A framework for SPMD applications with load balancing

Alexandre Plastino[1], Celso C. Ribeiro[2], Noemi Rodriguez[2]

[1] Universidade Federal Fluminense
Departamento de Ciência da Computação
Niterói 24210-130, Brasil
plastino@dcc.ic.uff.br

[2] Pontifícia Universidade Católica do Rio de Janeiro
Departamento de Informática
Rio de Janeiro 22453-900, Brasil
{celso,noemi}@inf.puc□rio.br

Abstract—
This work describes SAMBA, a framework for the development of parallel SPMD applications with load balancing. SAMBA contains the structure common to different SPMD applications and a library of load-balancing algorithms. This structure allows the developer to concentrate on the specific problem at hand. The load-balancing library makes easier the identification of the most appropriate strategy for a given application. Experiments conducted with SAMBA on three different parallel applications are described, illustrating the ease of use of the proposed framework and the relevance of load balancing.

Keywords— **SPMD, load balancing, framework, data parallelism, parallel programs.**

I. Introduction

Parallel algorithms are classically designed using *functional* (*control*) or *data* parallelism. Associated with MIMD (Multiple Instruction, Multiple Data) architectures, data parallelism originated the SPMD (Single Program, Multiple Data) programming model [Qui94]: the same program is executed on different processors, over distinct data sets. In this programming model, each *task* is characterized by the data over which the common code is to be executed.

The SPMD programming model has been widely used in parallel programming, due to the ease of designing a program that consists of a single code running on different processors. Moreover, data decomposition is a natural approach for the design of parallel algorithms for many problems [Mat96].

In spite of this relative simplicity, coding and debugging a new SPMD application may not be a trivial task. It is particularly hard to determine the most appropriate load-balancing technique for each application, partly because of the large variety of load-balancing algorithms that have been proposed [AM96], [FG96], [FTI90], [LPR97], [WLR93], [ZLP97]. The selection of a suitable strategy is essential for an efficient control of dynamic imbalance factors, which may include the lack of information about the processing requirements associated to each task, the dynamic creation of new tasks, or the variation of processor load external to the application.

The aim of the work we present in this paper is to support the development of SPMD applications, with special emphasis on the identification of an appropriate load-balancing strategy. The central contribution is the SAMBA (Single Application, Multiple Load Balancing) framework [Pla00], which captures the structure and characteristics common to different SPMD applications, offering a basis for their development. The goal of the proposed framework is to facilitate the following tasks:

- Coding new SPMD applications with load balancing: given the routines that create and execute tasks, an SPMD application with load balancing is automatically generated. This allows the programmer to concentrate on the specific problem at hand, reducing the programming effort.
- Identification of the most appropriate load-balancing strategy for a specific application: the designer can easily generate a different version of the application for each of the load-balancing strategies offered in SAMBA's library. This allows different strategies to be tested and compared at no extra programming cost.
- Design and evaluation of new load-balancing algorithms: SAMBA offers a set of functions that are used in many load-balancing algorithms. Besides, the strategies in the load-balancing library can be used as a basis for the development of new algorithms.

A first version of the SAMBA framework has been implemented [PRR99] in C using the MPI library [MPI93], and has been tested on three different environments: an IBM SP2 system, a network of Sun workstations, and a cluster of IBM PCs.

This paper also proposes classification criteria for load-balancing algorithms designed for SPMD applications. The proposed set of criteria provides a terminology for describing different algorithms and establishes a background for their comparison and classification, helping the programmer to understand the differences between them. The design of

SAMBA's load-balancing library was guided by these classification criteria. By implementing algorithms in different classes, we have been able to check that the framework effectively captured the functionality of a wide range of load-balancing algorithms.

Finally, we also discuss the analysis and the effectiveness of load-balancing strategies in the context of three parallel applications developed with SAMBA. The goal of this analysis was to determine the effectiveness of SAMBA for developing different applications and for supporting the performance analysis of load-balancing strategies for them. The three applications – matrix multiplication, adaptive quadrature for numerical integration, and a genetic algorithm – were chosen for the different dynamic imbalance factors they present. The experiments were carried out on a cluster of IBM PCs.

Section II describes the SAMBA framework. Next, Section III presents the classification criteria for load-balancing algorithms. SAMBA's load-balancing library is described in Section IV. Section V discusses the computational experiments. Finally, Section VI presents the related work and, in Section VII, we draw some conclusions.

II. Framework

The code of an SPMD application can typically be structured in three major components: (a) the single code which is replicated for the execution of a task (In this work, we use the term *task* to identify a subset of data which corresponds to a unit of work in an SPMD application. In this sense, a task does *not* correspond to a process.); (b) the load-balancing strategy; and (c) a skeleton (which initializes and terminates the application, manages the tasks, and controls the execution of the other parts). The first of these components is specific to each application, but the other two usually are not. SPMD applications can thus be modeled by a *framework* [GHJ+94], [Pre95], [Pre96], [RBP+91]. A framework acts as a specification that models applications with common structure and characteristics, and as a partial implementation that can be used as a basis for the development of these applications.

A framework differs from a program in that some of its parts are purposefully incomplete. These parts are called *hot spots*. The developer must fill in these hot spots to obtain a complete program. This process is known as *framework instantiation*. Thus, the typical user of a framework is an application developer.

Developing an application starting from a framework is essentially different from programming with libraries. When using a library, the programmer writes the main program and invokes the library functions as necessary. When using a framework, the main program is reused and the developer must code a number of functions that are activated by this program. Frameworks capture design decisions common to all applications of a given domain. In this way, they promote reusability to a high degree: not only parts of the code are reused, but also, and most importantly, the application architecture itself.

The framework proposed in this work will be described through the specification of classes and their relationship. We will use a graphical notation based on UML (Unified Modeling Language) [BR96].

A. Framework Definition

Figure 1 depicts the SAMBA framework. Methods that define hot spots in the framework have their names shifted to the left. Next, we discuss the main classes in the framework.

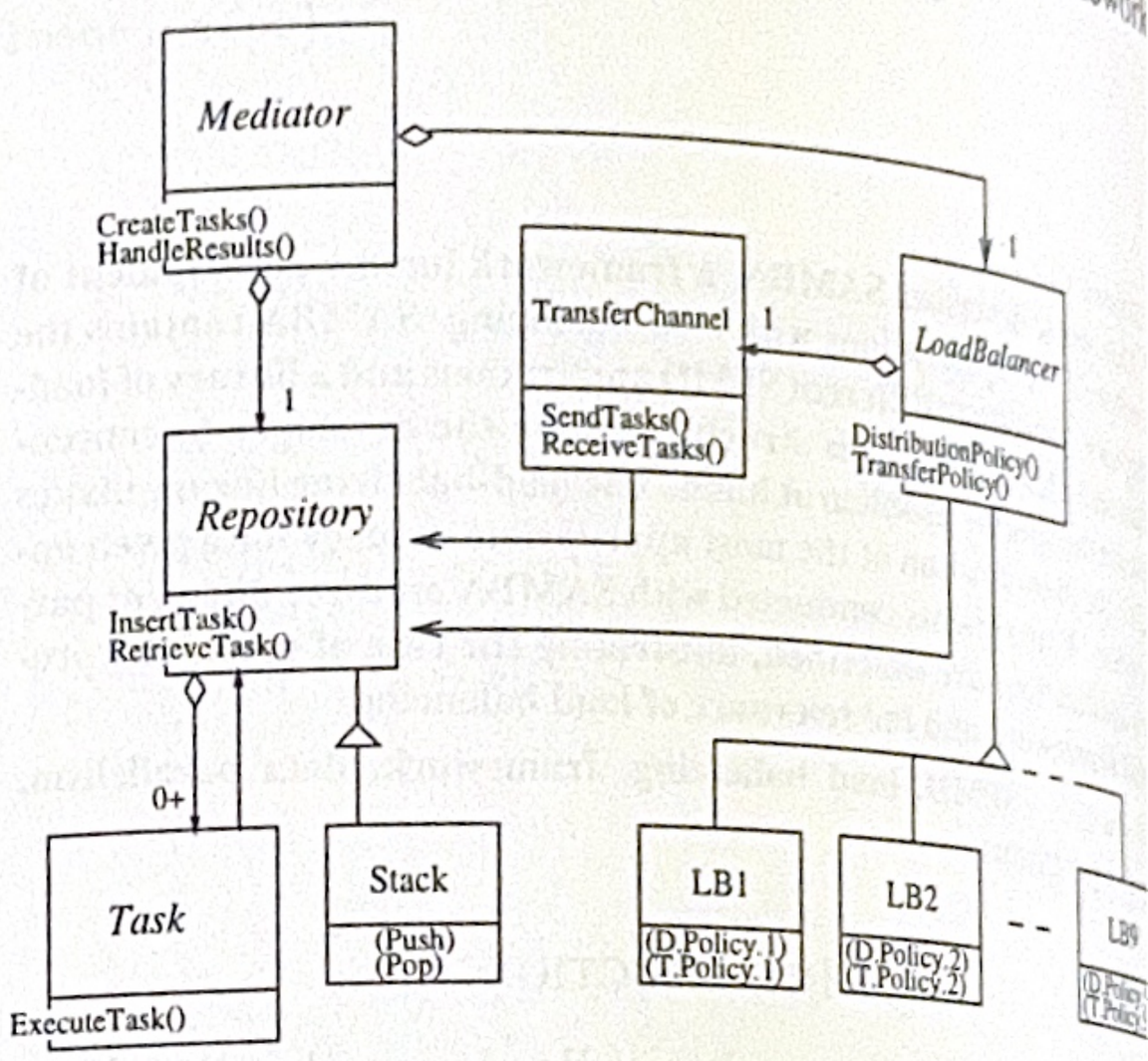

Fig. 1. Framework for the development of SPMD applications

Class *Mediator* defines the basic SPMD application structure. Its behavior is basically defined by two methods: a method that initially generates the tasks and the method that handles the results from the execution of these tasks. A *Mediator* object owns two other objects: a task repository and a load balancer.

Class *Repository* is responsible for managing a pool of tasks. Its behavior is defined by two methods, one for inserting and the other one for retrieving tasks.

Class *Task* represents a task of the application. Its main method is the one for executing a task – the single code. Objects in this class must have access to the repository, because the execution of a task may generate a new task to be inserted into it.

Class *LoadBalancer* is responsible for balancing the load among the processors. Its two main methods implement policies for distribution and transfer of tasks. Objects in this class own a transfer channel that is used for sending and receiving tasks. *LoadBalancer* objects must also access the repository in order to retrieve tasks that are to be executed.

Class *TransferChannel* provides two main methods that are used by the processsors for sending and receiving

Objects in this class must also access the repository to retrieve tasks that are to be sent and to insert tasks that are received.

Methods *ExecuteTask()* (the kernel of an SPMD application), *CreateTasks()*, and *HandleResults()* are specific to each application. These three methods must be implemented when a new SPMD application is created. Thus, these methods are hot spots in the SAMBA framework.

SAMBA also offers a load-balancing library. This library is described in the framework by defining methods *DistributionPolicy()* and *TransferPolicy()* as hot spots. Class *LoadBalancer* currently has nine subclasses, each of them implementing a different load-balancing strategy.

Because specific applications may benefit from specific organizations of the task repository, SAMBA allows the programmer to redefine the organization of the repository. In the framework, methods *InsertTask()* and *RetrieveTask()* are also defined as hot spots. The predefined repository organization (a stack) is a subclass of class *Repository*.

In the model we have just described, there is no reference to parallelism. In fact, each processor must instantiate this model. However, different processors may follow different paths in the control flow, selected through tests on variables that indicate the executing processor. This allows processors, for instance, to play different roles when executing *LoadBalancer* methods.

B. Object Interaction

The interaction diagram in Figure 2 illustrates the behavior of objects as execution of a SAMBA-derived application proceeds. This diagram depicts the interactions between objects along time [JCJ+92], [Boo94].

When execution begins, the *Mediator* object is instantiated, with the subsequent instantiation of the *Repository* and *LoadBalancer* objects. The *Mediator*'s first activity is to initiate task creation. In this process, the *Repository* is typically activated several times for task insertion. A *Task* object is instantiated (a new task is created) at each of these activations.

Next, object *LoadBalancer* is activated so as to execute its distribution policy. In the process of distributing tasks, the *LoadBalancer* object invokes the *TransferChannel* object to send and receive tasks. *TransferChannel* retrieves the tasks to be sent, and inserts the received ones, invoking the *Repository*.

According to its distribution policy, the *LoadBalancer* can also request the execution of tasks. In this case, it invokes the *Repository* to request a task and activates the retrieved *Task* requesting its execution. The execution of a task may at any point determine the insertion of new tasks in the *Repository*.

When execution of the distribution policy is completed, the *Mediator* activates *LoadBalancer* for the execution of the transfer policy. In this phase, as before, tasks can be sent, received, and retrieved for immediate execution.

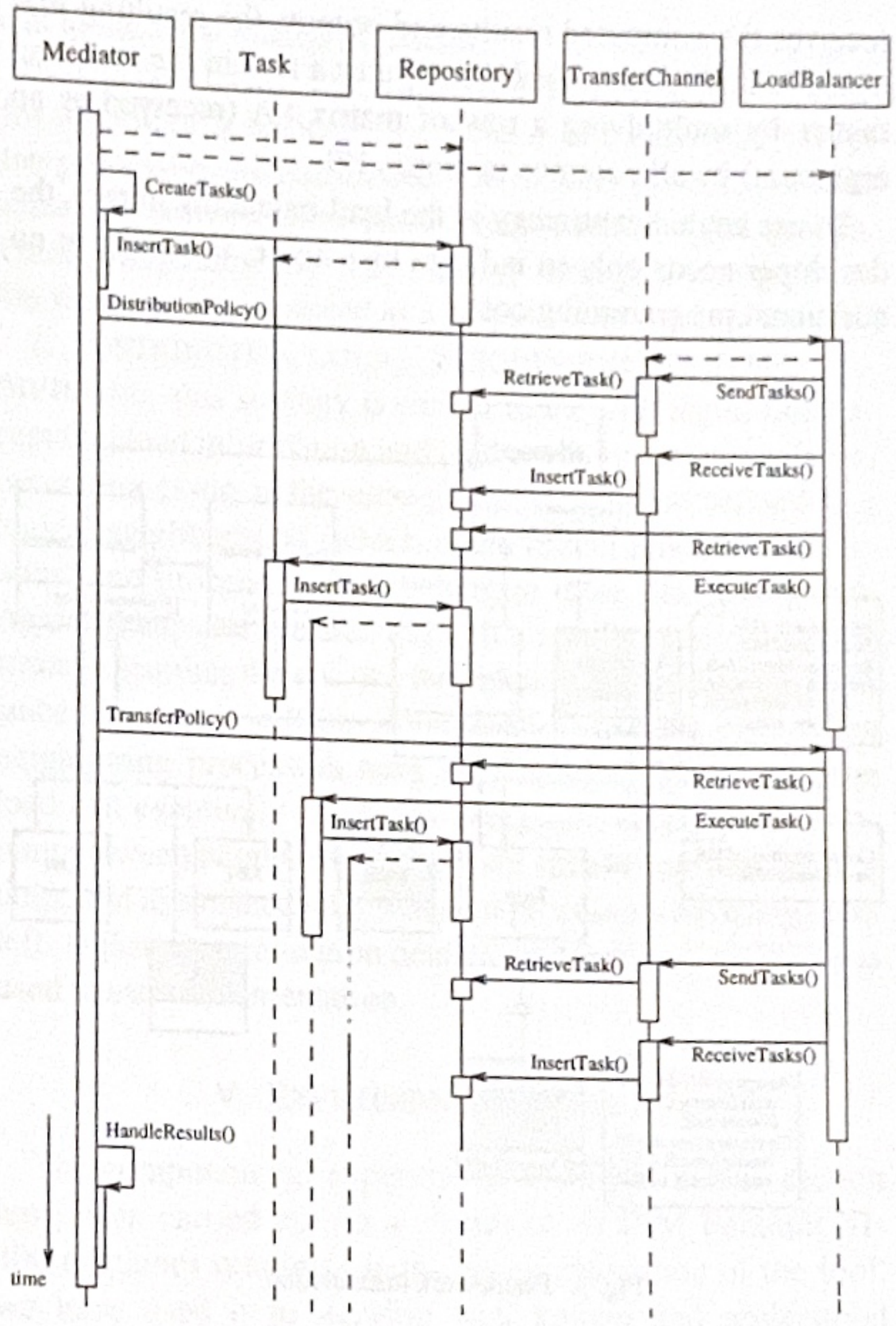

Fig. 2. Interaction diagram

Finally, the *Mediator* executes any necessary final actions, such as processing the results generated by the execution of the different tasks.

C. Framework Instantiation

To derive a complete application from a framework, the developer must code a method for each hot spot in the framework.

Figure 3 illustrates SAMBA's instantiation for parallel matrix multiplication. Subclasses that provide the code for the hot spots appear as shadowed objects in the figure. *LoadBalancer* is substituted for strategy $LB2$. Only one implementation is currently available for the *Repository* class, the one implementing a stack of tasks. Subclasses specific for the application at hand must be developed for classes *Mediator* and *Task*. The pseudo-code for the methods implementing hot spots in these classes is shown in the figure. Method *CreateTasks()* reads matrixes `MA` and `MB`, replicates matrix `MB` across all processors, and defines a task for each row in matrix `MA`. In method *HandleResults()*, a central processor

Objects in this class must also access the repository to retrieve tasks that are to be sent and to insert tasks that are received.

Methods *ExecuteTask()* (the kernel of an SPMD application), *CreateTasks()*, and *HandleResults()* are specific to each application. These three methods must be implemented when a new SPMD application is created. Thus, these methods are hot spots in the SAMBA framework.

SAMBA also offers a load-balancing library. This library is described in the framework by defining methods *DistributionPolicy()* and *TransferPolicy()* as hot spots. Class *LoadBalancer* currently has nine subclasses, each of them implementing a different load-balancing strategy.

Because specific applications may benefit from specific organizations of the task repository, SAMBA allows the programmer to redefine the organization of the repository. In the framework, methods *InsertTask()* and *RetrieveTask()* are also defined as hot spots. The predefined repository organization (a stack) is a subclass of class *Repository*.

In the model we have just described, there is no reference to parallelism. In fact, each processor must instantiate this model. However, different processors may follow different paths in the control flow, selected through tests on variables that indicate the executing processor. This allows processors, for instance, to play different roles when executing *LoadBalancer* methods.

B. Object Interaction

The interaction diagram in Figure 2 illustrates the behavior of objects as execution of a SAMBA-derived application proceeds. This diagram depicts the interactions between objects along time [JCJ+92], [Boo94].

When execution begins, the *Mediator* object is instantiated, with the subsequent instantiation of the *Repository* and *LoadBalancer* objects. The *Mediator*'s first activity is to initiate task creation. In this process, the *Repository* is typically activated several times for task insertion. A *Task* object is instantiated (a new task is created) at each of these activations.

Next, object *LoadBalancer* is activated so as to execute its distribution policy. In the process of distributing tasks, the *LoadBalancer* object invokes the *TransferChannel* object to send and receive tasks. *TransferChannel* retrieves the tasks to be sent, and inserts the received ones, invoking the *Repository*.

According to its distribution policy, the *LoadBalancer* can also request the execution of tasks. In this case, it invokes the *Repository* to request a task and activates the retrieved *Task* requesting its execution. The execution of a task may at any point determine the insertion of new tasks in the *Repository*.

When execution of the distribution policy is completed, the *Mediator* activates *LoadBalancer* for the execution of the transfer policy. In this phase, as before, tasks can be sent, received, and retrieved for immediate execution.

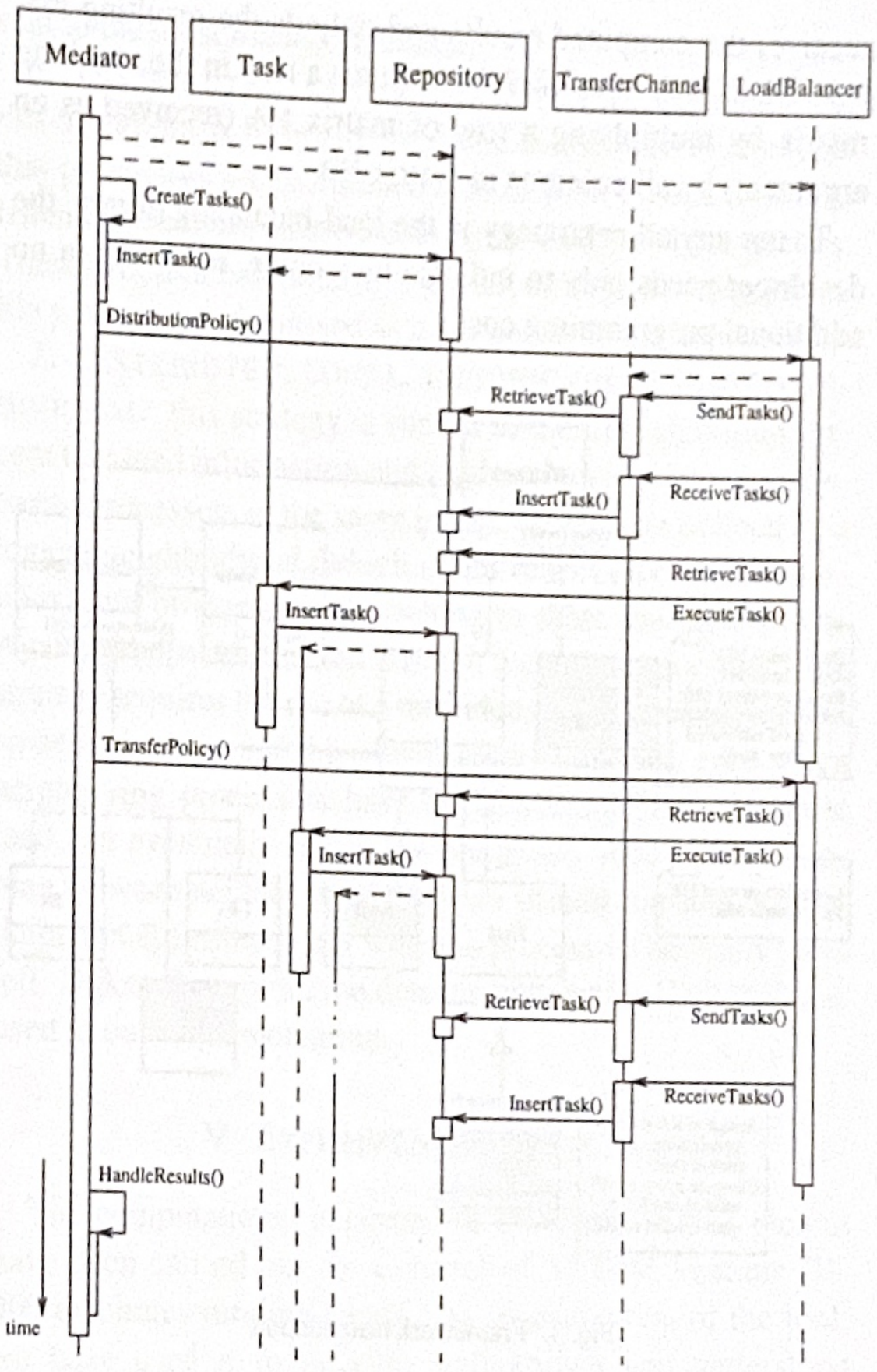

Fig. 2. Interaction diagram

Finally, the *Mediator* executes any necessary final actions, such as processing the results generated by the execution of the different tasks.

C. Framework Instantiation

To derive a complete application from a framework, the developer must code a method for each hot spot in the framework.

Figure 3 illustrates SAMBA's instantiation for parallel matrix multiplication. Subclasses that provide the code for the hot spots appear as shadowed objects in the figure. *LoadBalancer* is substituted for strategy $LB2$. Only one implementation is currently available for the *Repository* class, the one implementing a stack of tasks. Subclasses specific for the application at hand must be developed for classes *Mediator* and *Task*. The pseudo-code for the methods implementing hot spots in these classes is shown in the figure. Method *CreateTasks()* reads matrixes `MA` and `MB`, replicates matrix `MB` across all processors, and defines a task for each row in matrix `MA`. In method *HandleResults()*, a central processor

receives the computed results and outputs the resulting matrix. Method *ExecuteTask()* computes a row in the resulting matrix by multiplying a row of matrix MA (received as an argument) by all columns in matrix MB.

To use any other strategy in the load-balancing library, the developer needs only to indicate his choice, incurring in no additional programming cost.

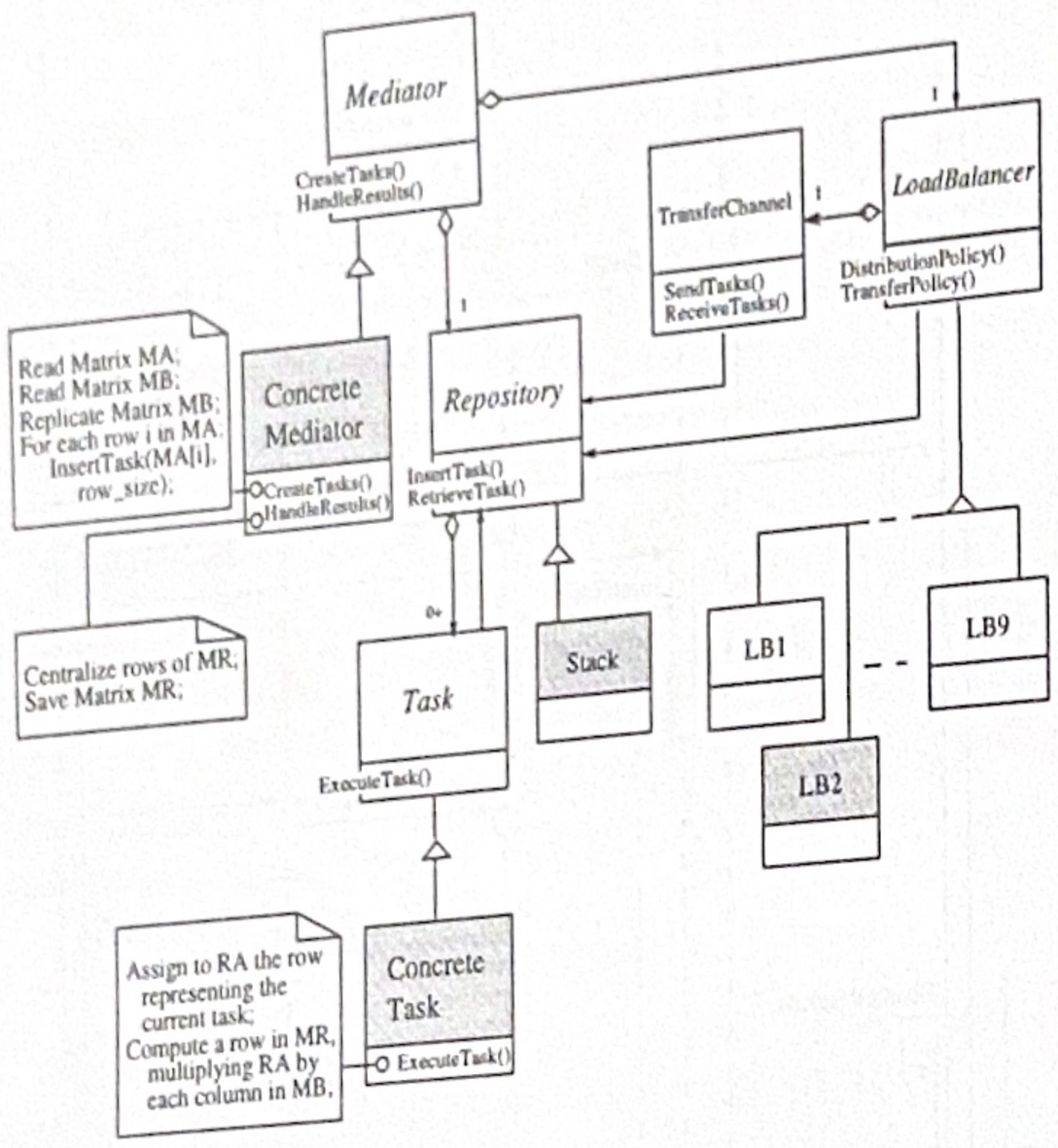

Fig. 3. Framework instantiation.

III. Classification Criteria

This section briefly describes classification criteria for load-balancing algorithms for SPMD applications. [PRR98], [Pla00] discusses these criteria in detail, using examples from the SPMD load-balancing literature to illustrate each class.

The first classification criterion is based on the way and moment tasks are distributed among the processors. *Static* strategies distribute tasks before execution. In contrast, in the case of *dynamic* strategies, tasks can be distributed or transferred among the processors along the application execution, so as to compensate dynamic factors which lead to load imbalance.

In the case of *demand*-driven dynamic strategies, distribution is performed along the execution of the application, according to the behavior of the processors. Initially, a master processor distributes a block of tasks to each processor. Whenever a processor finishes the execution of the current block, it asks for and receives a new block of tasks. This procedure goes on until the master terminates the distribution of all tasks. In *transfer*-based dynamic strategies, all tasks are distributed among the processors before execution. Tasks are redistributed during execution, being transferred from overloaded processors to underloaded ones.

Variation and heterogeneity of external load in different processors represent an important load-imbalance factor. Strategies are also classified as either *integrated* or *isolated*, according to whether or not they use information about the load external to the application for balancing decisions.

In some transfer-based strategies, load transfers may occur between any two processors, while in others they are restricted to processors within pre-defined groups. Such strategies are referenced as *global* and *local*, respectively.

In local *partitioned* algorithms, each processor belongs to exactly one group. In *neighborhood*-based local algorithms, groups are defined by processor neighborhoods associated with the physical topology of the interconnection network or with the logical topology of the aplication.

In transfer-based *centralized* algorithms, only a single processor performs load balancing. This processor computes the necessary reassignments and informs the other processors involved of the necessary task transfers. A transfer-based *distributed* algorithm runs locally within each processor, with its code replicated in all processors. Each processor defines and performs by itself the transfers in which it is involved.

The execution of distributed *synchronous* algorithms is characterized by the fact that all processors perform the balancing procedures at the same time. In the case of an *asynchronous* algorithm, each processor runs the balancing algorithm independently, without any interprocessor synchronism.

Transfer-based algorithms can also be classified according to their pattern of activation. In the case of *periodic* strategies, load balancing is activated from time to time, after intervals which may be defined by elapsed times or by a number of executed operations. In the case of *event*-oriented strategies, a load-balancing algorithm is activated by the occurrence of some pre-defined action or condition, such as the reduction of its load to a level below a certain threshold.

The goal of transfer-based *collective* algorithms consists in balancing the load of several processors, while transfer-based *individual* algorithms aim to resolve underloading or overloading at a single processor.

Individual *receiving* strategies try to eliminate an underload condition by identifying overloaded processors from which the underloaded processor can receive tasks. On the other hand, *sending* strategies focus on an overloaded processor and try to identify underloaded processors to which tasks can be sent from the overloaded processor.

Non-blind individual algorithms use load information from recipient or sender candidate processors. *Blind* algorithms, on the contrary, do not take this kind of information into account. In some cases, the processors involved in each load transfer are selected randomly or through some round-robin scheme.

IV. Load Balancing in SAMBA

The design of SAMBA's load-balancing library was guided by the classification criteria of load-balancing algorithms for SPMD applications described in Section III. This ensured the inclusion of load-balancing algorithms with a wide variety of characteristics. In this section, we describe seven implemented algorithms. Two other variants will be discussed later.

1. STATIC: the central processor distributes the initial set of tasks in equal parts among all processors, including itself. Each processor simply executes the tasks it has received, without any dynamic load balancing.

2. DEMAND-DRIVEN: a central processor allocates an initial set of tasks to each other processor. Each time a processor finishes the execution of a block of tasks, it requests and receives a new one. The number of tasks in a block is given as a parameter in the activation of the application.

3. DISTRIBUTED, GLOBAL, COLLECTIVE: a central processor distributes the initial set of tasks in equal parts among all processors, including itself. Whenever a processor consumes all of its tasks, it sends a message to all others, asking them to execute a load-balancing step. At this point, each processor sends to all others its *internal load index*, i.e., the remaining number of tasks to be executed. Next, each processor checks whether load balancing is to take place or not. Load balancing takes place only if the load index of at least one processor exceeds a certain threshold, which is given as an argument in the activation of the application. In this case, each processor computes the necessary task exchanges and performs the exchanges in which it is involved.

4. CENTRALIZED, GLOBAL, COLLECTIVE: this algorithm is similar to the previous one, except that all decisions are taken by a central processor. Whenever a processor consumes all of its tasks, it sends a message to all other processors asking them to send their internal load index to a central processor. After receiving this information from all processors, this central processor checks the need for load balancing, in the same way as before. If load is to be balanced, the central processor defines the necessary exchanges, and informs each processor of the task transfers in which it is involved. Finally, each processor executes the indicated transfers.

5. DISTRIBUTED, GLOBAL, INDIVIDUAL: once again, load balancing is triggered by the end of the tasks to be executed at a processor. However, here the goal is to correct an underload condition at this single (individual) processor. Whenever a processor consumes all of its tasks, it sends a message to all others and they send back their internal load indices. After receiving load information from all processors, the underloaded processor checks whether load balancing is to take place or not, in the same way as in the third algorithm. If load balancing is necessary, a request for load transfer is sent to the most loaded processor.

6. DISTRIBUTED, LOCAL, PARTITIONED, COLLECTIVE: this algorithm is basically the same as the third one, except that processors are partitioned into disjoint groups. All load balancing activity takes place inside each of these groups. These groups are defined by the user, through a configuration file whose name is passed as a parameter to the application.

7. DISTRIBUTED, LOCAL, NEIGHBORHOOD-BASED, INDIVIDUAL: this strategy is similar to the fifth algorithm, except that load information and load exchanges occur only between processors in the same group. Groups are defined by a logical neighborhood (which could reflect physical connections) and processors may belong to more than one group. Again, groups are defined through a configuration file. This strategy requires the use of a termination detection algorithm, since there may still be global load to execute even when neighboring processors have no remaining load. Because load can eventually reach any processor, successively hopping between neighbors, a processor should consider his participation as finished only when no processors have any tasks left. Dijkstra's termination detection algorithm [DSG83] was used in our implementation.

V. Experimental Results

The computational experiments described in this section have been carried out on a cluster of 32 IBM Pentium III-400 machines running Linux. As an evaluation of the tool, we have used it to develop well known and understood SPMD applications and to analyse load-balancing strategies for them. In this section we report the results obtained with three applications: computation of $\int_0^{15} e^x dx$ by adaptive quadrature with a tolerance of 10^{-16} (application #1), computation of the product of two integer valued square matrices of dimension 2 000 (application #2), and a parallel genetic algorithm (application #3).

The choice of these applications was due to the different imbalance factors they present and to their different task creation mechanisms. In application #1, tasks are dynamically generated, as the outcome of the execution of other previously created tasks. The total number of tasks is initially unknown. Differently, in application #2 the number of tasks is known beforehand and all tasks have the same complexity. Finally, in application #3, tasks may present quite different computational loads.

Seven load-balancing strategies available with SAMBA have been tested and compared: static (S1), demand-driven (S2), distributed (S3), centralized (S4), individual (S5), partitioned (S6), and neighborhood-based (S7). For the last strategy, two logical topologies have been evaluated: ring (S7r) and hypercube (S7h). Each group of the locally partitioned strategy (S6) was, in this analysis, formed by exactly four processors.

Application #1 computes $\int_0^{15} e^x dx$ using adaptive quadrature. Numeric quadrature computes the definite integral of a function over an interval by dividing the interval into subintervals and approximating the area of each subinterval using the trapezoidal rule. The approximations computed for the subintervals are then added to obtain the approximation of the total area. The creation of subintervals is dynamically determined. The approximation of the area defined by an interval is compared to the sum of the two areas corresponding to the two subintervals obtained by dividing the whole interval in the middle. If the difference is small enough, the approximation is returned. Otherwise, recursive (and parallel) computation proceeds over the two subintervals.

In our experiment, the original integration interval was initially divided into 64 subintervals, generating 64 tasks. The computational effort of each of the original 64 tasks is not the same, as far as there are different levels of recursion for each subinterval. Table I shows elapsed times in seconds for different load-balancing strategies, using 4, 8, 16, and 32 processors. The experiment was performed in exclusive mode (i.e., with no other applications running on the cluster), so as that the only imbalance factor was the dynamic generation of tasks along the execution of the application. The demand-driven strategy (S2) led to the best results for this application, independently of the number of processors. The poor behavior of dynamic strategies based on transfer of tasks (S3-7) seems to be natural for small granularity applications with a large number of dynamically generated tasks (total of 110 933 464). To illustrate this point, we notice that the load-balancing strategy was activated 548 times along the execution using the distributed strategy (S3) and four processors. Also, the very bad behavior of the locally partitioned strategy (S6) is due to the fact that different processor groups receive initial subsets of tasks with very different computational demands, but are not able to redistribute them to other groups.

TABLE I

APPLICATION #1 USING 4, 8, 16, AND 32 PROCESSORS

	S1	S2	S3	S4	S5	S6	S7r	S7h
4	248	114	126	132	115	—	117	—
8	162	49	82	82	74	122	91	81
16	103	23	49	51	56	88	68	58
32	52	13	32	32	39	53	49	36

Application #2 computes the product of two integer valued square matrices of dimension 2 000. The basic parallel multiplication algorithm is used. The second (right) matrix in the multiplication is replicated across all processors. Each task is described by a row of the first (left) matrix and corresponds to the computation of one of the 2 000 rows of the resulting matrix. All tasks in this application present roughly the same computational demand. The total number of tasks is known beforehand and equal to the dimension of the matrices. The only imbalance factor is the external load.

External load was simulated by load processes which continuously computed arithmetic expressions. The simulated external load and the matrix multiplication were the only processes running in the controlled environment. Out of each group of four processors, the first processor had no external load, the second one executed a single load process, the third one executed two processes, and, finally, the fourth executed three processes.

Table II presents elapsed times in seconds for application #2 with different load-balancing strategies, using 4, 8, 16, and 32 processors. In contrast with the previous application, the total number of tasks is relatively small – 2 000. With four and eight processors, the algorithms based on task transfer (S3-7) presented comparable results, and were more appropriate than the demand-driven algorithm (S2). This possibly occurs due to the fact that, with only a few processors, the communication overhead imposed by the exchange of load indexes is low. On the other hand, in the demand-driven strategy, one processor is dedicated to task distribution and does not contribute to the computation itself. With only a few processors, this represents a heavy loss in computing power.

TABLE II

APPLICATION #2 USING 4, 8, 16, AND 32 PROCESSORS

	S1	S2	S3	S4	S5	S6	S7r	S7h
4	762	496	391	383	379	—	386	—
8	410	261	247	252	244	252	245	260
16	265	168	189	190	196	177	191	195
32	182	128	171	163	175	145	153	166

As the number of processors grows, the demand-driven strategy is again the one which achieves the best results. However, it is interesting to note the relatively good performance of the partitioned strategy (S6). In this experiment, the same load-imbalance pattern was generated inside each group. In this setting, balancing in the scope of a group can attain the same balancing results as a global strategy and implies less communication overhead. The comparison of strategies S3 (distributed) and S6 (partitioned) thus reflects the communication overhead due to a totally distributed computation.

In the experiment reported in Table II, the balancing algorithms did not explicitly take the external load into consideration. Indeed, most parallel environments are not dedicated, i.e., several concurrent applications compete for existing computational resources. We modified algorithms S3 and S6 to take this competition into account, originating strate-

gies S8 and S9, respectively. We used the number of executed tasks per unit of time as the external load index. When all tasks present the same computational requirements, this index reflects the computational capacity of each processor, determined by the amount of external load of each processor.

Table III presents elapsed times in seconds for application #2 with the two new strategies. External load was simulated in the same way as before. Elapsed times for strategies S3 and S6 are repeated for better understanding. The gain in performance obtained with the new algorithms is still very small. However, we believe this gain could be higher if the observed "ping–pong" effect was eliminated. When a processor is handling little external load, there is a tendency for too many processors to send their load to it, causing the previously underloaded processor to become a bottleneck in the system, in which case eventually it will have to send back part of the load it received.

TABLE III
LOAD-BALANCING STRATEGIES USING EXTERNAL LOAD INDEXES

	S3	S8	S6	S9
4	391	374	—	—
8	247	241	252	248
16	189	187	177	171
32	171	166	145	140

The experiment described next was carried out by integrating an existing iterative application within SAMBA. Application #3 is a variant of a parallel genetic algorithm (GA) originally developed to find good solutions to the optimization of oil fields [And97]. Each iteration of a GA handles a population of individuals. First, each individual in the population is evaluated in terms of a fitness function. Next, a selection procedure selects the best individuals for mating. After reproduction (crossover), some individuals are subject to a mutation process. The fittest individuals from the current generation survive to form the population of the next generation. In this experiment, only the phase of fitness evaluation is performed in parallel. The individuals in the population are distributed to the processors and their fitness values are computed. Once all individuals have been evaluated in parallel, the next phases in the same iteration are performed in sequential mode.

Each SPMD task in application #3 corresponds to the computation of the fitness value for some individual. The only imbalance factor in this application is the variation in the time needed for the evaluation of each individual. The number of tasks is equal to the number of individuals in the population, which was set at 1 000. Table IV presents results for seven load-balancing algorithms, depicting elapsed times in seconds observed for one single iteration of the GA using 4, 8, 16, and 32 processors, with the times needed for fitness evaluation ranging from 1 to 5 seconds for each individual.

TABLE IV
APPLICATION #3 USING 4, 8, 16, AND 32 PROCESSORS

	S1+	S1-	S2	S3	S4	S5	S6	S7r	S7h
4	733	708	955	721	721	722	—	721	—
8	387	345	410	366	370	376	368	370	370
16	202	170	194	196	189	204	188	195	195
32	107	74	96	106	109	107	98	106	107

The first two columns in Table IV give the elapsed times of the slowest (S1+) and fastest (S1-) processors using the static balancing algorithm, which illustrate the maximum possible gains which could be achieved by more effective load-balancing strategies. With only four processors, the demand driven strategy (S2) showed the worst performance, since one processor does not take part in the computations. The strategies based on task transfers (S3-7) achieve similar performances. The reduction in elapsed time observed for the best dynamic strategies is small, since the imbalance observed for the static strategy S1 itself was already small. However, this changes when the number of processors increases up to 32. In this case, load imbalance incurred by the static algorithm increases and the demand driven strategy (S2) presents the best results, leading to a gain of 10% in the elapsed time with respect to the static algorithm.

The first part of Table V illustrates the load imbalance observed for the static algorithm using 4, 8, 16, and 32 processors. Intervals in this table represent the variation in the time needed for the evaluation of each individual (in seconds). We note that load imbalance increases with both the upper limit of these intervals and the number of processors. The second part of this same table gives the gains (percentual reductions in elapsed times) observed for the best algorithm in each case. We note that the largest gains in elapsed times are observed exactly for the situations with the largest load-imbalance indices, illustrating the importance of choosing an adequate load-balancing strategy.

TABLE V
LOAD IMBALANCING AND GAINS

	Imbalancing			Gains		
	[1,5]	[1,10]	[1,20]	[1,5]	[1,10]	[1,20]
4	3%	3%	3%	2%	2%	2%
8	11%	12%	14%	5%	6%	6%
16	16%	18%	20%	7%	6%	7%
32	31%	35%	37%	10%	13%	14%

VI. Related Work

A large number of tools and environments that support the development of parallel applications have been proposed and implemented [CJ99], [Dec97], [FMD98], [Wei99].

[CJ99] presents the PMESC programming tools. PMESC is a library that offers routines typically used in SPMD applications. SAMBA allows the resuse not only of specific routines, but of the whole SPMD structure. Also, one of SAMBA's main goals is to support the choice of an appropriate load-balancing strategy. PMESC does not deal with this issue.

The VDS library [Dec97] offers mechanisms for the distribution of tasks, with the goal of balancing the load among the participating processors in a parallel application. However, the library does not offer alternative strategies. Once the type of application to be developed is defined, VDS adopts one out of two available strategies.

[FMD98] describes a theoretical analysis of the behavior of seven load-balancing algorithms for data-parallel applications. The goal of this analysis is to define criteria for evaluating and comparing these strategies. A cost and quality estimation model is derived for each of the algorithms. These models can be useful for parallel programmers, as well as for parallel programming language compilers.

The Prophet system [Wei99] tries to minimize total execution time by scheduling the tasks of an SPMD applications among heterogeneous processors in a network of workstations. Prophet assigns load to each processor according to its processing capacity, but does not attempt to balance load after the execution of tasks has begun.

VII. Final Remarks

In light of the discussion of related work in the previous section, SAMBA's main contributions to the support of SPMD applications can be summarized as follows. As a framework, SAMBA supports the development of SPMD applications by allowing reuse both of routines typically used in such applications and of the application architecture itself. Also, using SAMBA, the programmer can easily experiment with different load-balancing strategies when developing new applications. SAMBA's load-balancing library contains a significant number of load-balancing algorithms, covering a wide range of strategies. The computational experiments have shown the importance of using an appropriate load-balancing algorithm and the associated reductions that can be achieved in elapsed times. They also illustrate that the most suitable load-balancing strategy may vary with the type of application and with the number of available processors. From this point of view, the facilities offered by SAMBA in terms of load-balancing play also an important role in the development of efficient parallel applications.

References

[AM96] J.N.C. Árabe and C.D. Murta, "Auto-balanceamento de carga em programas paralelos", *Proc. of the VIII Brazilian Symp. of Computer Architecture and High Performance Processing*, 1996, 161–171.

[And97] A.C.B. de Andrade Filho, *Optimizing hydrocarbon field development using a genetic algorithm based approach*, Ph.D. Dissertation, Dept. of Petroleum Eng., Stanford University, 1997.

[Boo94] G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamim/Cummings, 1994.

[BR96] G. Booch and J. Rumbaugh, *Unified Method*, Rational Software Corporation, 1996.

[CJ99] S. Crivelli and E.R. Jessup, "The PMESC programming library for distributed-memory MIMD computers", *Journal of Parallel and Distributed Computing* 57 (1999), 295–321.

[Dec97] T. Decker, "Virtual Data Space - A universal load balancing scheme", *LNCS* 1253 (1997), 159–166.

[DSG83] E. Dijkstra, W. Seijen, and A. Gasteren, "Derivation of a termination detection algorithm for a distributed computation", *Information Processing Letters* 16 (1983), 217–219.

[FG96] M.A. Franklin and V. Govindan, "A general matrix iterative model for dynamic load balancing", *Parallel Computing* 22 (1996), 969–989.

[FMD98] C. Fonlupt, P. Marquet, and J. Dekeyser, "Data-parallel load balancing strategies", *Parallel Computing* 24 (1998), 1665–1684.

[FTI90] M. Furuichi, K. Taki, and N. Ichiyoshi, "A multi-level load balancing scheme for or-parallel exhaustive search programs on the multi-psi", *Proc. of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1990, 50–59.

[GHJ+94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design pattern - Elements of reusable object ooriented software*, Addison-Wesley, 1994.

[JCJ+92] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-oriented software engineering - A use case driven approach*, Addison-Wesley, 1992.

[LPR97] S. Lifschitz, A. Plastino, and C.C. Ribeiro, "Exploring load balancing in parallel processing of recursive queries", *LNCS* 1300 (1997), 1125–1129.

[Mat96] T.G. Mattson, "Scientific computation", em *Parallel and Distributed Computing Handbook* (A.Y. Zomaya, editor), 981–1002, McGraw-Hill, 1996.

[MPI93] Message Passing Interface Forum, "MPI: A message passing interface", *Proceedings of Supercomputing'93* (A.Y. Zomaya, editor), 878–883, IEEE Computer Society, 1993.

[Pre95] W. Pree, *Design patterns for object-oriented software development*, Addison-Wesley, 1995.

[Pre96] W. Pree, *Framework patterns*, SIGS Management Briefings, SIG Books & Multimedia, 1996.

[Pla00] A. Plastino, *Balanceamento de carga de aplicações paralelas SPMD*, Doctorate thesis, Departament of Computing, Catholic University of Rio de Janeiro, 2000.

[PRR98] A. Plastino, C.C. Ribeiro, and N. Rodriguez, "Uma taxonomia de algoritmos de balanceamento de carga para aplicações SPMD", Technical Report PUC-RioInf.MCC49/98, Department of Computing, Catholic University of Rio de Janeiro, 1998.

[PRR99] A. Plastino, C.C. Ribeiro, and N. Rodriguez, "A tool for SPMD application development with support for load balancing", *Proc. of the ParCo Parallel Computing Conf.*, Delft, 1999, to appear.

[Qui94] M.J. Quinn, *Parallel Computing: Theory and Practice*, McGraw-Hill, 1994.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object oriented modeling and design*, Prentice-Hall, 1991.

[Wei99] J.B. Weissman, "Prophet: Automated scheduling of SPMD programs in workstation networks", *Concurrency: Practice and Experience* 11 (1999), 301–321.

[WLR93] M.A. Willebeek-LeMair and A.P. Reeves, "Strategies for dynamic load balancing on highly parallel computers", *IEEE Trans. on Parallel and Distributed Systems* 4 (1993), 979–993.

[ZLP97] M.J. Zaki, W. Li, and S. Parthasarathy, "Customized dynamic load balancing for a network of workstations", *Journal of Parallel and Distributed Computing* 43 (1997), 156–162.