# MODELING DISTRIBUTED COMPUTATIONS

Isabelle M. Demeure
Gary J. Nutt

CU-CS-371-87 July, 1987

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

This research has been supported by NSF cooperative agreement DCR-8420994.

## ABSTRACT

This technical report describes a means for characterizing and modeling the structural framework of schedulable units of computation in a distributed computation, which is called the *process architecture* for the computation. The characterizations focus on the complexity of the structure of the computation in terms of the amount of code that has to be written to implement the computation, the similarity of different parts, and the degree of dynamism in component creation and termination. The discussion includes several diverse examples of distributed computations along with models to represent different aspects of each.

The process architecture model introduced in this paper is still under development. We expect that it will change as our knowledge about the topic increases. This technical report is a working paper describing the model, and can be expected to change from time to time.

# 1. INTRODUCTION

There is an ongoing research effort in distributed computing at the University of Colorado. This effort is concerned with writing parallel, distributed computations, as well as supporting systems and tools to deal with these computations. The environment of research includes a spectrum of physical systems including multiprocessors and networks of workstations.

This paper describes a model of process structure and interaction of the components that make up a parallel, distributed computation. This study started with a desire to understand more about the nature of parallel, distributed computations. As a result, we explored various characteristics of the computations, and a model began to emerge. It is intended to be used to represent computations, to study their behavior, and to serve as the basis of a set of tools to assist the user. We will employ these tools in creating, monitoring, debugging and measuring the performance of distributed computations. We intend to use a graphical representation of this model as a visual support for our tools.

## 1.1. Background

Given a problem definition, a programmer who is assigned the task of writing a sequential program must identify the data structures needed and find an appropriate way to represent them, find a good modular decomposition of his program, and design the algorithms needed to implement the required functionalities. A programmer who is asked to write a parallel, distributed computation has additional work to perform. He must derive an appropriate set of processes that will cooperate to produce the expected results, identify which processes have to share information with one another and what are the appropriate types of communication between them. He must also find out which processes need to be created at initialization time, which must or can be created later during the execution, which will terminate at the end of the computation and which will terminate before. In other words, he must come up with a *process architecture* for the computation. In the same way that, for sequential programs, there are different possible modular decompositions, different possible designs for the data structures, and different possible implementations of algorithms, there are different ways of breaking a distributed computation into processes, of picking the topology of communications among them, of choosing the time at which each will be created and destroyed. We have proposed a methodology that focuses on the design of the process architecture [2, 3]. The methodology entails the identification of tasks in the solution, finding relationships among the tasks, and defining the basic elements of the process architecture; we have used binary logic precedence graphs to accomplish the preliminary part of the methodology [4]. This paper concentrates on means of viewing process architectures, and in considering abstractions of the basic architecture which focus on code modularization, computational semantics, and complexity.

## 1.2. Overview

In general, we assume that the computational environment includes several processors and a mechanism for communications among them. Some interprocess communication facility (message passing or shared memory) must be available, and a fork/join mechanism such as the one provided by Unix is assumed.

A given implementation of a distributed computation can be characterized in several ways. The number of types of processes which define the process architecture is a measure of the amount of code required to implement the computation; it also helps to identify common components. (Note that the

same code can be used for two different processes in which case the processes are of the same type.) The types and topology of communications describe the means by which the individual parts share information. The degree of dynamism with which the processes are created and destroyed are an indication of the complexity of the process interrelationships. These criteria are influential in the decision as to the acceptability of a particular process architecture on a physical computing environment such as a network of computers versus a shared memory multiprocessor. This paper proposes a characterization of the distributed computations in these terms, and suggests a way of modeling them. Such a characterization and a model will help us to identify, understand and describe aspects of distributed computations that do not exist in sequential programs.

In Section Two, we will informally introduce our characterization through two examples of distributed computations, *the chaotic relaxation algorithm,* and *the listener/client/server model.* In Section Three we will give a definition of this characterization in more formal terms. In Section Four, we will apply it to several examples of distributed computations. Section Five will conclude this document.

## 2. MOTIVATION: TWO EXAMPLES OF DISTRIBUTED COMPUTATIONS

### 2.1. Chaotic Relaxation Algorithm

Chaotic relaxation is a method of solving a system of n equations and 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 $k \neq j$. For the given class of equations, the solution vector, X, will converge.

This is a system that is well-suited to parallel computation; 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 processes. 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 process is created with a copy of a row of A (say row k), and the corresponding element of B (B[k]). 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 the row of A and the value of B he knows about. After computing X[k], the process disseminates the value of X[k] so that the halting evaluator and the other n-1 processes can use 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.

Only two "types" of processes are involved in this computation; the first is the "parent" process that handles the initialization and the termination; the second defines the replicated "worker" processes. All the processes are created during the initialization phase, and terminate during the termination phase. As we saw, the processes have to exchange information with one another. There are several ways of setting up communications between them. We introduce two of them. The first one is to have each "worker" process be in a point-to-point communication with the parent process, that takes care of forwarding the useful information to the other worker processes. The topology of communications is therefore a star whose

center is the parent process. The second way of setting up communications, is to have all processes involved in a multicast communication, where each process listens and talks to all other processes. In this case, the topology of communications is a ring.

### 2.2. Listener/client/server model

In the listener/client/server model, one or more services (such as name servers or address servers) are available to the users. We have n listener processes standing on an arbitrary number of processors on the network, waiting for incoming requests. When a listener receives a request from a user, it forks a server and puts the user and the server in communication.. The user can then use the service until it does not need it anymore, in which case the server terminates. In the remainder of the paper, it is assumed that a given client uses at most one service at a given time.

Here, there are three types of processes involved in the computation: the listeners, the clients, and the servers. The listeners are created at initialization time, and terminate only when the computation terminates (if ever). New clients can be created and terminate at any time. Servers are forked when incoming requests come and can terminate at any time.

As far as the communications between processes are concerned, we have the following situation. In order to request a service, a client has to establish a point-to-point communication with a listener. The listener and newly-created server communicate with a point-to-point mechanism. The server establishes a point-to-point communication with the requesting client. The communications are therefore established following precise rules which define the topology of communications for this computation.

### 2.3. Comparison of the two examples

Through these two examples, we see that the knowledge of the types of processes, and the communication between them, gives a good overall view of the design of the computation. The topology is a way of identifying the rules under which processes are created and communications are set up between them.

The two examples of computations exhibit different behaviors. The listener/client/server model is dynamic in the creation and the termination of its processes, while the chaotic relaxation algorithm creates and destroys all processes at one time. Having or not having a dynamic behavior appears to be an important characteristic of parallel, distributed computations. In particular, if we are to build graphical tools, the features required to show views (in terms of processes and communications between them) of computations, are not the same for computations which have a static behavior such as the chaotic relaxation example does, and for computations which have a dynamic behavior as it is the case for the listener/client/server model. In the first case, we must be able to build the corresponding picture during the initialization phase. The picture then remains the same until the end of the computation. The second case, however, requires the ability to dynamically draw and erase processes and communication between them.

# 3. CHARACTERISTICS OF DISTRIBUTED COMPUTATIONS: DEFINITIONS

## 3.1. Process Architecture

A *parallel, distributed computation* is commonly defined as being a set of processes that run concurrently on a number of processors, exchanging information and synchronizing via shared memory or message passing. In this document we describe distributed computations using a *Process Architecture Model (PAM)* composed of *components*, and *communication relations*. The characterization of distributed computations which interests us is made up of components, means of sharing information, and patterns of interconnection among these items. Informally, this is what we mean by the process architecture of the distributed computation. Figure 1 is a static representation of the process architecture for the point-to-point approach to the chaotic relaxation algorithm, which is itself a static computation. (Note that the listener/client/server model employs dynamic creation and destruction of components, hence a complete pictorial representation would need to be dynamic.) In the Figure, circles represent components (processes in this case), and squares represent properties of the communication relation that exists among components.

A component is a schedulable unit of computations such as a process or a thread within a process. Components of a distributed computation exchange information and synchronize using an arbitrary communication mechanism.

A communication relation among components describes the means by which they communicate (or synchronize) with one another. In particular, there is a communication relation or *shared variable communication relation*, between the processes that share a given variable. There is another form of relation or *multicast communication relation*, between the processes that exchange messages through a multicast medium. There is a third form of relation or *point-to-point communication relation*, between two processes that exchange messages in a point-to-point way.

We say that a communication relation is *permanent* if it exists as long as the components involved in it exist. It is *temporary* otherwise.

Through the notion of communication relation, we therefore have means of representing any form of communication that can take place between components of a distributed computation (the three forms mentioned above are only examples of the possible kinds of relation that we might want to describe. Another example of communication relation is sharing a disk file).

## 3.2. Representing Instances with Classes

Component and communication relation instances can be represented by some characterization analogous to a process class, which we call a *component class* and a *communication relation class*, respectively.

A component class characterizes the behavior of its *component instances*. The instantiation of a component is made through a *create-component* operation. It results in the creation of an executable instance of the component class called a *component instance*. In cases where we refer to "components", we mean "component instances" unless otherwise made clear. An example of component class is a worker process of the chaotic relaxation algorithm. All the workers are instances of the same class - the worker class - initialized with a different set of data (a row of A and the corresponding value of B).

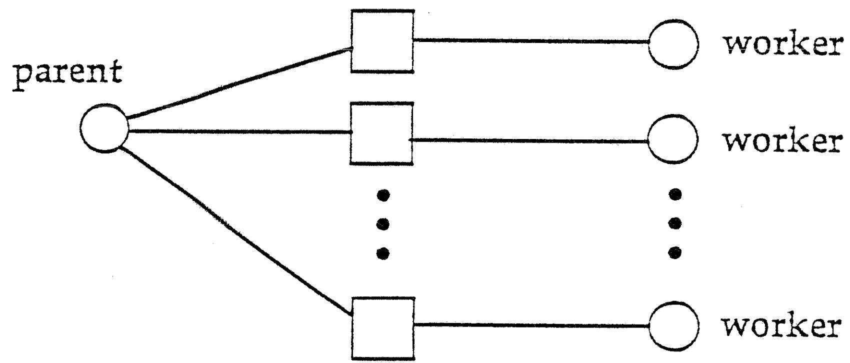



Figure 1

A relation class is characterized by the number of component instances involved and the class of each of them, the type of communication (eg. point-to-point, multicast or shared-variable), the protocol of communication used, etc. All these characteristics are *properties* of the relation class.

The *instantiation* of a relation class takes place at the establishment of the communication among the component instances involved (for example once a shared variable has been declared, and at least one of the processes that want to share it has declared its intention, or once the two ends of a point-to-point communication are ready to exchange information). In particular, the instantiation of a relation class maps the formal component instances referenced in the relation class description, to actual component instances.

Figure 2 is a static representation of the union of the different states of the process architecture for the listener/client/server model. The Figure also incorporates a representation of the relevant classes for the various instances in the process architecture. We find several examples of communication relations; there is a point-to-point communication relation between each client instance that requested a service and the corresponding server instance. There is a matching communication relation class between the client class, and the server class. This communication relation class is instantiated when the server instance establishes communication with the client instance. It is a permanent communication relation because it exists as long as the server instance exists.

### 3.3. Phases of Computation

All computations start with an initialization phase during which a number of processes are created, establish communication with one another, and acquire their initial data. All well-built computations also have a termination phase in which all running processes terminate in a graceful manner.

Between the initialization and the termination phases is a middle or computational phase. For some computations, this "middle" phase is well-structured, and distinct subphases can be identified at the level of the computation, without going into the algorithmic details of each process. For example, some computations such as the "global optimization" computation that we will introduce in the example section, proceed by successive iterations in which all processes take part. The information about the phase in which the computation exists is a precious indication about its progression

For some other computations, such subphases cannot be identified. It is the case for the listener/client/server model in which one cannot predict when a new client will come in, and when a service will be requested.

### 3.4. Static Versus Dynamic Process Architecture

As we saw, the process architecture can be static or dynamic, and this gives the corresponding distributed computation different characteristics.

We say that a system is *static* if it is composed from a fixed number $N$ of instances of an arbitrary number of component classes, all of which are created during the initialization phase of the computation, and terminated during the termination phase of the computation, $N$ being known prior to the execution or determined during the initialization phase. If $k_i$ designates the number of component instances of the class $i$, and $n$ the number of component classes in the computation, we have $N = \sum_{i=1}^{i=n} k_i$.

A system is *dynamic* if it is not static. We distinguish between two types of dynamic systems: *dynamic bounded* and *dynamic unbounded* systems.

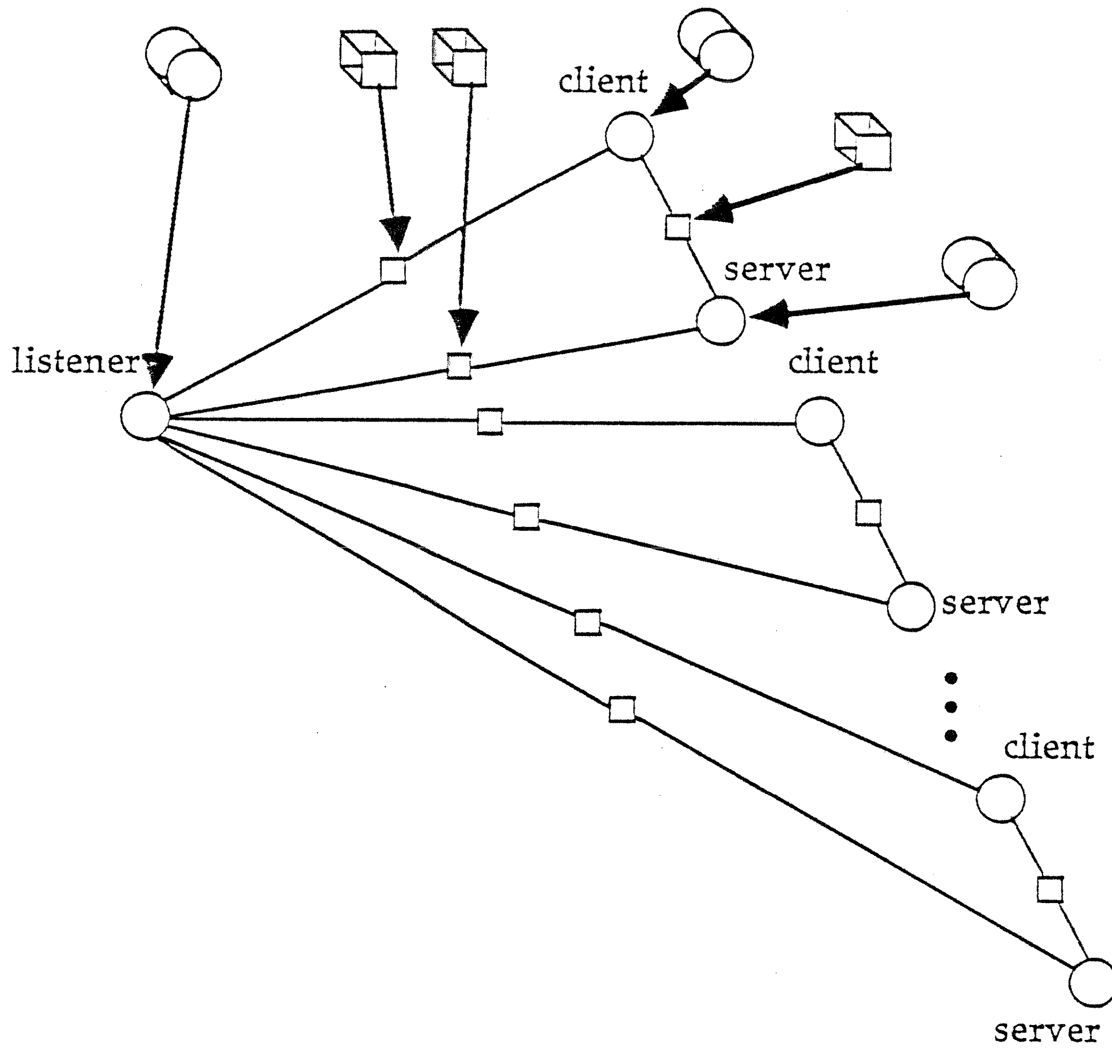



Figure 2

A system is *dynamic bounded* if its maximum number of component instances and the class of each of these component instances is known prior to the execution or determined during the initialization phase, but the component instances can be created and can terminate at different times during the computation. If $n$ is the number of component classes, $k_i$ the maximum number of component instances of class $i$ and $N$ is the actual number of component instances at a given time, we have $N \leq \sum_{i=1}^{i=n} k_i$ (i.e. $N$ is bounded).

A system is *dynamic unbounded* if its maximum number of component instances is not known prior to the execution, and cannot be determined during the initialization phase. However, we assume that the different classes of components are known. This time we know the number $n$ of classes but we do not know the $k_i$ as we did previously. Therefore, we cannot say how many component instances $N$ will exist or even bound $N$.

An example of static computation is given by the above described implementation of the chaotic relaxation algorithm. The number of worker processes is determined during the initialization phase, and the exact number of processes is therefore determined at this time too. The client/listener/server model gives an example of a dynamic unbounded computation, as the number of clients and servers cannot be known or even bounded prior to the execution or during the initialization phase.

Although there might be many other ways of characterizing parallel, distributed computations, we believe that this one captures some of their important properties. As mentioned in the motivation section, if we are to build graphical tools, the features required to show views of distributed computations (in terms of components and communication relations between them), are not the same depending on the type of the computation. To show such a view of a static computation, we must be able to build the corresponding picture during the initialization phase. The picture then remains the same until the end of the computation. To show views of a dynamic bounded computation, it is enough to be able to build a maximal picture of the computation showing the processes and the communication relations that will exist at some time, and to find a way to "highlight" the ones existing at any given time. Showing pictures of dynamic unbounded computations, however, requires the ability to dynamically draw and erase processes and communication relations.

### 3.5. Partitioning Computations

In the Process Architecture Model, a computation may be represented by a collection of components. Depending on the use to be made of the model, it is often convenient to group the individual components into sets of components. We have discussed one such grouping, namely that of common definition, ie., component classes. There are other criteria that are useful for grouping the component instances: For example, the semantics of operation for the listener/client/server model suggest that while the number of instances of the listener class is independent, the number of instances of the client and server classes is highly related (there is one client instance for each server instance). Therefore, a partition of the component instances as shown in Figure 3 provides an alternative view of the distributed computation, where the criteria for observation are related more to the semantics of the model than the definition of the processes.

As it may be interesting to introduce several levels of grouping in order to capture the complexity of a computation, we want to be able to deal with hierarchies of partitions; in this case, we shall refer to each level of description as a set of *subcomputations*. A *leaf subcomputation* is defined as being an arbitrary collection of instances from the same computation, either from the same component class, or from

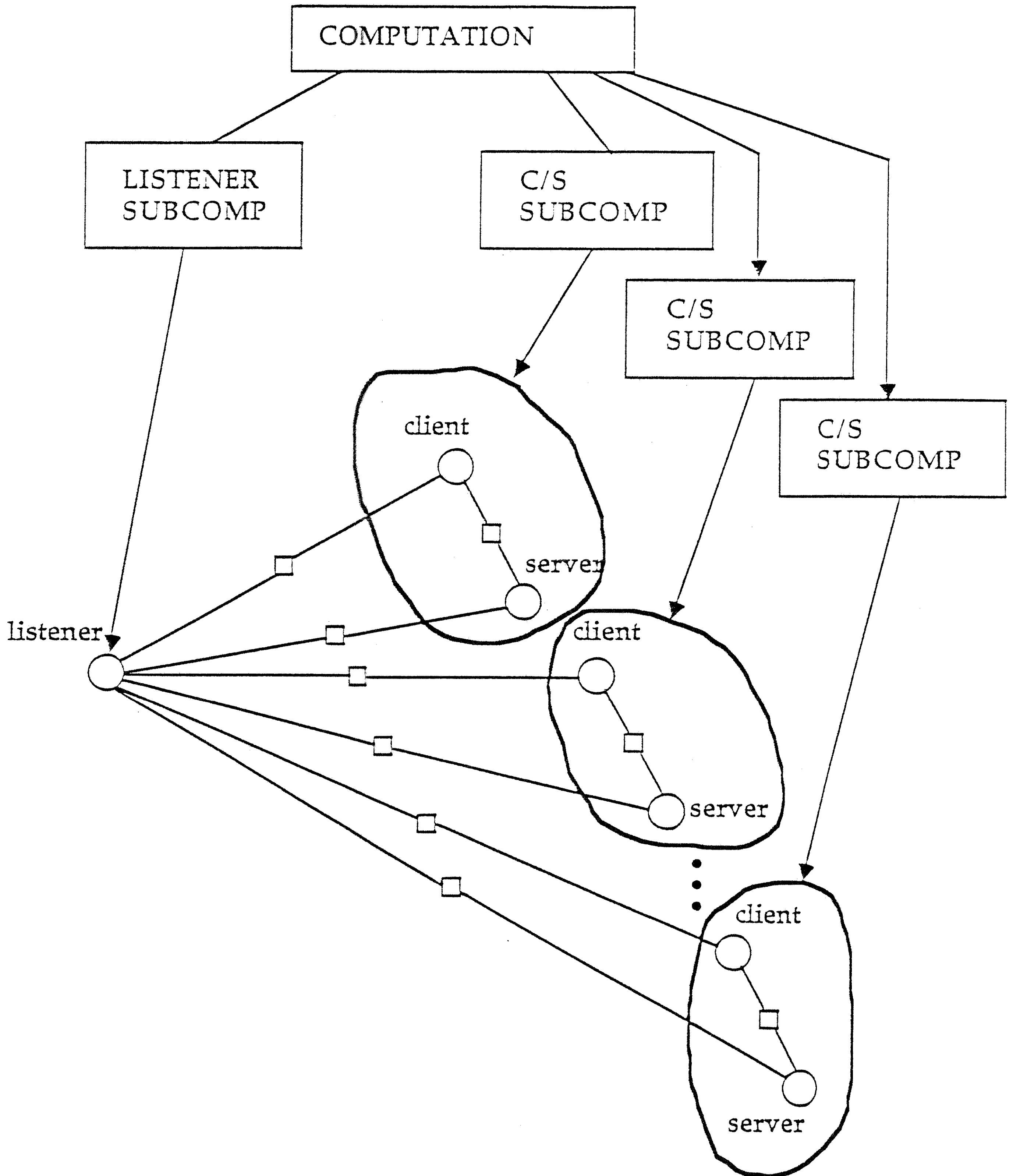



Figure 3

different component classes. A *non-leaf subcomputation* is an arbitrary collection of leaf and non-leaf subcomputations from the same computation. The term "subcomputation" is used when we do not wish to distinguish between leaf and non-leaf subcomputations. In particular, the computation is itself a subcomputation.

A subcomputation can be represented by some characterization called a *subcomputation class.*

The idea behind the notion of subcomputation is to be able to partition a possibly big computation, into **disjoint** smaller units, the leaf subcomputations, that are easier to deal with. If the computation is complex, one or more non-leaf subcomputation levels can be added on top of the leaf-level partition.

In the following we will also use the notion of *proper partition.* If we suppose that at some instant, all instances of a distributed computation are partioned into leaf subcomputations, then such a partition is *proper* if all component instances within a given leaf subcomputation are derived from the same subset of component classes, and no other leaf subcomputation within the partition has any component instance derived from any class for the subset. Thus for each proper partition of the component instances there is a corresponding partition of the component classes for the computation.

A criterion for the choice of the partition is to isolate into distinct subcomputations the parts that have a static behavior, from the ones that have a dynamic bounded behavior and the ones that have a dynamic unbounded behavior. It is indeed easier to deal with a static computation (or subcomputation), than it is to deal with a dynamic unbounded, or even a dynamic bounded one.

The decomposition into subcomputations must reflect as much as possible the properties, and the behavior of the computation. If the computation appears to be a collection of n "independent" pieces of computation running in parallel, the partition should show n subcomputations. In the same idea, if processes are created to work during one of the phases of a computation and disappear at the end of the phase, there should exist a subcomputation that groups them.

Another criterion, is to try to come up with a proper partition. By doing so, we come up with subcomputations whose characteristics are defined by the ones of the classes they are made from, and that are therefore easier to visualize.

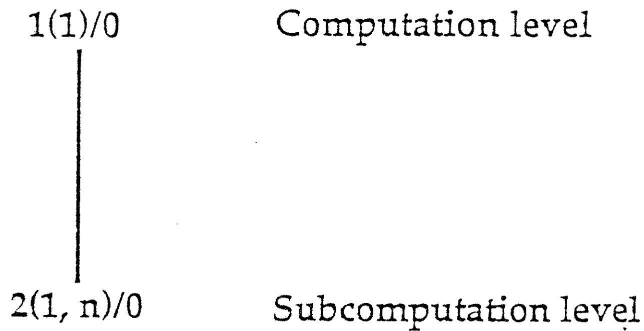
Several levels of subcomputations, corresponding to several levels of grouping, can be introduced to capture the complexity of some computations. The subcomputations form a tree (all the leaf subcomputations are disjoint, and a subcomputation is included in exactly one subcomputation of the above level).

### 3.6. Complexity of a Subcomputation

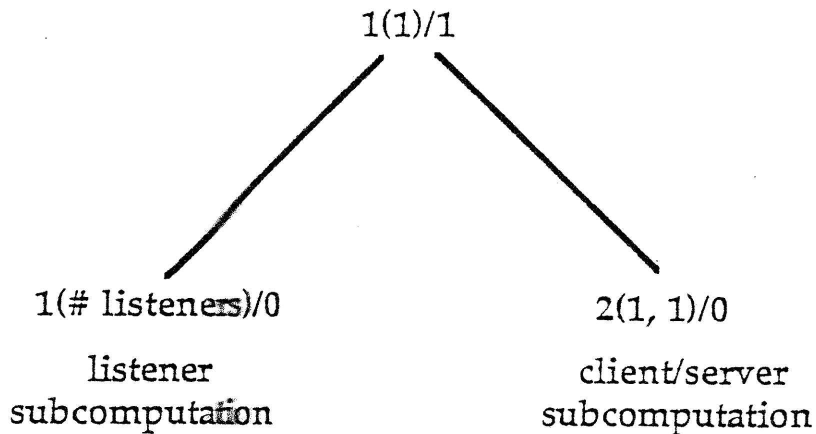
In order to characterize the complexity of a computation, we use the notion of subcomputation.

A computation (or non-leaf subcomputation) is said to be *$n( N_1, N_2, ..., N_n )/m$-structured* if it can be broken into a set of subcomputations involving $n+m$ classes of subcomputations, such that there is a fixed or bounded number of subcomputation instances of n of the subcomputation classes, and an unbounded number of instances of m of the subcomputation classes. $N_i$ is the bound for the number of subcomputation instances of subcomputation class number $i$.

A leaf-subcomputation is said to be *$n( N_1, N_2, ..., N_n )/m$-structured* if it involves components from $n+m$ component classes, such that there is a fixed or bounded number of component instances of n of the component classes, and an unbounded number of instances of m of the component classes. $N_i$ is the bound for the number of component instances of component class number $i$.




a) Chaotic Relaxation




b) Listener/Client/Server

Figure 4

We can therefore traverse the hierarchy of subcomputations of a given computation and characterize the complexity at each level.

For example, *Chaotic relaxation* can be modeled by a single subcomputation. It is 1(1)/0-structured.

The subcomputation itself is composed of 1 instance of the parent class, and n instances of the worker class. It, therefore, involves 2 classes and is 2(1, n)/0-structured.

Figure 4a shows a tree giving the complexity of each level of subcomputation for the chaotic relaxation algorithm.

*The listener/client/server model* can be broken into 1 subcomputation containing all the listeners, and an unbounded number of subcomputations of the same type, each containing a pair client/server (the server being the one forked to answer a request from the client). It is therefore 1(1)/1-structured.

The listener subcomputation is itself 1(# listeners)/0-structured. Each of the "client/server" subcomputation involves two component classes, the server class, and the client class, and one instance of each. It is therefore 2(1, 1)/0-structured.

Figure 4b shows a tree giving the complexity of each level of subcomputation for the listener/client/server model.

### 3.7. Topology

As we saw, the components as well as the communication relations are instantiated following precise rules that constrain the number of components and the establishment of communications between them. These rules define the *topology* of the computation. The topology of a computation therefore appears as the set of rules used to derive the process architecture of the computation from the class definitions. However, we will often use the word "topology" as a synonym for "structure among the components in a process architecture," thereby identifying the set of rules with the result they produce.

As we saw in Section Two, the chaotic relation computation has a star topology if the worker processes are in point-to-point communication with the parent process, and in ring topology if all the processes are involved in a multicast communication relation.

The listener/client/server computation does not have an identified topology shape such as ring, star, or grid. The topology is defined by the rules followed to instantiate the servers, and to instantiate communication relations between a client and a listener, a listener and the server it forks, a server and the corresponding requesting client.

### 3.8. Physical Structure of the Computation

The physical structure of a computation is given by the mapping of the logical structure of the computation to the actual network of processors, and communication media between them. The physical structure may vary from one execution to another, depending on how processes (component instances) are scheduled to processors.

The scheduling of component instances to processors is usually left to the scheduling unit of the supporting system. In certain cases, however, the designer of the computation might have to or want to put some requirements on how the component instances must be scheduled to processors. He might, for example, require that each component instance be scheduled to a different processor, or that a given

instance be scheduled to a given processor, and design the computation consequently. In this case, the physical mapping requirements become part of the design of the computation and must be considered as a characteristic of the computation itself.

It is often recognized that the analog of our process architecture is influenced by the physical structure of the system on which the computation will be executed. We expect that the process architecture model is an appropriate vehicle for considering alternative process architectures for specific computer architectures.

## 4. EXAMPLES OF CHARACTERIZATION OF COMPUTATIONS

### 4.1. Introduction

The purpose of this Section is to illustrate the above characterization with several examples of distributed computations. We will describe the chaotic relaxation algorithm, the listener/client/server model, an example of the global optimization algorithm and the "employee's telephone bill" computation.

For each example, we will give a general overview of the computation, the implementation chosen, the Process Architecture Model and a description of the characteristics of the computation. The characteristics will be further described in terms of classes of components, subcomputations, classes of communication relations and topology; we will also provide a diagrammatic representation showing the classes involved in the computation, and some of their instances. Note that the syntax used for the description is informal. It could be made more formal if it were to be used in the context of a system, eg., the description could be formalized to the degree that a machine could parse it. In the same way, the diagrammatic representation chosen is informal. The diagrams can be inferred from the descriptions. A legend for the diagrammatic representations is given in Figure 5.

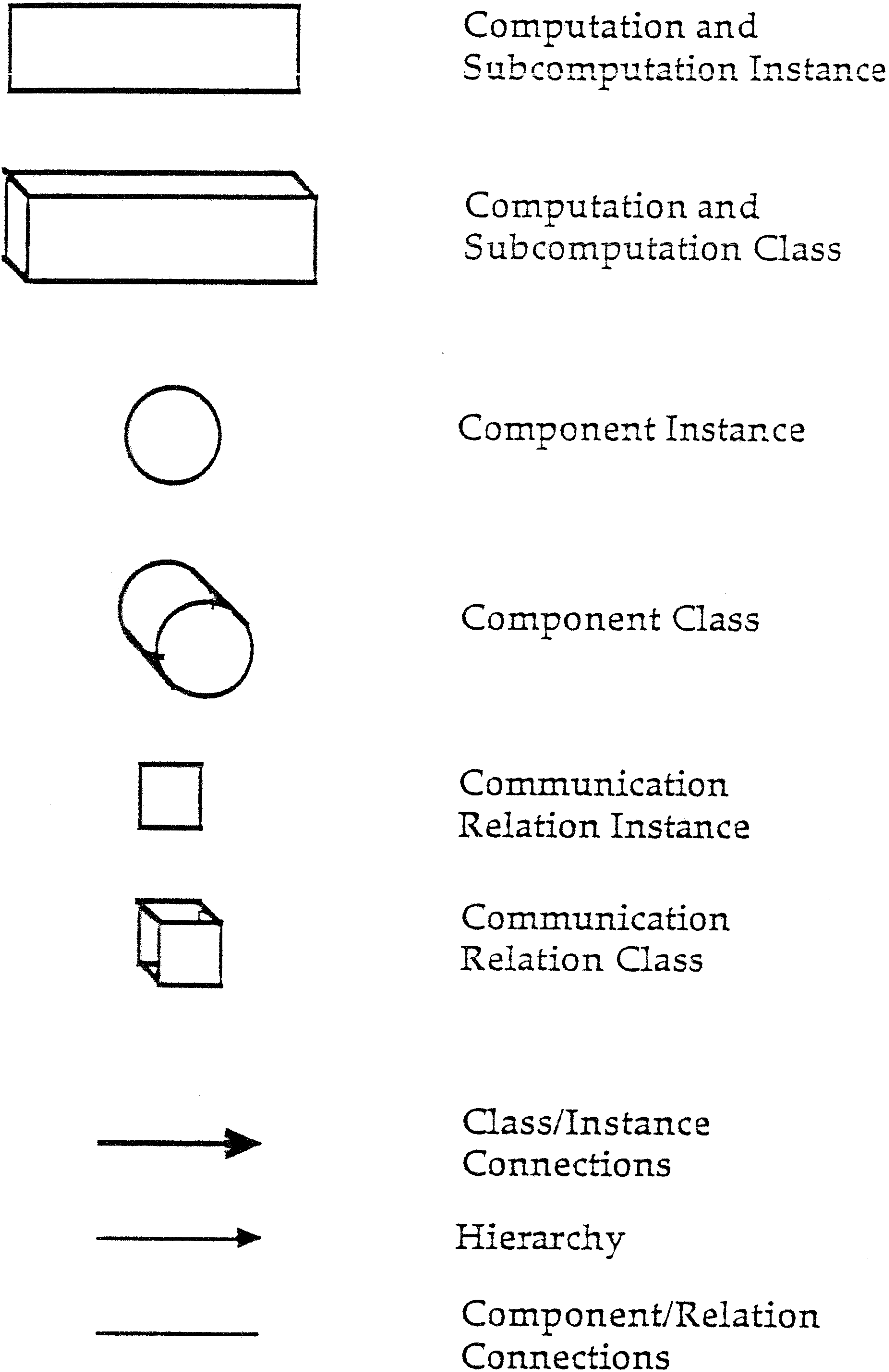
### 4.2. Chaotic Relaxation

As we saw, the program is constructed from only two process classes: the "parent" process and the worker processes. The computation is static, as it infers its structure during the initialization phase (when it reads the value of $n$, the number of equations).

It is a very simple computation, and therefore it has a very simple partition into subcomputations: the whole computation can be modeled as a single subcomputation. It is 1(1)/0-structured. The leaf subcomputation is itself 2(# equations, 1)/0-structured.

Phases of computation could be forced, by having all the worker processes wait for a complete set of new values for the X[i], before they compute the next value. In this case, all the worker processes would iterate at the same time. This is however not the implementation we have chosen. In ours, each worker process iterates with the current value of the X[i], whether changed since the previous iteration or not, and there is therefore no sub-phases.

In Section Two, we mentioned two of the different communication schemas that can be adopted; in the first one, or "point-to-point version of the chaotic relaxation algorithm", each worker process is in point-to-point communication with the parent process; in the second one, or "multicast version of the chaotic relaxation algorithm" all the processes are in communication with one another through a multicast communication.




Legend
Figure 5

Figure 6 shows a description of the point-to-point version of the chaotic relaxation algorithm. Figure 7 shows a diagrammatic representation of it. Figures 8 and 9 show the description and the diagrammatic representation for the multicast version of the chaotic relaxation algorithm.

---

**Figure 6 - Description of the Point-to point Version of Chaotic Relaxation**

**components**

Component class 1: parent class.
Description: a parent component reads the values of the matrix A and the vector B; then, it determines the number n of equations to solve in order to find X such that AX = B, and spawns n worker processes; it checks the evolving values of the solution vector X and terminates the computation when the halting condition is reached.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;

List of relation classes involving this component class:
- relation class 1 (parent/worker);

Component class 2: worker class.
Description: a given worker process, say process k, is responsible for computing the value of X[k], using the kth row of A - A[k][] - B[k], and the current best guess at X[0], ... , X[k-1], X[k+1], ..., X[n]. Each time it has computed a new value for X[k], it sends it to the parent process, and acquires the current value of X from it. It stops when the parent process has determined that the stopping condition is reached.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;
- row (or equation) number;

List of relation classes involving this component class:
- relation class 1 (parent/worker);

**Subcomputations**

number of levels: 2;
level 1 or leaf:
- 1 class of subcomputation;
- class 1 subcomputation: worker/parent subcomputation class;
  * composed from 2 classes of component worker and parent;

* static;
* 2(1 + #equations)/0-structured;

- one instance of subcomputation class 1;

Computation level:

- 1 instance of worker/parent subcomputation;
- the computation can be modeled as a single subcomputation;
- proper;
- static;
- 1(1)/0-structured;

**communication relations**

Relation class 1: parent/worker communication relation.
Properties (of the class):

- type: point-to-point full duplex;
- number of ends: 2;
- end 1: parent;
- end 2: worker;
- permanent relation;

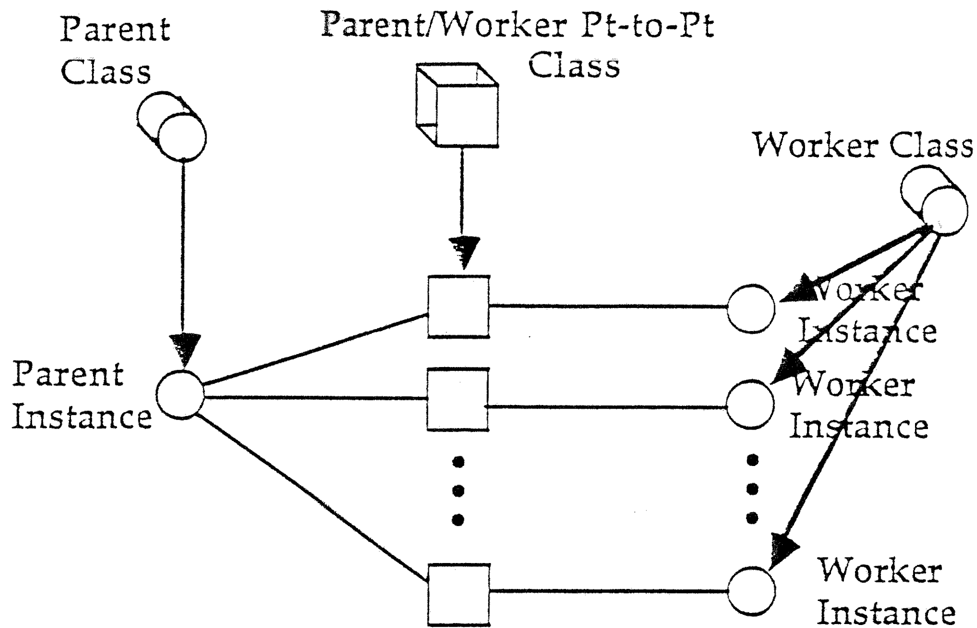
Attributes (defined for each instance):

- process number of end 1;
- process number of end 2;

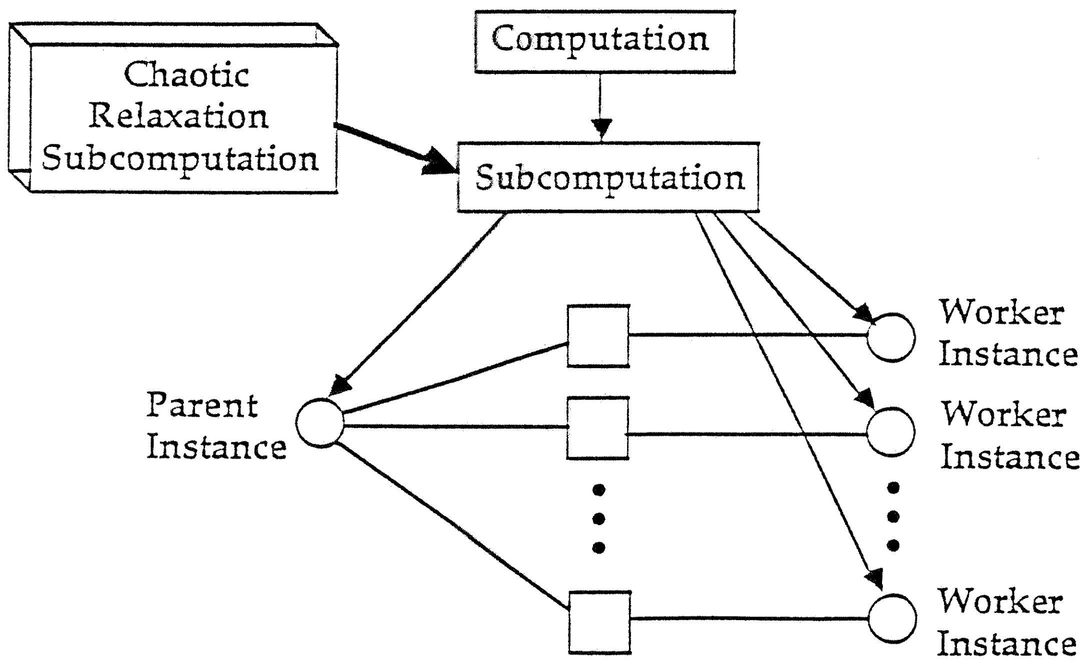
**topology**

Constraint on number of class 1 components: 1 parent;
Constraint on number of class 2 components: determined by parent at initialization time;
Constraint on communication relation instance: 1 instance for each worker;
Topology shape: star;

---




a) Class-Instance Mapping




b) Computation Structure

Figure 7

**Figure 8 - Description of the Multicast Version of Chaotic Relaxation**

/* the aspects of this description that differ from the previous one are printed in bold characters */

**components**

Component class 1: parent class.

Description: a parent component reads the values of the matrix A and the vector B; then, it determines the number n of equations to solve in order to find X such that AX = B, and spawns n worker processes; it checks the evolving values of the solution vector X and terminates the computation when the halting condition is reached.

Properties (of the class): ;

Attributes (defined for each instance):

- process number;

List of relation classes involving this component class:

- relation class 1 (parent/worker);

Description: a given worker process, say process k, is responsible for computing the value of X[k], using the kth row of A - A[k][] - B[k], and the current best guess at X[0], ... , X[k-1], X[k+1], ..., X[n]. Each time it has computed a new value for X[k], it sends it to the parent process, and acquires the current value of X. It stops when the parent process has determined that the stopping condition is reached.

Properties (of the class): ;

Attributes (defined for each instance):

- process number;
- row (or equation) number;

List of relation classes involving this component class:

- relation class 1 (parent/worker);

**Subcomputations**

number of levels: 2;

level 1 or leaf:

- 1 class of subcomputation;
- class 1 subcomputation: worker/parent subcomputation class;
  * composed from 2 classes of component worker and parent;
  * static;
  * 2(1 + # equations)/0-structured;
- one instance of subcomputation class 1;

Computation level:

- 1 instance of worker/parent subcomputation;
- the computation can be modeled as a single subcomputation;
- proper;
- static;
- 1(1)/0-structured;

**communication relations**

Relation class 1: parent/worker communication relation.

**Properties (of the class):**

- **type: multicast**
- **number of ends: # of worker processes + 1;**
- **ends: all (all processes can "talk" with one another);**
- **permanent relation;**

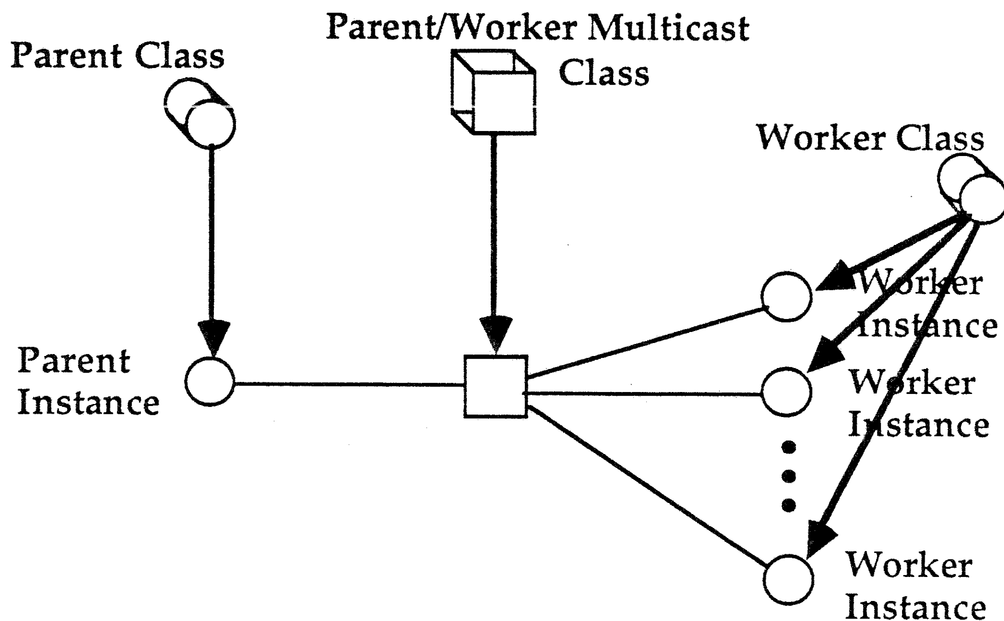
**Attributes (defined for each instance):**

- **list of process numbers (corresponding to all processes involved in the communication relation instance);**

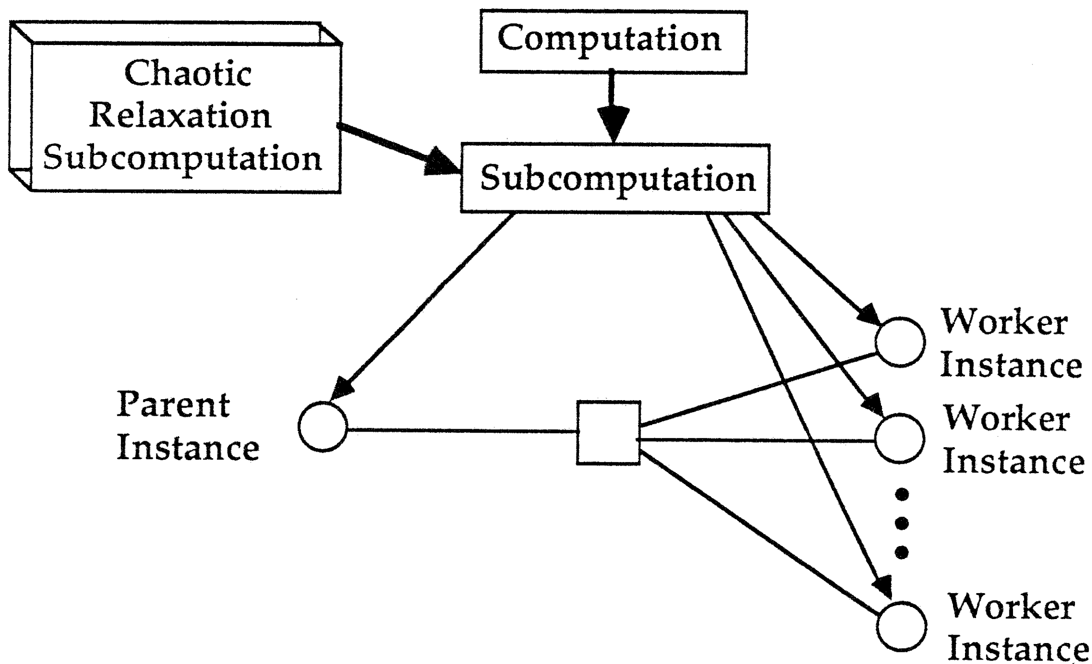
**topology**

Constraint on number of class 1 components: 1 parent;
Constraint on number of class 2 components: determined by parent at initialization time;
**Constraint on communication relation instance: 1 instance;**
**Topology shape: ring;**

---




a) Class-Instance Mapping




b) Computation Structure

Figure 9

### 4.3. Listener/client/server model

As we saw, the computation is dynamic unbounded. It involves three classes of components (user processes, listeners, servers), and several instances of each class. There is a fixed number of listeners, but an unbounded number of clients and servers.

It can be broken into 1 subcomputation containing all the listeners, and an unbounded number of subcomputations of the same type, each containing a pair client/server (the server being the one forked to answer a request from the client). It is therefore 1(1)/1-structured.

The listener subcomputation is itself 1(# listeners)/0-structured, and therefore static. Each of the "client/server" subcomputation involves two component classes, the server class, and the client class, and one instance of each. It is therefore 2(1, 1)/0-structured and static.

Figures 10 and 11 show the description and the diagrammatic representation for the listener/client/server model.

---

**Figure 10 - Description of the Listener/client/server Model**

**components**

Component class 1: listener class.
Description: listeners wait for new incoming requests for services from
client processes. When it receives a request, a listener forks the
corresponding server and gives it the "address" of the requesting
client. It then goes on waiting for new requests.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;

List of relation classes involving this component class:
- relation class 1 (listener/client);
- relation class 2 (listener/server);

Component class 2: server class.
Description: when a server is forked, it is given the "address" of the client
who needs his services. It uses it to establish a communication with
him and answers his request until it is terminated by the client who
does not need him anymore.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;

List of relation classes involving this component class:
- relation class 2 (listener/server);
- relation class 3 (server/client);

Component class 3: client class.

Description: when a client needs a service, it requests it from the "closest" listener. It waits for the server that will be created to answer his request to initiate a communication with him. When it does not need the service anymore, it terminates the server.

Properties (of the class): ;

Attributes (defined for each instance):

- process number;

List of relation classes involving this component class:

- relation class 1 (listener/client);
- relation class 3 (server/client);

**Subcomputations**

number of levels: 2;

level 1 or leaf:

- 2 classes of subcomputations;
- class 1 subcomputation: listener subcomputation;
    * composed from one component class: listener component class;
    * static;
    * 1(# listeners)/0-structured;
- class 2 subcomputations: client/server subcomputation;
    * composed from two component classes: server and client component class;
    * static;
    * 2(1, 1)/0-structured;
- One instance of class subcomputation 1;
- unbounded number of instances of subcomputation class 2;
- not a proper partition;

Computation level:

- unbounded number of subcomputations
- 1 instance of class 1 subcomputation (listener subcomputation);
- unbounded number of class 2 subcomputations (server/client subcomputation)
- not a proper partition;
- dynamic unbounded;
- 1(1)/1-structured;

**communication relations**

Relation class 1: listener/client communication relation.

Properties (of the class):

- type: point-to-point full duplex;
- number of ends: 2;
- end 1: client;
- end 2: listener;
- permanent relation;

Attributes (defined for each instance):

- process number of end 1;
- process number of end 2;

Relation class 2: listener/server communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: server;
- end 2: listener;
- permanent relation;

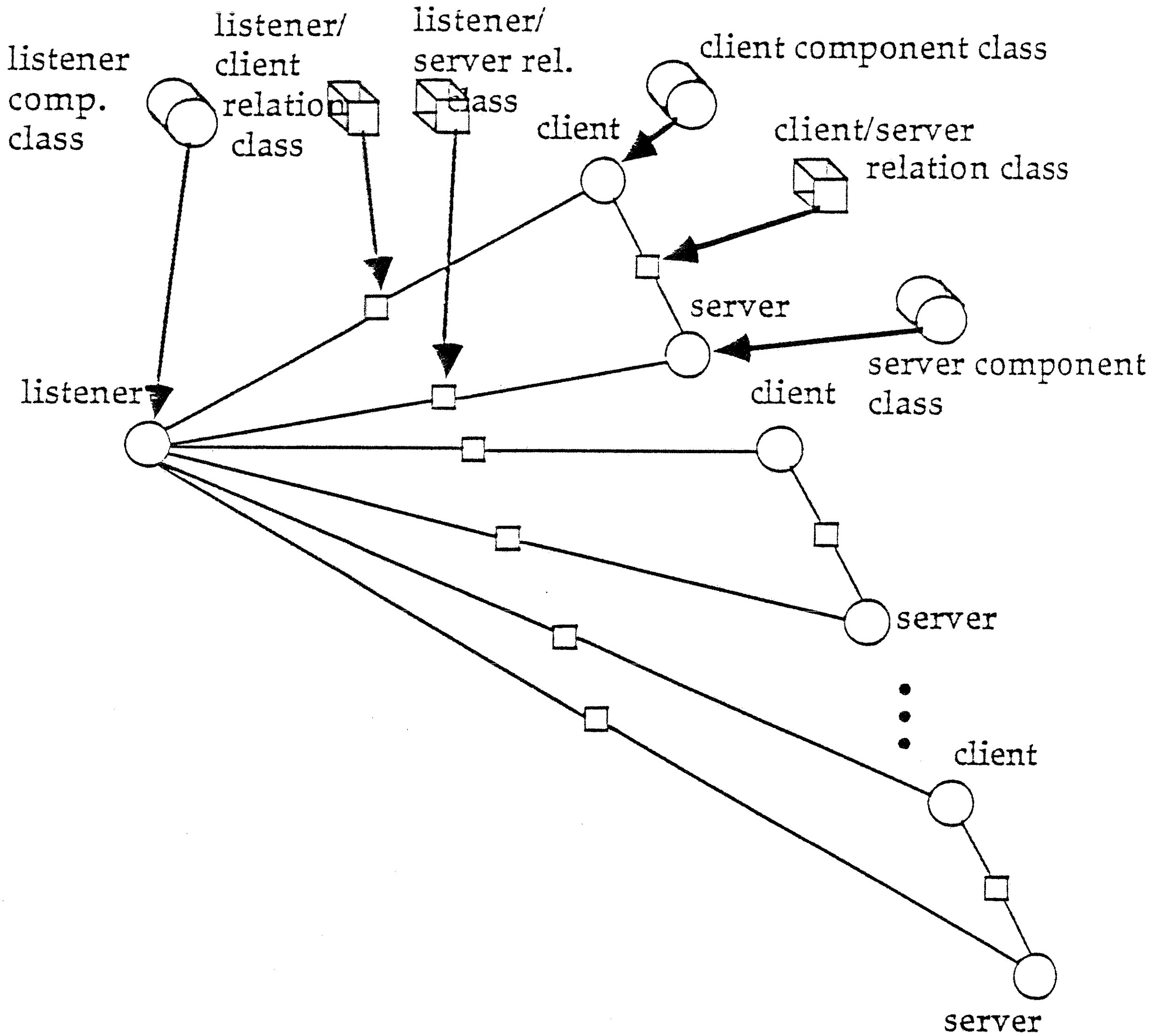
Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

Relation class 3: server/client communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: client;
- end 2: server;
- permanent relation;

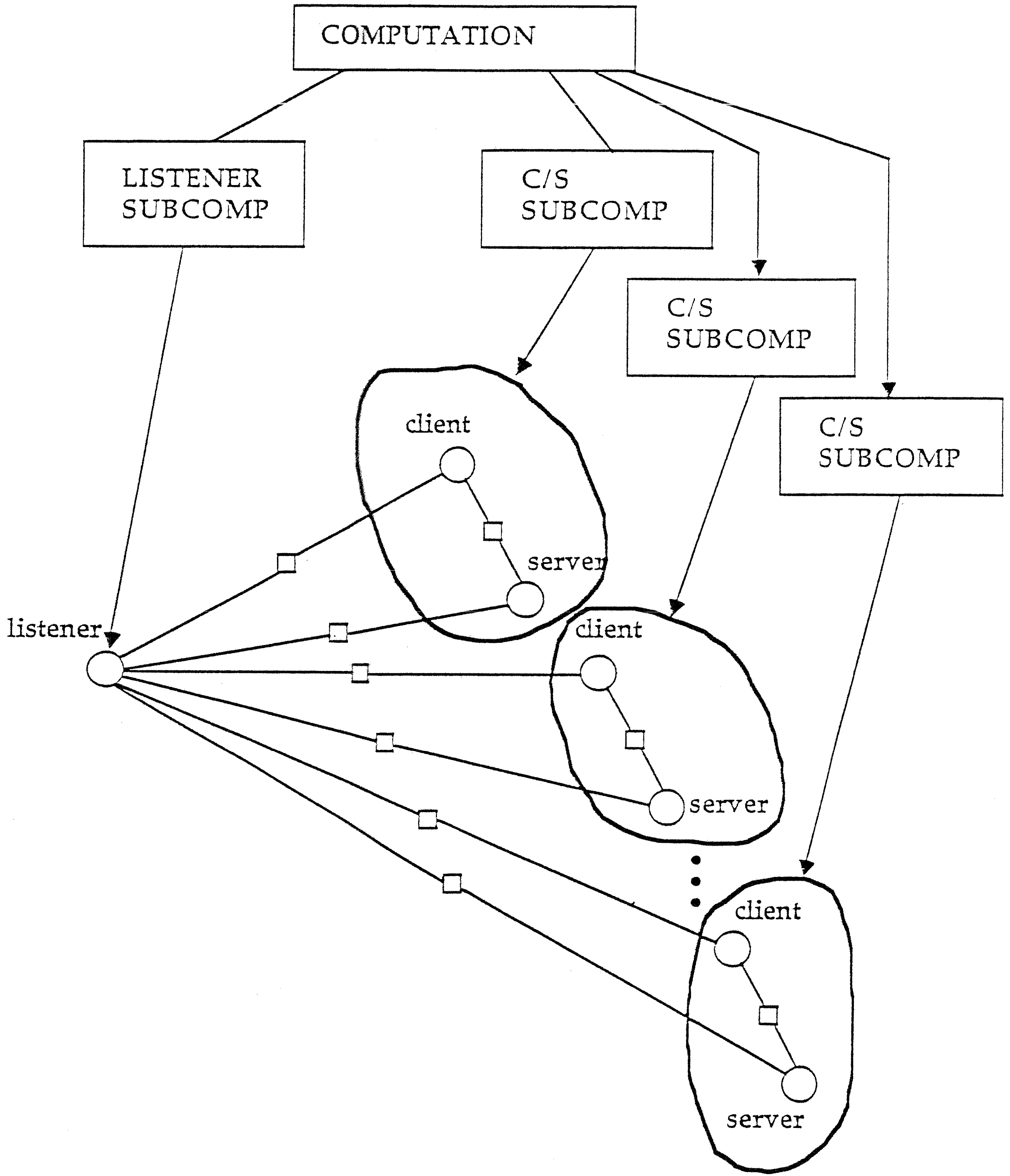
Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

**topology**

Constraint on number of class 1 (client) components: ;
Constraint on number of class 2 (server) components: as many as there are requests from clients currently processed;
Constraint on number of class 3 (listener) components: fixed;
Constraint on communication relation instance of class 1 (listener/client): one for each new request of a client;
Constraint on communication relation instance of class 2 (listener/server): one for each pair server/listener such that the server has been forked by the listener (on request of a client).
Constraint on communication relation instance of class 3 (client/server): one for each pair client/server such that the server has been forked following a request from the client.
Topology shape: ;




Class-Instance Mapping
Figure 11a




Computation Structure
Figure 11b

## 4.4. The concurrent stochastic method for global optimization

### 4.4.1. Introduction

We use the Concurrent Stochastic Method for Global Optimization algorithm described in [1]. However, the alert reader may find some slight differences - adopted for the purpose of the example - with the original algorithm, and an implementation of the algorithm which is not exactly the one suggested by Byrd, Dert, Rinnooy Kan and Schnabel.

### 4.4.2. Overview of the algorithm

We will only give a quick description of the algorithm.

The global optimization problem is to find the lowest function value of a function that may have multiple local minimizers. The algorithm we are going to present is iterative. Each iteration consists of a sampling phase, in which the function on which the global optimization is performed is evaluated at a number of randomly sampled points; it is followed by a minimization phase in which a local minimization procedure is started from a subset of the sample points; a probabilistic stopping rule is then applied to determine whether the algorithm should be continued, and if it should, the next iteration is begun.

The concurrent version of the algorithm proposed in [1], is based on the fact that the feasible region can be divided into P subregions; the generation of sample points for each subregion can be done concurrently and local minimization can be performed simultaneously on several sample points.

The main steps of the algorithm are given in Figure 12.

---

**Figure 12 - A Concurrent Method for Global Optimization**

Given: $f$, the function, $S$, the feasible region, $P$, the degree of parallelism, and $N$ is the global size of the sample at each iteration.

**step 0: partition $S$**

Subdivide $S$ into $P$ equal size, regular shaped subregions $S_i$, $i=1,...,P$, and assign subregion $S_i$ to process $i$ for $i=1,...,P$.

At iteration number k:

**step 1: generate sample points and function values**

For $i=1,...,P$

Add $N/P$ points, drawn from a uniform distribution over subregion $i$, to the (initially empty) set of sample points, and evaluate $f(x)$ at each new sample point.

**step 2: select start points for local searches**

For $i=1,...,P$

Determine a (possibly empty) set of start points in subregion $i$, disregarding sample information

from all other subregions. Resolve start points near border between subregions.

**step 3: perform local minimizations from all start points**

Collect all start points and distribute one to each of a series of processes created to perform local minimizations. If there are more start points than there are processes to perform searches from them, start a new local minimization as soon as one of the processes terminates, until local searches from all start points have been completed.

**step 4: decide whether to stop**

If stopping rule is satisfied, regard the lowest local minimizer found, as the global minimizer. Otherwise, go to step 1.

---

### 4.4.3. Implementation

We use three types of processes to implement the above described algorithm: master process type, region-point-selector process type, and local-minimizer process type.

The master process acquires the description of the function $f$, the feasible region $S$, the size $N$ of the sample at each iteration, and the degree of parallelism $P$. It then subdivides $S$ into $P$ subregions, creates $P$ region-point-selector process instances and assigns a subregion to each of them. At each iteration, it is responsible for collecting all the points that are near the subregions border, and to distribute them to all region-point-selector processes. It is also responsible for collecting all the start points from the region-point-selector processes. It then initiates the local searches (by successively creating local-minimizer processes), collects all the results, performs the stopping test, and starts the next iteration if required.

We can distinguish between two approaches: in the first one or "maximal degree of parallelism" one, the master creates as many local-minimizer instances as there are searches to perform, at each iteration. The second or "bounded" approach, consists of fixing a maximal number of authorized simultaneous searches. Only that number of searches can be started in parallel, and an additional one can be started as soon as one terminates. We will adopt this second option and consider that the maximal number of simultaneous researches, $M$ is acquired by the master during the initialization phase.

The region-point-selector processes are responsible for extending their set of sample points by $N/P$ new points, at each iteration, and evaluating $f(x)$ at each of them. They then send the values of the sample points that are near one of their borders to the master, and in exchange they receive all candidate start points that are within the critical distance of a border between subregions. The idea is that some of this start points would not be selected by the sequential algorithm because some sample point in another subregion, but within the critical distance, has a smaller function value, and the purpose of this phase is to get rid of them. Once it is done, each region-point-selector process sends the points it selected to the master. It sits idle until the next iteration begins, or the computation terminates.

The local-minimizer processes are started by the master, and given a start point from which to perform a search. When they are done, they report their result to the master, and terminate.

The region-point-selector, as well as the local-minimizer processes are in point-to-point communication with the master. The topology is therefore a star.

### 4.4.4. Characterization of the computation

As we saw, three types of components, master, region-point-selector and local minimizer, are involved in this computation.

It can be broken into three different subcomputations: the master subcomputation, the region-point-selector subcomputation and the local minimizer subcomputation. The master subcomputation as well as the region-point-selector subcomputation both involve one class of subcomputation, and one instance of it. The local minimizer subcomputation can itself be broken into an unbounded number of subcomputations, each corresponding to an iteration. Each of these subcomputations are instances of the same class. The computation therefore appears to be 2(1, 1)/1-structured.

The master subcomputation is 1(1)/0-structured and is therefore static.

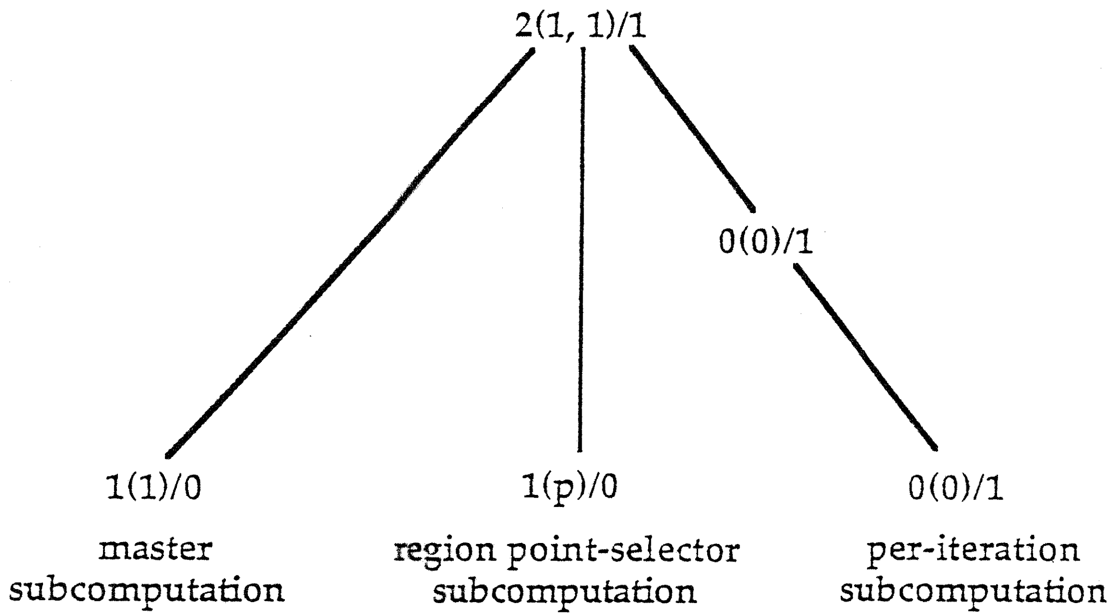
The region-point-selector subcomputation involves one class of component, the region-point-selector class, and p instances of it - p representing the chosen degree of parallelism -. It is therefore 1(p)/0-structured and static.

As we said, the local-minimizer subcomputation can be broken into an unbounded number of "per iteration" subcomputations. It is therefore 0(0)/1-structured.

Each "per-iteration" subcomputation involves an unbounded number of local minimizer components. At any time, there is a bounded number $M$ of coexisting local-minimizer processes but the global number is unbounded as there can be more than $M$ created at a given iteration and the number of iterations is unknown and unbounded. It is therefore 0(0)/1-structured.

This decomposition is interesting because it helps pinning down the very organized structure of the computation into sub-phases (the iterations).

Figure 13 gives the tree of the complexity of subcomputations at each level. Figure 14 gives the description of the computation; Figure 15 shows a diagrammatic representation of it.




Global Optimization Complexity
Figure 13

**Figure 14 - Description of the global optimization algorithm**

**components**

Component class 1: master class.
Description: The master process acquires the description of the function $f$, the feasible region $S$, the size $N$ of the sample at each iteration, the maximum number $M$ of simultaneous local-minimizer, and the degree of parallelism $P$.

It then subdivides $S$ into $P$ subregions, creates $P$ region-point-selector process instances and assigns a subregion to each of them.

At each iteration, it is responsible for collecting all the points that are near the subregions border, and to distribute them to all region-point-selector processes. It is also responsible for collecting all the start points from the region-point-selector processes. It then initiates the local searches (by successively creating local-minimizer processes without creating more than $M$ at the same time), collects all the results, performs the stopping test, and starts the next iteration if required.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;

List of relation classes involving this component class:
- relation class 1 (master/region-point-selector);
- relation class 2 (master/local-minimizer);

Component class 2: region-point-selector class.
Description: the region-point-selector processes are responsible for extending their set of sample points by $N/P$ new points, at each iteration, and evaluating $f(x)$ at each of them. They then send the values of the sample points that are near one of their borders to the master, and in exchange they receive all candidate start points that are within the critical distance of a border between subregions. The idea is that some of this start points would not be selected by the sequential algorithm because some sample point in another subregion, but within the critical distance, has a smaller function value, and the purpose of this phase is to get rid of them. Once it is done, each region-point-selector process sends the points it selected to the master. It sits idle until the next iteration begins, or the computation terminates.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;
- subregion assigned;

List of relation classes involving this component class:
- relation class 1 (master/region-point-selector);

Component class 3: local-minimizer class.
Description: the local-minimizer processes are started by the master, and given a start point from which to perform a search. When they are done, they report their result to the master, and terminate.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;
- start point assigned;
List of relation classes involving this component class:
- relation class 2 (master/local-minimizer);

**communication relations**

Relation class 1: listener/client communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: client;
- end 2: listener;
- permanent relation;
Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

Relation class 2: listener/server communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: server;
- end 2: listener;
- permanent relation;
Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

Relation class 3: server/client communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: client;
- end 2: server;
- permanent relation;
Attributes (defined for each instance):

- process number of end 1;
- process number of end 2;

### Subcomputations

number of levels: 3;
level 1 or leaf:
- 3 classes of subcomputations;
- class 1 subcomputation: master subcomputation;
    * composed from one component class: master component class;
    * static;
    * 1(1)/0-structured;
- class 2 subcomputation: region-point-selector subcomputation;
    * composed from one component class: region-point-selector component class;
    * static;
    * 1(p)/0-structured;
- class 3 subcomputation:
    * per-iteration subcomputation;
    * composed from one component class: local-minimizer component class;
    * dynamic unbounded;
    * 0(0)/1-structured;
- one instance of subcomputation class 1;
- p instances of subcomputation class 2 - where p is the degree of parallelism -;
- unbounded number of instances from subcomputation class 3;
- not a proper partition;

level 2:
- 1 subcomputation class;
- class 4 subcomputation: local-minimizer subcomputation;
    * composed from one subcomputation class: per-iteration subcomputation;
    * dynamic unbounded;
    * 0(0)/1-structured;
- unbounded number of instances from subcomputation class 4;

Computation level:
- 1 instance of class 1 subcomputation (master subcomputation);
- 1 instance of class 2 subcomputation (region-point-selector subcomputation);
- unbounded number of class 4 subcomputation (local minimizer subcomputation)
- not proper;
- dynamic unbounded;
- 2(1, 1)/1-structured;

### communication relations

Relation class 1: master/region-point-selector communication relation.
Properties (of the class):
- type: point-to-point full duplex;

- number of ends: 2;
- end 1: master;
- end 2: region-point-selector;
- permanent relation;

Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

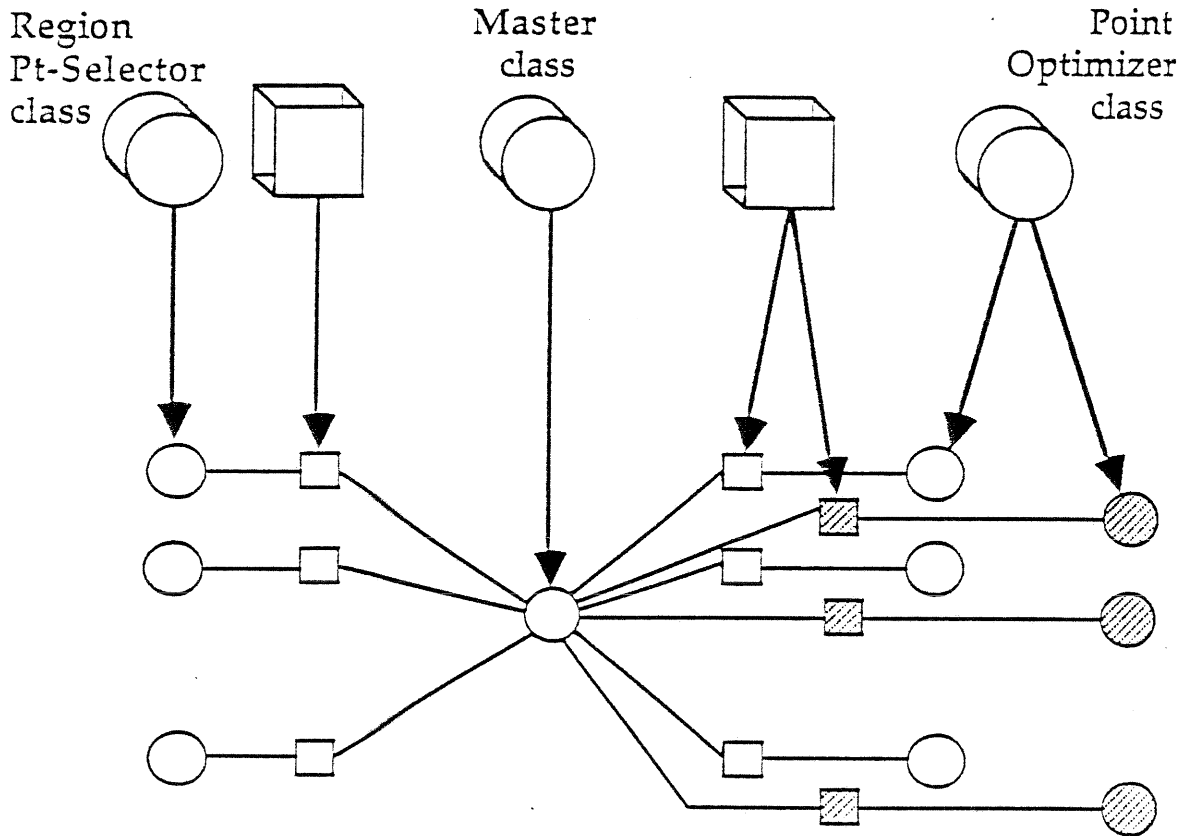
Relation class 2: master/local-minimizer communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: master;
- end 2: local-minimizer;
- permanent relation;

Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

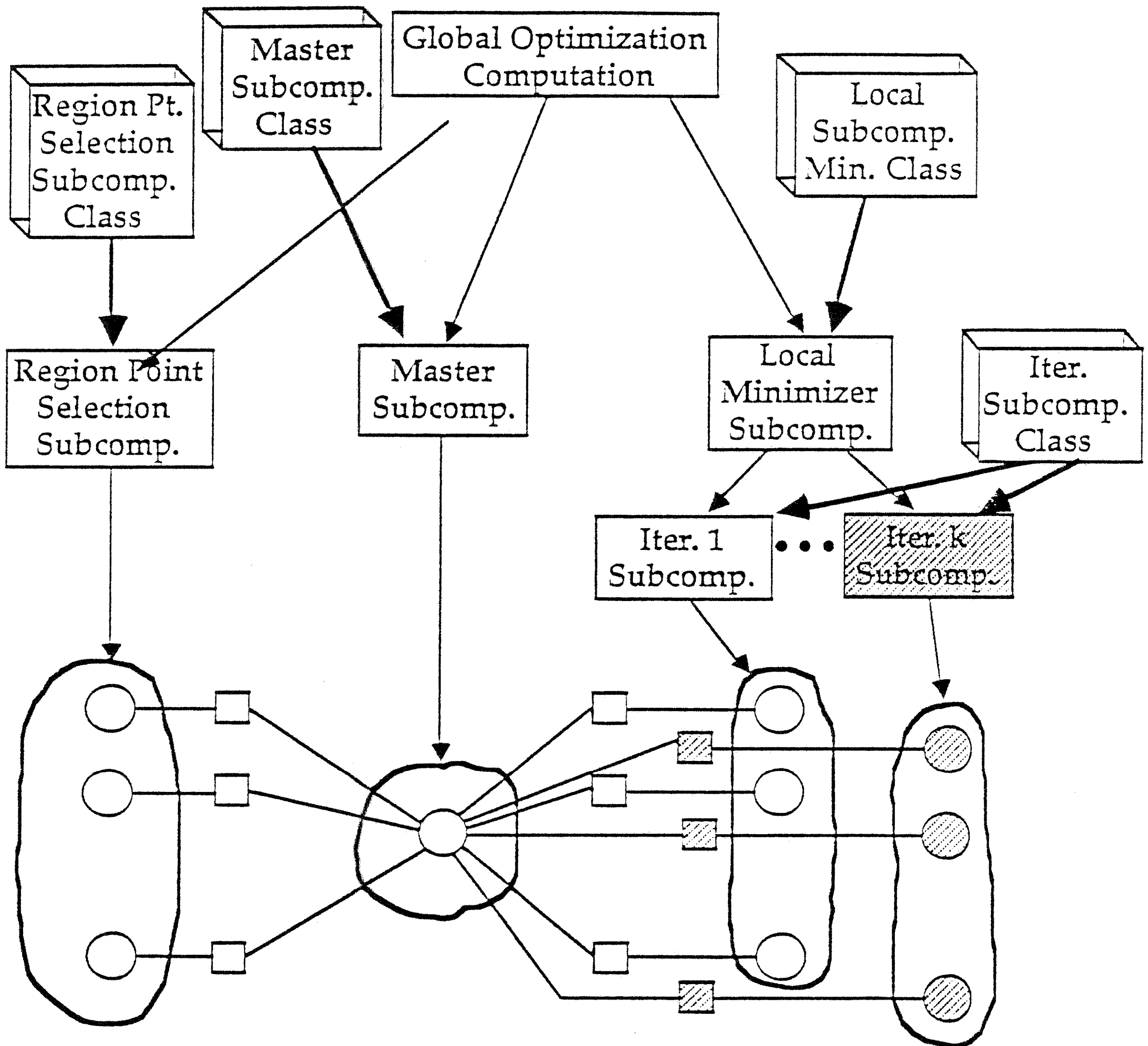
**topology**

Constraint on number of class 1 components: 1 master;
Constraint on number of class 2 components: acquired by master at initialization time;
Constraint on number of class 3 components: acquired by master at initialization time;
Constraint on communication relation instance of class 1: 1 instance for each region-point-selector;
Constraint on communication relation instance of class 2: 1 instance for each local-minimizer;
Topology shape: star;

---




Class-Instance Mapping
Figure 15a




Computation Structure
Figure 15b

### 4.5. Employee's telephone bill computation

#### 4.5.1. Overview of the computation (see Figure 16)

A company has several buildings in a big city. Each building has a private autobranch commutator (or PABX), which keeps track of all the telephone calls given either locally (from an extension of the building to another), or to the outside (from an extension of the building, to any outside telephone number). In practice, each time an employee gives a telephone call, a record containing the extension number of the originator, the destination number, and the duration of the call, is composed and stored by the PABX. Every night, the resulting file is sent by the PABX to a central computer which is in charge of the facturation for the company.

The central system is connected to a network of machines. Every night, an operator starts the telephone bill computation. When doing that, he specifies a command file specifying which treatments are to be performed. The options are (for each building): getting a listing of the calls, getting statistical information on the calls of the day ( total number of local calls, average number of local calls per employee, total number of outside calls, average number of outside calls per employee, average duration of the calls, ....), getting a bill for each employee.

#### 4.5.2. Implementation

The computation starts with a master process which receives the files from each building, forks a worker for each file, and collects the final results. The worker processes are given the part of the command file they are concerned with. They, themselves, spawn a "level two" worker process to perform each option requested (listing of the calls, statistical information and billing). If the number of calls recorded in the file exceeds a threshold that is a parameter of the system, a worker process is automatically forked to compute statistical information, whether requested or not. The "level two" worker processes write their results into files and report to the "level one" worker process that forked them before they terminate.

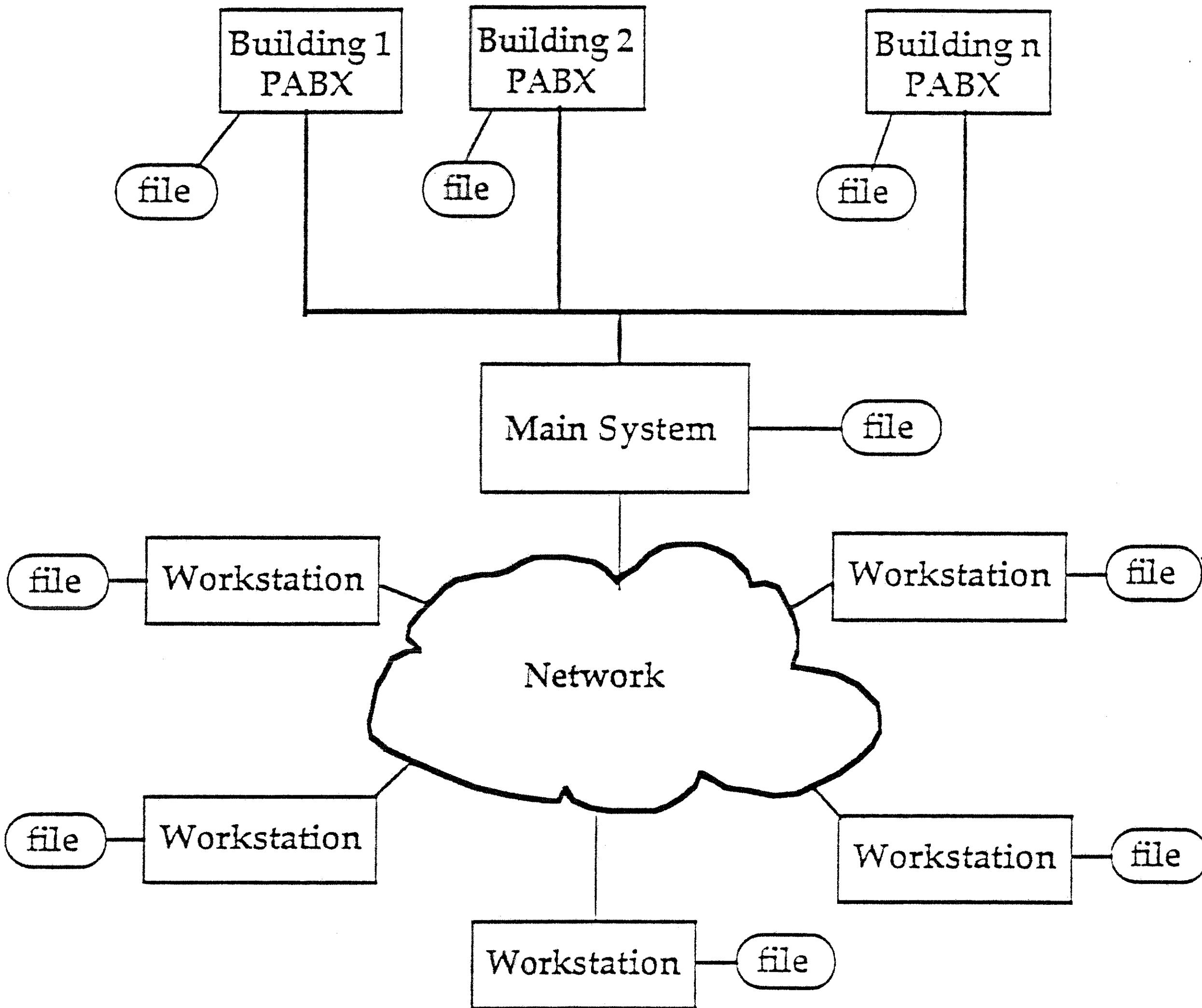
#### 4.5.3. Characterization of the computation

There are five classes of components: the master, the "level one" workers, the level two workers responsible for listing the calls or "listing workers", the level two workers responsible for gathering statistical information or "statistical workers", the level two workers responsible for establishing the bills or "billing workers".

The computation is dynamic bounded. There is only one master and as many "level one" workers as there are buildings (the number is fixed and known). There are as many "listing workers" and "billing workers" as requested (at most as many as there are buildings), and the number can be determined at initialization time. There are at most as many "statistical workers" as there are buildings. The exact number can only be determined at runtime after counting the number of records in each file. This is why the computation is dynamic bounded.

The master is in point-to-point communication with each "level one" worker. Each level one worker is in point-to-point communication with the level two workers it forked.

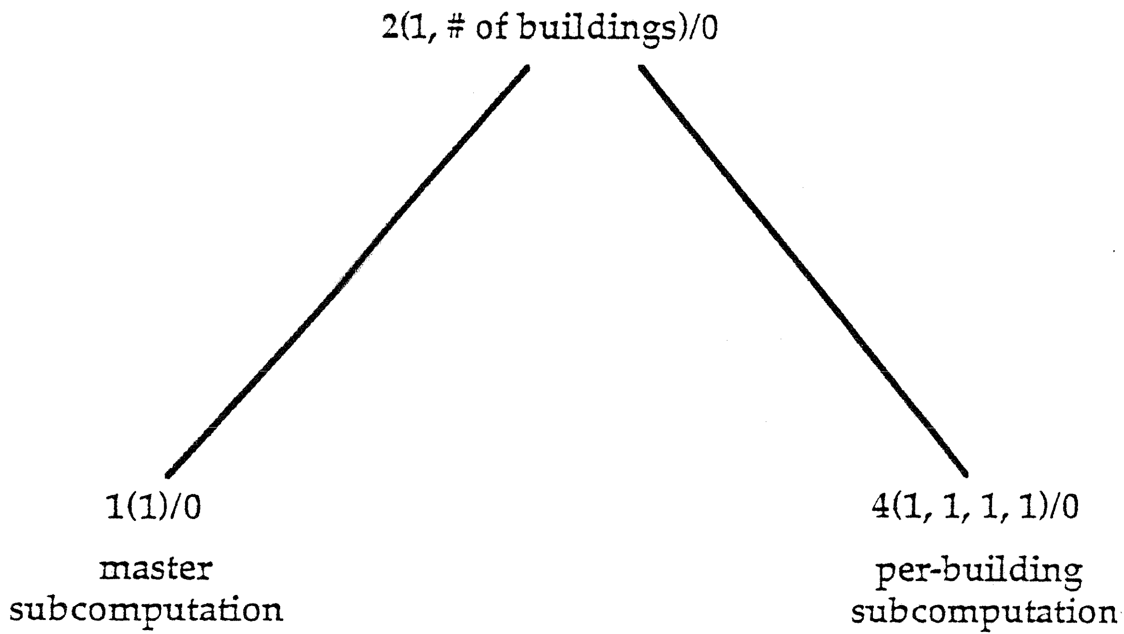
The topology is a tree with two different levels (level one workers, and level two workers).




Overview of the System
Figure 16

*The employee's telephone bill computation* can be broken into 1 master subcomputation, and as many "per building" subcomputations as there are buildings. It is therefore 2(1, #buildings)/0-structured. The master subcomputation is 1(1)/0-structured. Each "per building" subcomputation involves four component classes - level 1 worker class, billing worker class, listing worker class, and statistical worker class -, and at most one instance of each (depending on the options selected). It is therefore 4(1,1,1,1)/0-structured. This partition is the one that seemed the most natural, and the most appropriate to model the behavior of the computation. We see the computation as a number of subcomputations (the "per building" subcomputations), running in parallel.

Figure 17 shows the tree of complexity of the computations at each level. Figure 18 gives the description of the computation; Figure 19 provides the Component-Instance and the Computational Structure views of the computation.




Complexity Diagram
Figure 17

**Figure 18 - Description of the employee telephone billing system**

**components**

Component class 1: master class.
Description: The master process acquires the the files from each PABX. It then
forks a level 1 worker for each file, gives it the part of the command
file it is concerned with, and collects the final results.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;

List of relation classes involving this component class:
- relation class 1 (master/level 1 worker);

Component class 2: level 1 worker class.
Description: The level 1 worker processes are given the part of the command
file they are concerned with. They, themselves spawn a "level two"
worker process to perform each option requested (listing of the calls,
statistical information and billing).
If the number of calls recorded in the file exceeds a threshold that
is a parameter of the system, a worker process is automatically forked
to compute statistical information, whether requested or not.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;
- file to work on;

List of relation classes involving this component class:
- relation class 1 (master/level 1 worker);
- relation class 2 (level 1 worker/ billing worker);
- relation class 3 (level 1 worker/ listing worker);
- relation class 2 (level 1 worker/ statistical worker);

Component class 3: billing worker class.
Description: The billing workers establish a bill for each employee of the
building they are processing the file of, write the result in a local
file, and report to the level 1 worker who forked them, before they
terminate.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;
- file to work on;

List of relation classes involving this component class:
- relation class 2 (level 1 worker/ billing worker);

Component class 4: listing worker class.
Description: The listing workers output a listing of the calls made from the building they are processing the file of, and report to the level 1 worker who forked them, before they terminate.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;
- file to work on;

List of relation classes involving this component class:
- relation class 3 (level 1 worker/ listing worker);

Component class 5: statistical worker class.
Description: The statistical workers compute statistical information on the calls given from the building they are processing the file of, write the results in a local file, and report to the level 1 worker who forked them, before they terminate.
Properties (of the class): ;
Attributes (defined for each instance):
- process number;
- file to work on;

List of relation classes involving this component class:
- relation class 2 (level 1 worker/ statistical worker);

**Subcomputations**

number of levels: 2;
level 1 or leaf:
- 2 classes of subcomputations;
- class 1 subcomputation: master subcomputation;
    * composed from one component class: master component class;
    * static;
    * 1(1)/0-structured;
- class 2 subcomputation: per-building subcomputation;
    * composed from one component class: per-building component class;
    * dynamic bounded;
    * 4(1, 1, 1, 1)/0-structured;
- one instance of class 1 subcomputation;
- as many instances of class 2 subcomputations as there are buildings;
- not a proper partition;

Computation level:
- 1 instance of class 1 subcomputation (master subcomputation);
- # buildings instances of class 2 subcomputation (per-building subcomputation);
- not proper;
- dynamic bounded;
- 2(1, # buildings)/0-structured;

**communication relations**

Relation class 1: master/level 1 worker communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: master;
- end 2: level 1 worker;
- permanent relation;

Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

Relation class 2: level 1 worker/ billing worker communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: level 1 worker;
- end 2: billing worker;
- permanent relation;

Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

Relation class 3: level 1 worker/ listing worker communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: level 1 worker;
- end 2: listing worker;
- permanent relation;

Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

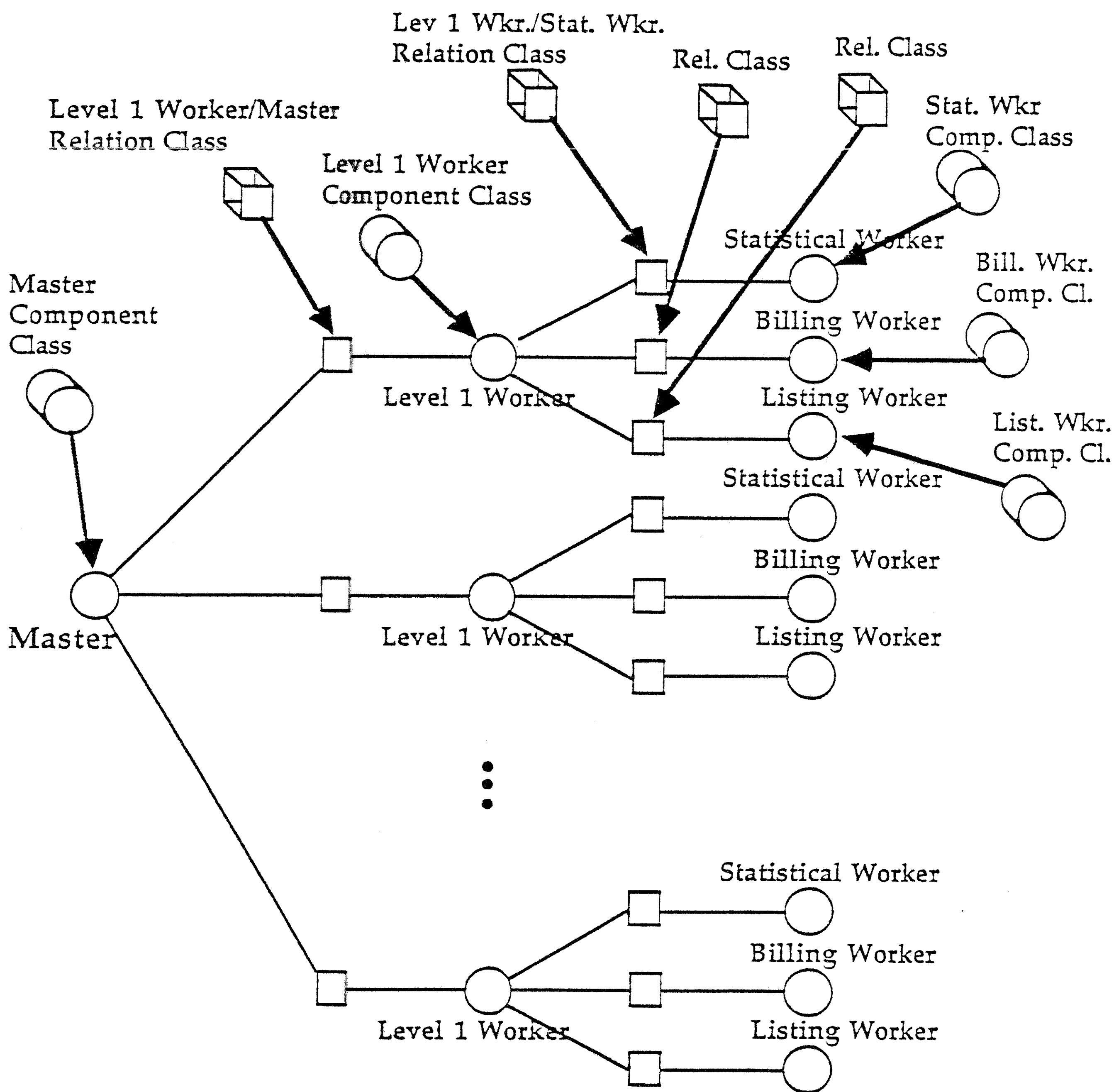
Relation class 4: level 1 worker/ statistical worker communication relation.
Properties (of the class):
- type: point-to-point full duplex;
- number of ends: 2;
- end 1: level 1 worker;
- end 2: statistical worker;
- permanent relation;

Attributes (defined for each instance):
- process number of end 1;
- process number of end 2;

**topology**

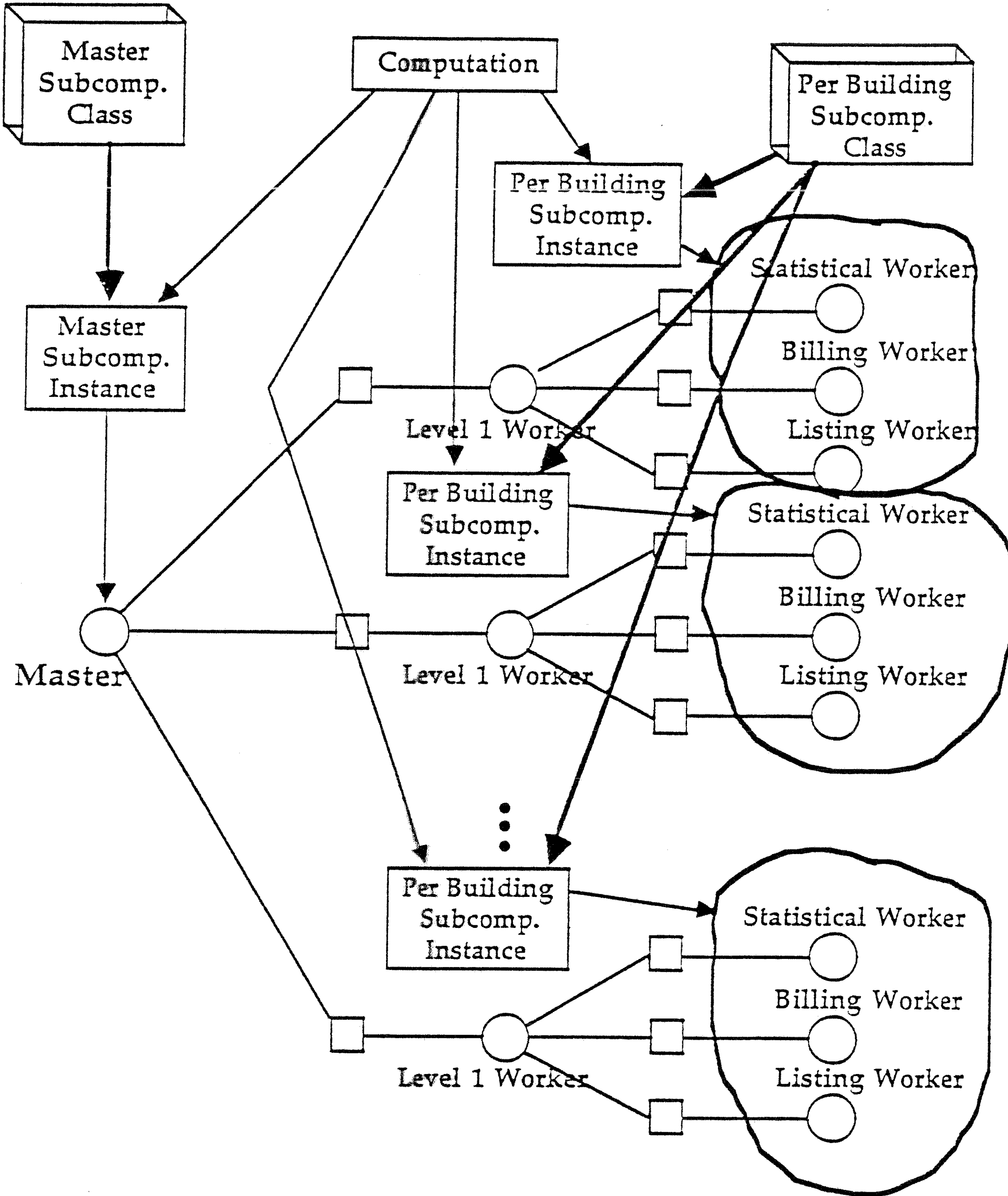
Constraint on number of class 1 components: 1 master;
Constraint on number of class 2 components: # of buildings;
Constraint on number of class 3 components: number requested in command file;
Constraint on number of class 4 components: number requested in command file;
Constraint on number of class 5 components: at most # of buildings;
Constraint on communication relation instance of class 1: 1 instance for each Building;
Constraint on communication relation instance of class 2: 1 instance for each billing worker component;
Constraint on communication relation instance of class 3: 1 instance for each listing worker component;
Constraint on communication relation instance of class 4: 1 instance for each statistical worker component;
Topology shape: tree with two levels;

---




a) Class-Instance Mapping

Figure 19




b) Computational Structure

Figure 19

## 5. CONCLUSION

We have identified some important characteristics of parallel, distributed computations, namely the process classes and the communication relation classes involved in a computation, its topology, its degree of dynamism, its partition into subcomputations, its main phases. We used these characteristics to build a Process Architecture Model (PAM) of parallel, distributed computations; we suggested an informal diagrammatic representation for the model and an informal syntax to describe the characteristics of computations; we, finally exercised the characterization, the model and the description on several examples of parallel, distributed computations.

We believe that this work provides a way of describing and visualizing important aspects of parallel, distributed computations. In addition, this approach encourages us to focus on the macroscopic aspect, or process architecture aspect, of parallel, distributed computations.

As mentioned in the introduction of this document, we are planning to design tools using PAM to build distributed computations, to debug them, and do performance measurements on them. The PAM diagrams are intended to complement the Bilogic Precedence Graphs mentioned earlier. Our tools will use the two types of graphs depending on the level of the view required (PAM diagrams, to get an overall or macroscopic view, and BPGs to get a more detailed view of the computation).

References

1. R. H. Byrd, C. L. Dert, A. H. G. Rinnooy-Kan and R. B. Schnabel, ''Concurrent Stochastic Methods for Global Optimization'', Technical Report CU-CS-338-86, Department of Computer Science - University of Colorado, Boulder, June 1986.

2. I. M. Demeure and G. J. Nutt, ''Modeling Parallel Program Architectures'', Technical Report submitted for publication, Department of Computer Science - University of Colorado, Boulder, July 1987.

3. G. J. Nutt, ''Visual Programming Methodology for Parallel Computations'', to appear in the MCC-University Research Symposium proceedings, Austin, Texas, July 1987.

4. G. J. Nutt, ''Bilogic Precedence Graph Models'', Technical Report CU-CS-363-87, Department of Computer Science - University of Colorado, Boulder, May 1987.