Modeling Parallel, Distributed Computations using ParaDiGM - A Case Study: the Adaptive Global OptimizationAlgorithm.

Isabelle M. Demeure, Sharon L. Smith, Gary J. Nutt

CU-CS-419-88 December 1988

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

This researach has been supported by NSF Cooperative Agreement DCR-8420944, NSF grant CCR-8802283 and AFOSR-85-0251

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION

Abstract

ParaDiGM, the Parallel Distributed computation Graph Model, was designed to model implementations of parallel computations to be run on distributed message-based computer systems. We have used it to model two implementations of a complex adaptive parallel global optimization algorithm. In this paper, we introduce the ParaDiGM constructs, describe the algorithm, and then present the models of the implementations. These examples illustrate ParaDiGM's utility as a modeling formalism for representing and studying implementations of parallel, distributed algorithms.

Contents

1 Overview of the Approach - Background 5

1.1 ParaDiGM 6

1.2 The Adaptive Global Optimization Algorithm 7

2 ParaDiGM 8

2.1 The DCPG - Distributed Computation Precedence Graph - Model 8

2.1.1 Control Flow Constructs 8

2.1.2 Data Flow Constructs 10

2.2 PAM - Process Architecture Model - 11

2.2.1 The Basic Model 12

2.2.2 Classes 13

2.3 Mapping between the DCPG and the PAM Models 13

3 The Adaptive Global Optimization Algorithm 14

3.1 Phases of the Adaptive Algorithm 16

3.1.1 Sampling and Start Point Selection Phase 16

3.1.2 Local Minimization Phase 17

3.1.3 Other Tasks in the System 17

3.2 Two Implementations of the Adaptive Algorithms 18

3.2.1 Homogeneous Implementation: A Detailed Description of the Software Implementation 18

3.2.2 Heterogeneous Implementation: A Conceptual Description of the Algorithm 22

4 The Modeling Experiment 25

4.1 Comparison of the Adaptive Algorithm Models 25

4.2 Evaluation of ParaDiGM 27

5 Conclusions and Future Work 29

6 Acknowledgements 30

References 31

A APPENDIX **33**

A.1 Levels of DCPGs 33

List of Figures

1 Legend for the Control Flow Constructs of DCPGs 9
2 Legend for the Data Flow Constructs of DCPGs 9
3 DCPG Graph for the Fictitious Computation 11
4 Legend for the PAM Graphs 12
5 PAM Instance Graph for the Fictitious Computation 13
6 PAM Class Graph for the Fictitious Computation 14
7 Mapping between DCPG and PAM for the Fictitious Computation 15
8 DCPG for the homogeneous implementation 20
9 PAM Class Model homogeneous implementation 20
10 Mapping between DCPG and PAM of the homogeneous implementation . . . 21
11 DCPG for the heterogeneous implementation 23
12 PAM Class Model heterogeneous implementation 23
13 Mapping between DCPG and PAM of the heterogeneous implementation . . 24
14 Level 1 DCPG for the homogeneous implementation 34
15 mapping between level 1 and level 2 DCPGs for the homogeneous implementation 35
16 Refinement of node 1 for the homogeneous implementation 36
17 Refinement of node 2 for the homogeneous implementation 36
18 Level 1 DCPG for the heterogeneous implementation 37
19 mapping between level 1 and level 2 DCPGs for the heterogeneous implementation 38
20 Refinement of node 1 for the heterogeneous implementation 39
21 Refinement of node 2 for the heterogeneous implementation 39

1 Overview of the Approach - Background

This paper describes our experience with modeling two implementations of an adaptive parallel global optimization algorithm using ParaDiGM - Parallel Distributed computation Graph Model. There were two goals for this study. First, we were interested in evaluating the modeling technique by using it to represent a real and quite complex parallel, distributed algorithm. Second, we needed a way to represent and describe the algorithm in order to evaluate and compare different implementations of it.

When this study began, the work on the adaptive global optimization algorithm had been in progress for several months. The algorithm was fully designed and several implementations were under way. As the work progressed, finding a way to represent the various implementations of the algorithm became an important issue and a challenging problem. The implementations involved many types of different processes that display both dynamic and static behavior and have complex interprocess communication requirements. We needed a representation that could capture the partitioning of an implementation into processes, effectively describe data and its movement between parts of a computation, and distinguish between the different types of process control, process scheduling, interprocess communication, and synchronization that the algorithm required.

In this experiment, we feel that ParaDiGM was successful in capturing all of these aspects. Moreover, the ParaDiGM graphs provide a comprehensive overview of each implementation that allows us to use these graphs to compare the various implementations. Considering that we are able to model complex implementations such as the ones presented here, we expect that ParaDiGM will be an effective and useful technique to model other parallel, distributed computation implementations.

In the remainder of this section, we introduce the ParaDiGM model and give an overview of the parallel global optimization algorithm. In section 2, we give a more complete description of ParaDiGM. In section 3, we describe the global optimization algorithm in more detail and present the ParaDiGM graphs for the two implementations. In section 4, we compare these graphs, and evaluate ParaDiGM.

1.1 ParaDiGM

ParaDiGM is a graph model that can be used to represent implementations of parallel, distributed computations. By "parallel, distributed computations" we mean computations to be run on a local memory multiprocessor machine such as a Hypercube or a network of workstations, using message passing as the interprocess communication mechanism. The model was created to serve other purposes as well as representing and conveying ideas about implementations of parallel, distributed algorithms. It can be used: (1) as a basis to study the characteristics of a given implementation; (2) as a basis to compare and contrast different possible implementations; (3) as a visual interface for graphic tools (we shall not deal with this issue in this paper).

ParaDiGM is composed from two distinct but complementary models: the DCPG - Distributed Computation Precedence Graph - model and the PAM - Process Architecture Model - model. DCPGs are useful for expressing the functionality of an algorithm in terms of tasks, information sharing, and synchronization. The execution of the computation can be simulated using tokens in much the same way that they are used in Petri Nets [Pet81]. The PAM model is intended to represent a different aspect of parallel, distributed computations; it is defined in terms of schedulable units of computations (or processes) and the pattern of information sharing (or communication) among them. It also focuses on the identification of processes that can be implemented by the same module of code (or process class) and of communication taking place among processes of the same class (or communication class). Finally, there is a natural mapping between the two models. We describe ParaDiGM in more detail in section 3.

The DCPG model is based on the BPG - Bilogic Precedence Graph - model created by Nutt [Nut87]; it was designed to model parallel, distributed computations more naturally than the general purpose BPG model did. The literature on graph models is rich. Current graph models generally fall into the class of control flow models such as precedence graphs, [Bro85], Petri Nets [Pet81], or data flow models [BD87]. The Bilogic Precedence Graphs and their DCPG cousins are most similar to other models used in control flow modeling, particularly interpreted Petri nets and E nets [NN73], Information Control Nets [Ell79], performance Petri Nets [HV85], and Timed Petri Nets [Ram74].

The basic graph in the Process Architecture Model resembles resource graphs such as the

one used by Holt [Hol72] to study deadlock properties of computer systems. It is also similar to communication graphs such as the one used in the POKER system [Sny84]. Finally, we see similarities between ParaDiGM and the underlying precedence graph model used by the PARET system [NE88] to represent underlying architectures, algorithms, and the assignment of tasks to processors.

The originality and the strength of ParaDiGM lie in the fact that the model focuses on a specific model of computation: parallel distributed computations using message-passing as an interprocess communication mechanism. The constructs and features used for such computations are therefore built directly in the model, thus providing a very natural way to represent them (unlike more general purpose models do). Another important aspect of the model is that it can be used to provide complementary views of an algorithm.

1.2 The Adaptive Global Optimization Algorithm

The algorithm that we chose for our experiment is a modification of an earlier, static parallel algorithm for global optimization [BDRS86]. Both algorithms are stochastic methods based on the work of [RT84].

The global optimization problem is to find the minimum value of a function that may have multiple local minimizers in some space S. In general, iterative, stochastic methods have approached the problem in the following manner. At iteration k:

(1) Generate sample points and function values.

(2) Select start points for local minimizations.

(3) Perform local minimizations from all start points.

(4) Wait for all minimizations to complete then decide whether or not to stop.

The earlier, static parallel version of this algorithm exploits parallelism in two ways: First, the space, S, is divided evenly into P regions, where P is the number of available processors. The first and second steps of the algorithm are conducted simultaneously on the P machines. The processors synchronize after the second step. Next, the start points selected in step 2 are distributed to processors so that they can conduct local minimizations in parallel from these points. The third step continues until all the start points generated by step 2 have been used.

In the adaptive algorithm, the synchronizations after step 2, step 3, and step 4 are removed. This allows parts of several iterations to be in progress at the same time. The mo-

tivation for this change is to realize increased efficiency from the processors without changing the type of tasks that are being allocated to processors. Contrasted with the static parallel algorithm, the distribution of work within the problem domain S may be uneven. Many functions may have regions within the space S where more local minimizers will be observed; therefore, the regions are allowed to subdivide so that on subsequent iterations a single processor will be dedicated to a smaller region.

Although the ideas behind the new parallel version are conceptually simple, it proved to be a difficult task to describe the algorithm as easily as the earlier version. The primary reason for this difficulty is that the new algorithm necessarily have more complex communication and process generation patterns than the previous version. It is no longer sufficient to simply list the steps of the algorithm. It is necessary to employ a specific mechanism to represent details of the implementations.

2 ParaDiGM

As mentioned before, ParaDiGM is composed from two submodels: the DCPG model and the PAM model. In subsection 1 we describe the DCPG model, in subsection 2 we introduce the PAM model, and in subsection 3 we describe the mapping between the two submodels. The application of ParaDiGM described in this paper is limited to representation; in [DN89] we describe additional applications related to other aspects of design and implementation.

2.1 The DCPG - Distributed Computation Precedence Graph - Model

From the point of view of the DCPG model, a distributed computation is a collection of nodes that represent tasks, local data structures or messages, a control graph that captures the precedence among tasks, and a data flow graph that captures access to local data structures as well as message exchange among tasks. Tasks are organized into threads of control, each thread being a schedulable unit of computation or process. Several threads can be active simultaneously, and two distinct threads can communicate through message exchanges.

2.1.1 Control Flow Constructs

The control flow subgraph in a DCPG is made of nodes of various types and arcs joining the nodes. *"Task"* nodes are used to model any sequence of code (see figure 1 (a)). *"or"*

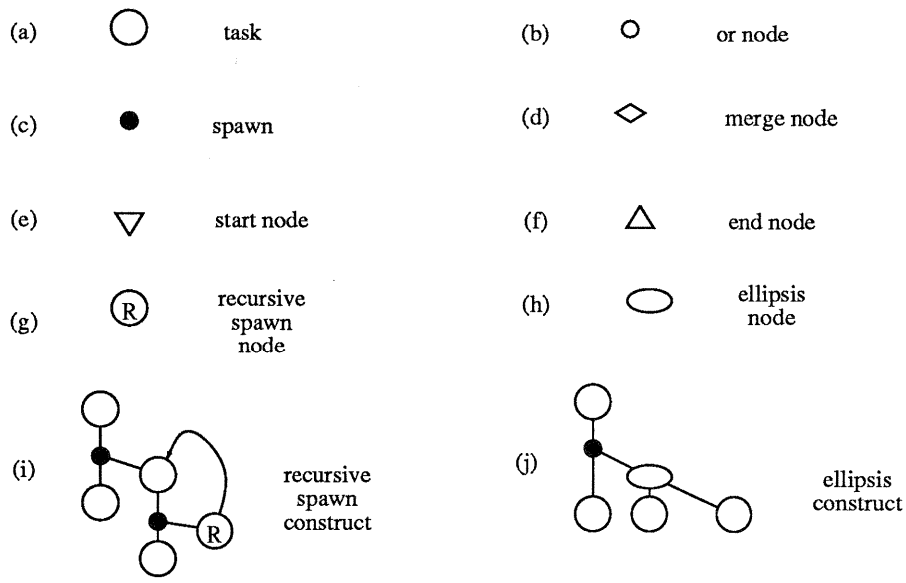

Figure 1: Legend for the Control Flow Constructs of DCPGs

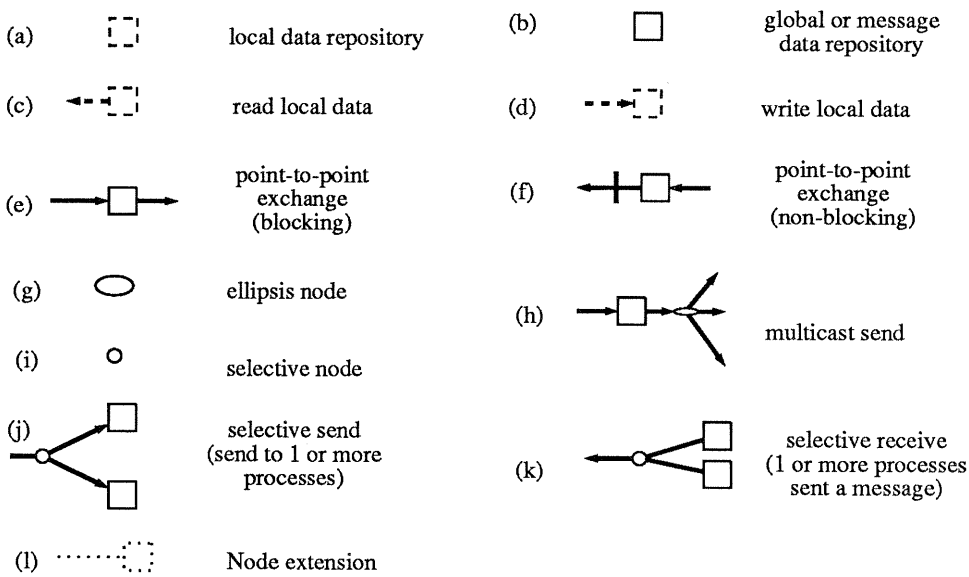

Figure 2: Legend for the Data Flow Constructs of DCPGs

nodes represent alternative choices (see figure 1 (b)). *"Spawn"* nodes model the creation of processes (see figure 1 (c)). More than one process can be created as the output of a spawn node. To model the creation of a number of identical processes, we introduce the *"ellipsis"* construct (see figure 1 (h) and (j)). To model computations in which a process spawns a process identical to itself (or to one of its parents) which in turn spawns a process identical to itself (or to one of its parent), and so on, we employ the *"recursive construct"* (see figure 1 (g)). As shown in figure 1 (i), this is modeled by having a control arc go from a spawn node to a recursive node and another one from the recursive node to the first task node modeling the process that is recursively spawned. We introduce the *"start"* node to model the creation of a process by direct action of the user (to start the first process or processes of a computation); the *"end"* node construct is used to represent the termination of an existing process (see figure 1 (e) and (f)). Finally, the *"merge"* nodes are used as merging points of several control flow arcs to simplify the graph (see figure 1 (d)). Control flow arcs are used to model the control flow of the computation by indicating the order in which nodes must be parsed.

Figure 3 shows an example of a DCPG graph representing a fictitious computation. In this computation there is a *"main process"* represented by nodes 1 to 10 and two "secondary processes" represented by the two identical threads under the ellipsis node 10. These two threads represent the first and the last of a number of identical processes spawned but not represented on the graph.

2.1.2 Data Flow Constructs

The data flow graph in a DCPG is made up of *"local data repository nodes"*, *"message repository nodes"*, data flow nodes and data flow arcs.

Local data repositories are used to model any data structure local to a process (see figure 2 (a)). "Reading from" a local repository is modeled by a data flow arc going from a local data repository to a control flow node; "writing to" a local repository is modeled by a data flow arc going from a control flow node to a local data repository (see figure 2 (c) and (d)).

Message repositories are used to model message passing among processes (see figure 2 (b)). A *message repository* is used to model each exchange (or message) among two or more processes. Message passing arcs link control flow nodes, data flow nodes and message repositories. Several message passing constructs are available. In particular, we can model a

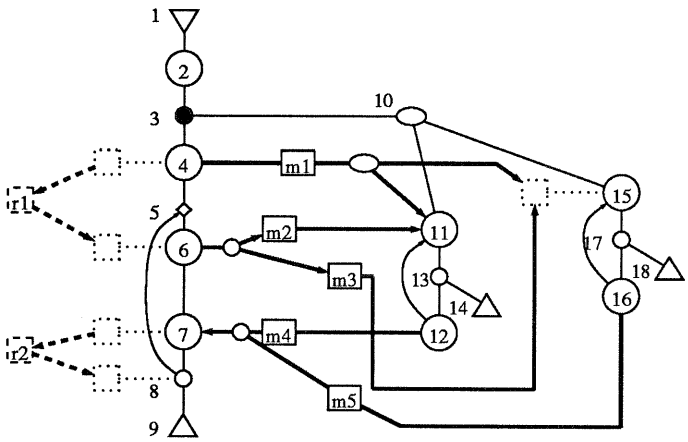

Figure 3: DCPG Graph for the Fictitious Computation

basic "point-to-point" message passing where there is one sender and one receiver (see figure 2 (e) and (f)); we can represent a "multicast" message where there is one sender and a number of receivers (see figure 2 (h)) using the *"ellipsis data flow node"* (see figure 2 (g)); using the *"selective data flow node"* (see figure 2 (i)), we can model a *"selective send"* where there is one sender and one or more possible receivers (see figure 2 (j)) and a *"selective receive"* where there is one receiver and one or more possible senders see figure 2 (k)). The receive primitives can be blocking (this is the default) or non-blocking (see figure 2 (f)).

"Node extension" constructs (see figure 2 (l)) can be added to any node without changing its semantics. They are used in order to make a DCPG graph easier to read.

In figure 3, there are 2 data repositories, r1 and r2, local to the "main process". There are also several examples of message passing primitives. The message exchange that goes through repository m1 is an example of multicast send. The message exchange that takes place through repositories m2 and m3 is an example of a selective send. The message exchange that takes place through repositories m4 and m5 is an example of a selective receive.

2.2 PAM - Process Architecture Model -

From the point of view of the PAM model, a computation is a collection of processes and communication relations that capture the need for processes to exchange messages with one another. Such a view of a computation can be useful in studying the "partitioning" into processes and the "connectivity" of a computation, in evaluating and comparing the communication patterns of two implementations of a computation, and in deciding on the scheduling

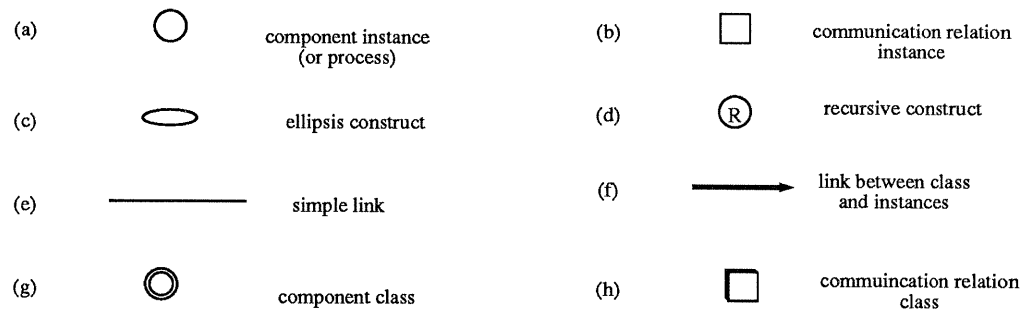

Figure 4: Legend for the PAM Graphs

of processes to processors. The PAM model also provides ways of grouping processes. In particular, it focuses on the identification of identical processes (implementing the same functionalities) or process class, and on that of the identical communication relations (involving the same number of processes of the same class).

2.2.1 The Basic Model

The basic elements of the Process Architecture Model are *components* (that we also refer to as processes) and *communication relations* among these components. A component is a schedulable unit of computation. The purpose of the communication relations is to describe the pattern of sharing information among the components. In component PAM graphs we have four types of nodes: "Component" nodes (see figure 4 (a)), "communication" nodes (see figure 4 (b)), "ellipsis construct" nodes (see figure 4 (c)), and "recursive construct" nodes (see figure 4 (d)). "Simple link arcs" (see figure 4 (e)) indicate how component nodes relate to each other. Figure 5 shows a PAM instance graph for the fictitious computation. The components in this graph correspond to "the main process" and the two "secondary processes". A *multicast communication relation* involving the "main process" and all the "secondary processes" models the need for the "main process" to multicast messages to the "secondary processes". A *point-to-point communication relation* is used to represent the need for each "secondary process" to communicate on a point-to-point basis with the "main process". Note that on the PAM graph we also represent only the first and the last of the "secondary processes". The ellipsis construct is used to model the fact that a process is in a communication relation with a number of identical processes not represented on the graph.

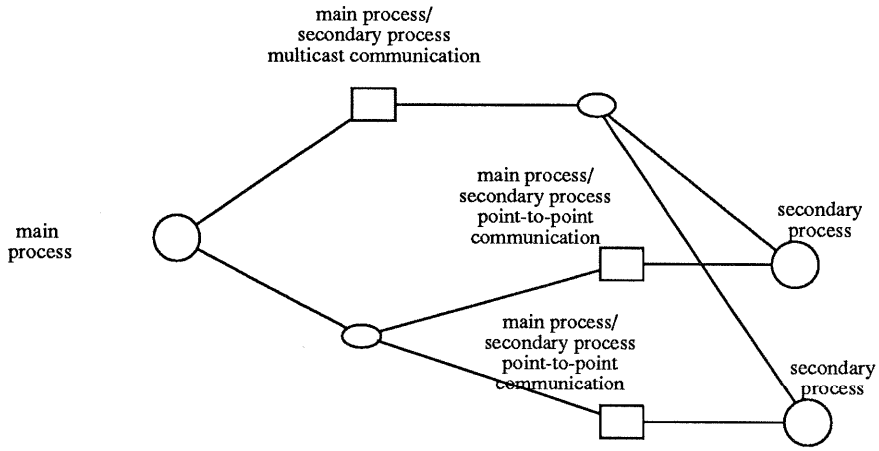

Figure 5: PAM Instance Graph for the Fictitious Computation

2.2.2 Classes

There can be several instances of the "same process"; such processes are instances of the same *process class*. Similarly, identical communication relations (involving processes of the same class, following the same pattern of communication) are considered to be members of the same *communication relation class*.

A PAM class graph can be drawn for each computation. It shows the instances involved in the computation as described in the previous section as well as the classes they belong to (see figure 4 (g) and (h)). Bold arrows (see figure 4 (f)) are used to represent the fact that a given instance belongs to a given class.

Figure 6 shows a PAM class graph for the fictitious computation.

2.3 Mapping between the DCPG and the PAM Models

PAMs and DCPGs are alternative methods to model the same distributed computation depending on the aspects we want to stress. There is a natural mapping between the two submodels: Each thread of control in a DCPG maps onto a process in the PAM and each process in a PAM corresponds to a thread of control in a DCPG. Similarly, the set of message repositories in a DCPG can be partitioned into a number of subsets each of which can be

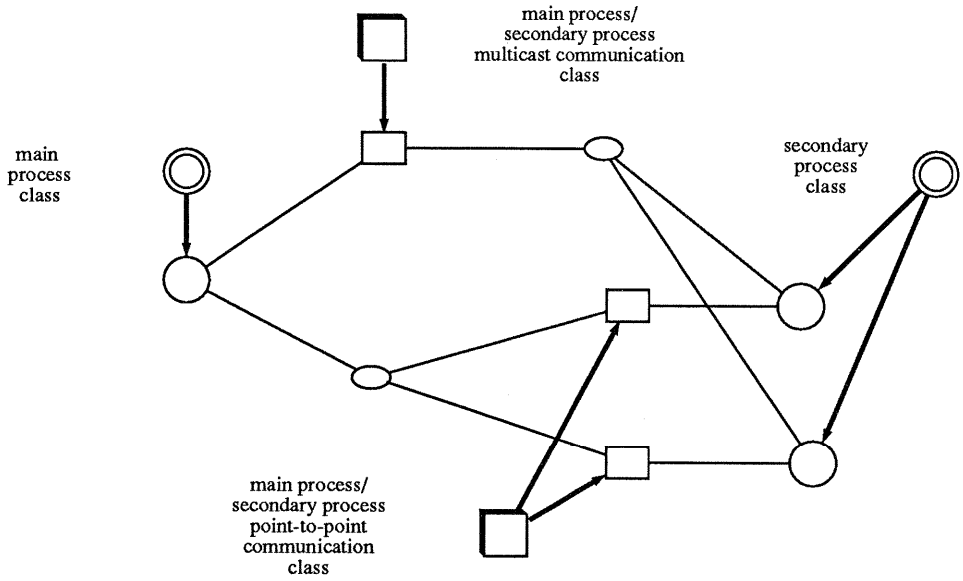

Figure 6: PAM Class Graph for the Fictitious Computation

mapped onto a communication relation in a PAM and each communication relation in a PAM corresponds to a subset of message repositories in a DCPG. Moreover, all instances of a given class should be represented by DCPG parts following the same pattern. Thus, this DCPG pattern can serve as a definition for the class. A mapping graph can be used to show the mapping between a DCPG and PAM. It is made of a DCPG graph and a PAM graph. To show the mapping, we draw a rectangle around portions of the DCPG that correspond either to a process, or to a group of message repositories that must be mapped onto a communication relation. Numbers indicate the correspondence between the DCPG portions delimited by the rectangles, and the instances represented on the PAM graph. Figure 7 shows a mapping graph for the ficticious computation.

We are now ready to apply ParaDiGM to the adaptive global optimization algorithm implementations.

3 The Adaptive Global Optimization Algorithm

Because load imbalance can lead to serious degradation in the overall performance of a parallel system, it is important to ensure that the processors have similar amounts of work to do. It became apparent in observing and modeling the performance of the first parallel version that

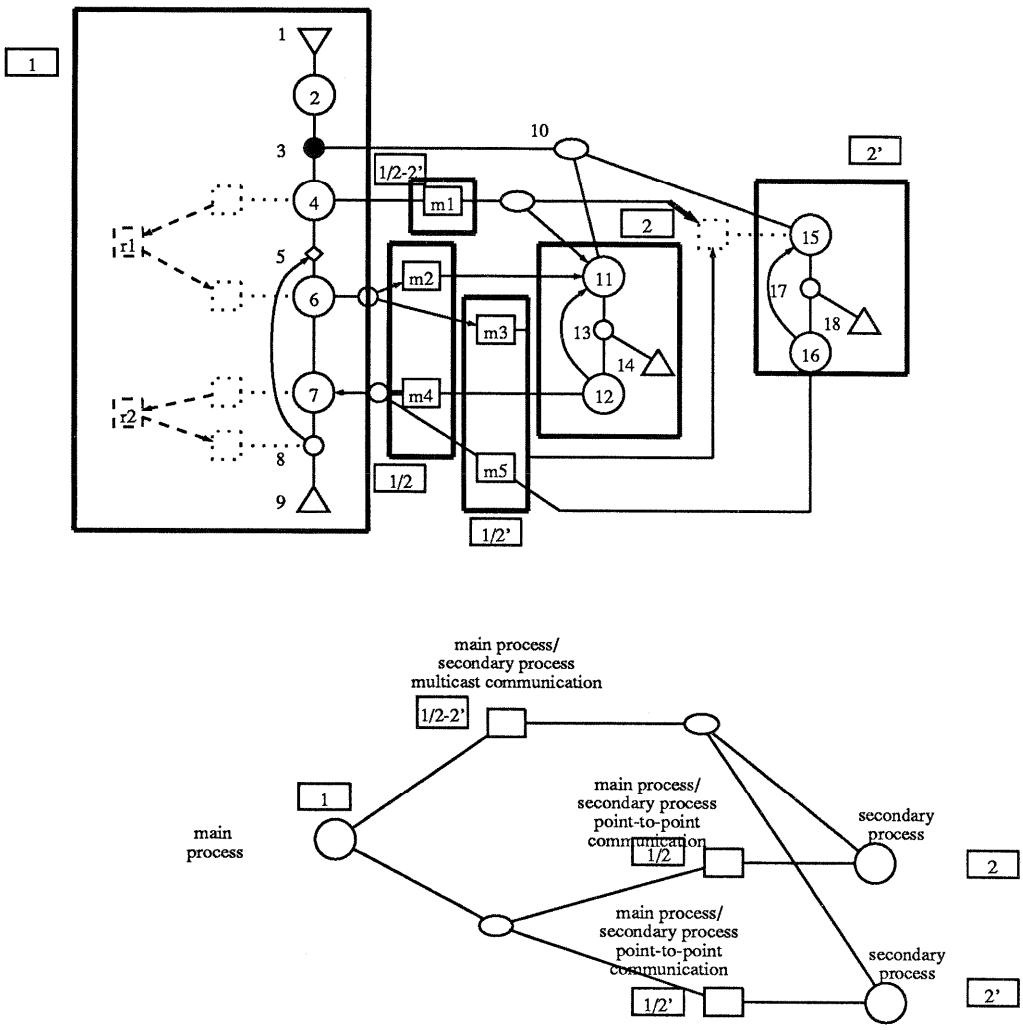

Figure 7: Mapping between DCPG and PAM for the Fictitious Computation

the tasks assigned to processors at various points during the computation could vary greatly in computational requirements, creating large variations in processor utilization [ES88]. One motivation for developing an adaptive parallel version of the global optimization algorithm is to address the problem of load balancing in distributed computations. In the remainder of this section provide a detailed discussion of the adaptive algorithm and two implementations.

3.1 Phases of the Adaptive Algorithm

As mentioned earlier, the first parallel version of the global optimization algorithm can be described as a computation consisting of a collection of *iterations* each having a *sampling phase*, a *start point selection phase*, and a phase where *local minimizations* are executed. The adaptive version of the algorithm uses the same types of phases as the static parallel version. Work is divided in a similar fashion, but the synchronizations between steps are removed. In addition, the problem domain may be further subdivided, adaptively, when more work is required. Phases are scheduled by a master process that distributes work based on processor demand. Load imbalance is addressed by allowing phases and (consequently) iterations to overlap. The asynchronous character of the algorithm is illustrated by the following pseudo code of the scheduler process:

```
REPEAT
    if a processor is available
       then schedule phases of work

    receive new phases of work:
       start point = new local minimization phase
       subregion division = new sample and start point selection phase

    if all phases of an iteration have completed
       then decide whether or not to stop

UNTIL stopping criteria have been satisfied
```

3.1.1 Sampling and Start Point Selection Phase

The purpose of this phase is to select candidate start points for local minimizations from each of the subregions comprising the problem domain S. In each subregion start points are

selected from a set of randomly generated sample points. These start points are then sent to the process which schedules phases of work.

The details of this phase are complicated by two requirements: First, the sample data generated for a subregion during each sampling phase must be accessible to subsequent sampling and start point selection phases occurring in that subregion for the duration of the computation. This requirement is trivially satisfied in the static algorithm because the data associated with a particular subregion are bound to one processor for the entire computation. In the adaptive algorithm, however, subregion boundaries are allowed to be divided, thus producing more subregions than available processors. The concern is how to best manage the sample data for the various subregions among the processors. The two implementations address this issue differently, so we will present the approaches when we discuss the implementations. The second requirement is that the start point selection phase, as it operates on a particular subregion, will need to generate a few points that lie in bordering subregions in order to prevent unnecessary local minimizations. These points should also be considered part of the sample information for the subregion that they belong to. This requirement is satisfied by sending the "oversample" points to a process that determines where they belong. When a sampling and start point selection phase is conducted in one of these bordering subregions, the oversample points are requested from the process and sent to the subregion.

3.1.2 Local Minimization Phase

The local minimization phase of the algorithm performs a local minimization from the start point sent by the scheduler process. When the local minimization is complete, the resulting minimizer is sent to the scheduling process and more work is allocated to that processor.

3.1.3 Other Tasks in the System

Above, we have only alluded to several other "processes" that must be performed in order to manage the parallel computation. These processes are responsible for managing the asynchronous character of the computation. The *scheduler* process is responsible for scheduling phases of the computation. It is also responsible for any initialization that must take place for the computation, including the initial creation of processes for the first phase of the computation. During the computation the scheduler distributes work to processors as they become available and determines when the computation should stop. In addition to keeping

track of which phases are pending and where active phases are running, it keeps track of all the minimizers that have been computed and the size of the sample space for all of the subregions.

The process which manages the oversampling points is known as the *global information handler.* Associated with a subregion is a list of oversample points that have been generated by other subregions. During the sample generation phase of the algorithm, oversample points are requested from the global information handler.

3.2 Two Implementations of the Adaptive Algorithms

In this section we will present two implementations of the adaptive global optimization algorithm and their corresponding ParaDiGM models. The first one, referred to as the "homogeneous implementation", has been fully implemented. The second one, referred to as the "heterogeneous implementation", has been designed but is not yet implemented. The heterogeneous implementation distinguishes between processes that service different types of tasks, whereas the homogeneous implementation does not. The heterogeneous implementation is the one that more closely matches the original conceptual design of the algorithm.

In describing the two implementations, we shall adopt the following terminology. The unit of distribution is referred to as a *process.* Processes are made of a collection of *tasks.* In the previous discussion we described the parts of the adaptive algorithm as *phases.* Each phase is implemented either as a single task or as part of a task. There are two tasks of primary interest to both implementations, the *subregion* task and the *local minimization* task. The subregion task consists of the sampling phase and start point selection phase, while the local minimization task is comprised of the local minimization phase. Finally, the scheduler and the global information handler, are present in both implementations. A primary distinction between the two implementations is how the identified tasks are assigned to processes.

3.2.1 Homogeneous Implementation: A Detailed Description of the Software Implementation

The ParaDiGM model for the homogeneous implementation is described by the graphs presented in figures 8, 9 and 10. Figure 8 is a DCPG for the implementation; figure 9 shows a PAM class graph describing the process architecture of the implementation; figure 10 shows

the mapping between the PAM instance graph and the DCPG graph. In figure 8 the leftmost part of the graph corresponds to the task performed by the scheduler process. In figure 10 all these tasks are mapped to the process labeled number 1. This number is also the one associated with the scheduler process represented in the PAM instance graph below and specifies the mapping of the collection of tasks to the scheduler process.

The top and upper right parts of the graph in figure 8 correspond to the task performed by the global information handler process. The mapping for this process is indicated by the circles labeled number 2 in figure 10.

In addition to the scheduler and global information handler processes, this model requires *worker* processes. The worker processes are created when the scheduler initiates processing and remain active for the duration of the algorithm. Both subregion tasks and local minimization tasks are scheduled for worker processes. The mapping for this process is indicated by the circles labeled number 3 in figure 10.

When a subregion or local minimization task completes execution, it requests additional work from the scheduler. In figure 8 the scheduling of a task is modeled by a message construct coming from the *schedule tasks* node and feeding into the "*MORE WORK?*" node. The scheduler has a local data repository called *work queue* that contains information about pending local minimization or subregion tasks to be executed. The scheduler chooses a task from this queue and send a message to the worker that requested it. An attempt is made to schedule subregion tasks for the same worker process each time; this is not always possible because subregions may split or the same worker process may not be available. The sample information associated with a subregion is maintained by the worker process at the processor where the last iteration of a subregion took place. If a subregion is rescheduled for another processor, the worker processes of the old site sends the sample information to the new site. This is modeled in figure 8 as a message coming from the node *get prev-samp, new-samp* of one worker and flowing into the same node on another worker.

We have also used data repositories to represent the *global information* data structure that contains all the oversample information, and the *minimizer list* that contains the current minima.

The PAM graph presented in figure 9 shows the various processes involved in the implementation and their need to communicate with one another. In particular it shows the three classes of processes and their cardinality; there is a single scheduler process and a single

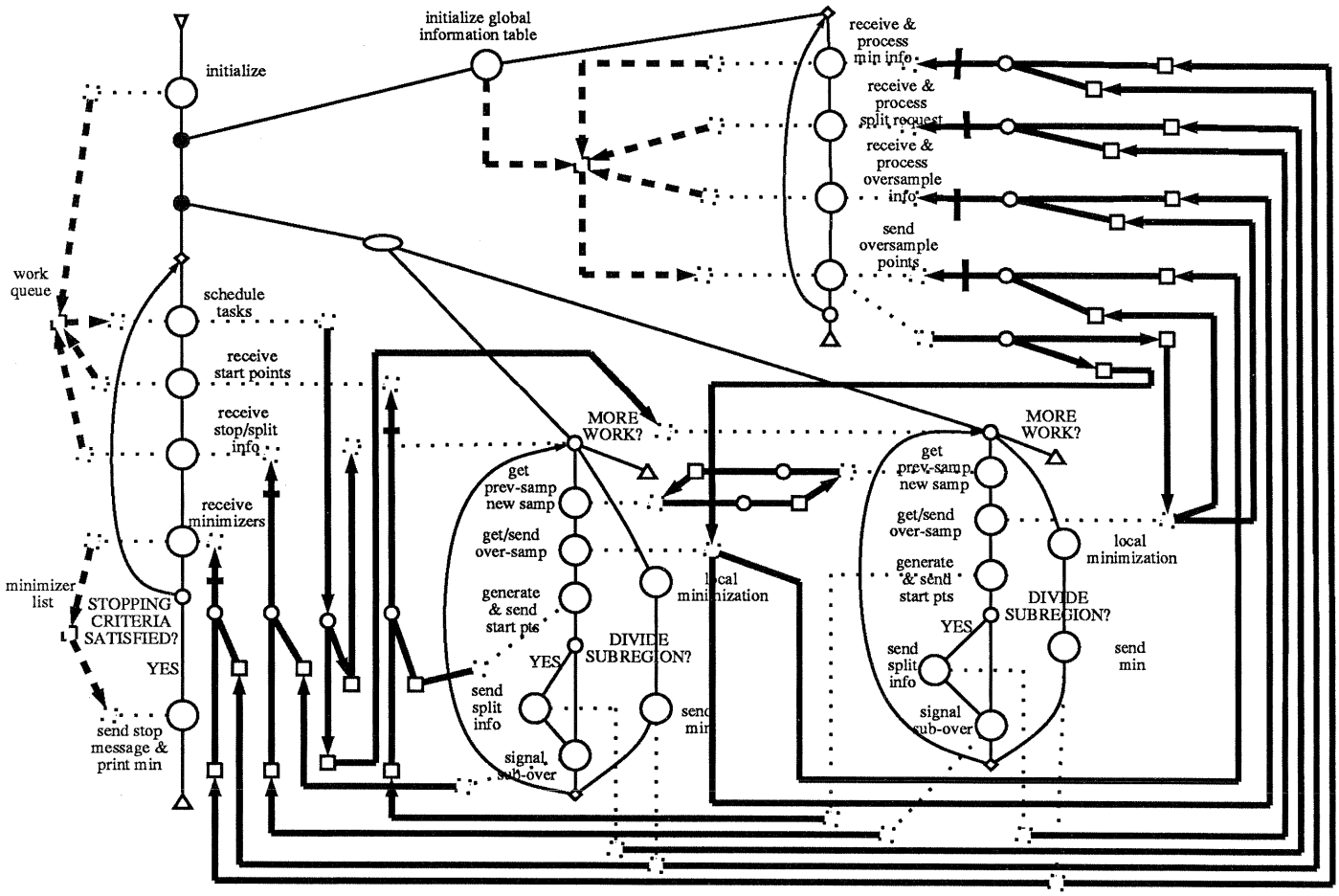

Figure 8: DCPG for the homogeneous implementation

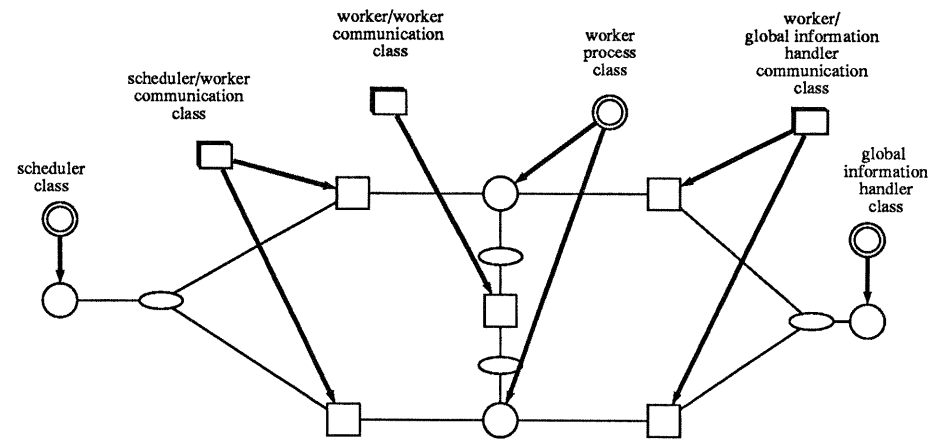

Figure 9: PAM Class Model homogeneous implementation

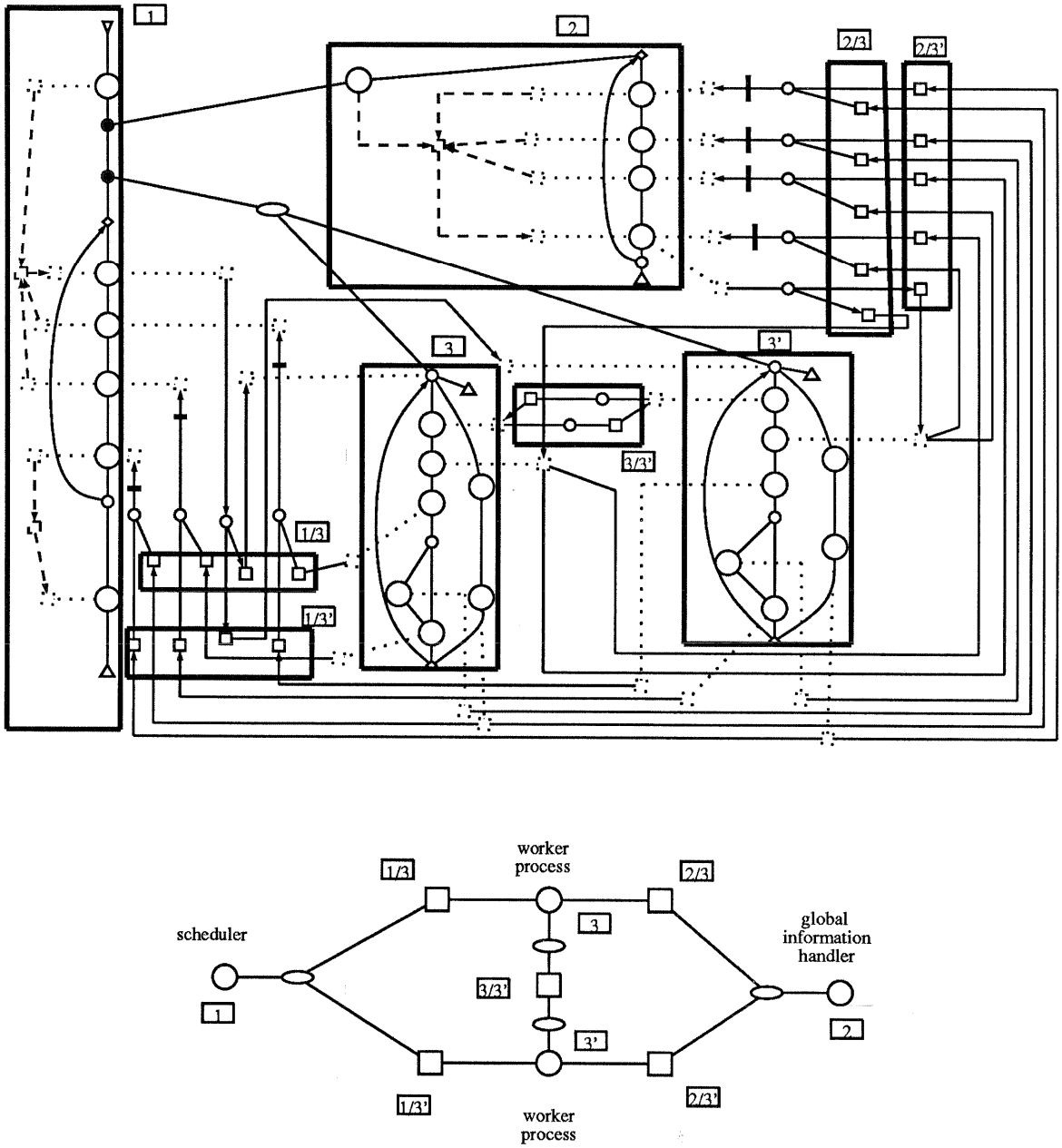

Figure 10: Mapping between DCPG and PAM of the homogeneous implementation

global information handler process, but a number of worker processes. All the messages exchanged between a given worker process and the scheduler form a communication instance of the communication class within the PAM model called the scheduler/worker communication class (see figure 9). Communication classes also exist between the worker processes, forming the worker/worker communication class, and between the worker and the global information handler, forming the worker/global information handler communication class. Note that there is no scheduler/global information handler communication class.

3.2.2 Heterogeneous Implementation: A Conceptual Description of the Algorithm

The ParaDiGM model for the heterogeneous implementation is described by the graphs presented in figures 11, 12 and 13. Figure 11 is a DCPG for the implementation; figure 12 shows a PAM class graph describing the process architecture of the implementation; figure 13 shows the mapping between the PAM graph and the DCPG graph.

In this model the global information handler process is the same as in the first model. The task nodes corresponding to this process are the same as those in the homogeneous implementation, and the mapping from the DCPG model to the PAM model is also the same, labeled as number 2 in figure 13.

The task nodes for the scheduler are presented in figure 11. These task nodes are circled and labeled number 1, mapping to the scheduler process in the PAM model as shown in figure 13. The changes to process creation and scheduling are as follows. First, the scheduler creates an initial set of *subregion* processes, equal to the number of processors available, that perform the subregion task. The subregion processes consist of task nodes corresponding to the *sampling phase* and the *start point selection phase* of the algorithm. When a subregion task decides to divide the subregion, a new subregion process is spawned that corresponds to half of the divided subregion. This is modeled in the DCPG with a recursive construct. The sample information associated with this half of the subregion is managed by the new process. The sample information corresponding to the other half of the subregion is retained by the subregion process that initiates the division. The scheduler is still responsible for determining where the subregion processes execute. The essential difference between a subregion process and a worker process is that a subregion process is not bound to any one processor.

When a subregion task completes its start point selection phase, it sends the start points to

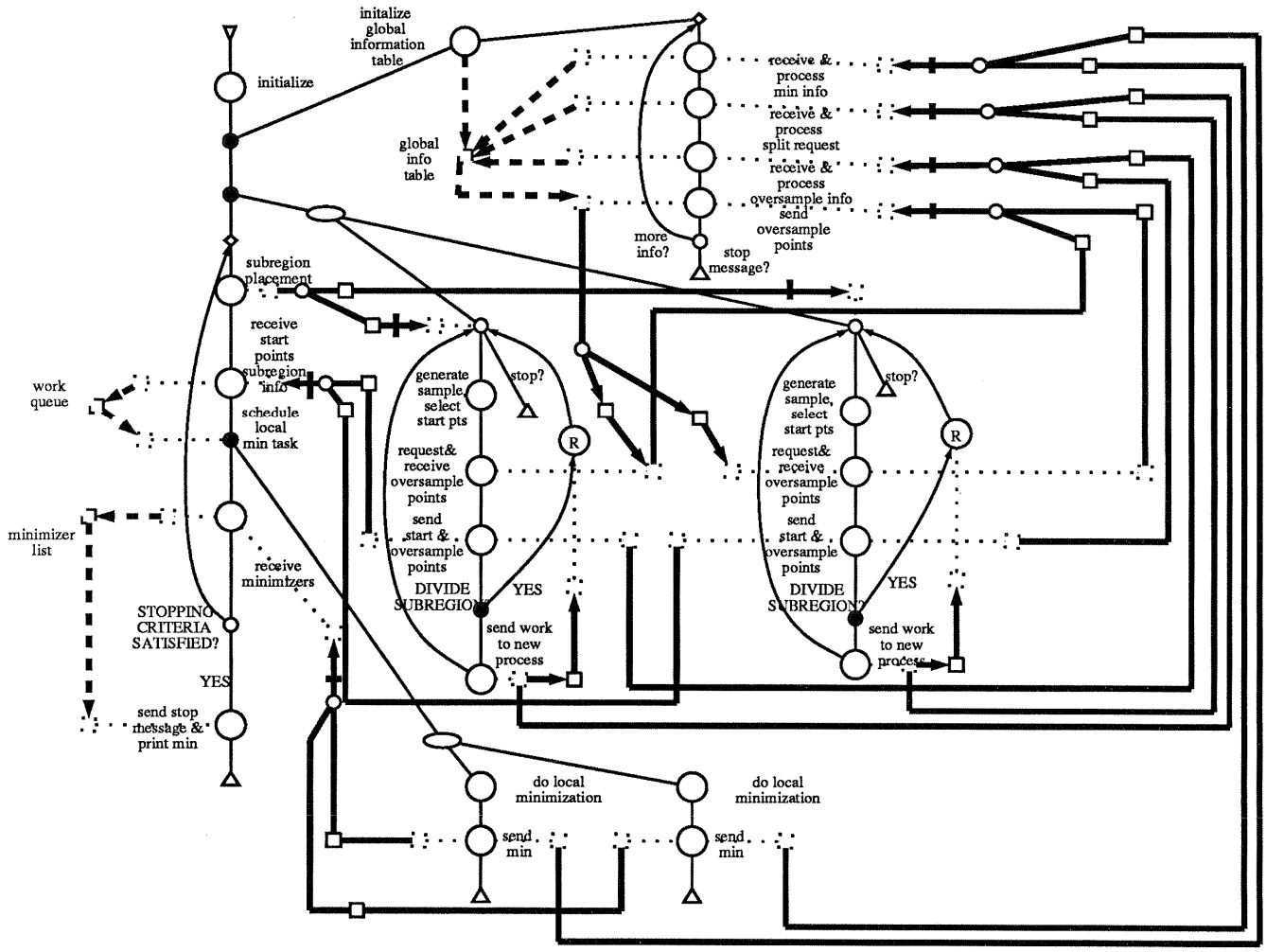

Figure 11: DCPG for the heterogeneous implementation

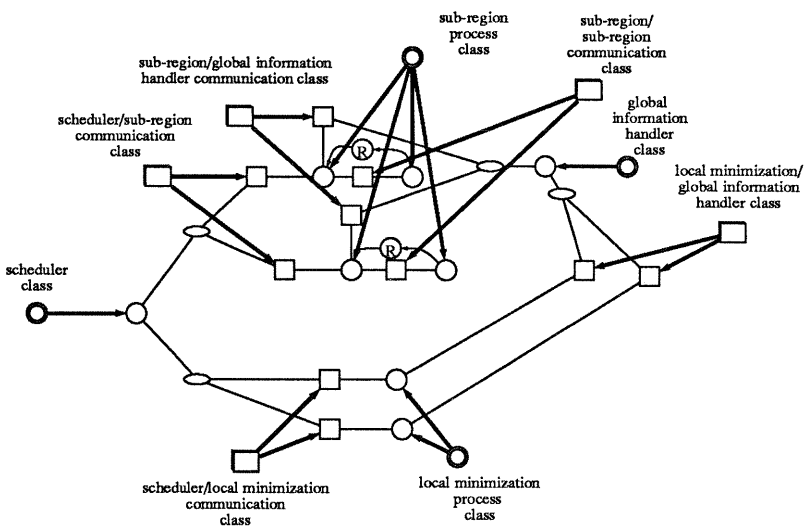

Figure 12: PAM Class Model heterogeneous implementation

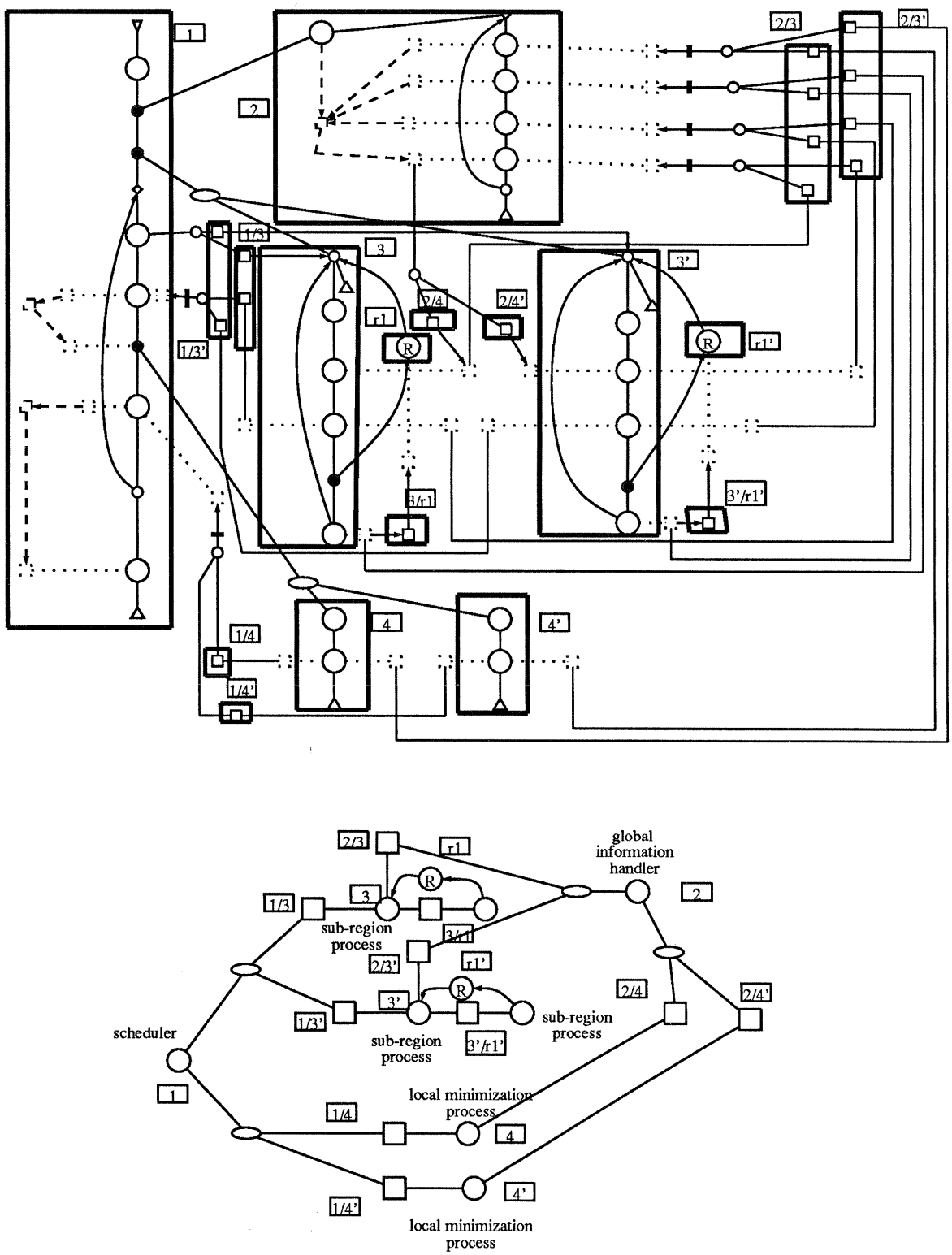

Figure 13: Mapping between DCPG and PAM of the heterogeneous implementation

the scheduler. When a processor becomes available, the scheduler creates a *local minimization* process, which executes on the desired node. The process is active only for the duration of the local minimization *task*. A distinction between local minimization processes and subregion processes is that subregion processes are active throughout the lifetime of a computation, servicing several iterations of subregion tasks, while local minimization processes service local minimization tasks for one iteration only.

The DCPG/PAM mappings for the subregion processes and local minimization processes are labeled number 3 and 4, respectively, in figure 13.

In this implementation there are 4 process classes and 5 communication classes, shown in the PAM model of figure 12.

4 The Modeling Experiment

4.1 Comparison of the Adaptive Algorithm Models

In this application of ParaDiGM, one of our goals is to compare structural aspects (processes and interprocess communication relationships) in the two implementations of the adaptive global optimization algorithm. (In other studies, we are also interested in relative performance, process/processor mapping strategies, software distribution, etc.) Here, we only attempt to infer design aspects of the two implementations from static representations of the implementations, without actually implementing the algorithms.

The two models describe alternative implementations of the same algorithm: In the homogeneous implementation, a static set of P worker processes is created during the initialization phase and the scheduler process is used to assign work when a worker process completes one phase and is ready to move onto the next one. In the heterogeneous implementation, the worker processes are replaced by subregion processes. Initially, there are P subregion processes but that number will change as the algorithm logically changes the number of subregions in which minimizations are to be conducted. Additionally, the scheduler process creates new local minimization processes to process the start points within a subregion.

The homogeneous implementation is communication-intensive, coordinating the activity in the various worker processes, while the heterogeneous implementation tends to trade off communication for process creation and destruction overhead. A comparison of Figures 10 and 13 quickly illustrates the proliferation of processes employed in the heterogeneous

implementation. That is, the homogeneous implementation is based on a static number of $P+2$ processes (the scheduler, the global information handler, and the P worker processes), while the heterogeneous implementation employs a dynamic set composed of one scheduler process, one global information handler process, N local minimization processes, and $P+k$ subregion processes (N and k are determined by conditions that exist in the data, and reflects the adaptivity of the algorithm).

The verbal description of the heterogeneous implementation defines the processes, but Figure 13 makes the number and intercommunication relationships among them much more specific. It should be clear from the comparison of Figures 10 and 13 that the implementations are considerably different in terms of process management.

Based only on the PAM models, one may infer that the homogeneous implementation is superior to the heterogeneous implementation. However, when we compare the two DCPGs (Figures 8 and 11), the differences in interprocess communication loads become apparent. The global information handler has similar communication responsibility in both cases. However, the heterogeneous implementation intends to package functionality in such a manner that communication between the scheduler and the processes that execute the fundamental parts of the algorithm is minimized (compared to the homogeneous implementation). While there are more types of communication in the heterogeneous implementation (see Figures 9 and 12), the nature of each kind of communication is simplified; overall, Figures 8 and 11 indicate that there are fewer paths of communication among processes in the heterogeneous implementation than in the homogeneous implementation.

Figures 8 and 11 also suggest the additional control complexity of the heterogeneous implementation over the homogeneous implementation. The control flow subgraph for the heterogeneous implementation is more complex than that for the homogeneous implementation, since it must reflect the dynamic process management versus static process management. The dynamism in the heterogeneous implementation is evident in the scheduler's main loop, and in the recursion in each subregion process.

Despite the proliferation of processes in the heterogeneous implementation, it is a closer conceptual match to our intuition of the operation for adaptive global optimization. The homogeneous implementation was implemented because there was concern about the performance of the heterogeneous implementation. In particular, we felt that the heterogeneous implementation would suffer from process creation and destruction overhead, and in dissem-

inating process topology information as the topology changed. To address these issues, we implemented a form of "lightweight" processes [A*86]. Since the host operating system did not support this concept, we coalesced logical processes into a general-purpose worker process. In this case, a worker process can take on different roles, depending upon the need at the time; the effect is similar to multiprogramming lightweight processes. Unfortunately, the worker processes are more complex than the ones we had forseen prior to implementation. The most difficulty arises in multiplexing and demultiplexing the interprocess communication among the scheduler and workers. However, as desired, the number of processes is static and smaller than in the heterogeneous implementation strategy.

In hindsight, it appears that we could have reached the same conclusions by working on ParaDiGM models that we reached by experimenting with actual implementations. Since we did the implementation before we did the modeling it is difficult to quantify the amount we would have learned from the models if we had done them before the implementation.

4.2 Evaluation of ParaDiGM

A second goal of this study was to experiment with ParaDiGM as a practical tool for representing process architectures. Upon casual implementation, adaptive global optimization appears to differ from the conventional static distributed global optimization in minor ways. However, the resulting algorithm has considerably different performance aspects under some loading conditions. The verbal description of the adaptive approach attempts to emphasize that synchronization is removed after various stages by allowing processes to lag behind, or proceed ahead of, the majority of the processes. Without a model such as ParaDiGM, our means of describing this idea more precisely is a pseudo code representation.

Thus, ParaDiGM can be used as an intermediate language between concepts and (pseudo) code. In this paper, we illustrate the utility of ParaDiGM by representing two different implementations of the same algorithm. The fact that we have a language that is detailed enough to compare and contrast the implementations suggests utility for the model. Our subjective claim is that, while the reader may take exception to ParaDiGM, it is considerably more precise and useful for describing the operation of the implementation than is a verbal description, and more useful than a conventional pseudo code description.

ParaDiGM provides a mechanism by which the algorithm designer can represent the implementation: He can identify processes, and refine the representation to illustrate process

control, interprocessor communication types and structures, synchronization, and scheduling. In a verbal description, each of these aspects of the implementation can be described, although extreme care must be taken to assure that each issue is addressed, and that the explanation is unambiguous and sufficiently detailed. A ParaDiGM description is not guaranteed to be unambiguous nor to contain sufficient detail for all purposes; however, the model encourages that these issues be addressed at the appropriate level of detail.

The experience with modeling the implementations has not been exclusively positive. Like most languages, certain ideas may be difficult to express in the language. For example, we wanted to be able to give an overview of the implementation to illuminate interesting features without minimizing important details. We found it useful to be able to represent a complex algorithm by a DCPG graph that fits on a page, even if it appears "dense". We used the two DCPG graphs to present the two implementations to colleagues and they generally proved to be a good medium. We could easily point at parts of the graphs to support explanations.

There is a learning curve associated with using ParaDiGM. While our modeling work is not sufficiently advanced to have complete documentation, our observation was that it was relatively easy to learn to use ParaDiGM. New ParaDiGM users read a technical report describing the approach [DN89] and review several example models. We expect that our subsequent machine aids for supporting ParaDiGM will allow that process to be considerably less formidable. The modeling primitives were easily assimilated, although the novice user would sometimes have no particular understanding about when to use one construct over another.

The DCPG constructs were found to be adequate for the particular function, control, and communication aspects of the algorithm implementations that we considered.

The PAM models were more difficult to learn and develop, not because the constructs are abstruse or insufficient, but because the utility of the PAM model is not obvious initially. Since process abstractions are sufficiently different from procedural descriptions, it is difficult to appreciate PAMs until one concentrates on processes instead of tasks, on the partitioning of the computation and the identification of the types of processes needed, and on the interaction among processes. We envision that PAM will be extremely useful in deciding how to assign processes to processors, or, more specifically, as an aid in minimizing the communication between processors and in load balancing.

The primary limitation of the ParaDiGM formalism at this time is the size of problems

that the model can address. While we were able to represent a fairly complex algorithm on a single sheet of paper, it is not difficult to envision distributed computations that would require much more detail. This issue has been addressed in part with the concept of levels (see appendix), but a satisfactory solution is not yet available. This limitation seems to be an inherent problem of most visual modeling formalisms [SG88].

5 Conclusions and Future Work

There have been a number of different software models over the last twenty years. The spectrum has ranged from mathematical specifications that precisely define the action of the system, to purely representational models that ignore specification. Distributed computations challenge model methodologies at all parts of the spectrum, since they not only introduce parallelism, but they also rely heavily on correct synchronization and communication. Many of these traditional models tend to either ignore the issues associated with concurrency, or to treat it in an overly formal manner.

ParaDiGM is intended to represent distributed computations in terms of the functionality, process partitioning, interprocess communication, synchronization, control tasks, and data flow. We know of no other model that addresses all of these issues in a single framework.

The model has been continually refined over a couple of years, based on experiments in representing various computations and algorithms. In our first attempts at using ParaDiGM, we studied diverse classes of systems and algorithms, ranging from the chaotic relaxation (successive overrelaxation) for solving a system of linear equations, to client-server operation, to a model of accounting practices used in a PBX. Early experiments caused us to reevaluate ParaDiGM, and to modify the features correspondingly. The model has changed an insignificant amount in the last year (when the work described in this paper was conducted).

On a pragmatic level, it is difficult to talk about strategies for employing parallelism. In the case of adaptive global optimization, the verbal description conveys the general idea, but abstraction necessary to choose between worker processes and subregion processes is difficult to grasp until one designs the code or the systems. We have illustrated an intermediate representation mechanism, and argued for its utility, by describing the adaptive global optimization implementation verbally and with ParaDiGM. While we have no quantitative argument that the ParaDiGM models are superior to the verbal models, we are able to re-

port that the models provide a more lucid description of the tradeoffs involved in evaluating the two approaches than was available using pure abstraction or verbal descriptions. *One systematic way of representing the implementations is better than none at all.*

The adaptive global optimization implementation models could be criticized as being overly complex ("busy"). However, these implementations are nontrivial; the models merely reflect the complexity of the implementation rather than modeling overhead or a particular modeling approach. The differences describe alternative process partitioning strategies, which is exactly the purpose of such a model.

Even so, the question remains about the scalability of ParaDiGM on larger problems. One of the directions of the continuing research is to incorporate a hierarchy within DCPGs to model computations at varying levels of detail. At the first level of the hierarchy the DCPG graph is represented with as few nodes and communication arcs as possible. At subsequent levels, parts of the graph can be refined to provide more details about the computation. This approach allows one to model bigger computations, as well as to encourage top-down development of the design.

Other research directions include the development of tools that employ ParaDiGM as a graphical interface to design, simulation, and execution of parallel, distributed computations.

We will also continue our modeling work for other implementations of the adaptive global optimization algorithm, as it has highlighted many issues in the current work. As part of this ongoing effort, we have developed a simulation of the homogeneous implementation using queueing network models. We will subsequently analyze and compare the queueing network simulation model with a ParaDiGM simulation model.

6 Acknowledgements

We would like to thank Betty Eskow and Bobby Schnabel for their helpful comments on the paper and the models of the adaptive global optimization algorithm.

References

[A*86] Mike Accetta et al. Mach: a new kernel foundation for unix development. In *USENIX Conference Proceedings*, June 1986.

[BD87] R. G. Babb and D. C. DiNucci. Design and implementation of parallel programs with large-grain data flow. In L. H. Jamieson, D. B. Gannon, and R. J. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 335–349, MIT Press, 1987.

[BDRS86] 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.

[Bro85] J.C. Browne. Formulation and programming of parallel computations: a unified approach. In *14th International Conference on Parallel Processing*, pages 624–631, St-Charles, IL, August 1985.

[DN89] I. M. Demeure and G. J. Nutt. *Modeling Parallel, Distributed Computations with ParaDiGM.* Technical Report, Department of Computer Science - University of Colorado, Boulder, 1989. To appear.

[Ell79] C. A. Ellis. Information control nets: a mathematical model of office information flow. In *Proceedings of 1979 ACM Conference on Simulation, Measurement and Modeling of Computer Systems*, August 1979.

[ES88] E. Eskow and R. B. Schnabel. *Mathematical Modeling of a Parallel Global Optimization Algorithm.* Technical Report CU-CS-395-88, Department of Computer Science - University of Colorado, Boulder, April 1988.

[Hol72] R. C. Holt. Some deadlock properties of computer systems. *Computing Surveys*, 4(3):179–196, September 1972.

[HV85] M. A. Holliday and M. K. Vernon. *A Generalized Timed Petri Net Model.* Technical Report 593, Computer Sciences Department - University of Wisconsin, Madison, May 1985.

[NE88] K. M. Nichols and J. T Edmark. Modeling multicomputer systems with paret. *Computer*, 39–48, May 1988.

[NN73] J. D. Noe and G. J. Nutt. Macro e-nets for representing parallel systems. *IEEE Transactions on Computers*, C-12(8):718–727, August 1973.

[Nut87] Gary J. Nutt. *Bilogic Precedence Graph Models.* Technical Report CU-CS-363-87, Department of Computer Science - University of Colorado, Boulder, May 1987.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., 1981.

[Ram74] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* PhD thesis, MIT, 1974.

[RT84] A. H. G. Rinnooy Kan and G. T. Timmer. A stochastic approach to global optimization. In P. Boggs, R. Byrd, and R. B. Schnabel, editors, *Numerical Optimization*, pages 245 – 262, SIAM, Philadelphia, 1984.

[SG88] S. L. Smith and S. L. Gerhart. Statemate and cruise control: a case^2 study. In *The Twelth annual international Computer Software and Applications Conference*, October 1988.

[Sny84] L. Snyder. Parallel programming and the poker programming environment. *Computer*, 27–36, July 1984.

A APPENDIX

A.1 Levels of DCPGs

In this appendix we present additional views of the two implementations of the global optimization algorithms. These views illustrate the use of levels to build hierarchies of graphs. At the time this technical report was written, we were still experimenting with the notion of levels and the possible ways to structure them. Figure 14 shows a simpler DCPG for the homogeneous implementation than the one displayed in figure 8. We shall refer to this simpler DCPG as the level 1 DCPG and to the former one as the level 2 DCPG. The reason the level 1 DCPG is simpler than the level 2 DCPG is that some details have been omitted from it. In figure 15 we have circled portions of the level 1 DCPG with thick, dotted lines. These portions are represented in more detail in the level 2 DCPG. Refinements of these portions are showns in figures 16 and 17. By replacing the circled portions in the level 1 DCPG with the corresponding Refinements, we obtain the level 2 DCPG. All arcs have been labeled with matching numbers in figure 15 as well as in figures 16 and 17 to show the mapping between the two levels.

Similarly, figure 18 shows a level DCPG for the heterogeneous implementation. Figure 19 shows the mapping between the two levels. Figures 20 and 21 are refinements of two level 1 nodes.

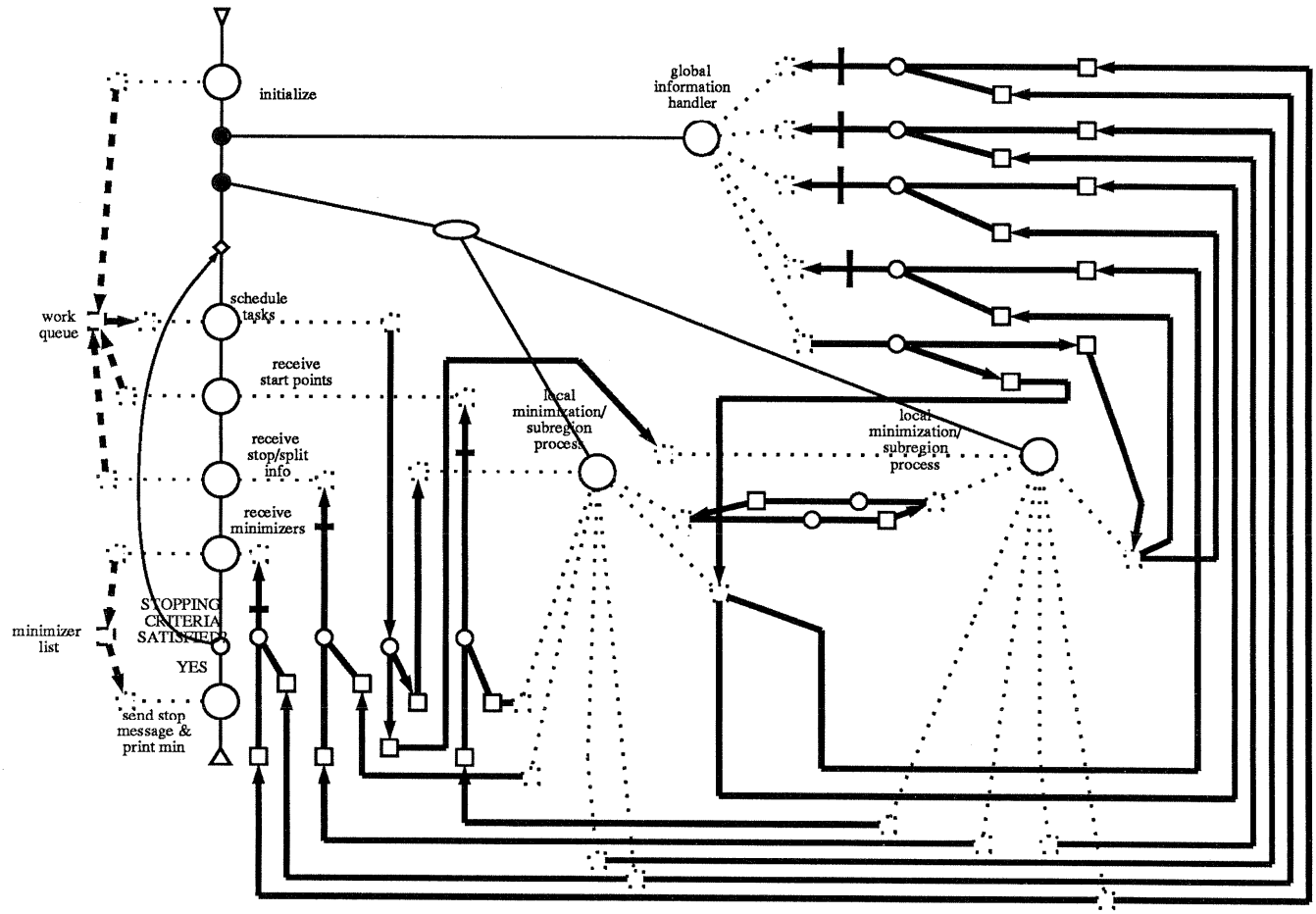

Figure 14: Level 1 DCPG for the homogeneous implementation

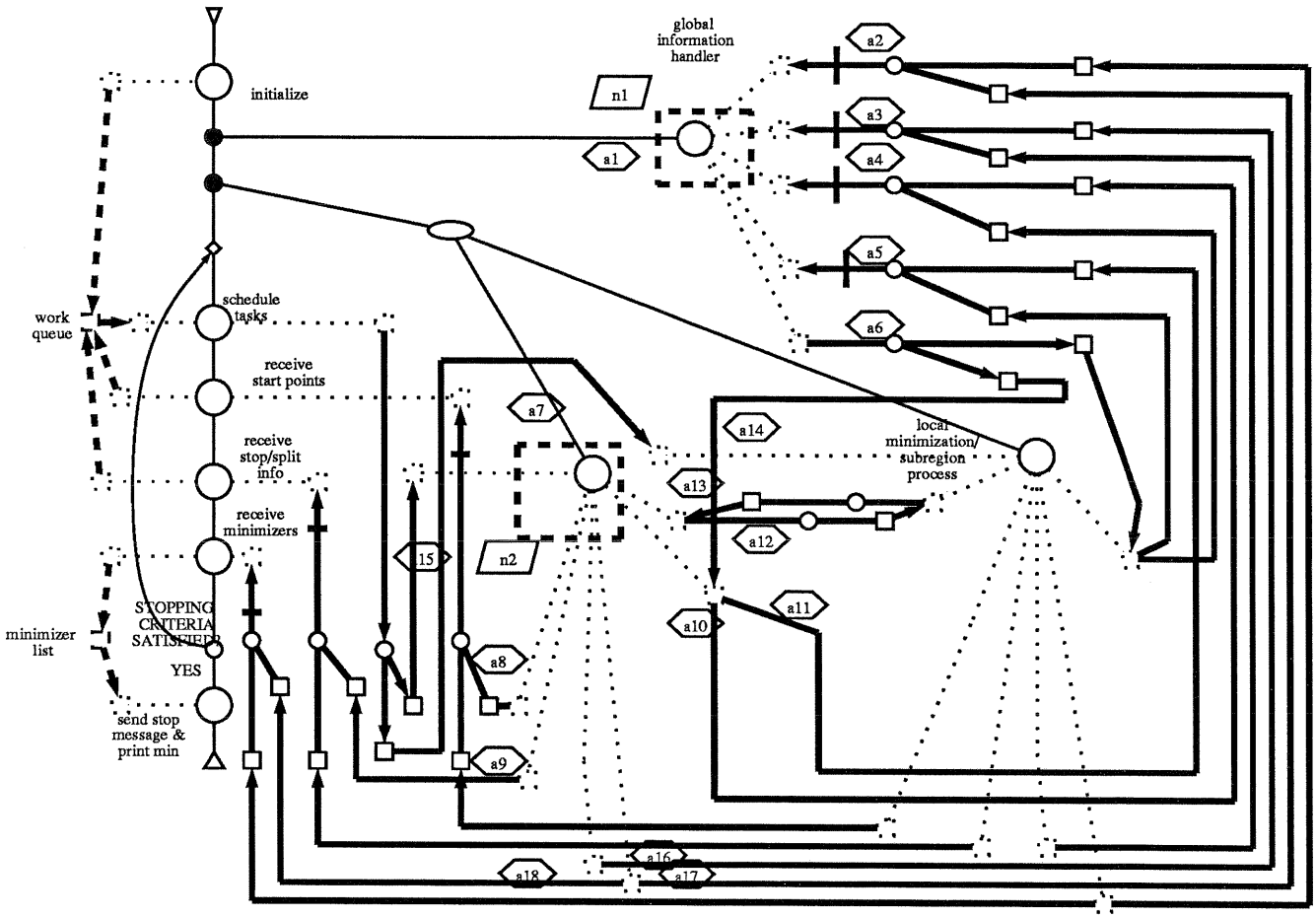

Figure 15: mapping between level 1 and level 2 DCPGs for the homogeneous implementation

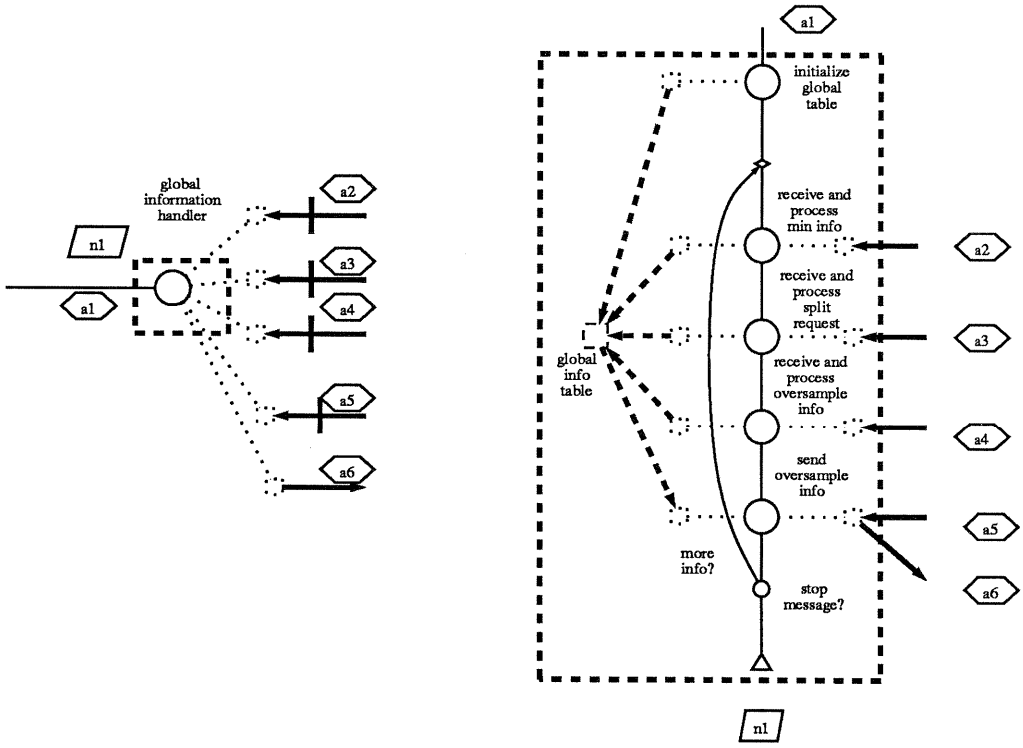

Figure 16: Refinement of node 1 for the homogeneous implementation

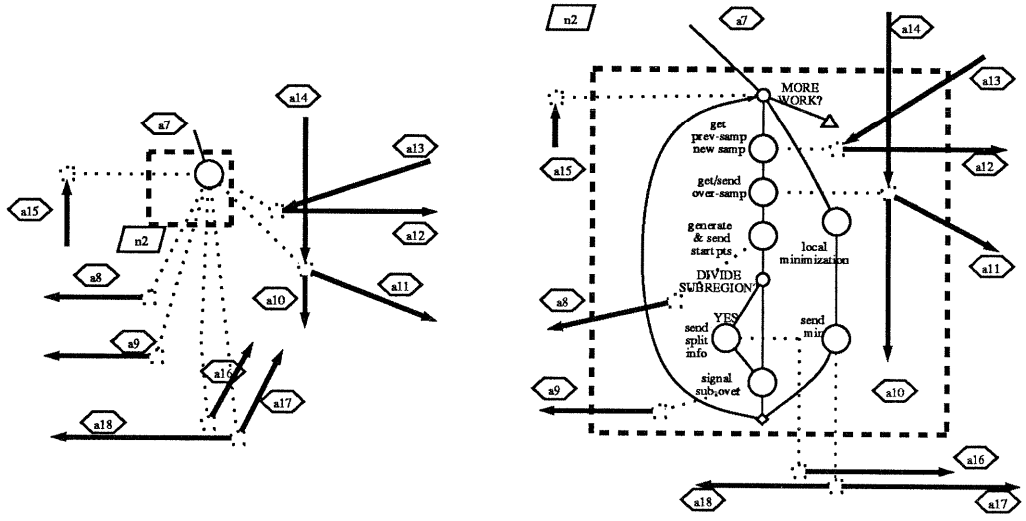

Figure 17: Refinement of node 2 for the homogeneous implementation

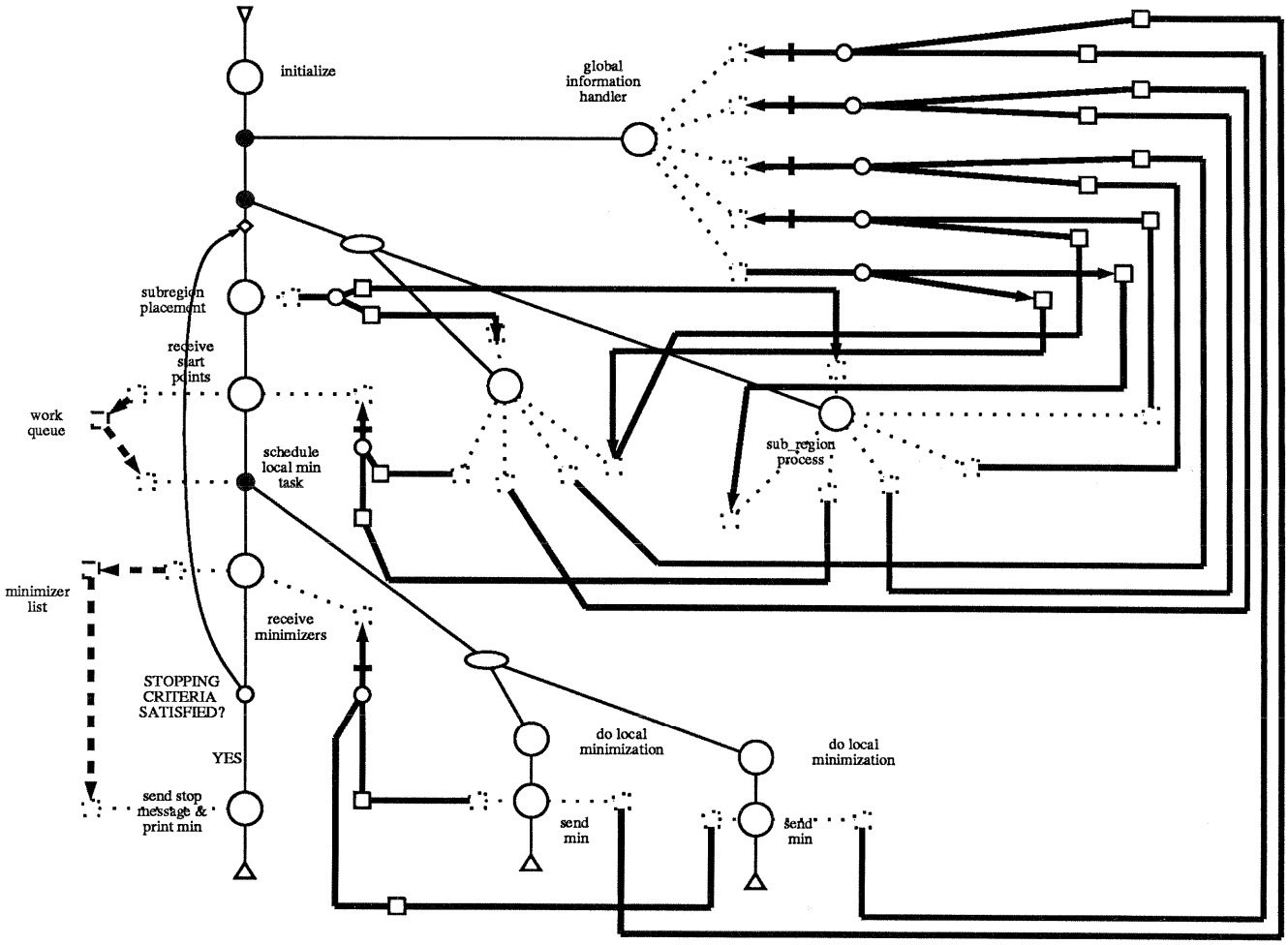

Figure 18: Level 1 DCPG for the heterogeneous implementation

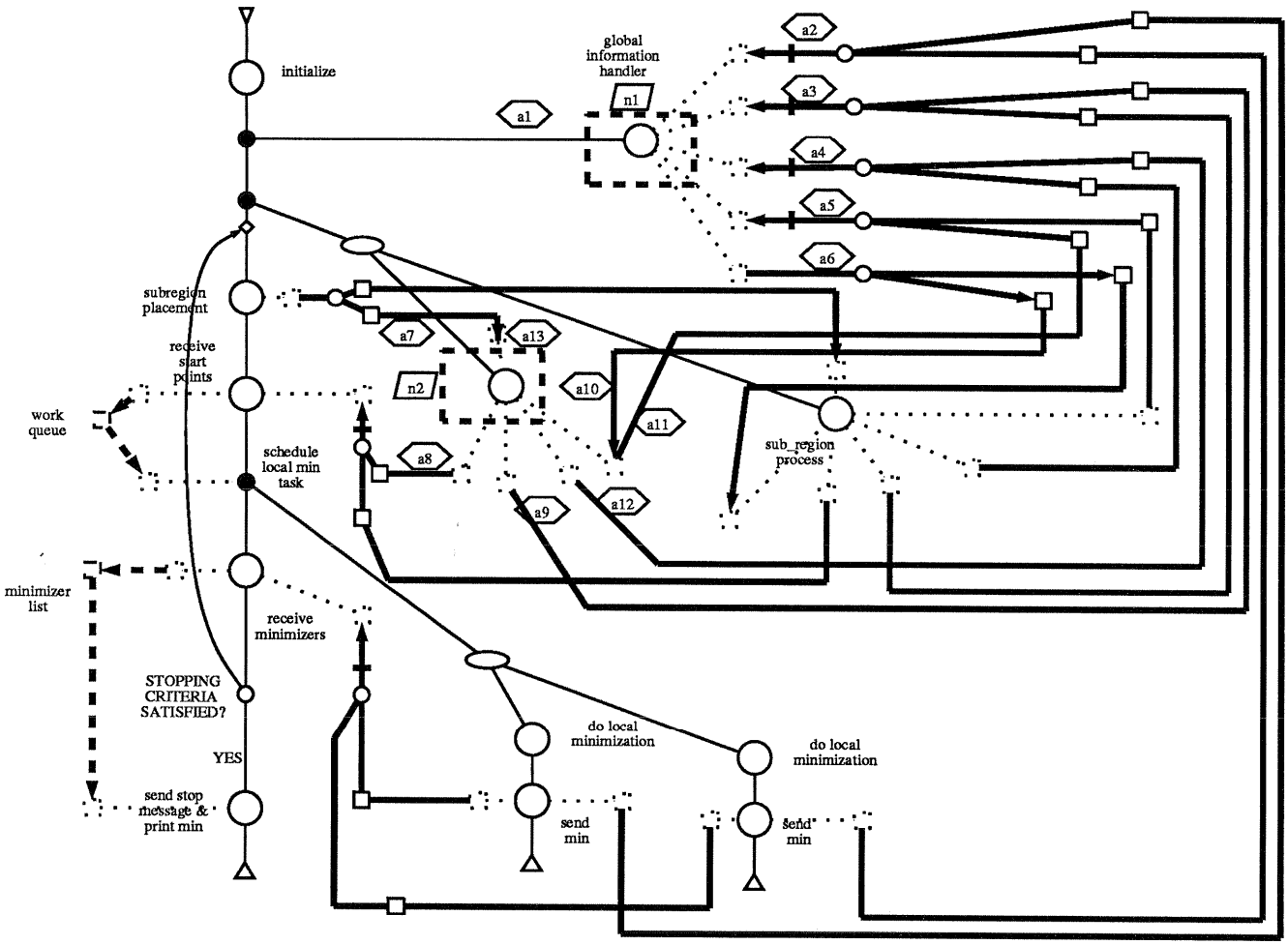

Figure 19: mapping between level 1 and level 2 DCPGs for the heterogeneous implementation

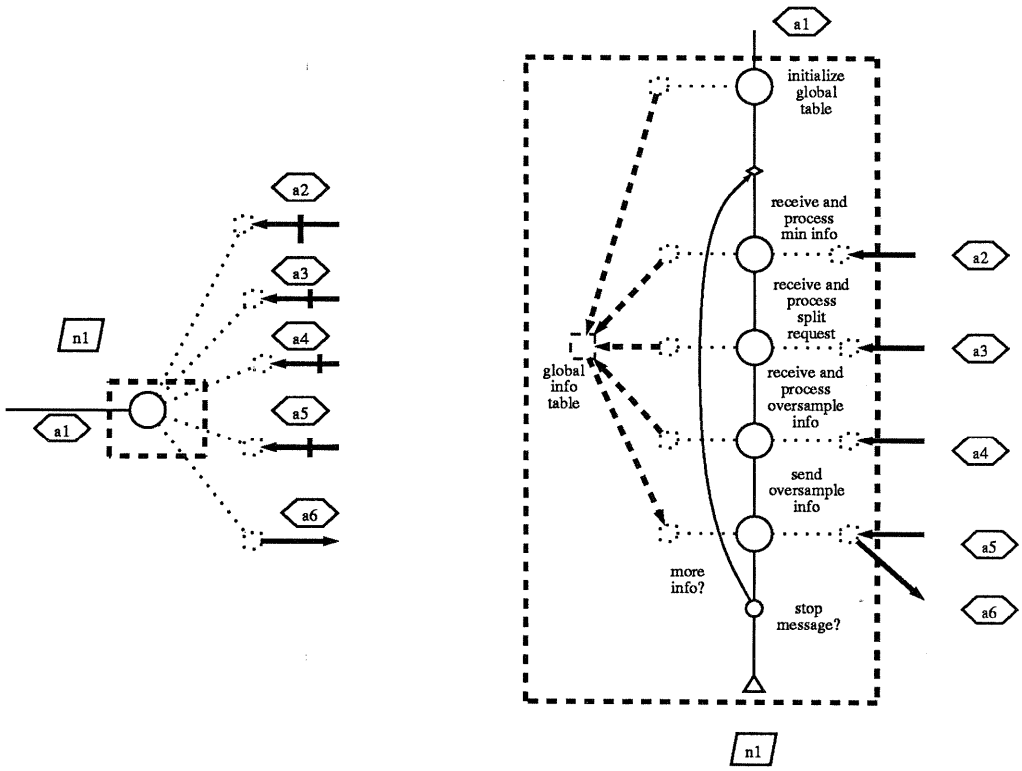

Figure 20: Refinement of node 1 for the heterogeneous implementation

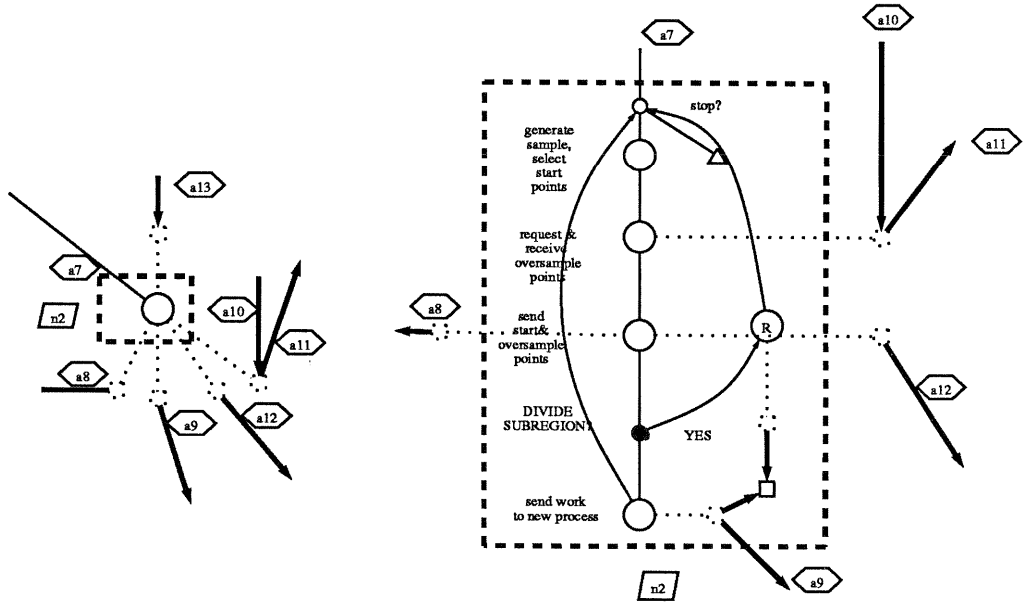

Figure 21: Refinement of node 2 for the heterogeneous implementation