, each task in the bpg could be interpreted by a probability service time distribution; during simulation, the distribution is sampled to reflect a service time for that task to execute. It is also necessary to use interpretations to control the number of iterations in loops, etc. It is then feasible to model the implementation through simulation techniques, and to illustrate the simulation through animation of the bpg.

### 5.3. Functional Interpretations

While simulation is useful to study the performance, it does not provide true functional descriptions of the tasks. However, the procedural descriptions that may appear as a refinement of any node in a bpg are not restricted, thus they are a mechanism for including procedures which are invoked under the direction of the control flow model. This means that the control flow model can be used as the task scheduler for a real implementation of a system modeled by the bpg.

## 6. AN EXAMPLE: CHAOTIC RELAXATION

Chaotic relaxation is a method of solving a linear system of n equations in n unknowns, AX=B. The general idea is that for a particular class of systems of equations, it is possible to repeatedly solve the system for each x[k] simultaneously, using the current best guesses at the values of x[j], where k <> j. For the given class of equations, the solution vector, X, will converge.

When the chaotic relaxation algorithm is executed as a single-process program, by solving for only a single x[i] at a time, then the process is usually called Gauss-Seidel iteration. It is possible to solve for the n values of the vector X simultaneously by using a program made up of n processes. Each process repeatedly computes one of the x[i], based on the most recently values of the other elements, x[j]. Again, the solution vector will converge.

This is a system that is well-suited to parallel computation, since it is easy to decompose the computation into processes, and since the interprocess communication pattern is easily understood. Chaotic relaxation has been implemented in a workstation/LAN environment using both point-to-point and broadcast message passing among the processings computing the individual values of X [4]. The program that implements the approach typically determines the dimension of the system of equations, n; reads the values for the matrix A and the vector B into its memory; then spawns n identical instances of the following process class. The creating process then begins evaluating AX and comparing the result to B, for the current value of X. When the error in the solution is judged to be sufficiently small, the process sets a global flag to tell the n spawned processes to halt. When all processes have halted, the main process can save the result and terminate.

Each instance process is created with a copy of A and B. The process first obtains a copy of the current best-guess at the value of the solution vector, X, then it computes x[k] from the guess vector X and A and B. After computing x[k], the process disseminates the value of x[k] so that the halting evaluator and the other n-1 processes can used the newly-computed value. If the global stop flag has been set by the termination process, then the process halts; otherwise, it obtains the updated values for the vector X, and continues the process above.

### 6.1. Modeling Chaotic Relaxation

The following discussion is a design scenario for implementing the chaotic relation algorithm, using the bilogic precedence graphs as a modeling tool. The essence of the algorithm is that it is possible to simultaneously solve for the x[i] using n different tasks. These tasks are all started at the same time, and the computation is complete when all of the tasks have completed. Thus, Figure 6 is an initial cfs of the chaotic relaxation algorithm. The task labeled "initialize" models the process of reading the values of A and B, initializing variables, etc.; the task labeled "clean up" represents the process of reporting the results. The three tasks labeled "compute x[i]" for i=1, 2, and n -- and the ellipsis -- represent the n tasks which repeatedly compute the values of the various components of the solution vector, X. The model captures the basic structure of the chaotic relaxation algorithm, but it ignores a great deal of detail. How is information shared among the n tasks? How is a halting condition determined? How does one compute x[i]? The tasks should repeatedly compute x[i], but that is not represented. To answer these questions, the model needs to be refined.

---

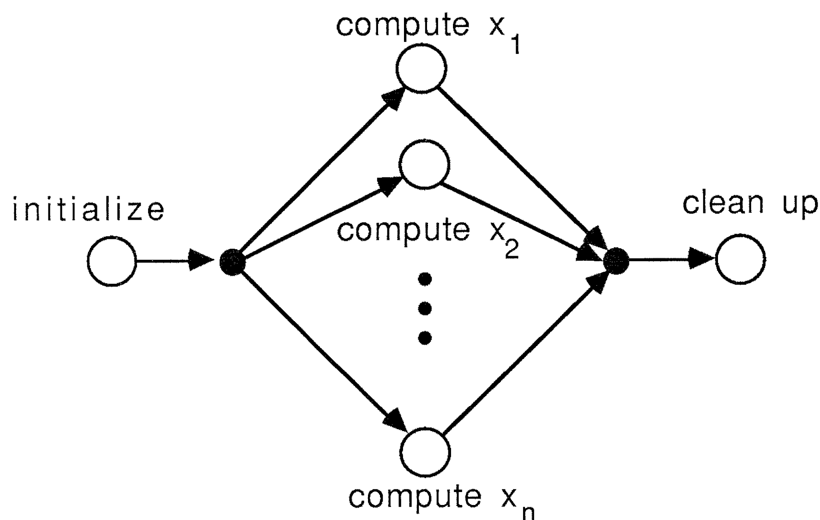



Figure 6

The process by which the n tasks share information is an important criteria for the performance of the algorithm in the workstation/LAN environment, [4]. Given that the algorithm is implemented in a ethernet LAN environment, and that tasks might map to processes which exist on independent workstations, then one can build a common data repository at an n+1st workstation -- called the broadcast algorithm -- or distribute newly computed results to the n individual processes -- called the point-to-point algorithm. Figure 7a is a bpg representing the point-to-point approach, and Figure 7b represents the broadcast approach.

The halting condition is determined by evaluating the system AX=B with the most recent approximations to X. Figure 8 models this approach (using the data broadcast algorithm). Notice that the new task, "halt test," is assumed to execute in parallel with the n worker processes, and that it reads the X data repository and writes a "halt" repository. The "halt" repository may be implemented as a flag in shared memory, a signal among processes, a broadcast message, etc.; that is not determined with this level of model.

The changes to the original model in Figure 6 have been additions to the original bpg. Figure 6, or the cfs subgraph for the bpg shown in the figure, represent most of the art of the algorithm. Details can be added by refining the nodes in the bpg. In order to model the steps in computing x[i], one can construct a refinement of the tasks labeled "compute x[i]." Notice that the subject tasks has a single control flow input arc and a single control flow output arc; it also has two data flow input arcs and a single data flow output arc. Therefore, a refinement of the task -- another bpg -- should have the same configuration at its boundaries. Consider the bpg shown in Figure 9. The large, dashed circle encapsulating the bpg represents the original task from Figure 6, and it has the same orientation of data and control flow edges as the original task. Within the dashed line are 5 new tasks and 2 new repositories. The bpg indicates that control flow initiates processing through the first OR task (which also merges the feedback loop control), then causes the task "get X" to be executed. Once the values of X have been obtained, they are passed to the task labeled "evaluate x[i]" where the actual computation takes place. Once x[i] has been determined, then the updated version is written to the global store, and the final OR task determines whether or not the global halting condition has been met. If it has, then control flows out of the subgraph, otherwise control loops back to "get X."

In order to adequately model the broadcast mechanism it is necessary to refine the data repository shown in Figure 8; Figure 10 is one model for that process. This model shows 2n+1 subtasks for the repository subgraph. We have used an abbreviation of the control flow subgraph by excluding all control flow edges from the figure; the intent is that the 2n+1 tasks run asynchronously. The proper picture would show a feedback loop on each task.

Similar refinements can be done for the other nodes in Figure 6, eg., "initialize," and "halt test," as shown in Figures 11 and 12, respectively.

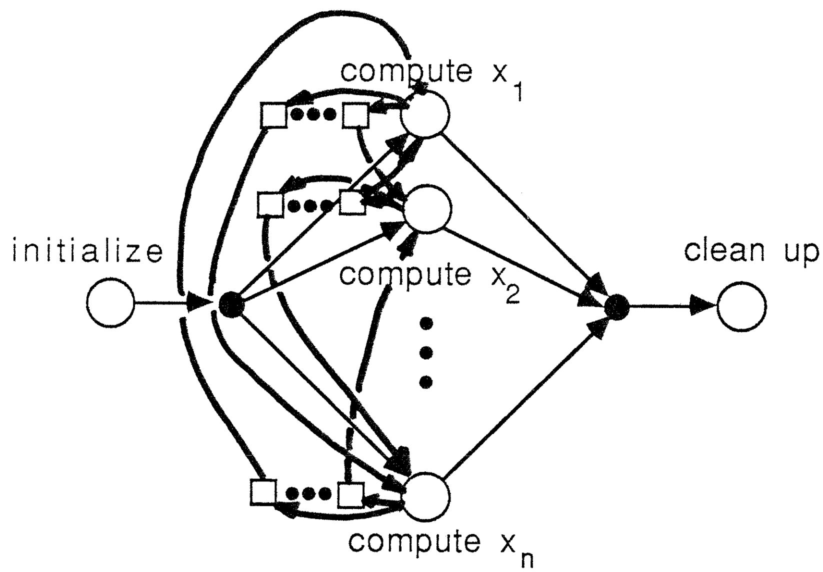



Figure 7a

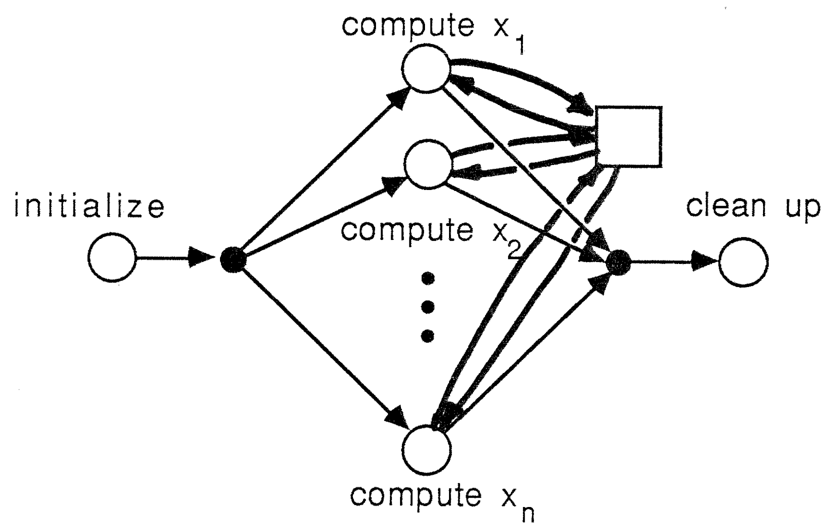



Figure 7b

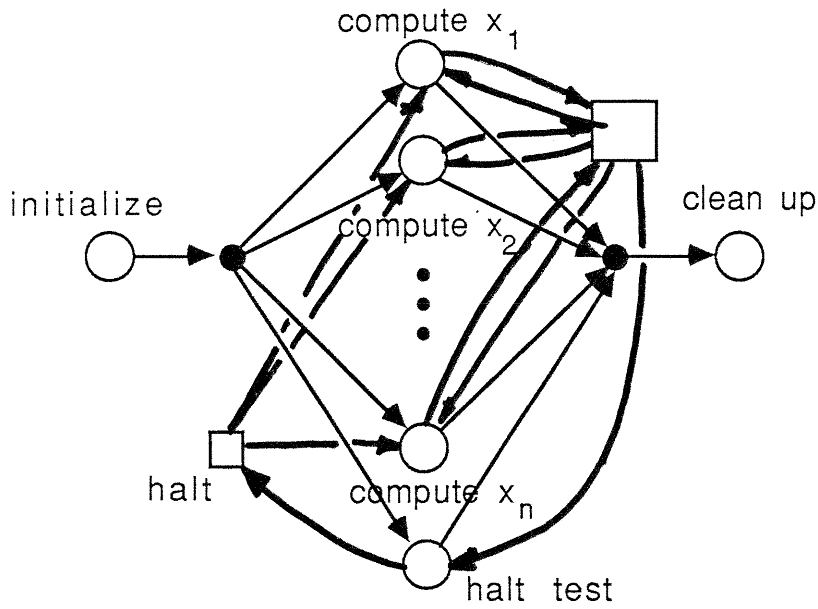



Figure 8

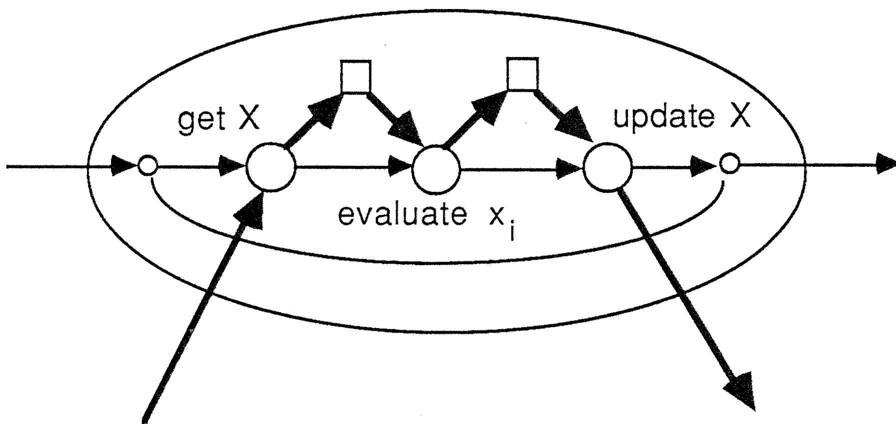



Figure 9

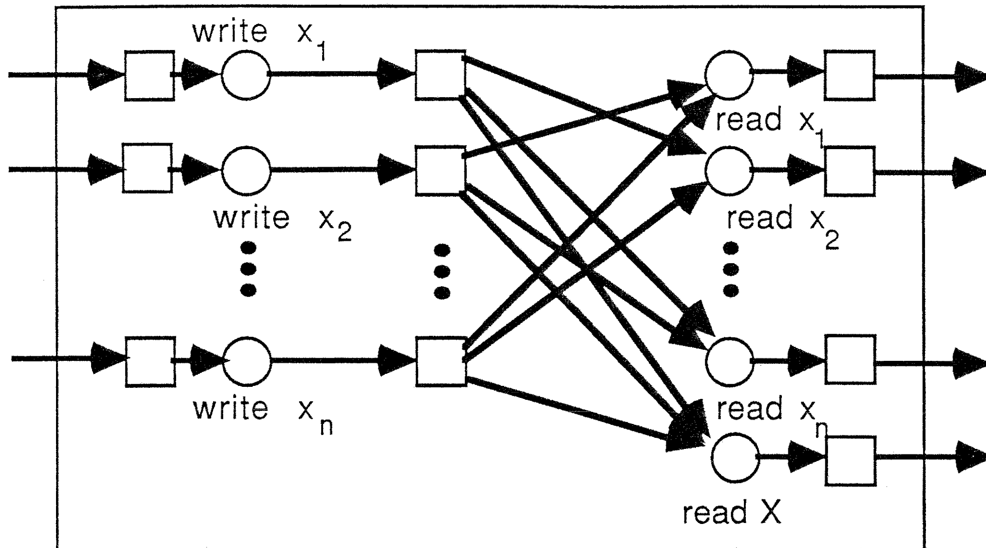



Figure 10

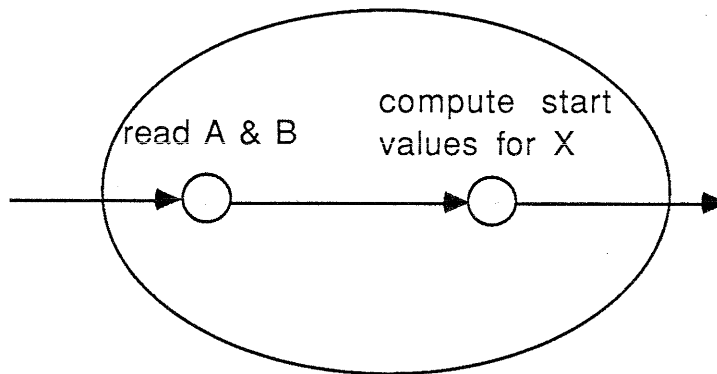



Figure 11

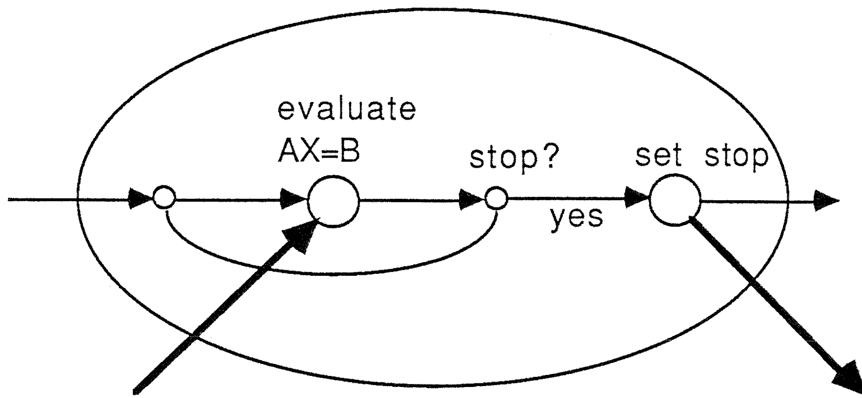



Figure 12

### 6.2. Simulation Interpretations

In order to simulate the performance of the chaotic relaxation algorithm under the two schemes for distributing values of X, one can simulate the performance of the the two models shown in Figures 7a and 7b. The control flow subgraph is the same in the two models, although the interpretations will be different for the two cases.

#### 6.2.1. The Point-to-Point Algorithm

The estimate for the time to initialize the system is going to be a function of the degree of the linear system, and possibly of the time required to obtain resources to run the program. The latter problem is ignored, and the time to execute the "initialize" task is f(n) = overhead + k*n. The "clean up" task can be interpreted similarly, with the expression for the simulation time being g(n) = overhead + k*n.

The time to "compute x[i]" is dependent upon the time to get X, solve for x[i], and then to send the updated value to each of the other tasks (see Figure 9). The other factor is the number of iterations -- or the amount of time spent iterating -- within each "compute x[i]" task. For the model in Figure 7a, one can estimate the execution by the following expression:

```
sample(service_distribution) + (n-1)*xmit_time
```

A more detailed interpretation would cause each "compute x[i]" to interact with data repository nodes as follows: Each time the simulation interprets "compute[i]", execute the following code segment:

```
wait(sample(service_distribution));
for i=1 to n-1
{
        signal(data_repository[i]);
        wait_for_signal(from_data_repository[i]);
};
```

Each data repository in Figure 7a then has an interpretation as follows:

```
wait_for_signal(from_evaluator[i]);
wait(sample(service_distribution));
signal(evaluator[i]);
```

#### 6.2.2. The Broadcast Algorithm

The "initialize" and "clean up" tasks will have the same interpretation as those in the Point-to-Point simulation. The simple model of "compute x[i]" will use an interpretation such as:

```
sample(service_distribution) + broadcast_time
```

A more detailed interpretation for "compute x[i]" follows:

```
wait(sample(service_distribution));
signal(data_repository);
wait_for_signal(from_data_repository);
```

And the data repository must simulate the broadcast facility. If we ignore conflicts, then the data repository uses the same interpretation as in the Point_to_Point interpretation

with a different service distribution.

### 6.3. Functional Interpretation

A functional interpretation would ordinarily be applied to the most detailed bpg. Logically, this is equivalent to the bpg shown in Figure 13 (for the Broadcast algorithm). The following pseudo-code segments provide an example of how a functional interpretation could be supplied for the bpg.

```
"read A & B"
{
        -- initialize the program --
        get(n);
        for i = 1 to n
          for j = 1 to n
                get(A[i, j]);
        for i= 1 to n
                get(B[i]);
}


"compute start values for X"
{
        for i = 1 to n
                put(compute_initial_guess(X[i]), central);
}


"spawn evaluators"
{
        for i= 1 to n
        {
        -- Create a new token for each new control flow sequence --
                spawn_token(i);
                send_token(out_arc[i]);
        };
}


"get X"
{
        -- retrieve the current estimates for X[j] --
        for i = 1 to n
                get(X[i], central);
}
```

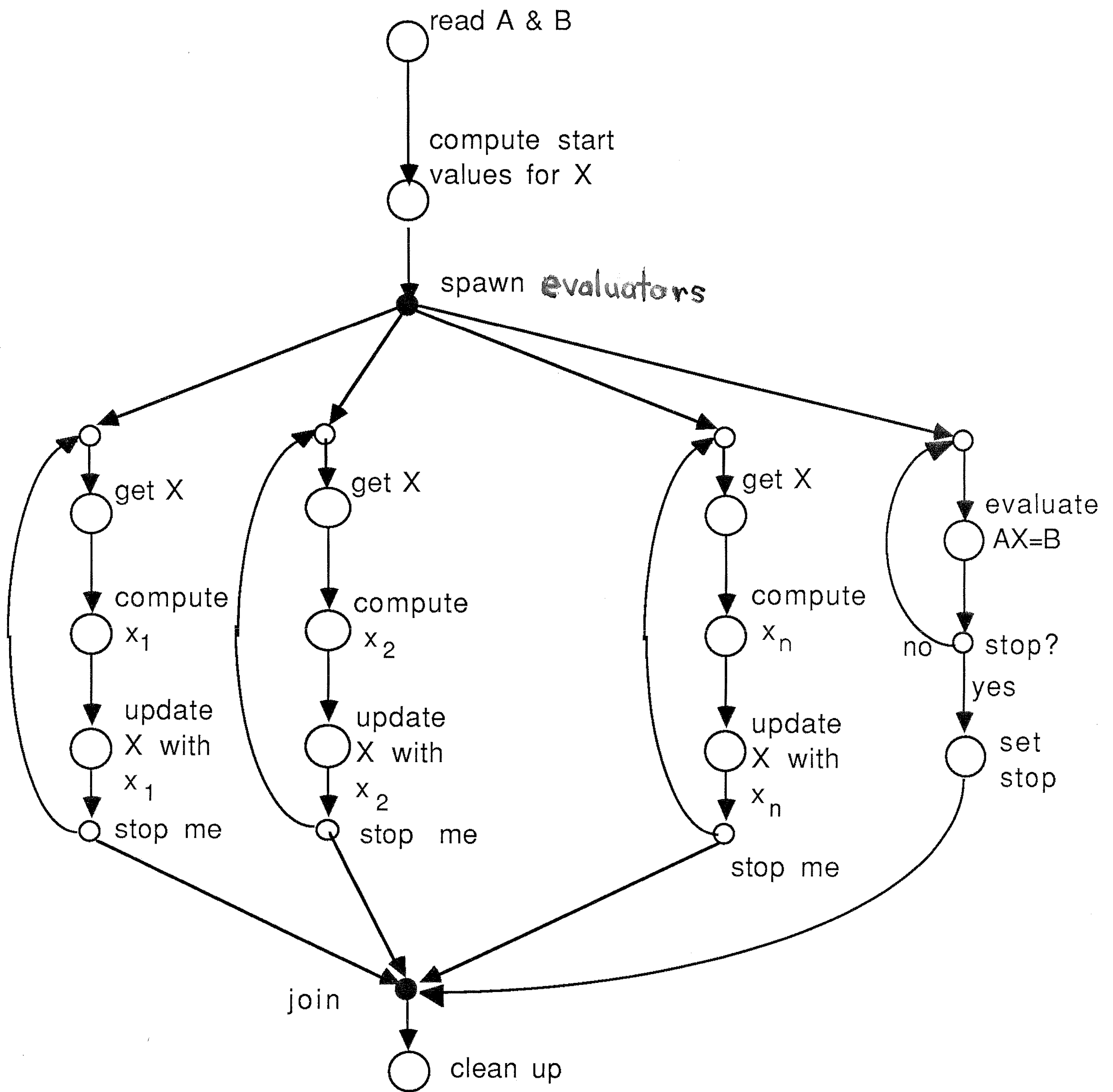



Figure 13

```
"compute x[i]"
{
        solve_for(X[i]);
}


"update X"
{
        -- restore the latest estimate for X[i] --
        put(X[i], central);
}
"stop me"
{
        if (time_to_stop) then out_arc["join"]
        else out_arc["get X"];
}


"evaluate AX=B"
{

        C = evaluate(AX);
}


"stop?"
{
        if ((B-C) < error) then out_arc["set_stop"]
        else out_arc["evaluate AX=B"];
}


"set stop"
{
        set_stop_flags;
}


"join"
{
}


"clean up"
{
        put(X);
        terminate();
```

```
}

"write x[i]"
{
        set_dr(x[i], value);
}

"read x[i]"
{
        for i = 1 to n
                put(value(x[i], out_arc[...]);
}

"read X"
{
        for i = 1 to n
                out(value(x[i], out_arc[...]);
}
```

## REFERENCES


[1] Dennis, J. B., "Data Flow Computation," *Control Flow and Data Flow: Concepts of Distributed Programming,* ed. by M. Broy, Springer Verlag, 1985, pp. 345-398.

[2] Ellis, C. A., "Information Control Nets: A Mathematical Model of Office Information Flow," *Proceedings of 1979 Conference on Simulation, Measurement and Modeling of Computer Systems,* (Boulder, Colorado, August 13-15), ACM, New York, pp. 225-239.

[3] Ellis, C. A. and G. J. Nutt, "Office Information Systems and Computer Science," *ACM Computing Surveys,* Vol. 12, No. 1, (March, 1980), pp. 27-60.

[4] Gardner, T. J., I. M. Gerard, C. R. Mowers, E. Nemeth, and R. B. Schnabel, "DPUP: A Distributed Processing Utilities Package," Department of Computer Science Technical Report No. CU-CS-337-86, July, 1986.

[5] Noe, J. D. and G. J. Nutt "Macro E-Nets for Representation of Parallel Systems," *IEEE Transactions on Computers,* Vol. C-22, No. 8, (August, 1973), pp. 718-727.

[6] Nutt, G. J., "An Experimental Distributed Modeling System," *ACM Transactions on Office Information Systems,* Vol. 1, No. 2, (April, 1983), pp. 117-142.

[7] Nutt, G. J. and P. A. Ricci, "Quinault: An Office Modeling System," *IEEE Computer,* Vol. 14, No. 5, (May, 1981), pp. 41-57.

[8] Peterson, J. L., "Petri Nets," *ACM Computing Surveys,* Vol. 9, No. 3, (September, 1977), pp. 223-252.

[9] Raeder, G. "A Survey of Current Graphical Programming Techniques," *IEEE Computer,* Vol. 18, No. 8, (August, 1985), pp. 11-25.