Collected Papers
on
VISA and ParaDiGM

Isabelle M. Demeure†
Gary J. Nutt‡

CU-CS-488-90 August, 1990

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

---

† This work was done while this author was at the University of Colorado, supported by NSF Cooperative Agreement DCR-8420944. Her current address is Département Réseaux, Ecole Nationale Supérieure des Télécommunications, Piece C234, 46 rue Barrault, 75634 Paris Cedex 13, FRANCE. Her electronic mail address is demeure@enst.enst.fr.

‡This author was supported by NSF Grant CCR-8802283. His electronic mail address is nutt@boulder.colorado.edu.

ABSTRACT

Collected Papers on VISA and ParaDiGM

Several papers and technical reports have been written describing the Parallel, Distributed computation Graph Model (ParaDiGM) and the VISual Assistant (VISA). Demeure's dissertation is the final word on most of the ideas in ParaDiGM and VISA. This collection includes three of the most significant papers about the work:

(1) I. M. Demeure and G. J. Nutt, "Prototyping and Simulating Parallel, Distributed Computations with VISA," June, 1990, submitted for publication.

(2) I. M. Demeure and G. J. Nutt "The VISA Distributed Computation Modeling System", June, 1990, submitted for publication.

(3) I. M. Demeure, S. L. Smith, and G. J. Nutt, "Modeling Parallel, Distributed Computations using ParaDiGM -- A Case Study: The Adaptive Global Optimization Algorithm", appears in *SIAM Conference on Parallel Processing*, Chicago, IL, December 11-13, 1989.

The first paper provides some background on ParaDiGM, then illustrates how VISA can be used to support the ideas. The second paper focuses on the design of the VISA modeling system. The third paper, coauthored with Sharon L. Smith, illustrates the use of ParaDiGM to model an adaptive global optimization algorithm.

Prototyping and Simulating Parallel, Distributed Computations with VISA

Isabelle M. Demeure †
Maître de Conférences
Département Réseaux
Ecole Nationale Supérieure des Télécommunications de Paris.

Gary J. Nutt ‡
Professor
Computer Science Department
University of Colorado, Boulder.

---

† This author was supported by NSF Cooperative Agreement DCR-8420944.

‡ This author was supported by NSF Grant CCR-8802283.

Address any correspondence to:
Professor Gary J. Nutt
Department of Computer Science
University of Colorado
Campus Box 430 - Boulder, CO 80309.
(303) 492-7581
nutt@boulder.colorado.edu.

Abstract

In this paper, we describe the VISual Assistant (VISA), a software tool for designing, prototyping, and simulating parallel, distributed computations. In particular, VISA is meant to assist a designer in the choice of partitioning and communication strategies for computations, based on performance goals. VISA uses the Parallel Distributed computation Graph Model (ParaDiGM) as the basis of its operation, including its graphical interface. VISA supports the editing, animation, and simulation of ParaDiGM graphs. The animation is used to provide qualitative feedback about any particular design, while simulation provides the corresponding quantitative results of an experiment.

We introduce the ParaDiGM constructs and describe the functionality of VISA. We illustrate its utility by providing simulations of two implementations of relaxation algorithms for solving linear systems.

1 Introduction - Motivation

In recent years, decreasing hardware costs have stimulated large investments in networks of computers. Designing efficient software for these environments is a challenging task. One of the challenges lies in the *partitioning* activity which involves dividing a problem into units (e.g, *processes*) that can be distributed over various processors in a distributed hardware environment. A second challenge lies in the nature of the communication and synchronization among the processes, i.e., the *communication* strategy. The choice of the partitioning strategy is closely related to the choice of the communication strategy, and vice versa; to design an efficient distributed computation one must choose a suitable combination of both strategies.

This paper describes a software tool called the *VISual Assistant* (VISA), to simulate and prototype distributed computations. We illustrate how VISA can be used to experiment with alternative partitioning and communication strategies for the same problem, in order to design efficient distributed computations.

VISA supports computational models expressed in a formal modeling language called the *Parallel, Distributed computation Graph Model* (ParaDiGM). The language employs complementary submodels that provide both macro and micro views of a computation. The macro view concentrates on interrelationships among processes, while the micro view describes the details of each process's activity.

VISA and ParaDiGM are designed to address large-grain parallel computations for distributed memory MIMD architectures. In this context, a *parallel, distributed computation* is a collection of units (e.g., processes) to be distributed over the various processors of a distributed memory architecture, running simultaneously, and exchanging information and synchronizing via *message passing*. We assume that the supporting system for such computations includes features to spawn new processes, as well as message passing primitives [15].

Performance of parallel computations is often measured by the ratio of the time to execute "the best" sequential version of the computation to the time to execute the same parallelized computation. For systems with N individual processors this *speedup* has an upper bound of N. This is, however, an optimistic upper bound. Amdahl's law stipulates that, if a parallel computation is made of N units that run in parallel for $(1-f)$ fraction of the time, and one unit that is active for the remaining fraction f of the time, then the speedup that can be attained is $1/[f+(1-f)/N]$

(e.g., with $N = 1,000$ and $f = 0.01$, the speedup is 91) [1]. Even small amounts of serial operation (e.g., that caused by synchronization), can dramatically affect the speedup. Therefore, partitioning and communication strategies for parallel, distributed computations must attempt to minimize the fraction of serial operation to maximize performance.

Factors involved in choosing partitioning and communication strategies include the ratio of computation to synchronization (its *granularity*), the nature of the communication and synchronization mechanism, and the identification of specific function to place in a process, (e.g., partitioning by data or functional criteria). The partitioning and communication strategy also has direct consequences on how a computation will perform if the various processors of the distributed hardware environment are unevenly loaded, or operate at different speeds.

While some aspects of the partitioning problem can be handled algorithmically for certain classes of algorithms (e.g., see [9]), there currently are no general solutions to this problem. Instead, it appears that the most promising direction for advancing the technology is to provide tools to support the software engineer as he designs the computation.

Since we describe a model and a system, we will use a simple example to explain both; ParaDiGM is introduced with the example in Section 2. In Section 3 we describe how VISA provides support for ParaDiGM, and how it is constructed. Section 4 addresses VISA usage in the context of the example. Related work is described in Section 5 and our conclusions are presented in the final section.

2 Modeling Relaxation Algorithms with ParaDiGM

2.1 Relaxation Computations

Relaxation is an iterative technique for 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 X, based on previous estimates of each $X[i]$. The computation is initialized by establishing initial values for each $X[j]$, then by computing a new estimate for $X[i]$ based on the previous estimates of $X[j]$, where $j \neq i$.

The algorithm has an obvious parallel, distributed implementation. By partitioning the computation into n processes, each process can independently compute some $X[i]$, provided that it has

a copy of row i of A, $B[i]$, and estimated values of $X[j]$.

Successive over relaxation (SOR) and *chaotic relaxation* are two well-known variants of the general relaxation technique. They differ in the details of synchronization among the processes that compute the $X[i]$ values. In the remainder of this Section, we will use ParaDiGM to describe those approaches, (preceded by a brief overview of ParaDiGM).

2.2 ParaDiGM Overview

The ParaDiGM macro model – the *Process Architecture Model* (PAM) – identifies processes (circles), interprocess communication mechanisms (boxes), and illustrates their interconnection (with edges); Figure 1a is an illustration of a PAM for the SOR implementation. The model is used to describe the processes that make up the computation, and their interprocesss communication relationships. The usage of PAM is illustrated in the discussion of the relaxation algorithms.

The micro model – the *Distributed Computation Precedence Graph* (DCPG) – uses many different atoms to represent tasks, control flow, and data flow as local references and interprocess communication; Figure 1b is an example of a DCPG. The control flow is a subgraph with various node types being interconnected with thin control flow arcs, with an inverted delta representing a start point and a normal delta representing the termination of control flow. Large circles represent processing activity, called a *task*; alternative control flow paths are represented by small, open circles; small, filled circles represent the initiation of concurrent control flow through a *spawn* operation; and elliptical nodes represent replication of atoms or composites without explicitly specifying them.

Data flow is represented in a DCPG by bold lines (dashed and solid) interconnecting tasks and *local data repositories* (dashed boxes) and *message repositories* (solid boxes).

Each DCPG node represents a basic block of computation, including all forms of computation one would employ in a conventional procedural programming language. While the DCPG represents the relationship among the execution of these individual blocks (through precedence and control flow represented by tokens), each node's computational detail can be described through the use of a function expressed in a procedural language (the precise nature of the language is relatively unconstrained in the ParaDiGM definition, but is specific in the VISA system, see Section 3 below). We include an example interpretation in the chaotic relaxation model.

We elaborate on the ParaDiGM model explanation by describing the SOR and chaotic relaxation algorithms.

2.3 The SOR Algorithm

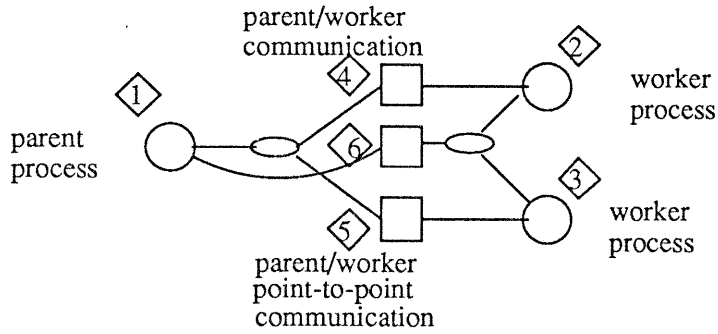
The SOR approach is initiated as a single *parent* process which determines the dimension of the system of equations, n; it reads the values for the matrix A and the vector B into its memory. The parent then spawns n identical *worker* processes and initializes each with an appropriate row of A (row i for the process computing $X[i]$), $B[i]$, and the initial value of X. Each of the workers is allowed to proceed with the computation of an individual $X[i]$; when it has completed this computation, it sends the value to the parent, then waits for a new set of X values to begin the next iteration.

Whenever the parent has received all $X[i]$ from the workers, it decides whether a solution has been reached or not, i.e., when the error in the solution is judged to be sufficiently small, the parent process tells the n worker processes to halt. Otherwise, the parent redistributes the latest estimate for X to each worker and waits for the n individual results to be recomputed. When all processes have halted, the parent process saves the result and terminates.

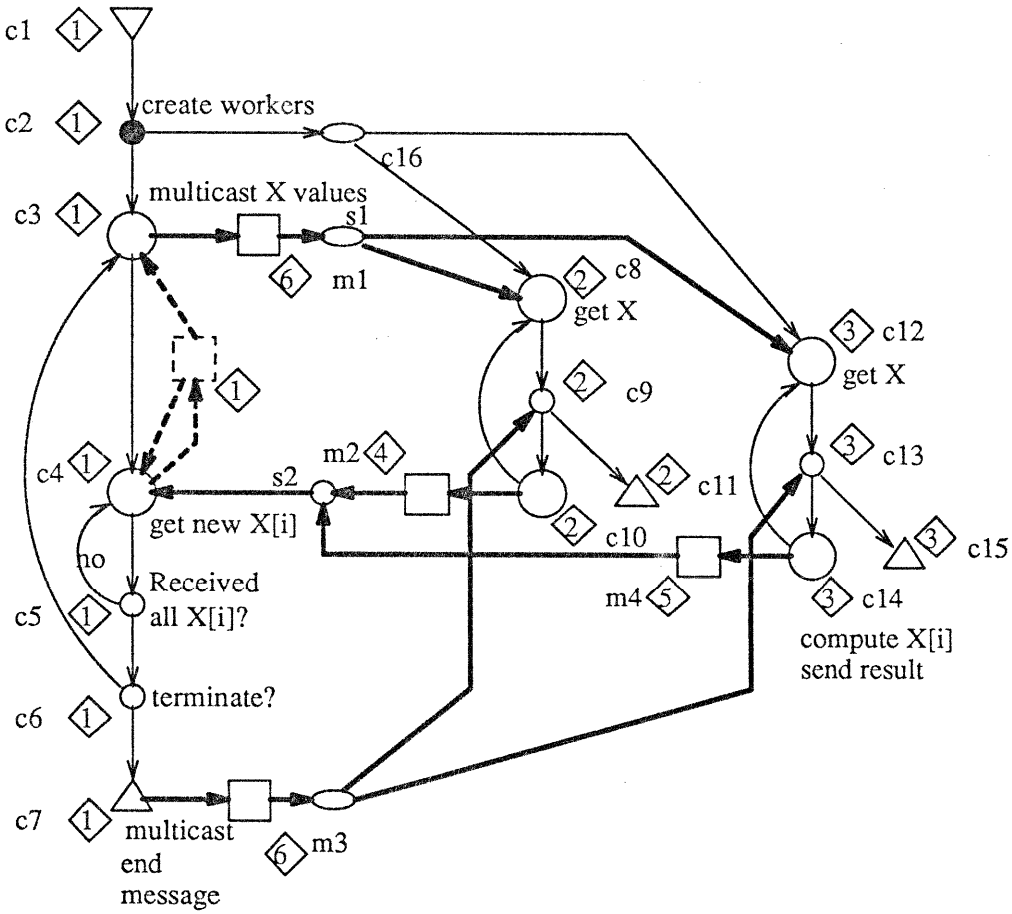
Figure 1b is a DCPG representing the parent as a set of tasks initiated by the inverted delta symbol. The DCPG spawn primitive leads to an ellipsis, which describes two of the resulting n workers. Thus the DCPG describes the micro level processing performed by each of the $n + 1$ processes.

The PAM graph is shown in Figure 1a. The node labeled "1" represents the parent process, and the nodes labeled "2" and "3" represent two of the worker processes. The square nodes labeled "4"and "5" illustrate that each worker process (2 and 3) communicates on a point-to-point basis with the parent process (1) to return a newly computed value for $X[i]$. The ellipse node indicates that while the model represents only two of the worker processes, there may be more.

The diamond-boxed labels describe a mapping between the nodes in the PAM and DCPG graphs. The unboxed labels are identifiers for DCPG nodes where a letter indicates the type of the node (c for control node, m for message node, s for selective node and e for extension node). Many of the nodes are also labeled with abbreviated text indicating the nature of the task or data structure they model.



(a) The PAM



(b) The DCPG

Figure 1: Successive Over Relaxation

In this graph, three threads of control are explicitly represented: All the nodes in the first thread (c1, c2, c3, c4, c5, c6, c7) are labeled with a boxed 1 because they correspond to the parent process also labeled 1. The nodes in the second thread are c8, c9, c10 and c11 corresponding to the worker process labeled 2. Process 3 is identified similarly. The ellipse node labeled c16 indicates that while the model represents only two workers, there may be more.

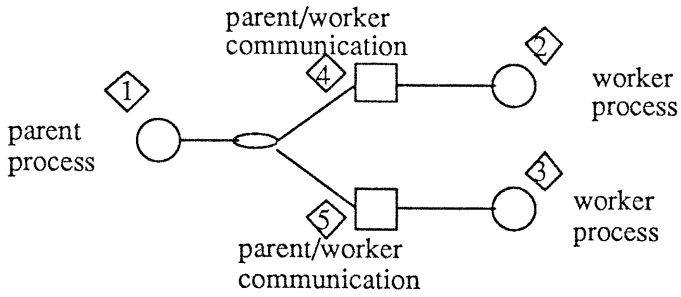
Task c3 sends messages to the worker processes, indicated by the construct made of nodes s1 (*multicast node*) and message node m1, i.e., c3 sends a message to all of the worker processes. DCPG nodes s2, m2, and m4, indicate that the parent receives a message from one of the worker processes at task c4. Notice that node m2 models messages between the parent process and the worker process labeled 2, which correspond to the communication labeled with a boxed 4. Similarly, node m5 maps to the communication node labeled with a boxed 5. Nodes m1 and m3 represent the multicast messages from the parent to the workers, and map into the node labeled with a boxed 6.

2.4 The Chaotic Relaxation Technique

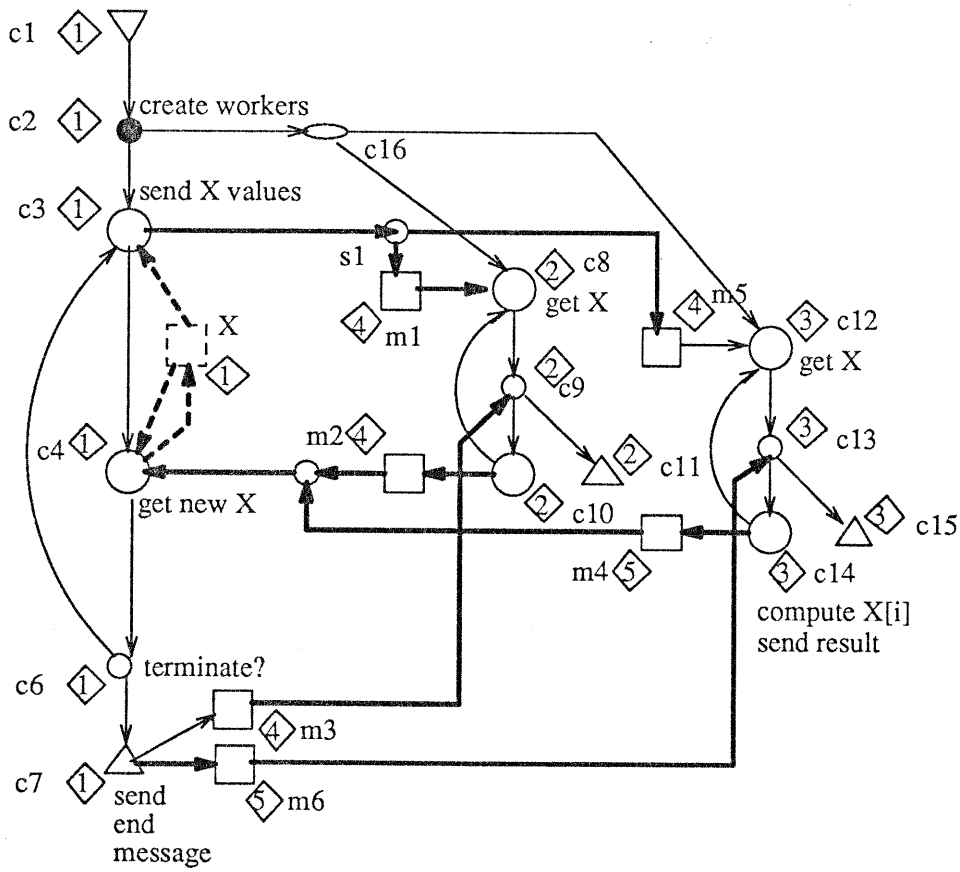
Chaotic relaxation is very similar to SOR, with the exception that the worker processes operate asynchronously. That is, each worker independently receives the $X[j]$ values from the parent, and after reporting the newly-computed $X[i]$, obtains new copies of the current estimates of $X[j]$ (independent of the iteration for each of those computations), and begins computing a new estimate for $X[i]$.

The ParaDiGM model for chaotic relaxation bears a remarkable resemblance to that for SOR, with differences being in the areas of dissemination of X, and in the iteration synchronization mechanism. Figure 2 illustrates these two distinctions: Chaotic relaxation eliminates the synchronization represented by node c5 and the corresponding loop from c5 to c4 (used to collect all results from the workers). Also, X values are sent to the individual worker that completed, rather than using the multicast mechanism; this is represented by the selective send node (s1). The chaotic relaxation PAM is less complex than the SOR PAM, since it makes more use of point-to-point communication instead of multicasting.

Both the PAM and the DCPG specifically identify similarities and differences between SOR and chaotic relaxation. Since there are few functional differences between the two approaches, there are few differences between DCPGs. However, the PAM highlights the differences in IPC mechanisms.



(a) The PAM



(b) The DCPG

Figure 2: Chaotic Relaxation

Previously, we have mentioned that nodes in a DCPG model may incorporate additional information, (e.g., to specify the semantics of an *or* node, or the number of processes to be created in an ellipsis construct). We elaborate on the chaotic relaxation DCPG to illustrate one way that those interpretations can be used.

Figure 3 provides an example of an interpretation procedure for node c4 of the parent process; this particular implementation is expressed in the C programming language. Node c4 represents the part of the parent process that receives X values from a worker process, adds it to the currently-known vector, then tests X to see if the solution has converged. The C code illustrates these steps, including a call to a VISA procedure (*sys_receive_block*) to represent a blocking receive of a message. Notice the simplicity of the code in the context of the DCPG control flow.

ParaDiGM, itself, does not require that node interpretations be expressed using the C language (although the VISA implementation currently only supports C). Further, as discussed below, the VISA design is not fundamentally dependent upon C, allowing one to provide relatively simple extensions to support other languages.

3 VISA

ParaDiGM allows designers to construct relatively complex models, illustrating the process architecture, the control flow within processes, and procedural models of the behavior of each task within a process. While ParaDiGM graphs are a useful form for representing parallel, distributed computations (e.g., with pseudo code interpretations for each node), they are considerably more useful if the models are executable.

Further, our experience with graphical models is that their utility can be limited by the ease with which the designer can construct and study the models. For example, if the graphical model is constructed using mechanical media, it is time-consuming to change the model so that it really reflects the designers notions. Simple graphical editors for managing graph models considerably enhance their utility as working models.

VISA supports the creation and execution of ParaDiGM models. It can be used during the design process to describe parts or all of the application under study and to compare possible alternative approaches. It is an interactive, graphic computer environment that allows the designer to create and edit ParaDiGM models, and to execute the models with an animated display of the

```
#define PRECISION     0.01
#define NUM_WORKERS   3
#define MSG_SIZE      32

/* globals */
float   X[NUM_WORKERS];
short done[NUM_WORKERS];

/* some code deleted */

proc4()
/*
 * Parent Process: task node
 * wait for a new X[k] value from one of the workers, get it, and
 * set up flag for stopping condition if relevant
 */
{
        char msg[MSG_SIZE];
        short msg_length;
        float result, diff, fabs();

        /* block waiting for a message from one of the workers */
        sys_receive_block(msg, &msg_length, &num, 0);
        sscanf(msg, "%f", &result);


        /* test for stopping condition */
        diff = fabs(result - X[num]);
        if (diff <= PRECISION)
           done[num] = TRUE;
        X[num] = result;

        task_return(0, 0);
}
```

Figure 3: Example Interpretation Procedure

execution as well as extensive data collection describing the effect of the execution, see Figure 4. The graphs are supplemented with various attributes, simulation parameters (e.g., distribution functions to be used when execution time values are needed, load factors for representing the effective speed of a processor), and interpretations (user-defined procedures that can be executed by the system).

While all ParaDiGM graph editing operations (including graph annotations) are supported directly in VISA, node interpretations can be prepared using a generic text editor. The resulting files are then compiled and made ready for subsequent usage at model interpretation time. The text editor and compiler are parts of the underlying environment to VISA, but can be invoked from within a VISA session.

VISA differs from many other machine-supported design environments in its specific focus on the design of parallel, distributed computations for distributed memory MIMD target configurations. The ParaDiGM constructs allow the designer to concentrate on the macro process architecture or on the micro details, depending upon the designers intent. VISA is not designed to favor a topdown nor bottom-up strategy (corresponding to PAM-before-DCPG and DCPG-before-PAM, respectively), but to provide a flexible testbed. The testbed encourages the modeler to distinguish among partitioning strategies (using PAMs), computational control and data flow (using the DCPG graph), and details of the operation (through DCPG node interpretations). VISA supports the designer when he focuses on any of these aspects, even if the other aspects are not currently defined.

While VISA does not contain an algorithmic mapping algorithm to correlate the DCPG and the PAM, it provides editing support so that both models are managed by a single entity (the editor), and the editor provides manual tools to establish the mapping. This ability to handle the macro and micro views of the model simultaneously is not feasible without machine assistance such as provided by VISA; and it add considerable flexibility to the computation design tools.

Software design, in general, can benefit from *qualitative* feedback from the modeling system in addition to the traditional *quantitative* feedback of a simulation or analysis tool. VISA provides qualitative feedback through a scaled realtime animation facility, and quantitative feedback through traditional simulation tools and reports. The qualitative feedback – animation – depends upon the visual aspects of ParaDiGM rather than the formal aspects. For example, qualitative feedback

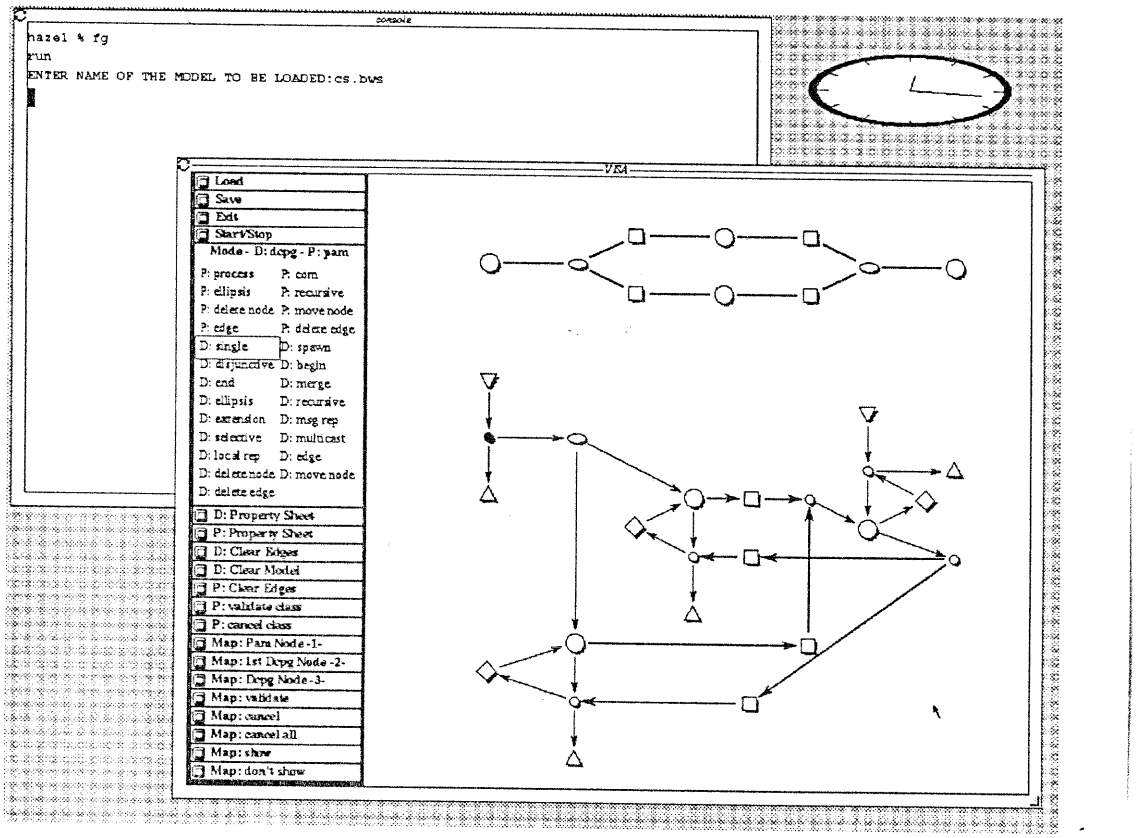


Figure 4: Example of an Editing Session

allows the designer to quickly perceive bottlenecks in the system design in the complete absence of any quantitative information. The visualization of the computation's control flow illustrates relative synchronization and allows the designer to quickly identify parts of the design that are likely to cause the computation to perform poorly because of the Amdahl effect.

ParaDiGM models employ tokens to represent the execution state of a model, thus VISA provides visual feedback in the form of tokens flowing through the DCPG graph, in the style of tokens used in Petri Nets [14].

3.1 System Overview

VISA is an instance of the Olympus architecture [11], with considerable extension for supporting the PAM portion of ParaDiGM and node interpretation in DCPGs. The Olympus architecture is, itself, a distributed system composed of a *frontend* and a *backend*. The frontend implements the user interface, including the specific presentation model and human factors aspects of the interface, e.g., windows and point-and-select interaction. The entire editor (except for the storage) is ordinarily implemented in the frontend. The frontend interacts with the backend using a general network IPC protocol, supporting a specific Olympus protocol.

The backend is a graph storage and interpretation engine. The frontend-backend interface allows a frontend to define or retrieve a model (ordinarily on a component-by-component basis). The backend will also interpret a specified model under the direction of a frontend. Model interpretations are separated into graph interpretation (representing the control flow through a DCPG) and node interpretation (corresponding to the execution of procedures associated with a node). Olympus employs another network-level interface between the graph interpreter and the node interpreter.

The node interpretation language is nearly independent of the backend, since the interface uses a remote procedure call (RPC) mechanism for invoking individual interpretations. Thus, task interpretation servers are started, accepting RPC calls from the backend as dictated by the graph interpretation. The interface is general, in the sense that the backend only expects that the node interpreter can accept an RPC call with parameters, then return other parameters when the interpretation is complete. Thus, node interpretations can be written in any procedural language that has an RPC interface.

There are several instances of the Olympus architecture, although most of these (including

VISA) originate from the BPG-Olympus system [13].

ParaDiGM makes different requirements on the support system than do other models supported by Olympus (these languages are similar to DCPGs; Olympus does not support a language like PAM). Therefore, while Olympus was a reasonable starting point for VISA, it was necessary to do substantial enhancement to achieve full support of PAMs and DCPGs.

VISA extends the Olympus architecture by creating an extensive distributed runtime environment under the control of a modified backend, and by enhancing the frontend so that it behaves as an editor for PAMs and DCPGs. VISA reuses considerable code from the BPG-Olympus frontend editor, but only about a quarter of the code from the backend, i.e. VISA not only extends the Olympus architecture, it contains considerable new implementation.

VISA DCPG node interpretations are expressed in C, supplemented by a library of functions that contains primitives to describe parallel, distributed computations (matching the ParaDiGM graphical constructs). These primitives include the *sys_spawn* primitive to create a single process, the *sys_spawn_ellipsis* to create a collection of processes (to match the DCPG ellipsis construct), the *sys_send* primitive to send a message, and the *sys_receive_block* and *sys_receive_noblock* primitives to receive messages.

The Olympus node interpreters described above are replaced by a client-server network of node interpretation engines. An *interpretation listener* is started on each machine that will support node interpretation. The graph interpreter executes the control flow for a DCPG, including the management of simulated time. When a node interpretation is to be executed, the VISA backend causes an interpretation listener to create an *interpretation server.* A particular interpretation server will then be used to interpret specific procedural declarations.

Each interpretation may include VISA *sys_*calls, related to graph dynamics. Thus, these calls are implemented in the backend, requiring the interpretation server to execute a call to the backend.

3.2 Supporting Simulation

Serial software execution times can be used to specify process execution times, as part of the annotation on a particular node (this information is used to derive performance predictions). However, it is often difficult to measure the execution time of a block of software to obtain the desired time values. For this reason, the simulator uses timing information provided by the user in the form of

distribution functions and time values in tables.

There can be several instances of the "same process" (e.g. several worker processes); they are instances of the same *process class*. Similarly, identical communication relations (involving processes of the same class, following the same pattern of communication) are members of the same *communication class*. The simulation information for the PAM graph, is specified for each class. For each process class, the parameters are a table of load factors, a table of time values, and a distribution function. For each communication class, the parameter is a table of delays. A distribution function may also be specified for each DCPG control node.

The simulator infers execution times from relevant DCPG node distribution functions if they are specified; otherwise, it attempts to obtain a value from the corresponding PAM class. If no distribution function was specified, a value from the "time value" table is used (the table is used circularly: The index of the current element in the table is given by the number of events for which the table has been used, modulo the number of elements in the table). Notice that while different samplings of a distribution function yield different values, a simulation done with time values only is deterministic, and therefore yields the exact same results when reproduced.

Computing the Simulation Times. Since VISA combines the execution of interpretations with the evaluation of simulation times, several problems can occur: (1) The estimated simulation time can be smaller than the execution time; (2) the events can get out of order because of the relative loads of the processors on which the various processes of VISA are executing; and (3) the simulator introduces some overhead and might therefore cause some delay in the simulation. In order to maintain relative ordering of the events, we introduce a scaling factor by which all the times are multiplied (which is equivalent to slowing down the simulation by that factor).

Let us introduce some notation:

- $T0$: Start time for simulation;
- $EI(e)$: Interval of time it takes for the simulator to execute the operations associated with event e;
- $SI(e)$: Estimated time associated with the event e to simulate the execution of the event e;
- $ASI(e)$: Adjusted simulation time for the execution of the event e (time the simulator will actually take to execute the event);

- SF: Scaling factor.

By definition:

$$ASI(e) = SI(e) * SF; \tag{1}$$

The aim is to choose SF such that for all events e: (1) $ASI(e) \geq EI(e)$, (2) SF is sufficiently large that the overhead due to the simulator is negligible compared to the adjusted simulation times, and that the actual relative loads of the processors can be ignored. Notice that the choice of SF depends on the typical real execution times and simulated execution times. (VISA allows the user to pick a value for SF.) The simulator detects cases in which the scaling factor is too small (when it gets events after their adjusted simulation time is passed), and provides feedback to the user.

Simulating Processor Load. The same idea of time adjustment can be used to simulate the relative loads or speeds of processors. This is done by assigning a load factor to each PAM component. The larger the load factor, the slower the processor running the PAM component (either because of a heavy load or because the processor is slower). So if PLF designates the Processor Load Factor, we have:

$$ASI(e) = SI(e) * SF * PLF; \tag{2}$$

A similar idea is used to simulate transmission times.

3.3 Collecting Results

The simulator collects information about a simulation as it runs and generates a report when the simulation terminates. This information is available to the user at the end of the simulation. The information collected is of three types: Global information, per-process information, and per-DCPG-node information. The global information includes:

- The number of processes created;
- The number of idle/active/terminated processes at the time the simulation was ended;
- The number of messages sent/the number of receives performed;
- The number of spawns performed;

- The number of times events were executed out of order (greater than zero if the scaling factor was too small).

The per-process information is collected for each of the spawned processes (whether represented on the ParaDiGM graphs or not). It includes:

- The process start time/end time/duration;
- Percentage of idle time (waiting for messages/for spawns);
- The number of messages sent/number of received performed;
- The number of spawns performed.

The per-DCPG-node information is collected for each DCPG control node (whether part of a process represented on the ParaDiGM graphs or not). It includes:

- The number of times executed (a node is executed several times if it is part of a loop);
- Average execution time/average idle time.

4 Using VISA

4.1 Experiments with the Example Computations

We describe two tests each with the SOR and chaotic relaxation computations. In all four tests, the time values were obtained from "time value" tables. The interpretation code used in these tests is a prototype of the computations. Although the interpretation yields the solution to the system of linear equations, due to the support provided by VISA, it requires significantly less effort than implementing the full computation, scheduling it on processors, and running it under various conditions. In other tests (see [4]), the interpretation is more of a simulation of the computation than an actual implementation.

Each of the four tests was run on a 3 by 3 matrix, with an initial condition such that the algorithm would converge. In these tests, we were interested in predicting the performance of the two computations under various load conditions. In the first test, the three worker processes were

simulated as running on equivalently loaded processors. In the second test, one of the worker processes was simulated running on a processor that is one third as fast.

The results of the four tests are summarized in Table I. For each of the tests, and for each process, we recorded the number of iterations needed for the algorithm to converge, the duration of the process, and the percentage of time the process spent waiting for messages to arrive.

Results with the Relaxation Computation. The first and third columns in Table I correspond to the tests for the chaotic relaxation computation. When one of the workers is one third as fast, the computation lasts 50 percent longer (as indicated by the duration of the parent, and the workers) than when they all run at the same speed; the two faster workers also need to do about two times as many iterations since they do not get an updated value of X[3] from the slower worker as often as they would if all workers ran at the same speed. The percentage of time spent idle is divided by two for the parent and the two faster workers, since they do about twice as many iterations in a time that is only fifty percent longer.

Results with the SOR Computation. The second and fourth columns correspond to the tests for the SOR computation. In this case, the computation lasts 100 percent longer when one of the workers is three times slower, than when they all run at the same speed. The number of iterations is the same, since all the processes synchronize after each iteration. However, the percentage of time spent waiting for messages for the parent and the two faster workers is three times larger than when the three workers are run at the same speed.

Comparison of the results for the two computations. The first two columns of this table indicate that the SOR computation performs slightly better than the chaotic one if all the processes run at the same speed. In the SOR version, the parent does not have as much computing to do as in the chaotic version: the parent waits for all three workers to complete an iteration before it itself iterates. In the chaotic version, the parent iterates each time any worker completes an iteration. The last two columns indicate that the chaotic relaxation computation performs better than the SOR computation under uneven load conditions. It does, however, use the CPU more intensively than the SOR version (since it iterates more in a shorter period of time).

From these experiments, one can conclude that if the load of the processors is equivalent (e.g. there are no other users on the system and all processors operate at the same speed), one should prefer the SOR computation over the chaotic one. Otherwise, one should prefer the chaotic com-

Table I: Comparison of Results for Four Tests

Process	data	bal chaotic	bal SOR	unbal chaotic num4 loaded	unbal SOR num4 loaded
parent num 1	num iterations	18	6	29	6
	duration (ms)	56	42	84	96
	% time idle (msg)	2	14	1	62
worker num 2	num iterations	7	6	13	6
	duration (ms)	55	41	83	95
	% time idle (msg)	29	17	16	64
worker num 3	num iterations	7	6	13	6
	duration (ms)	55	40	83	94
	% time idle (msg)	30	14	17	63
worker num 4	num iterations	6	6	5	6
	duration (ms)	52	38	87	102
	% time idle (msg)	35	11	0	0

putation.

These two tests are simple, but they illustrate the nature of the results that can be produced by VISA.

5 Related Work

With the technological advances that have been made in high-resolution and bit-mapped graphics, many software tools now use a graphic interface to a modeling or prototyping environment.

For example, STATEMATE [8] is a CASE (Computer Aided Software Engineering) system that incorporates the Statechart [7] mechanism developed by Harel, targetted at reactive systems. Statecharts are an extension of state diagrams used for studying the behavior of systems. In STATEMATE, statecharts are used as the controlling mechanism for activity charts which describe the functional characteristics of a system and are complemented by module charts which allocate the functionality to system components. The charts are supported by a graphical editor, a data dictionary, and a generalized query capability.

SARA (System ARchitects Apprentice) [6], is another CASE system. In SARA, the structure of a system is represented as a hierarchy of modules connected via sockets and interconnections. The behavior of the system is modeled using GMB (Graph Model of Behavior). GMB consists of a control flow graph very similar to Petri Nets, a data flow graph, and a description of the processes associated with the nodes of the data flow graph. SARA is well suited for the design of concurrent real-time systems, but could not very naturally support the design of parallel, distributed computations, as it does not have the appropriate primitives built into the system.

Finally, PARET (Parallel Architecture Research and Evaluation Tool) [10] is described by its authors as an environment for the study of interaction of algorithms and architectures. The underlying model is a directed flow graph. Through PARET, the effect of varying physical resources on system performance and alternate mapping, scheduling, and routing strategies can be studied. Interactive simulation, run-time measures, and summary statistics are supported. PARET is mostly targeted at modeling and studying architectural aspects as opposed to the algorithmic aspects on which we are focusing.

Finally, since part of the originality of VISA lies in the model on which it is based, let us provide some background for ParaDiGM. There have been a number of different graph models of software

over the last twenty years, e.g., see [2] and more recently, [3]. The immediate predecessor of the DCPG model is the BPG (Bilogic Precedence Graph) model [12] (which, in turn, is descended from Petri nets [14]); while BPGs were designed for general purpose modeling of parallel computations (see [13]), DCPGs focus on large-grained, message-based local computations. The basic PAM graph is most similar to communication graphs such as the one used in the POKER system [16].

6 Conclusion

The performance of a parallel, distributed computation is highly dependent upon the partitioning and communication strategy employed to implement it. Such a strategy determines the functionality implemented by each participant process, and the nature and amount of communication and synchronization required among the participant processes. Therefore, it determines how a computation will react to load imbalance or differences in speed of the processors of the supporting distributed hardware environment, and thus the efficiency with which these processors will be used.

There are no known general conditions to guarantee that partitioning and communication strategy for a computation will yield high performance. VISA is a prototyping and simulation tool intended to aid the designer of parallel, distributed computations in manually partitioning the computation, with rapid feedback on the behavior of a strategy.

VISA makes it easy to experiment with different implementations under various load conditions and with various input data. In these experiments, the designer can choose to focus on specific parts of the computation and to simulate others. The interpretation code can eventually be reused for the final implementation.

We have attempted to illustrate experimentation by describing simulation results for two computations that correspond to alternative partitioning and communication strategies for the relaxation problem. Although these computations are simple, VISA still proves useful: It allows the programmer to verify his intuition, and to support it with measurements, without having to fully implement alternative versions of the computation under study, and to run those computations under various conditions. The interpretations are quick prototypes of the corresponding computations. The measurements made on the SOR and chaotic relaxation computations support the intuition that chaotic relaxation should react more gracefully to load imbalance.

In other work [4], we have used ParaDiGM to model different kinds of systems and algorithms

ranging from client/server operation, to accounting practices used in a PBX, to various versions of an adaptive global optimization algorithm. ParaDiGM is intended to represent parallel, distributed computations in terms of the functionality, process partitioning, interprocess communication, synchronization, control tasks, and message and data flow, all in a single framework. ParaDiGM representations have proven to be concise and precise. Their visual nature makes them ideal candidates for serving as a basis for a software tool using a graphical user interface.

Although using VISA may appear to generate overhead in developing parallel, distributed computations, we believe that less total project time will be incurred with its use. It has been our experience that through VISA one can quickly design and prototype an implementation of a parallel, distributed computation, achieve a qualitative understanding through animation, evaluate its performance characteristics by running simulations of it under various load conditions, and try variants of it, without the cost of actually implementing it. The programmer can focus on such things as the way an implementation will react to load imbalance, the efficiency with which the processors are used by a computation, and the cost of the communication and synchronization for a given implementation. VISA is therefore useful for predicting the performance of a parallel, distributed computation during its design phase. Finally, the graphs given as input to VISA can be used as documentation for the computation under study.

In this paper, we have given a brief introduction to some of the ParaDiGM constructs; we have described the main functions of the VISA tool; finally, we have illustrated the use of VISA, obtaining simulation results for two implementations of a simple algorithm for solving a system of linear equations. A considerably expanded discussion of these and other issues appears in [4].

Experiments with the current version of VISA have revealed several ways in which the tool could be improved. In particular, they include user interface issues, issues related to the efficiency of the simulator, as well as ideas for additional functions (e.g., a function that would check that ParaDiGM graphs are correct). One of the directions of our continuing research is to refine VISA.

Another direction of the continuing research is to do experiments with more complex algorithms (e.g. Adaptive Global Optimization Algorithm, [5]), to use VISA to guide the partitioning and communication strategy for such computations, and to predict their performance.

7 Acknowledgements

We are grateful to Adam Beguelin, Jeff McWhirter, Mike Schwartz, and Dave Wagner, whose feedback helped us in the preparation of this paper.

References

[1] G. M. Amdahl. The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *AFIPS Conference Proceedings*, 1967.

[2] J. L. Baer. A Survey of Some Theoretical Aspects of Multiprocessing. *Computing Surveys*, 5(1):31–79, March 1973.

[3] J. C. Browne. A Unified Approach to Parallel Programming. June 1987. 1987 Summer Workshop in Parallel Computation - University of Colorado, Boulder.

[4] I. M. Demeure. *A Model, ParaDiGM, and a Software Tool, VISA, for the Representation, Design and Simulation of Parallel, Distributed Computations.* PhD thesis, University of Colorado, Boulder, August 1989.

[5] I. M. Demeure, S. L. Smith, and G. J. Nutt. Modeling Parallel, Distributed Computations Using ParaDiGM - A Case Study: The Adaptive Global Optimization Algorithm. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.

[6] G. Estrin, R.S. Fenchel, R. R. Razouk, and M.K. Vernon. SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems. *IEEE Transactions on Software Engineering*, SE-12(2):293–311, Feb 1986.

[7] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming - North-Holland*, 8:231–274, 1987.

[8] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[9] C. E. Houstis, E. N. Houstis, and J. R. Rice. Partitioning pde computations: methods and performance evaluation. *Parallel Computing*, 5(1I&2):141–163, July 1987.

[10] K. M. Nichols and J. T Edmark. Modeling Multicomputer Systems with PARET. *Computer (IEEE)*, 39–48, May 1988.

[11] G. J. Nutt. A Flexible, Distributed Simulation System. In *10th International Conference on Application and Theory of Petri Nets*, pages 210–226, Bonn, West Germany, June 1989.

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

[13] G. J. Nutt, A. Beguelin, I. Demeure, S. Elliott, J. McWhirter, and B. Sanders. Olympus: An Interactive Simulation System. In *1989 Winter Simulation Conference*, pages 601–611, Washington, D.C., December 1989.

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

[15] S.M. Shatz. Communication Mechanisms for Programming Distributed Systems. *IEEE Computer*, 21–28, June 1984.

[16] L. Snyder. Parallel Programming and the Poker Programming Environment. *IEEE Computer*, 27–36, July 1984.

The VISA Distributed Computation Modeling System

Isabelle M. Demeure† and Gary J. Nutt‡

ABSTRACT

This paper describes the implementation of a modeling environment to support the representation and performance prediction of parallel, distributed computations. The Parallel, Distributed computation Graph Model (ParaDiGM) is a formal graph model for the representation of such computations. The VISual Assistant (VISA) is the environment which uses ParaDiGM as the basis for its operation. VISA is, itself, a distributed system, providing separate processes to create and edit ParaDiGM models, and to simulate the resulting models, including the execution of model node interpretations. Mechanisms for supporting dynamic graphs and scaled realtime animation are described in detail.

1. INTRODUCTION

The proliferation of network-based computing has highlighted the difficulty in producing parallel, distributed software to make cost-effective use of these systems. Fundamental challenges lie in *partitioning* the computation into *processes* (or other schedulable units of computation), and in designing an effective interprocess *communication strategy* for the resulting partition.

While computations may be distributed to accommodate resource sharing or to allow for the flexible use of equipment, we focus on distribution to enhance performance. Performance bottlenecks in parallel computations can occur due to constraints on the serial processing within each process, due to fixed overhead costs associated with message passing, or variable delay due to synchronization.

Synchronization delays are generally unbounded in a parallel computation, the extreme case being deadlock. Synchronization delays are also the most difficult performance hazards to identify during the design of the computation, i.e., during the process of selecting partitioning and communication strategies.

While some aspects of the partitioning problem can be handled algorithmically for certain classes of algorithms (e.g., see [9]), there are currently no general solutions to this problem. Instead, it appears that the most promising direction for advancing the technology is to provide tools to support the software engineer as he designs the computation.

The *VISual Assistant* (VISA) system is used to interactively model and simulate partitioning and communication strategies for large-grained parallel computations in a distributed memory MIMD environment. Such computations are characterized by relatively large amounts of computation per partition with respect to the interprocess communication, and by the fact the communication is based on message-passing technique rather than shared memory access.

---

† This work was supported by NSF Cooperative Agreement DCR-8420944 at the University of Colorado. Current address is Département Réseaux, Ecole Nationale Supérieure des Télécommunications, Piece C234, 46 rue Barrault, 75634 Paris Cedex 13, FRANCE. Her electronic mail address is demeure@enst.enst.fr.

‡ This author was supported by NSF Grant CCR-8802283. His address is Department of Computer Science, Campus Box 430, University of Colorado, Boulder, Colorado, 80309-0430 USA. His electronic mail address is nutt@boulder.colorado.edu

VISA supports computational models expressed in a formal modeling language called the *Parallel, Distributed computation Graph Model* (ParaDiGM). The language employs complementary submodels that provide both macro and micro views of a computation. The macro view concentrates on interrelationships among processes, while the micro view describes the details of each process's activity.

VISA and ParaDiGM have been described in other papers, [5-7]; this paper focuses on the implementation of VISA. For completeness, we review the definition and use of ParaDiGM and provide a short description of the use of VISA.

1.1. Interactive Modeling Systems

There are a number of machine-supported, interactive modeling systems in existence, e.g., see STATEMATE [8], CODE [4] PAWS/GPSM [3], PARET [10], Quinault [11], GreatSPN [1], and VISA's sibling systems in the Olympus family, [2, 12, 13].

Each of these systems has a specific, underlying graph model. The system is built to implement the syntax of the graph model and to analyze or simulate the semantics determined by the formal behavioral specification of the model.

These systems are implemented in bitmap graphics environments in which a designer constructs models in the language supported by that system. Typically, the user employs a graphic point-and-select editor to construct a visual model; he may then provide annotation for the model through a wide variety of mechanisms, ranging from popup windows through preparing separate files in a distinct editing session. Systems such as PAWS/GPSM will also predict performance of the model using queueing network analysis techniques (a GPSM model is a queueing network).

The dynamics of model operation can be observed by supplying data to the model, then causing the model to execute in the supplied environment. Many of the systems provide a scaled realtime *animation* of the execution. The modeling system may allow the user to observe the operation of the model through the changing state of the model, and through various displays representing performance.

The power of these modeling systems is in their ability to allow a user to quickly and easily build a model and then to first focus on qualitative aspects of the model, followed by more intensive study of quantitative descriptions of the performance. The designer is able to quickly converge on appropriate models through a spectrum of interactive, experimental tools in a testbed environment.

1.2. VISA Goals

All instances of the Olympus family of modeling systems, including VISA, address the goals mentioned above.† Olympus systems are designed as a collection of processes that interoperate using message-passing protocols. Some processes implement the human-machine interface -- using workstations and bitmap graphics as well as conventional character-oriented terminal interfaces. Other processes implement the logic of the modeling language, independent of the user interface implementations. A final class of processes interpret procedural declarations associated with various parts of the model.

---

† In addition, the Olympus-BPG [13] and Phred [2] systems accommodate asynchronous operation between the user and the system (avoiding the need for checkpoints), and the simultaneous support of multiple users.

An Olympus user can "draw" a graph (within a formal modeling language), annotate the graph with various properties (that map into other aspects of the formal modeling language), then direct the system to analyze and interpret the model. Olympus systems are intended to allow the model designer to quickly hypothesize and test a system architecture with *minimal effort*, encouraging him to experiment with a wide variety of architectural approaches for a solution.

VISA and ParaDiGM focus on parallel, distributed computations for distributed memory MIMD machines. ParaDiGM has built-in mechanisms for representing interprocess communication in a network environment, and the modeling system is designed to represent dynamic process creation and destruction. VISA is intended to support the distributed computation user at the time he designs the architecture of his computation.

Distributed computations for distributed memory MIMD architectures are limited in the nature of parallelism and synchronization that can be applied and ParaDiGM focuses on corresponding mechanisms. For example, two processes can only synchronize through an explicit message-passing mechanism (even if the programming interface appears as shared memory). Deriving a partition of the function of a system involves careful thought as to the type and frequency of communication among the resulting partitions.

Insight into the operation of a particular partitioning and communication approach can be realized by providing complementary and distinct models of the computation, one identifying macro aspects of the computation and the other micro aspects such as are supported by other Olympus family models. This insight can become deeper if the supporting system analyzes and executes alternative models of the computation.

ParaDiGM and VISA depart from other Olympus models in a number of significant aspects: First, ParaDiGM supports both macro and micro models. The base Olympus architecture is useful for supporting the micro model, but it does not make provision for simultaneously supporting both a macro and a micro model.

Secondly, ParaDiGM relies heavily on the use of an ellipsis construct to represent dynamic, variable numbers of replications of defined subgraphs (ordinarily ones corresponding to a process). The effect is that the modeling system must support dynamic graphs; VISA extends the Olympus architecture by providing that support.

Finally, ParaDiGM models represent processes and procedures. The modeling system must accurately represent their execution in scaled realtime. This requires that procedure execution times be estimated by the user, then that the system accurately represent the relative values to reflect appropriate overlap (and mismatch).

VISA is an instance of the Olympus architecture, with considerable extension to both the frontend and the backend. These extensions follow from the need to simultaneously support macro and micro views of a model, to support dynamic graph models, and to implement a refinement of the Olympus scaled realtime representations.

In the remainder of this paper, we provide an overview of the user's view of VISA, then describe its implementation inasmuch as it is a generalization of the Olympus architecture and it advances the state-of-the-art in modeling systems design.

2. VISA CAPABILITIES

In an earlier paper, we have described the use of BPG-Olympus, another instance of the Olympus architecture [13]. BPG-Olympus allows a user to define a BPG (micro) model

using an interactive point-and-select graphic environment (one version of the frontend employs the SunView window systems and another uses Sun's NeWS window system). Once the model is defined, it can be executed in scaled real time, with the result being represented on the frontend display(s) as an animation of the model's execution. The architecture allows the user to alter parameters and the model itself as the animation runs. The system optionally manages time as scaled or simulated time, collecting statistics on the model execution. The animation provides qualitative feedback to the user, while the statistical reports provide quantitative feedback about the model's behavior.

ParaDiGM-VISA extends this modeling system, focusing on distributed memory, MIMD environments for distributed computations. That is, ParaDiGM models reflect individual threads of control which synchronize and exchange information explicitly using message-passing mechanisms. The macro model focuses on the message-passing patterns among processes, and the micro model addresses the details of the threads of control, consistent with the communication strategy.

2.1. ParaDiGM: The Underlying Model

The *Parallel, Distributed computation Graph Model* (ParaDiGM) is composed of two submodels: The *Process Architecture Model* (PAM) is the macro model that describes processes and their intercommunication patterns. The *Distributed Computation Precedence Graph* (DCPG) is the micro model that describes the collection of threads of control for various processes. Our purpose in this paper is to describe the VISA implementation; this is facilitated by providing an intuitive notion of ParaDiGM models without precise definition (which is supplied elsewhere [7]). To that end, we will illustrate ParaDiGM's usage with an example of listener/client/server computations.

In this computation, one or more services (such as a name server) are available to the users. There is a listener process waiting for incoming requests. When the listener receives a request from a user, it spawns a server and establishes a point-to-point connection between the user and the server. The user can then employ the service until it does not need it any more, at which time the server terminates. It is assumed that a given client uses at most one service at a time, and that the listener can only process one request at a time.

The listener/client/server computation is illustrated in Figure 1 as a PAM model at the top and a DCPG model on the bottom. (Because of the awkwardness in monochrome paper replicates, we have removed the explicit mapping of collections of DCPG nodes to each PAM node except in one case noted below.)

The PAM Model

Each PAM node represents an entire process (a circle) or a set of communication facilities that implements interprocess communication (a square). For example, the listener process is represented as a circle on the right side of the PAM and by a set of four connected nodes on the right side of the DCPG. (We have explicitly identified the mapping in this single case.)

Since computations may be dynamic, the PAM model makes provision for defining classes of processes for which any particular instance of a PAM is some set of instantiations of the relevant classes -- processes and communication instances. Ordinarily, the designer views snapshot configurations of instances rather than the class view.

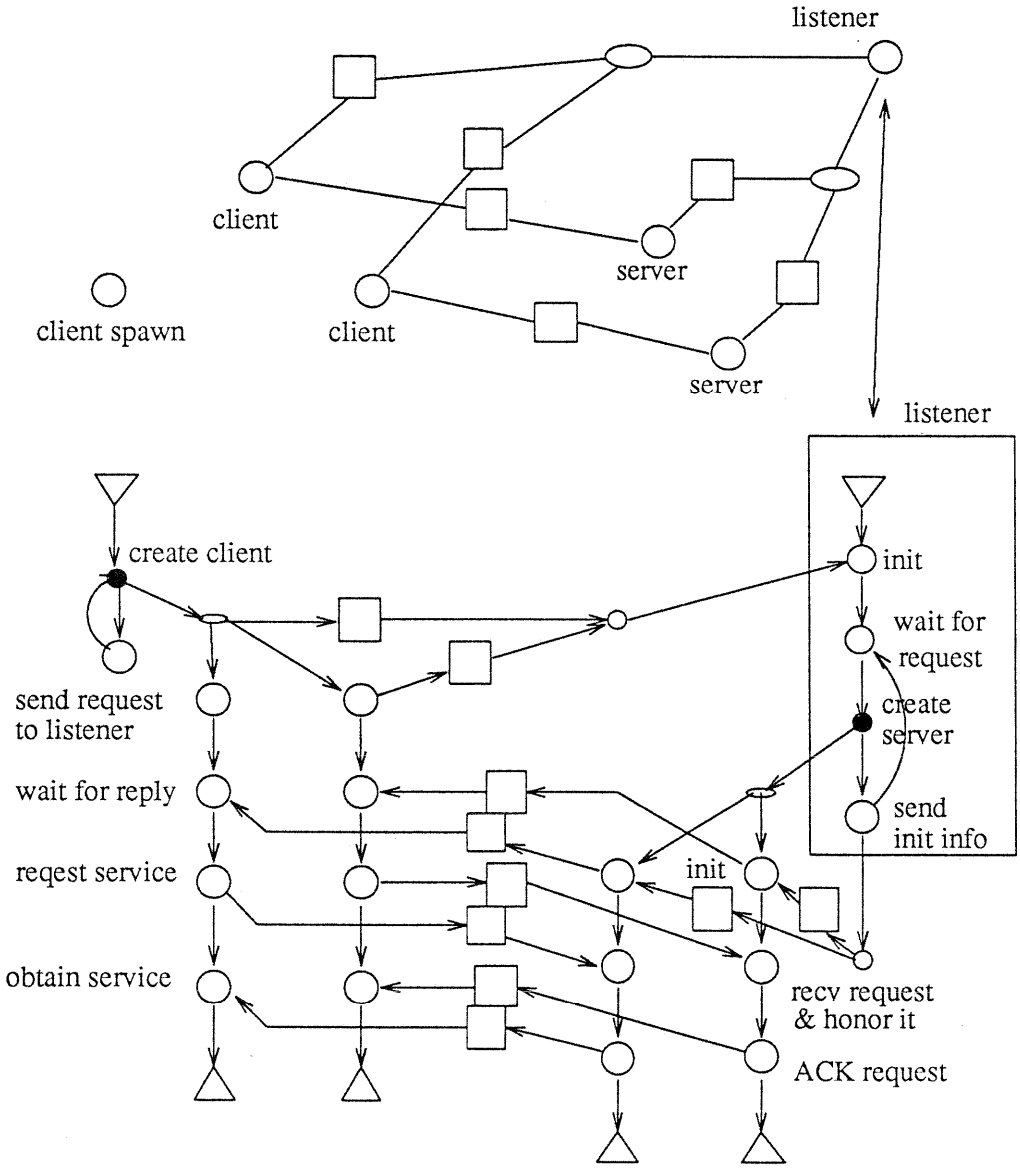


Figure 1: The Listener/Client/Server Computation

Dynamism in the target system can result in radically changing patterns in the process interconnection structure. However, it frequently manifests itself as varying numbers of instances of processes to perform more regular patterns of computation. Recognizing this distinction, PAM incorporates an *ellipsis* construct so that a PAM model can represent a variable number of instances of a single class without explicitly showing each instance. In the figure, the ellipsis is represented by an oval node; it is used to represent variable numbers of instances of the **client** and **server** processes.

The DCPG Model

A DCPG focuses on the control flow of each process that makes up the distributed computation. A large circle represents a fundamental computing *task*, while small, open circles

represent alternative control flow paths and small, filled circles represent creation of new processes through the *spawn* operation. Thus, a DCPG is a static representation of control flow, similar to a standard precedence graph with the addition of alternation. The four-node listener subgraph indicates that the listener first initializes itself, then begins waiting for an incoming service request. When the request arrives, the listener spawns a server (the small, filled node in the listener thread), then sends initialization information to the server.

DCPGs allow cycles, thus additional semantics need to be provided for them to adequately represent control flow. Borrowing from Petri nets, a DCPG node is said to be active if it is marked by a token. Spawn nodes release collections of tokens, while alternation nodes represent serial (OR) control flow for routing tokens.

Just as the ellipsis node was used to represent regular dynamism in PAMs, it is also used to represent the corresponding situation in DCPGs. That is, a thread of control that emanates from a spawn node can employ the ellipsis to represent the creation of a variable number of threads.

Each DCPG node represents a basic block of computation, including all forms of computation one would employ in a conventional procedural programming language. While the DCPG represents the relationship among the execution of these individual blocks (through precedence and control flow represented by tokens), each node's computational detail can be described through the use of a function expressed in a procedural language (the precise nature of the language is relatively unconstrained in the ParaDiGM definition, but is specifically C in the VISA system -- see below.)

Data is represented in the DCPG by incorporating square nodes corresponding to repositories. DCPG repositories may represent messages and message-passing mechanism (solid squares in the figures), or data repositories that are local to a process.

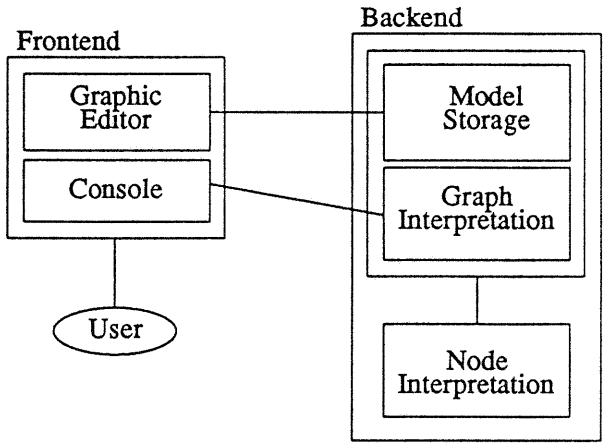
The Node Interpretation Language Extensions

The VISA task interpretation language is C supplemented by a library of functions that contains the primitives needed to write parallel, distributed computations (and to match the ParaDiGM graphical constructs). In this language, each process is identified by a process number; groups of processes can be defined that are identified by names (the notion of group is used for multicasting messages); arcs in DCPG graphs have a unique identifier or arc number. A mapping number is associated with each PAM component, and the corresponding part of the DCPG graph. VISA assists in mapping model atoms with variable names used in the interpretation.

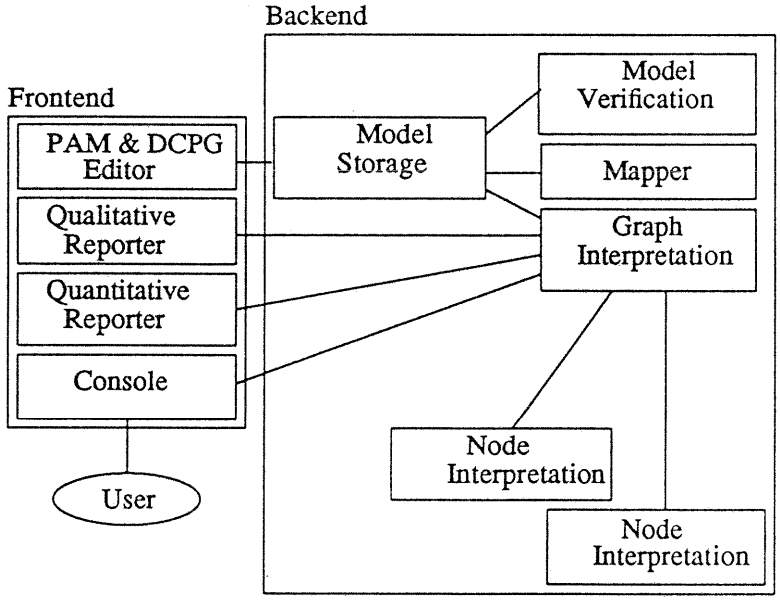
3. THE VISA IMPLEMENTATION

VISA extends the Olympus architecture to support dynamic graphs and ParaDiGM. The Olympus architecture focuses on the problems of providing a modeless user interface, generalized node interpretation facility, separating model syntax and semantics, and supporting multiple users working on one model.

The frontend portion of an Olympus system (Figure 2(a) implements all presentation aspects of the system, including the user's view of the model, the editor, and the means by which the human and the system interact. The backend implements model storage, graph interpretation (token movement, scaled realtime management, and node interpretation invocation), and node procedure execution.



(a) Olympus Architecture



(b) VISA Architecture

Figure 2: The Architecture

The frontend is implemented as a separate process from the backend functions -- the VISA frontend is implemented as a NeWS client and server. Olympus backends are implemented by a variable number of processes, one for graph storage and interpretation, and an arbitrary number for node interpretation. The various processes are independent, asynchronous processes with Olympus protocols used to coordinate their activity.

The frontend process dictates the full nature of the user interface. It is always in a state in which it can accept editing or console commands, based on the input multiplexing facility (in contemporary window systems, this amounts to an event-driven approach in which editing and console operations are implemented as event handlers). Each request is sent to the

backend independent of context, allowing editing and console operations to be intermixed.

The backend is independent of the user interface, dealing only with messages from the frontend. This not only allows the frontend to dictate the nature of the interface, it also allows it to present the model using any view (syntax) that is consistent with the internal definition kept in the backend. The syntax and the semantics of the model are independent.

By carefully designing the protocol between the frontend and the storage as a transaction-based protocol, any number of frontends can be connected to the backend. When a client performs an operation, e.g., a graph editing command, the command is sent to the backend. The backend executes the command, then acknowledges the completion by multicasting the result to all frontends. Each frontend can then update its local state to be consistent with the new state of the backend.

The node interpretation facility is distinct from the graph interpretation. The graph interpreter executes node interpretations using remote procedure call (RPC). Thus, the node interpretations are neither interpreted nor statically bound to the graph interpretation code. This allows the node interpretation language to be general.

The Olympus architecture is not designed to simultaneously handle multiple graph models such as PAMs and DCPGs. It incorporates a single model storage and the frontends are designed around one model. A more difficult problem is the existence of dynamic graphs in ParaDiGM, related to the use of the ellipsis construct. VISA extends the Olympus design to manage both of these problems. Also, ParaDiGM models deal with complex simulated time issues; while it is useful to provide scaled realtime execution, often the simulation routines (invoked via RPC) require more realtime than the real routines. This requires a careful redesign of the time management portion of the Olympus graph interpreter.

3.1. The System Organization

VISA is organized as a set of modules as shown in Figure 2(b).

The Frontend

PAM & DCPG Editor

The editor implements the graphic syntax for ParaDiGM models. It is a color bitmap, point-and-select editor written on top of the Sun NeWS window package. Thus, the editor is actually implemented as a NeWS client and server, with the server managing the display and the client implementing the logical parts of the editor.

The editor is partitioned into two logical parts (implemented in one process), one to manage PAMs and the other to manage DCPGs. The user can switch between the editors with command selection.

Qualitative Reporter

This is the animation facility. It uses the editor's display to represent the model. The qualitative reporter acts upon messages from the graph interpreter that describe the state of tokens in the graph. Thus, it is a simple module to remove and paint tokens upon command from the backend.

Quantitative Reporter

This module displays simulation results, currently as graphs and tables. In particular, such information can be monitored during the execution of a simulation run, or summarized at the end of the simulation. Results for several simulations can also be

presented in a way that makes their comparison easy.

Console

The console is essentially a menu handler, used to forward commands to the backend. It can be invoked at any time.

The Backend

Model Storage

The model storage provides a global mechanism for keeping the definition of the model. It is written by the editor, and read by various other modules.

Model Verification

The checking module functions include: (1) A function to check the correctness of the PAM graphs; (2) A function to check the correctness of the DCPG graphs; (3) A function to check that hierarchies of DCPGs are correctly built; (4) A function to check that the interpretation is syntactically correct - e.g: an interpreter or a compiler for the interpretation "language"; (5) A function to check the consistency between the interpretations and the DCPG graphs drawn; (6) A function to check that the mapping between DCPG and PAM is correct.

Model verification is not implemented in the current version.

Mapper

The purpose of the mapping module is to implement automatic algorithms for mapping a DCPG graph onto a PAM. Currently, all mapping is accomplished manually within the editor.

Graph Interpretation

The simulator bases its operation on the graphs, the interpretation procedures, and the various simulation parameters specified by the user. It describes the state of the running computation (which is interpreted by the qualitative reporter in the frontend), by stepping tokens through the DCPG graphs, and by identifying active components of the PAM graphs.

The graph interpreter implements the fundamental aspects of the simulation. It has a view of the model (which includes the simulation information). Upon request from the console, it starts the simulation. This consists of traversing the DCPG graph, taking appropriate action wherever necessary. In particular, the simulator interacts with the node interpreter(s) using a listener/client/server paradigm. Details of this interaction are provided below.

Managing the simulated time is a nontrivial task; we provide an extended discussion of the means by which this is accomplished below.

Node Interpretation

Each node interpreter executes C code with VISA library procedures whenever the graph interpreter determines that this should happen. Since VISA is intended to handle arbitrarily complex specifications of the computation, the node interpreters can be replicated so that the various interpretations can be evaluated concurrently. The details of this graph and node interpreter interactions implement dynamic graphs, so we extend the discussion of this part of the implementation.

3.2. Handling Dynamic Graphs

The model storage keeps a static description of the model, including ellipsis constructs. Each ellipsis construct (in the PAM and in the DCPG) represents regular, dynamic specification of the model. In the PAM, it means that the number of processes in execution at any time is determined by the semantics of the corresponding DCPG spawn node interpretation. Thus, there is a strong correspondence between the occurrence of a spawn node in the DCPG and the ellipsis in both the DCPG and the PAM.

Figure 3 is a PAM of the VISA implementation. The frontend processes represent the static NeWS client server implementation. However, most of the complexity of the model illustrates the implementation of the dynamic interpretation facilities.

When VISA is started, a variable number of interpretation listener processes are started, one per machine. The graph interpreter process establishes point-to-point IPC with each listener (a socket-based virtual circuit). The listener creates new node interpretation server processes upon request of the user, and the server executes the interpretation procedures upon request of the simulator. The user specifies a procedure name corresponding to the interpretation for each DCPG control nodes. The code for these procedures is compiled with the listener code. The invocation of a procedure is then made by an RPC call to the server corresponding to the process to which the node belongs.

Notice that interpretations can vary in detail from a simulation to the code that would be written in a real implementation of the modeled computation. So the tool can support and execute the code for the implementation. The information gathered by the simulator is then available to evaluate the implementation, and the model code can be easily ported to a more efficient supporting system.

The interpretation servers execute the interpretation procedures until they reach a Para-DiGM system call, at which time they call the simulator back and block until it calls them back. They also let the simulator know when they are finished executing a procedure.

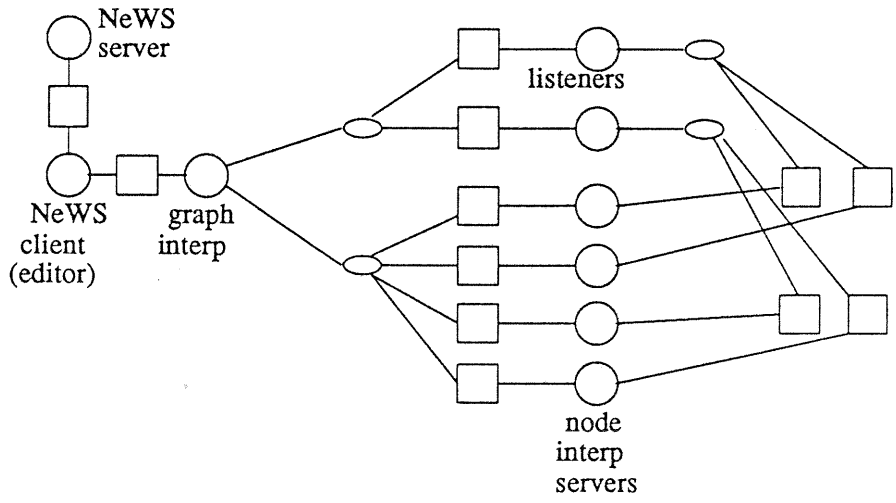


Figure 3: VISA PAM Diagram

3.3. Managing Simulated Time

All the message traffic goes through the graph interpreter; any message "sent" by a process will go to the simulator, and will be dispatched to the appropriate process(es) from there. The simulator is in charge of the timestamping of the events, and is therefore responsible for deciding on the relative ordering of the events.

The simulation is a combination of actual execution and simulation. The procedure specified for each of the nodes of the DCPG graph is executed by an interpretation server. Some of the interpretations can be simple (or complex) simulations of what the actual code would be. Therefore, we do not want to base the simulation of the computation on the time it takes to execute the interpretations, since the interpretation time may be unrelated to the simulation time. Given the precision of the clock on the SUN workstations, it is not possible to accurately time the executions. In addition, we want to be able to simulate processors with different loads, but we do not want simulation results that depend on the load of the network at the time the simulation is done. For all these reasons, the simulation times are not based on the interpretations, but on the simulation information provided by the user (which can be based on timing of the interpretation procedures if desired).

An "estimated" execution time is associated with each event in the computation. It is considered to be the real execution time for the event. This time can be obtained in several ways. If the user has specified a distribution time for the node, this distribution is sampled. Otherwise, if he has specified a distribution time for the component class corresponding to the process the node belongs to, the time is sampled from that distribution time. If no distribution times were specified, but a list of time values was provided, a value is taken from a list (the list is used circularly: The index of the element in the list that gives the value, is the number of events for which the list was used, modulo the number of elements in the list). Finally, if no distribution function and no list of values were specified, a fixed default value would be used.

Now several things can happen: (1) the estimated simulation time can be smaller than the execution time, in which case the simulator will timestamp the event "return from the procedure" at a time that is past; (2) the events can get out of order because of the relative load of the processors on which the interpretation servers are executing; and (3) the simulator introduces some overhead and might therefore cause some delay in the simulation. In order to avoid these problems, we introduce a scaling factor by which all the times are multiplied (which is equivalent to slowing down the simulation by that factor). Let

- T0: Start time for simulation;
- EI(e): Interval of time it really takes to execute the event e;
- SI(e): Estimated time associated with the event e, supposed to simulate the execution of the event e;
- ASI(e): Adjusted simulation time for the execution of the event e (time the simulator will actually take to execute the event);
- SF: Scaling factor;

Note that we are talking about intervals of times. We consider that the interval of time for an event is the interval of time between the immediately preceding event and the event considered.

By definition

$$ASI(e) = SI(e) * SF;$$

The aim is to choose SF such that for all events e: (1) $ASI(e) \geq EI(e)$, (2) SF is sufficiently large that the overhead due to the simulator is negligible compared to the adjusted simulation times, and (3) that the actual relative loads of the processors can be ignored. The choice of SF depends on the typical real execution times and simulated execution times; the user picks a value for SF. The simulator detects when the scaling factor is too small (when it gets events after their adjusted simulation time is passed), and provides feedback to the user who can then adjust it if desired.

Simulating Processor Load

The same idea can be used to simulate the relative loads of processors. This is done by assigning a load factor to each PAM component. The larger the load factor, the slower the processor running the PAM component (either because of a heavy load or because the processor is slower). So if PLF designates the processor load factor, we have:

$$ASI(e) = SI(e) * SF * PLF;$$

Transmission Time

The simulator is also responsible for simulating message passing. Message send and message receive are implemented as remote calls to the simulator. Let

- TT denote the average transmission time (average time required for a message to transit from a process to another one);
- s(m) denotes the event "process executes a send message m";
- r(m) denotes the event "process executes a receive message m";
- a(m) denotes the event "message m is made available to a process";
- ar(m) denotes the event "message m is actually received by a process";
- WT(m) denotes the time a process waits for a message;

Then

$$ASI(s(m)) = SI(s(m)) * SF,$$

where the interval of time considered is the one between the previous start procedure, send message or receive message event and the current send message event executed by the procedure.

$$ASI(r(m)) = SI(r(m)) * SF,$$

where the interval of time considered is the one between the previous start procedure, send message or receive message event and the current receive message event executed by the procedure. This gives the adjusted simulation time interval associated with the event "process performs the receive operation". The adjusted simulation time interval associated with the event "message is made available to a process" can be computed as follows:

$$ASI(a(m)) = (SI(s(m)) + TT) * SF,$$

So the adjusted simulation time interval associated with the event "message actually received by process" is:

$$ASI(ar(m)) = \max(ASI(r(m)), ASI(a(m))) = \max(SI(r(m)), SI(s(m)) + TT) * SF,$$

and the time a process waited for a message is:

$$WT(m) = \max(0, (ASI(ar(m)) - ASI(r(m)))).$$

We can simulate a slow communication channel, or load on the communication channel, by multiplying the average transmission time by a transmission delay factor, TDF. The time at which a message is made available to a process becomes:

$$ASI(a(m)) = (SI(s(m)) + TT * TDF) * SF,$$

so:

$$ASI(ar(m)) = \max(ASI(r), ASI(a)) = \max(SI(r), SI(s) + TT * TDF) * SF.$$

Using the Simulation Times

The parameters used to compute the simulation times are the scaling factor (SF), the average transmission time (TT), the process load factor for eachP PAM component (PLF), and the transmission delay factor for each PAM communication (TDF). TDF: transmission delay factor (for each PAM communication). The simulator is notified of all the events. It computes the adjusted simulation time interval associated with each event and deduces the time at which the event should occur from the simulation point of view. It queues the events in order (with respect to the computed time) and checks the queue at each "tick of the clock." It executes all the events in the queue for which the time is less than or equal to the clock time. If the time computed for an event is less than the current clock time (incorrect ordering of events), the simulator notifies the user of the system (who can then take action by adjusting the scaling factor SF).

4. SUMMARY

VISA and ParaDiGM represent the symbiosis of formal models and relatively complex support systems. Without VISA, ParaDiGM is of limited use, and ParaDiGM defines the semantics of VISA.

The modeling approach is different than others in that it embraces multiple views of a computation, at macro and micro levels. The macro model describes the relationship among independent threads of computation, and the micro model describes the details of a thread.

The VISA implementation extends the Olympus architecture by incorporating a mechanism for specifying dynamic graphs. The model specification is static, but incorporates an ellipsis construct. The node interpretations specify a value for the ellipsis construct when the model is executed.

Because of the wide range of possible simulation time values that might be useful for studying the qualitative aspects of a ParaDiGM model, the graph interpreter has been designed to accommodate scaling in such a manner that it preserves complex temporal relationships among the simulated processes. The scaling mechanism is also used to experiment

with IPC and processor performance factors.

We have focused on implementation issues in this paper, and believe that many of the ideas described here may assist other modeling system designers with their systems. While VISA is currently operational, the Quantitative Reporter and the Mapper are very simplified versions; the Model Verification module has not been implemented. The work has demonstrated the viability of the implementation approach, and of the general means of addressing the design of parallel, distributed computations.

5. ACKNOWLEDGEMENTS

This work benefited from continuing review and comment by the members of "the modeling group": Adam Beguelin, Dennis Heimbigner, Jeff McWhirter, Bruce Sanders, Carolyn Schauble, and Sharon Smith. We especially wish to acknowledge the use of the software built by Bruce Sanders and Steve Elliott. This work was supported by NSF under Cooperative Agreement DCR-8420944 and Grant CCR-8802283.

6. REFERENCES

1. G. Balbo and G. Chiola, "Stochastic Petri Net Simulation", *1989 Winter Simulation Conference Proceedings*, Washington, D. C., December 1989, 266-276.
2. A. L. Beguelin, "Deterministic Parallel Programming in Phred", University of Colorado, Department of Computer Science, Ph. D. Dissertation, May 1990.
3. J. C. Browne, D. Neuse, J. Dutton and K. Yu, "Graphical Programming for Simulation of Computer Systems", *Proceedings of the 18th Annual Simulation Symposium*, 1985.
4. J. C. Browne, M. Azam and S. Sobek, "CODE: A Unified Approach to Parallel Programming", *IEEE Software 6*, 4 (July 1989), 10-18.
5. I. M. Demeure and G. J. Nutt, "Prototyping and Simulating Parallel, Distributed Computations with VISA", submitted for publication, October 1989.
6. I. M. Demeure, S. L. Smith and G. J. Nutt, "Modeling Parallel, Distributed Computations using ParaDiGM -- A Case Study: The Adaptive Global Optimization Algorithm", *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.
7. I. M. Demeure, "A Graph Model, ParaDiGM, and a Software Tool, VISA, for the Representation, Design, and Simulation of Parallel, Distributed Computations", University of Colorado, Department of Computer Science, Ph. D. Dissertation, June 1989.
8. D. Harel, A. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman and A. Shtull-Trauring, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems", *IEEE Transactions on Software Engineering 16*, 4 (April 1990), 403-414.
9. C. E. Houstis, E. N. Houstis and J. R. Rice, "Partitioning PDE Computations: Methods and Performance Evaluation", *Parallel Computing V* (July 1987), 141-163.
10. K. M. Nichols and J. T. Edmark, "Modeling Multicomputer Systems with PARET", *IEEE Computer 21*, 5 (May 1988), 39-48.
11. G. J. Nutt and P. A. Ricci, "Quinault: An Office Environment Simulator", *IEEE Computer 14*, 5 (May 1981), 41-57.

12. G. J. Nutt, "A Flexible, Distributed Simulation System", *Tenth International Conference on Application and Theory of Petri Nets*, Bonn, West Germany, June 1989, 210-226.

13. G. J. Nutt, A. Beguelin, I. Demeure, S. Elliott, J. McWhirter and B. Sanders, "Olympus: An Interactive Simulation System", *1989 Winter Simulation Conference Proceedings*, Washington, D. C., December 1989, 601-611.

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

Isabelle M. Demeure (*)
Sharon L. Smith (**)
Gary J. Nutt (***)

Department of Computer Science
University of Colorado
Campus Box 430 - Boulder, CO 80309.
(303) 492-7581.
demeure@ulysse.enst.fr, smith@boulder.colorado.edu,
nutt@boulder.colorado.edu.

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.

1. Overview of the Approach - Background. This paper summarizes 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, to evaluate the modeling technique by using it to represent a complex, parallel, distributed algorithm. Second, to evaluate and compare different implementations of the algorithm.

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 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 different types of processes that have both dynamic and static behavior, and 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.

We feel that ParaDiGM was successful in capturing all of these aspects. Moreover, ParaDiGM graphs provide a comprehensive overview that allows us to use these graphs to compare 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 review the parallel global optimization algorithm; a more complete description appears in the companion paper [8]. In Section 2, we introduce ParaDiGM. In Section 3, we present ParaDiGM graphs for two implementations. In section 4, we compare these graphs, and evaluate ParaDiGM.

The adaptive, global optimization algorithm that we chose for our experiment is a modification of a static parallel algorithm for global optimization [1]. Both algorithms are stochastic methods

This research has been supported by (*) NSF Cooperative Agreement DCR-8420944, (**) AFOSR-85-0251 and (***) NSF grant CCR-8802283

based on the work of [7].

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. (*sampling phase*)

(2) Select start points for local minimizations. (*start point selection phase*)

(3) Perform local minimizations from all start points. (*local minimization phase*)

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

The 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 sampling and start point selection phases of the algorithm are conducted simultaneously on the P machines. Next, the start points selected in step 2 are distributed to processors so that they can conduct the local minimization phase 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 are removed, allowing phases of several iterations to be in progress at the same time. In addition, regions that are most likely to produce the global minimum are allowed to subdivide. In [8], it is demonstrated that these two techniques can yield dramatic performance improvements over the static parallel algorithm.

Although the ideas behind the new parallel version are conceptually simple, it proved to be a difficult task to describe the adaptive algorithm as easily as the static version. The primary reason for this difficulty is that the new algorithm necessarily has more complex communication and process generation patterns than the previous version, so that simply listing the steps of the algorithm is not sufficient to convey the adaptive, asynchronous character of the computation. It is thus necessary to employ a specific mechanism to represent details of the implementations.

2. ParaDiGM. ParaDiGM 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, using message passing as the interprocess communication mechanism. The model can also be used to study the characteristics of a given implementation, to compare and contrast different possible implementations, and as a visual interface for graphic tools (we shall not deal with the latter issue here).

ParaDiGM is composed from two complementary models: the Distributed Computation Precedence Graph (DCPG) and the Process Architecture Model (PAM). 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 [6]. The PAM model represents 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. There is a natural mapping between the two models.

ParaDiGM is related to several other graph models [3, 4, 5, 9]. However, the originality and strength of ParaDiGM lie in the fact that it focuses on a specific model of computation: Parallel distributed computations using message-passing. The constructs used for such computations are therefore built directly in the model, providing a 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.

2.1. The DCPG. 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.

The control flow subgraph in a DCPG is made of nodes of various types and arcs joining the nodes (see Figure 1 (a)-(h)). *"Task"* nodes are used to model any sequence of code. *"Or"* nodes represent alternative choices. *"Spawn"* nodes model the creation of one or more processes. The creation of a number of identical processes is indicated by the *" ellipsis"* construct. A *"recursive spawn"* models computations in which a process spawns a process identical to itself (or to one of its

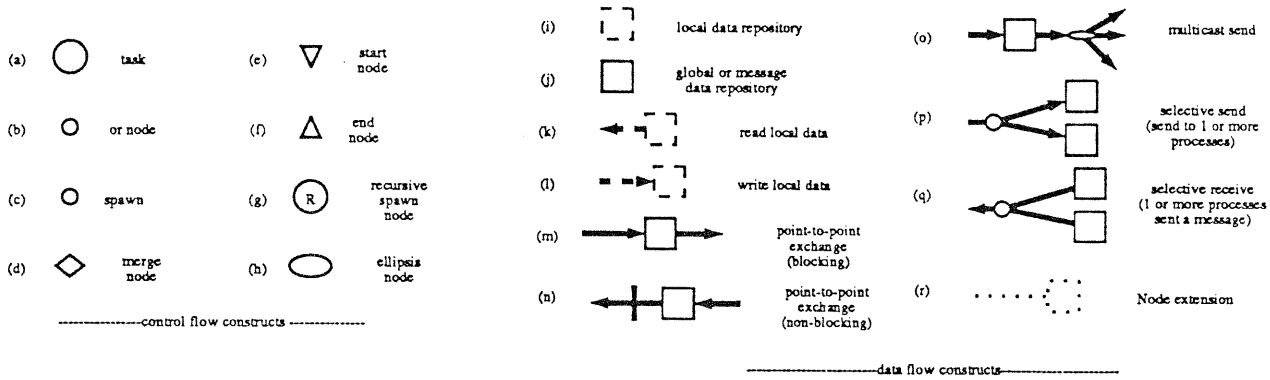


FIG. 1. *Legend for the Constructs of DCPGs*

parents) which in turn spawns a process identical to itself, and so on. The *"start"* node models the creation of a process by direct action of the user, and the *"end"* node construct is used to represent the termination of an existing process. Finally, *"merge"* nodes are used to join two or more control flow arcs together.

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 (see Figure 1 (i)-(r)). Local data repositories are used to model any data structure local to a process that can be "read from" or "written to". 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. The message passing constructs available are *"point-to-point"* messages, where there is one sender and one receiver; *"multicast"* messages, where there is one sender and a number of receivers; *"selective send"* messages, where there is one sender and one or more possible receivers; and *"selective receive"* messages, where there is one receiver and one or more possible senders. The receive primitives can be blocking (this is the default) or non-blocking.

"Node extension" constructs are used to simplify the readability of a DCPG graph without changing its semantics.

2.2. PAM. From the point of view of the PAM model, a computation is a collection of processes and communication relations. 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 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).

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.

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 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. We refer the reader to [2] for more details and how it arises in this context.

3. Two Implementations of the Adaptive Algorithm. In this section we will present two implementations of the adaptive global optimization algorithm and their corresponding ParaDiGM

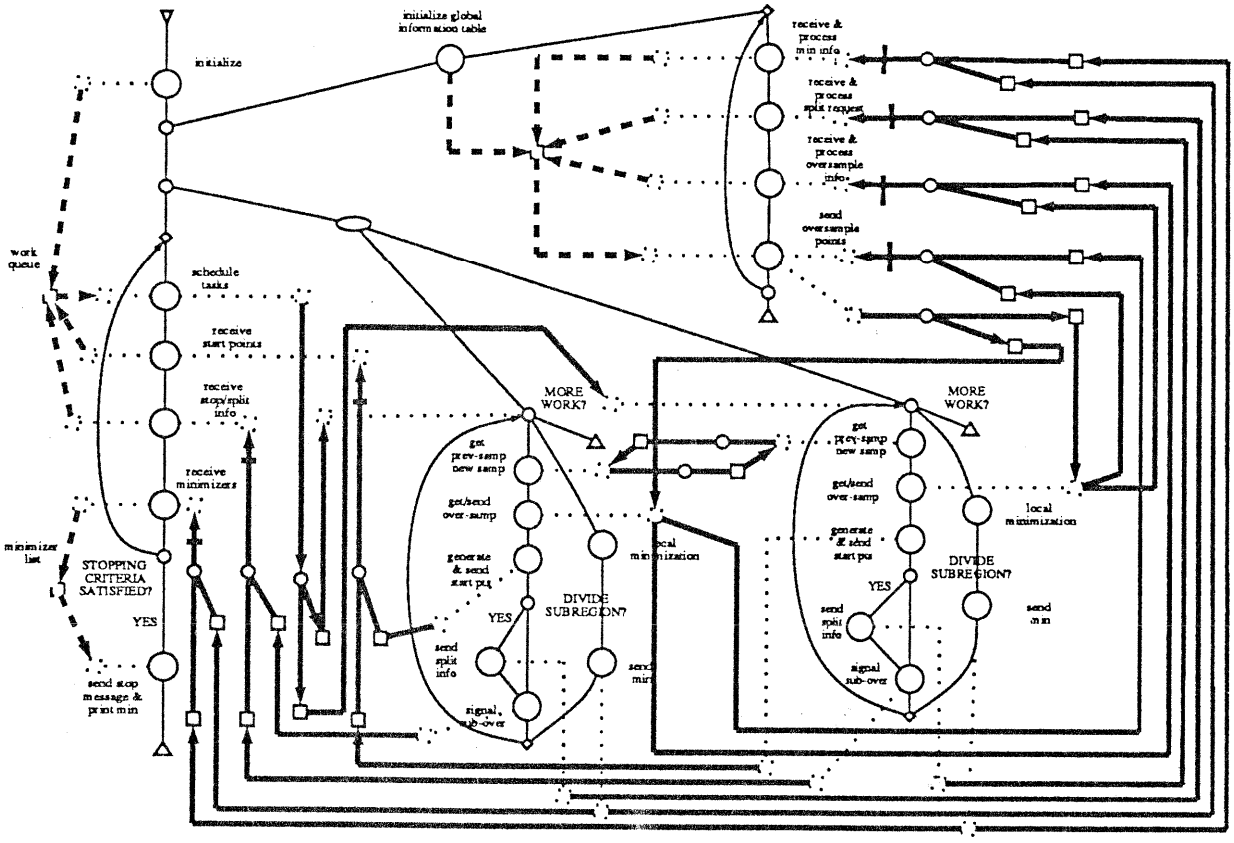


FIG. 2. *DCPG for the homogeneous implementation*

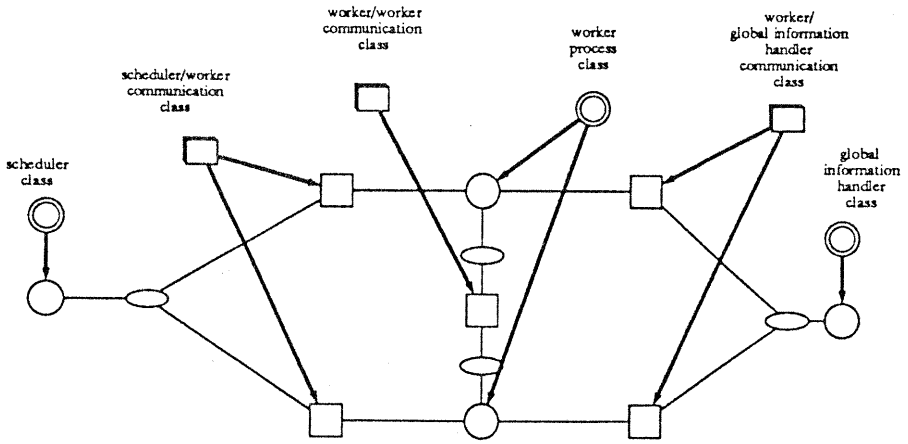


FIG. 3. *PAM Class Model homogeneous implementation*

models. The first one, referred to as the "homogeneous implementation", has been fully implemented. The second design is referred to as the "heterogeneous implementation". The heterogeneous implementation distinguishes between processes that service different types of tasks, whereas the homogeneous implementation does not. The heterogeneous implementation more closely matches the original conceptual design of the algorithm.

In describing the two implementations, we adopt the following terminology: The unit of distribution is a *process*; Processes are made of a collection of *tasks*. The two tasks of primary interest to both implementations are the *subregion* task, consisting of the sampling and start point selection phases mentioned in Section 1, and the *local minimization* task, consisting of the local minimization phase. A primary distinction between the two implementations is how the identified tasks are assigned to processes.

3.1. The Homogeneous Implementation. The DCPG and PAM models for the homogeneous implementation are shown in Figures 2 and 3. The leftmost part of the graph describes a scheduler process that distributes tasks for the computation. The top and upper right parts of the graph correspond to the global information handler process, that manipulates the global data. *Worker* processes, that service the computation of subregion and local minimization tasks, are created when the scheduler initiates processing and remain active for the duration of the algorithm.

When a subregion or local minimization task finishes execution, the worker process requests additional work from the scheduler. The scheduling is modeled by a message construct from the *schedule tasks* node 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. The scheduler selects a task and sends a message to the idle worker. The exchange of sampling information is modeled as a message from the node *get prev-samp, new-samp* of one worker and 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.

3.2. The Heterogeneous Implementation. The heterogeneous implementation is described by figures 4 and 5. The implementation is an alternate view of the algorithm and is a basis for comparing the graph models; and second, the graph illustrates the relationship between the DCPG and PAM graphs. A detailed discussion of the mapping between the DCPG and PAM graphs for both implementations is provided in [2].

In this model the global information handler process is the same as in the first model. Process scheduling and creation differs from the homogeneous implementation with the presence of *subregion* processes, initially created by the scheduler process and equal to the number of processors available. The subregion processes perform the *sampling phase* and the *start point selection phase* of the algorithm (i. e., the subregion tasks). When a subregion task divides the subregion, a new subregion process is spawned to handle the new subregion. The fundamental 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 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.

4. Conclusion.

4.1. Comparison of the Adaptive Algorithm Models. One of our goals was to compare structural aspects (processes and interprocess communication relationships) in the two implementations of the 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 their static representations, without implementing the algorithms.

In the homogeneous implementation, a static set of P worker processes is created and the scheduler assigns work when a worker process completes one phase and is ready to move onto

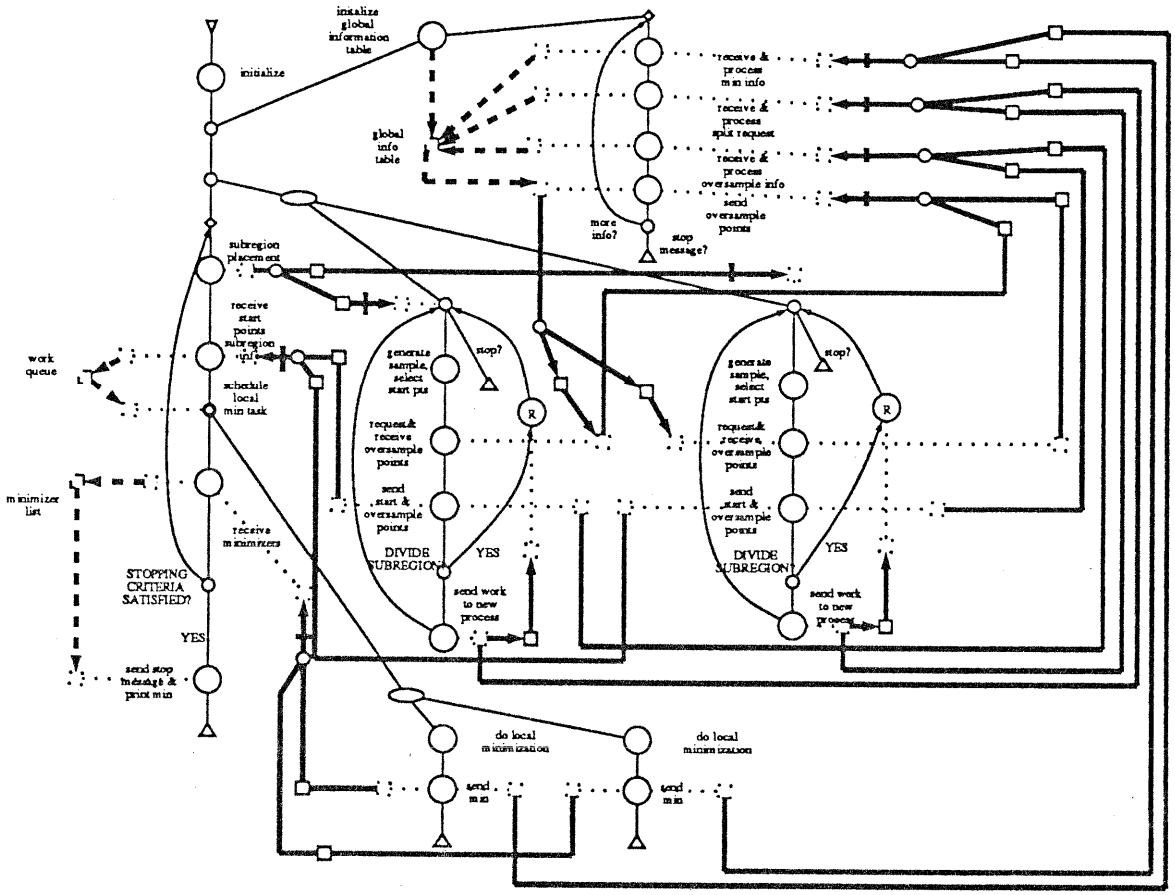


FIG. 4. *DCPG for the heterogeneous implementation*

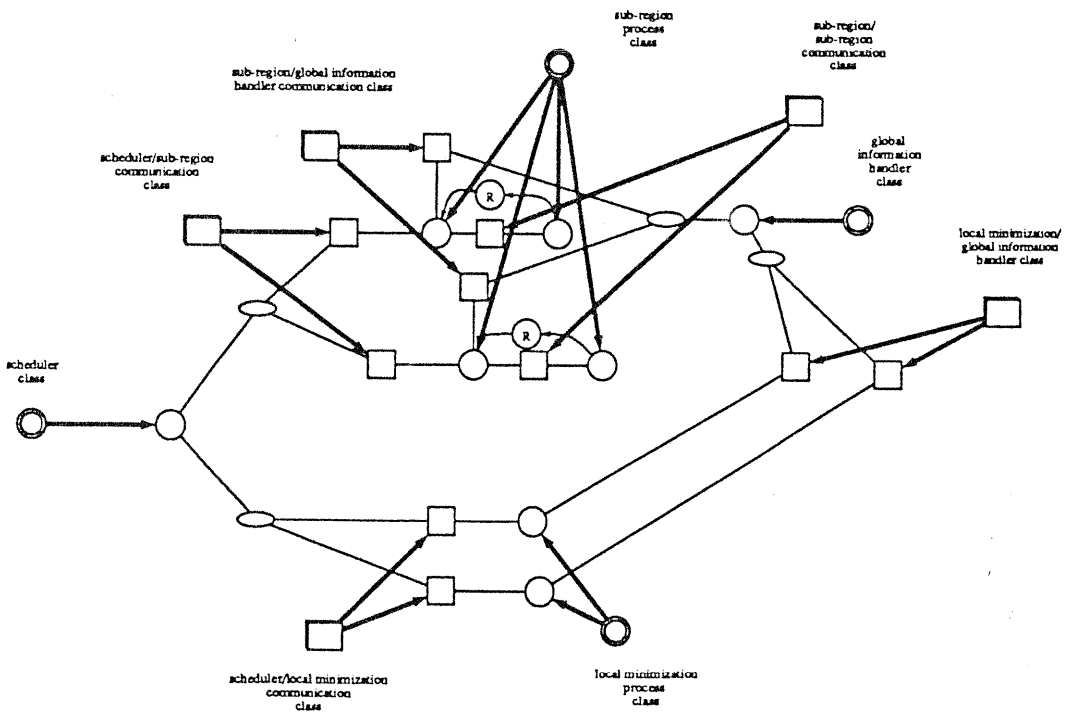


FIG. 5. *PAM Class Model heterogeneous implementation*

the next one. In the heterogeneous implementation, worker processes are replaced by subregion processes. Initially, there are P subregion processes but that number varies with the number of subregions. Additionally, the scheduler creates 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 trades off communication for process management overhead. A comparison of Figures 2 and 4 illustrates the proliferation of processes employed in the heterogeneous implementation. 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.

The PAM graphs in Figures 3 and 5 further illustrate the differences in the types of processes and their intercommunication requirements for the homogeneous and heterogeneous implementations. A detailed discussion of this comparison can be found in [2].

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. We felt that the heterogeneous implementation would suffer from process creation and destruction overhead, and in disseminating process topology information as the topology changed. To address these issues, we implemented a form of "lightweight" processes. Since the host operating system did not support this concept, we coalesced logical processes into a general-purpose worker process. 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 foreseen 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, we could have reached the same conclusions by working with ParaDiGM models that we reached by experimenting with actual implementations. Since we did part of 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 parallel, distributed computations. Initially, adaptive global optimization appears to differ from the static distributed global optimization in minor ways. However, the resulting implementation 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.

ParaDiGM can be used as an intermediate language between concepts and (pseudo) code. 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 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. 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 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 the PAM model 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.

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 is addressed with the concept of hierarchy of DCPGs [2]. At the first level of such a hierarchy, the DCPG graph is represented with a few nodes and communication arcs. 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, and encourages top-down development of the design.

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 represents distributed computations in terms of 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 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.

On a pragmatic level, it is difficult to talk about strategies for employing parallelism. ParaDiGM is an intermediate representation mechanism. While we have no quantitative argument that ParaDiGM models are superior to verbal models, we are able to report 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.*

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, S. L. Smith, and G. J. Nutt. *A Case Study: ParaDiGM and the Adaptive Global Optimization Algorithm.* Technical Report CU-CS-419-88, Department of Computer Science - University of Colorado, Boulder, December 1988.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] S. L. Smith, Elizabeth Eskow, and Robert B. Schnabel. An adaptive, asynchronous parallel global optimization algorithm. In *Fourth SIAM Conference on Parallel Processing for Scientific Computation*, December 1989.

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