Contents

Chapter 1

Load Sharing and Fault Tolerance Manager

BERTIL FOLLIOT AND PIERRE SENS

Laboratoire d'Informatique de Paris 6 / CNRS
University of Paris VI, Paris, France

Email: *folliot@src.lip6.fr, sens@src.lip6.fr*

1.1 Introduction

The increasing need for processing power for scientific calculations combined with the large development of distributed information systems makes the computation distribution over workstation networks very attractive. A large range of studies has shown that the workstations are idle from 33 % to 78 % of the time. A similar study in one of our campus facility (18 workstations) over a six-month period shows that the workstations were free 69 % of the time and the processors were free 93 % of the time. Consequently, to achieve high performance the system has to provide an efficient mechanism for program allocation. Two load sharing policies may be applied to react to dynamic system change: dynamic placement, that allocates programs according to the current system state, and migration, that moves processes according to system and application evolution.

In spite of the advantage of process allocation, only a few systems have implemented load balancing strategies [3] and most of the proposed algorithms use only simulation results indicating their efficiency. In such systems, processes are initially allocated on a set of workstations according to multi-criteria placement algorithms taking into account the host loads and the user activities. These algorithms use mechanisms for system state monitoring and application executions supervising. Based on this knowledge, the allocation policies allow the improvement of application response time as well as the effective use of the distributed system resources. A migration facility that moves processes from one workstation to another one usually copes with three main cases: a user comes back to his/her workstation after an initial placement decision has been done, a host becomes overloaded and there exists an underloaded host in the system, a fast host is underloaded and a slow one is loaded.

Load sharing is dedicated to long-lived application. For this kind of application, fault tolerance becomes an essential component. Many hours of computation can be lost not only if a hardware failure occurs, but also if one of the processors is rebooted, turned off or disconnected from the network. As a consequence, developing parallel programs also requires expertise in fault tolerance. To release application designers from this task, systems must include fault tolerant facility where processes affected by a crash are automatically recovered on a new set of workstations.

The main scope of this chapter is to describe the interaction between load balancing and fault tolerance in a cluster of workstations and to show that the combination of the two facilities can outperform reliable parallel applications.

This chapter is organized as follows: Sections 1.2 and 1.3 describe usual techniques to balance load between hosts and to tolerate host failures. Then, Section 1.4 presents results of an experimental platform which integrates both facilities. Finally, Section 1.5 outlines some related works in fault tolerant and load sharing management.

1.2 Load Sharing in Cluster Computing

Supports for parallel programming over workstations network often ignore programs' needs, real load conditions and users' activities (as PVM [9]). Some degree of load sharing has been adapted in existing packages such as lsPVM [14] or Dynamic PVM [5]. For better usage of unused hosts, several different policies and systems have been realized to balance the load [3], [11], [14]. However, there are few published methods that allow programmers to specify parallel application in a load sharing perspective. Indeed, optimum load sharing strategies are very hard to implement and are well-known as NP-complete, even in simple cases. The main reasons are:

1. It is impossible to get an exact knowledge of the global state of the system at a given instant.
2. The execution of complex allocation algorithms may represent a heavy burden.
3. The instability behavior of allocation algorithms is due to inaccurate information.

Parallel program placement has been widely studied. A number of works concentrated on static allocation, adapted for multiprocessor systems (dedicated environment) without interference between users or concurrent applications. A well-known parallel application environment is the PVM system. It uses a user-managed host file indicating a list of hosts to run an application. This scheme is adequate for a dedicated environment, but may lead to over-utilization of resources in a local area networks of workstations, if different PVM users use overlapping sets of hosts. It may also lead to unacceptable interference with hosts supporting interactive users.

Concerning dynamic load sharing in a local area network of workstations, most of the works deal only with the processor load and are not adapted for the allocation of entire parallel applications. Some other works, such as Utopia [14] or Mosix [3], consider the program needs other than CPU, but do not offer a parallel programming environment : naming, communication, attraction/repulsion between components. These works are divided in two groups:

- dynamic placement (as Rem [13], Utopia [14]) where processes are allocated at start-up and stay on the same location, and
- process migration (as Condor [11], Mosix [3], Sprite [6]) where processes can move according to overload conditions or the reactivation of workstations by their owners.

1.3 Fault Tolerance by Means of Checkpointing

Checkpointing and rollback recovery are well-known to provide fault tolerance in distributed systems. This section reviews the common techniques.

1.3.1 Checkpointing a Single Process

The checkpoint of a single process is a snapshot of the process's address space at a given time. To reduce the cost of checkpointing, two complementary techniques can be applied: incremental and non-blocking checkpointing [8].

Incremental methods reduce the amount of data that must be written. Only those pages of the address space modified since the last checkpoint are written to stable storage.

Non-blocking methods allow the process to continue executing while its checkpoint is written to stable storage. However, if the process modifies any of its pages during the checkpoint, the resulting checkpoint may not represent the real state of the process. The internal copy-on-write memory protection may be used to protect pages during the checkpoint. At the start of the checkpoint, the pages to be written are write-protected. After writing each page to stable storage, the checkpoint manager removes the protection from the page. If a process attempts to modify a protected page, the page is copied to a newly allocated page, and the protection of the original page is removed. The newly allocated page is not accessible to the process. Only the checkpoint manager uses it to finish the checkpoint.

These complementary techniques improve significantly the response time: between 3.4 times and 4.7 times faster checkpointing, according to experimental results presented by [8].

1.3.2 Checkpointing of Communicating Processes

When processes exchange messages, the simple approach to recovery for independent processes is no longer adequate. To recover from a fault, the execution must be rolled back to a consistent global state The rolling back of one process could result in an avalanche of rollbacks of other processes before a consistent state is found. For example, when the sender of a message m is rolled back, m is recorded as "received but not yet sent." Message m becomes an *orphan message* and the system is inconsistent. To discard this orphan message, the corresponding receiver must also be rolled back. This domino effect is illustrated in Figure 1.1.

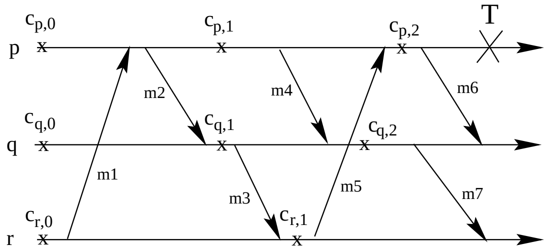

Figure 1.1 Domino effect.

The Xs indicate checkpoints and the arrows represent the messages. If p fails at time T, it would be rolled back to $Cp, 2$. Since message $m6$ becomes an orphan, q must be rolled back to $Cq, 2$. Then $m7$ becomes an orphan and r must be rolled back to $Cr, 1$ and so on. Finally, to recover from the failure of p, all processes must be rolled back to their initial checkpoints.

A substantial body of work has been published regarding fault tolerance by means of checkpointing. The main issues that have been covered are limiting the number of hosts that have to participate in taking the checkpoint or in rolling back [10], reducing the number of messages required to synchronize a checkpoint, or using message logging [8]. Checkpointing techniques can be classified into two categories: coordinated and independent checkpointing.

Coordinated checkpointing. With coordinated checkpointing, processes coordinate their checkpointing actions such that the collection of checkpoints represents a consistent state of the whole system. When a failure occurs, the system restarts from these checkpoints. Looking at the results of [8], the main drawback of this approach is that the messages used for synchronizing a checkpoint are an important source of overhead. Moreover, after a failure, surviving processes may have to rollback to their latest checkpoint in order to remain consistent with recovering processes. Alternatively, analyzing the interactions between processes can reduce the number of processes to rollback [10].

Independent checkpointing. In independent checkpointing, each process independently saves its state without any synchronization with the others. Message logging was introduced to avoid the domino effect [8]. Logging methods fall into two classes: pessimistic and optimistic. In pessimistic message logging, the system writes incoming messages to stable storage before delivering them to the application [12]. To restart, the failing process is rolled back to the last checkpoint and replies to outgoing messages are returned immediately from the log. The receiver is blocked until the message is logged on stable storage. Alternatively, the system delivers the message to the application immediately but disallows the sending of further messages until the message has been logged. In optimistic message logging, messages are tagged with information that allows the system to keep track of inter-process dependencies [8]. Inter-process dependency information is used to determine which processes need to be rolled back on restart. This approach reduces the logging overhead, but processes that survive a failure may still be rolled back. Alternatively, messages are gathered in the main memory of the sending host and are asynchronously saved to stable storage. The benefits listed above are obtained at the expense of the space and time required for logging messages. The space overhead is reasonable, given the current large disk capacities. Furthermore, at each new checkpoint all messages are deleted from the associated backup (a log is completely deleted after each checkpoint). The main drawback is the input/output overhead (i.e., the latency in accessing the stable storage).

1.4 Integration of Load Sharing and Fault Tolerance

This section presents and discusses a realization of a fault tolerant load sharing facility, GATOSTAR, that we developed at University of Paris 6. It is based on the implementation of two applications : GATOS and STAR. GATOS is a load sharing manager which automatically distributes parallel applications among hosts according to multicriteria algorithms. STAR is a software fault manager which automatically recovers processes affected by host failures.

1.4.1 Environment and Architecture

Unix is becoming a standard in networked environments. Granted, applications developed above an unmodified Unix version are portable at very low cost and can easily be distributed. These reasons led us to build GATOSTAR on top of Unix without any kernel modification.

System Environment

GATOSTAR relies on a BSD Unix operating system (SunOS). It works on a set of heterogeneous workstations connected by a local area network. We define migration domains of compatible hosts, where processes can freely move. In this environment, failures are uncommon events. Clark and McMillin measured the average crash time in a local area network to be once every 2.7 days [4] for a local area network of 10 workstations. Therefore, we favor solutions with a low overhead under normal operation, possibly to the detriment of an increase in recovery time. We consider that the system is composed of fail-silent processors, where the faulty nodes simply stop and the remote nodes are not notified. We do not consider the network partition problem.

Application Model

We have adopted the classic application model where an application is a set of communicating processes connected by a precedence graph. The graph contains all qualitative information needed to execute the application and helpful for the load sharing manager: precedences and communications between processes, and files in use. The application programmer can provide this information both by an execution graph (a file containing the graph description using a parallel application grammar), and/or by dynamic operations that allow, while running, to modify the execution graph. We are interested in applications running for extended periods of time, such as high number factoring, VLSI applications, image processing. This kind of application may be executing for hours, days or weeks. In that case, the failure probability becomes significant, and load sharing and the need for reliability are important concerns.

GATOSTAR Architecture

GATOSTAR architecture consists of a ring of hosts which exchange information about hosts' functioning and processes' execution. Having this type of information available on-line allows us to automatically recover the system and quickly return it to operation, thus increasing the availability of the distributed system resources. Each host maintains a load vector which contains a view of all host loads. Periodically, each host sends its vector to its immediate successor. Load messages are also used to detect host failure.

The current prototype is composed of four daemons running on each host (Figure 1.2): (1) a load sharing manager (LSM) in charge of allocation and migration strategies, (2) a fault tolerance manager (FTM) responsible of checkpointing and message redirection, (3) a ring manager (RM) in charge of load transmission and failure detection, and (4) a replicated file manager (RFM) that implements the reliable storage.

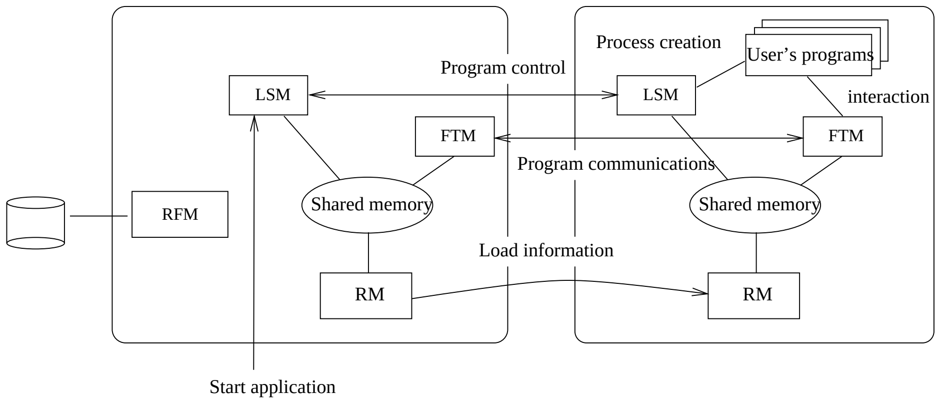

Figure 1.2 GatoStar architecture.

1.4.2 Process Allocation

Process Placement

To be executed, application processes need resources that vary from one process to another. To evaluate the need of each process, we transparently keep track of previous executions and/or we allow application designers to give appropriate indications. Dynamic process allocation can be operated according to the following criteria: hosts load, process execution time, required memory, and communication between processes. In the application description, the programmer can specify appropriate allocation criteria for each program according to its execution needs. He/she specifies the set of hosts involved in the load distribution and provides different binary code versions for each program in order to deal with system heterogeneity.

Selecting the less loaded host. Allocating processes according to load allows us to benefit from idle workstations power. To take into account the heterogeneous character of hosts, the load of a host is defined as directly proportional to CPU utilization and inversely proportional to its CPU speed [14]. On each host, the load sharing manager locally computes an average load and uses failure detection messages to exchange load information. Thus, no extra communication is needed when allocating programs. The program is allocated to the most lightly loaded host, if the load of this host is below the overload threshold. Otherwise, the program is started on the originating host as all hosts in the system are heavily loaded, or used by interactive users. The main difficulty is to find a good value for the overload threshold. A number of simulation studies on load sharing have been performed. Eager et al. [7] show that the overload threshold depends on the average load of the system and is between 1 and 2. These results and our experience with the GATOSTAR system lead us to set the overload threshold value to 1. Another problem is to get global state information. Indeed, every host cannot have exact knowledge about the load of the other hosts: first, because of network transmission time, and secondly there is a delay between the instant when a host receives the information and the instant it uses the information to decide where to allocate a process. Large load fluctuations can occur unexpectedly. If the delays mentioned above are significant, a program can be allocated to a host, which becomes heavily loaded. This problem is solved by re-directing the program allocation request, if the load of the selected host is higher than the overload threshold. We add to every remote execution request the expected load and a list of alternative hosts. Thus, re-allocating a program consists only in re-directing the execution request to one of these hosts. The load fluctuations coming after the initial placement can be solved only by migrating a sub-set of processes (see the next section).

Multicriteria load sharing algorithm. Encouraged by the fact that an application programmer has a good knowledge of his application execution needs, we specified program placement policies that take into account several criteria related to the applications execution needs. Indeed, application programs need resources (processor, memory, file, communication) which vary from one program to another. In the application description, the programmer may specify appropriate performance criteria for each program. In GATOSTAR, we propose the following ones:

1. Response time: The allocation policy corresponding to the response time criterion aims at minimizing the total execution time of the whole application. Reducing the response time of each program individually is not enough. Indeed, the application is run by batches of parallel programs that should be allocated at the same time. These considerations suggest that the longest running programs would be placed first. The selected hosts are allocated to programs in the order of their decreasing execution time values. Thus, the longest programs are allocated on the less loaded and fastest hosts (similar to the Longest Job First algorithm for a mono-processor machine).

2. File accesses: The policy for the file access criterion consists of finding the host where a program should be allocated so that the total cost of file accesses is minimized. The allocation decision is based on a number of parameters such as number of accesses, size of file accesses, disks and network speed. Once a host is selected to execute the program, a complementary policy allows us to decide whether to migrate one or more of the remaining files or to access them remotely (according to the previous parameters and the files size). The file access algorithm has not been implemented.

3. Communication between programs: The usage of this criterion aims to minimize the total cost of communications between programs. This is equivalent, in practice, to reduce the number of remote communications. To make this possible, we keep information of the location of all the application's programs already allocated in the system. Given this information, the most communicating programs are clustered on the same host.

4. Memory needs: In this case, the host on which a program will be allocated must have enough memory to run it. This is done by determining the memory needs of the program to be allocated and the available memory on every host. In contrary to the other criterion, which resources may be shared, the memory criterion provides an exclusive allocation. Program allocation on one host must not exceed the maximum physical memory of this host. The memory allocation criterion in GATOSTAR is conservative: Programs using this criterion are managed by a batch queue system, waiting for the appropriate memory space to be available. This ensures the correct execution of the application but may increase its response time.

5. Performance criteria combination: The previous performance criteria may be combined to ensure program placement according to several policies at the same time. We take at the beginning the set of available hosts, then, this set is reduced according to the first criterion on the given list of criteria. This mechanism is repeated with the remaining criteria until only one host remains or all the refinements have been done. The obtained host is the best compromise between the intervening criteria in the order in which the programmer specifies them.

Other recent works have shown that the multicriteria approach is very efficient for a general-purpose load sharing system, as in Isatys [2], as it can adapt the allocation to a wide spectrum of applications and programs behaviors.

Process Migration

Process migration is complex and difficult to realize. The GATOSTAR system immediately incorporates a migration mechanism by means of the checkpoint/restart already developed for STAR.

The load sharing algorithms, described in the previous section, has been extended to process migration according to applications and environment evolution. Environment evolution results from foreign activities: A user logs in or sends a remote job to an already loaded host. Application evolution stems from variations in resource usage (processor occupation, communication between processes, and memory demands). Since mechanisms to monitor such variations were already integrated, the adaptation of the multi-criteria load sharing algorithms has been straightforward.

The three main questions that must be answered when migrating processes are: when to migrate a process, which process to choose, and where to move the chosen process. Our proposed answer to the first and the third questions is based on two opposite load thresholds: a migration and a reception threshold. In the symmetrical scheme, the load sharing server periodically computes two conditions: (1) a faster workstation load is under the reception threshold, (2) the local host load is above the migration threshold and at least one workstation load is under the reception threshold. The former condition allows the potential use of fast available hosts, and the latter one allows us to decrease the overload of the local station. If one or both conditions are true, the server starts the process selection step. In the process selection step, the local server chooses a process that has been executed locally for at least a given local execution time. It prevents short-lived process to be migrated and it also prevents a process from spending all its time in migration from one host to another.

If there is more than one process that can be migrated, execution graph information (containing program description) is used. This graph indicates the estimated running time of each process, and the longest remaining process is selected. If such information is not available, we use a combination of the age of the process and its size. A logarithm function with the age of the process in input returns a number between 0 and 100. 0 means the process has not be running locally during the local execution time. 100 means the process has been running for more than 30 minutes. If several processes have a 100 value, then the smallest program in size is selected.

As with the overload threshold (the one used when a host is initially chosen), it is difficult to choose good values for the migration and reception thresholds. This threshold seems to depend on the behavior of the migrated processes, on the load ratio (available processing power / mean number of processes), and on users' behavior. There are few studies on the reception threshold. Most of the time, it is assimilated with the overload threshold. Again, our experiments lead us to set the migration threshold to 2 and the reception threshold to 0.5. These values depend on the overload threshold (1.0), and the three values cannot be changed independently. A future work will attempt to model the execution presented in the next section, and to simulate the change in the three thresholds.

1.4.3 Failure Management

In order to get an efficient failure detection mechanism, we operate a structuring of hosts in a logical ring. Each host independently checks its immediate successor in the ring. The checking process is straightforward and the cost in messages is relatively low.

Our recovery method uses a checkpoint/rollback management. By checkpointing, user processes save their states to prevent their local host failures. We adopt an independent checkpointing with a pessimistic message logging (see Section 1.3). This technique is particularly adapted for application composed of processes exchanging small streams of data. Otherwise, the cost of the logging is proportional to the number of messages. This method has the following advantages:

- the domino-effect is eliminated,

- checkpoint operation can be performed unilaterally by each process and any checkpointing policy may be used,
- only one checkpoint is associated to each process,
- checkpoint cost is lower than in a consistent checkpointing and recovery is implemented efficiently because all interprocess communications do not take place during a replay.

To reduce the cost of checkpointing, GATOSTAR implementes incremental and non-blocking checkpointing. We will see in the performance section that these techniques highly reduce the cost of checkpointing.

1.4.4 Performance Study

We present performance measurements of the process allocation and the migration strategies. These measurements were done in a real system within our laboratory. The supporting environment is composed of a set of Sun workstations connected by a LAN (Ethernet 10 Mbit/s). The environment has not been modified (usual daemons were running). All results presented are averages over a number of trials.

Cost of initial placement. The time to compute an initial placement for an application is rather low. Indeed, hosts selection is done locally when starting the application, and the only exchange of message is the request from the local load sharing manager to the remote one to execute a given program. The maximum size of such a request is 2 kilobytes (program name, options, global identification and some of the shell environment variables). In the presented measurements the request length is about 500 bytes, and the time for an initial placement is 100 milliseconds. Compared to other systems, the mean time for a placement request is 5 seconds in REM [13], and 12 seconds in DAWGS [4]. These allocation times are essentially due to highly communicating algorithms, while GATOSTAR does not perform any extra communication when allocating hosts.

Cost of process migration. The time for a process migration is proportional to the amount of data to be transferred from the source host to the selected one. Figure 1.3 shows the time to migrate a process (that is, the sum of the checkpoint time and restore context time) according to the process size.

In the following, the size of each test process is less than 100 kilobytes. Thus, migration time of these processes is less than 3 seconds.

Process allocation. Figure 1.4 presents the execution time of six parallel programs running on six hosts. We compare the non-preemptive placement strategy with the migration. Each program is computing a given number of iterations, from 50 million to 500 million. There is no communication between programs. The local execution

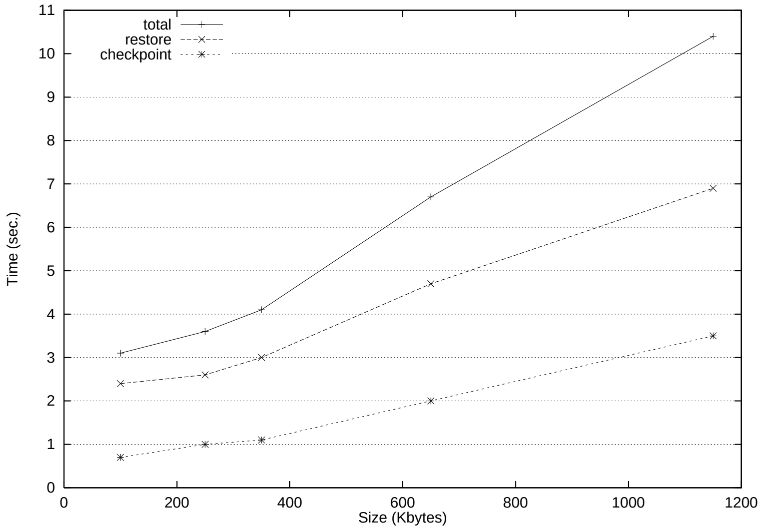

Figure 1.3 Migration time according to process size.

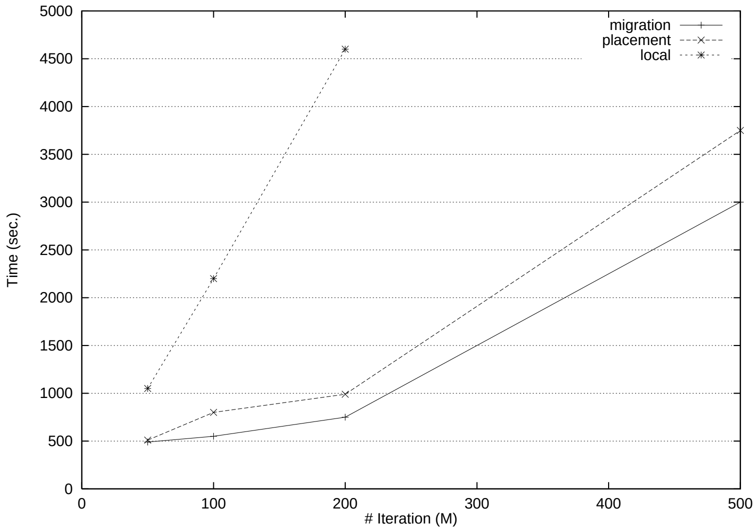

Figure 1.4 Execution time of six programs according to the number of iterations.

time (i.e., without any distribution) is from 1080 seconds for 50 million iterations to 11,160 seconds for 500 million iterations (not shown in the figure).

When there is a small number of iterations, less than 30 million, the execution time of the application is too short for the migration strategy to provide significant improvements. When the number of iterations increases, we obtain a difference of 28 % between execution with and without migration. This difference is due mostly to the correction of the initial placement. Indeed, when this application is launched, all the six hosts are possibly not unused. Then, when they become unused, migration allows us to share the work with them, while in case of non-preemptive allocation the hosts stay unused. On the other hand, when a station becomes overloaded because of some users' activities, migration allows us to select a more suitable host.

Figure 1.5 presents the speed-up obtained by migration compared to non-preemptive placement. There are six parallel programs, each program running for 1800 seconds, and the number of hosts varies from one to five. With one host we obtain the degradation of the migration strategy, because no migration is possible. Then, we observe that even with a limited number of hosts, migration reduces the execution time by adapting the number of programs on each host according to the users' activity. This good improvement is due mostly to migrations occurring when a program ends. For instance, with four hosts, a typical placement will be one program on each of the two first hosts and two programs on each of the two others. Consequently, the two first hosts will finish two times faster than the other ones. At this time migration allows us to move two programs in order to have one program on each host that reduces remaining execution time. The decrease of speed-up for three hosts is due to the initial optimal placement: two programs on each host. In that case, migration is used only to react to user activities.

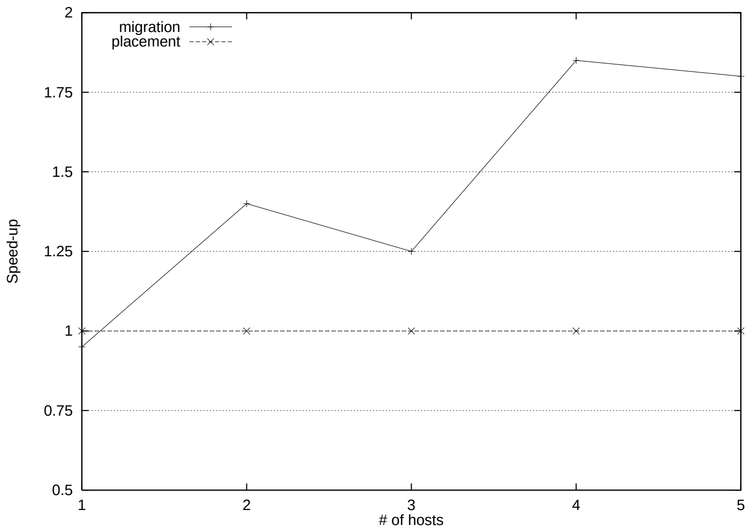

Figure 1.5 Speed-up according to the number of hosts.

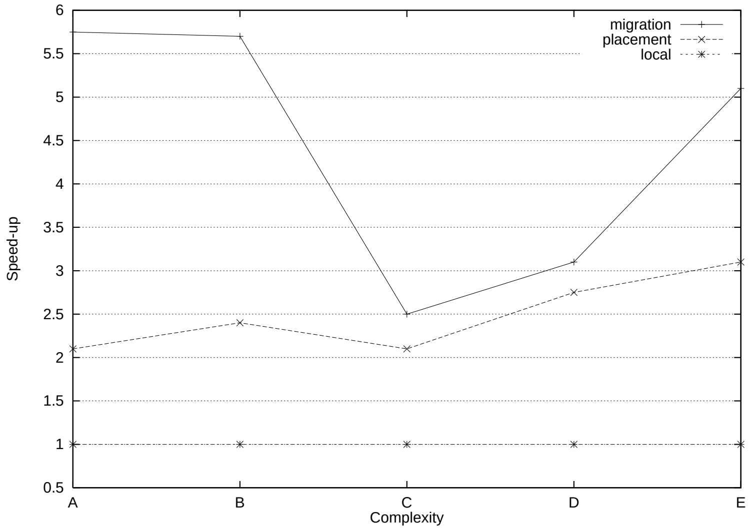

Figure 1.6 Speed-up according to application complexity.

Response time. Figure 1.6 compares the speed-up obtained by placement and migration when the complexity of an application evolves. These measures are related to the response time algorithm, where placement and migration policies know in advance the running time of each program. The various applications are given below:

A: 1.50M 1.100M 1.200M 1.500M (4 programs, 850M)
B: 3.50M 4.100M 2.200M 1.500M (10 programs, 1250M)
C: 5.50M 5.100M 2.200M 1.500M (13 programs, 1650M)
D: 10.100M 2.200M 2.500M (14 programs, 2400M)
E: 10.100M 5.200M 5.500M (20 programs, 4500M)

For instance, application A is composed of 4 parallel programs with a total number of 850 million iterations, the first one with 50 million iterations, the second one with 100 million, the third one with 200 million and the last with 500 million. Tests have been done with six hosts.

The single process of 500 million iterations bound the response time of the two first applications. In that case, migration improves greatly the speed-up. The other programs needing less execution time are kept on different hosts of the 500 million iterations program, and this last always profits from the better unloaded host. There is no such bound in applications C and D, which implies little difference between placement and migration. For the last application, the migration is beneficial for the five 500 million iterations programs.

Extrapolation by simulation. We present some results from a simulation study that show the impact of threshold values. We have simulated two strategies. In the classical one, which is implemented in GATOSTAR, allocation and migration decisions are based on fixed threshold. In the adaptive one, thresholds depend on the global load of the network. Figure 1.7 presents a performance evaluation of these two policies according to the number of hosts. We have simulated two applications: Application A is composed of 48 parallel CPU intensive processes with different execution times, application B is composed of 40 parallel processes with the same execution time. In all cases, the adaptive strategy improves the response time. We see that speed-up also depends on the application behavior.

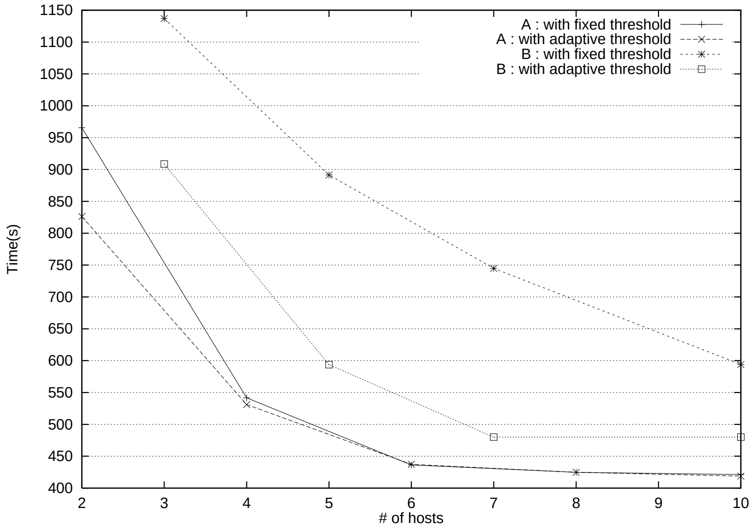

Figure 1.7 Simulation of response time according to the threshold policy.

In Figure 1.8, we vary the overload threshold value to optimize the response time of three different application configurations : 30 identical CPU intensive processes allocated on 10 hosts, 50 processes allocated on 10 or 20 hosts.

We can conclude, from these two simulation studies, that an allocation manager should adapt the threshold values not only according to the global load but also to the behavior of the current applications.

Performance of checkpointing. Finally, we evaluate the cost of checkpointing (used for process migration and to tolerate host failure). We chose three long-running, compute-intensive applications exhibiting different memory usage and communications patterns:

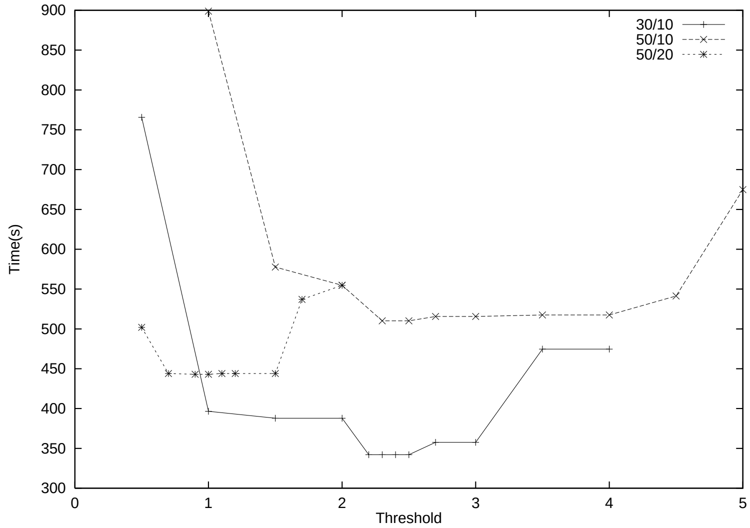

Figure 1.8 Simulation of response time according the overload threshold value.

1. The *Gauss* application performs Gaussian elimination with partial pivoting on a 1024 x 1024 matrix. The matrix is distributed among several processes. At each iteration of the reduction, the process that holds the pivot sends the pivot column to all other processes.

2. The multiplication application, called *matmul*, multiplies two square matrixes of size 1024 x 1024. The computation is distributed among several processes. No communication is required other than reporting the final solution.

3. The *fft* application computes the Fast Fourier Transform of 32768 data points. The problem is distributed by assigning each process an equal range of data points. Like the previous application, no communication is required other than reporting the final solution. Table 1.1 presents the overall running time and per process communication and memory requirements for the three applications when run without fault tolerance management (i.e. without checkpointing and message logging). Each application is distributed on five hosts: One host executes a master process and the four other hosts execute the computational processes.

Gauss and *matmul* require a sizeable amount of data, stressing the checkpoint and state restoration mechanisms. Moreover, the *Gauss* application exhibits a large amount of communication especially stressing the message logging. The *fft* application is long-running and requires a medium amount of data.

Table 1.2 presents the running times of the applications programs when run with independent checkpointing and pessimistic message logging. Applications run with

Table 1.1 Applications Requirements

Applications	Time (sec.)	Memory (Kbytes)	Sending data (Kbytes)
Gauss	344	1704	2700
matmul	723	2688	0.06
fft	1177	1200	0.06

a 2- minute checkpointing interval. Checkpoints and logs are duplicated.

Table 1.2 Parallel Applications Evaluation

Appli.	Full checkpoint		Non-blocking		Incremental	
	Time (sec.)	%	Time (sec.)	%	Time (sec.)	%
Gauss	567	64.82	505	46.80	457	32.85
matmul	844	16.79	768	6.34	748	3.57
fft	1244	5.75	1228	4.36	1194	1.50

In spite of checkpoint optimizations, we observe a high overhead for the *Gauss* application. In fact, this application is not a good candidate for message logging approaches, specifically because of its communication rate. The overhead due to message logging for this application is 14.53 %. The cost of message logging represents half of the global overhead when we apply incremental checkpointing.

For all three applications, incremental checkpointing provides a sizeable reduction of the overhead. Comparing to the full non-blocking checkpointing, we obtain reduction in overheads of between 24 % and 63 %. This difference is partly due to different communication rates. Moreover, applications can be divided into two categories: applications with an address space that is modified with high locality (matrix multiplication and *fft* applications) and applications with an address space that is modified almost entirely between any two checkpoints (*Gauss* application). For applications of the first category, incremental checkpointing is very successful (79 % of reduction for matrix multiplication and 75 % of reduction for *fft*). For the applications in the second category, incremental checkpointing is less effective (about 49 % of reduction for the *Gauss* application).

1.5 Related Works

Few experiences have been reported with load sharing systems with a fault tolerance mechanism. The major ones are the systems REM (Remote Execution Manager) [13], Paralex [1], Condor [11], and DAWGS (Distributed Automated Workload sharing System) [4]. These four systems have been developed essentially as load sharing manager, since fault tolerance is a limited extension. To our knowledge, no work has compared and unified the two concepts.

REM and Paralex are load sharing managers that support cooperative applications. REM provides a mechanism by which processes may create, communicate with, and terminate child processes on remote workstations. Active process replication is used to ensure the fault tolerant execution of the child processes. Thus, the fault tolerant property is gained at the expense of computing power. Moreover, failure of the local user's machine implies a failure of the whole computation. As the failure rate in a local area network is low, the process replication mechanism is not adapted to a load sharing scheme. Furthermore, the target applications of software fault tolerance are long-lived ones, and users are not supposed to be always connected with the same local machine. Paralex provides a package for distributing cooperative computation across a network. It uses passive process replication to achieve both fault tolerance and load balancing. Since process migration can be done only between replicas, the load sharing capability is strongly limited.

Condor and DAWGS rely on the same principle as GATOSTAR. Processes are initially allocated to an idle host, and a checkpoint/rollback mechanism allows the migration of processes and the restarting of processes after a host failure. In both systems, migration is essentially used to respect the privacy of workstation owners. The main drawback of Condor is that it deals only with independent processes, since Condor processes cannot communicate or synchronize. Some extensions, as Dynamic PVM [5], have adapted Condor to communicating processes. DAWGS is a complete load sharing fault tolerant facility, including I/O redirection. The two main differences between DAWGS and GATOSTAR are that (1) DAWGS relies on kernel modification and (2) DAWGS' fault tolerant part is not configurable and a process running on a faulty host must wait for the host to restart (there is no easy way to replicate a checkpoint file). The kernel modifications allow full user-transparency without the need to link with a special library, but it highly limits the portability of such a system (and thus, of the supported applications).

1.6 Conclusion

This chapter has described the advantages of load sharing and fault tolerance unification. Such unification may improve parallel applications' response time and decrease the overhead of the software fault management.

We presented GATOSTAR, a unified load sharing and fault tolerance facility. This system takes advantage of the overall system resources and allows the distribution of the applications processes on the most appropriate hosts. Process allocation policy takes into account both information about the system state and resource needs for the execution of applications. A checkpoint mechanism allows the recovery of processes after hosts' failures and is used to move processes in case of load fluctuations. Measures of GATOSTAR show the benefit of the load sharing and of the multi-criteria allocation algorithms in a local area network of workstations and indicate that the combination of dynamic placement and process migration outperforms dynamic placement only. Dynamic placement allows us to use the available machine at launch time at very low cost (one request). Then, for long-lived applications, process migration allows to correct this initial placement according to failures or changes of the program behaviors, in the hosts load, or in interactive users activities.

1.7 Bibliography

[1] O. Babaoglu, L. Alvisi, A. Amoroso, and R. Davoli. Paralex: An Environment for Parallel Programming in Distributed Systems. *Proceedings of the International Conference on Supercomputing*, pages 178–187, July 1992.

[2] M. Banâtre, Y. Belhamissi, V. Issarny, I. Puaut, and J-P. Routeau. Adaptive Placement of Method Executions Within a Customizable Distributed Object-Based Runtime System. *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 279–286, June 1995.

[3] A. Barak, O. Laden, and Y. Yarom. The NOW MOSIX and Its Preemptive Process. *Bulletin of the IEEE Technical Committee on Operating Systems*, vol. 7(2), pages 5–11, 1995.

[4] H. Clark and B. McMillin. DAWGS - a Distributed Compute Server Utilizing Idle Workstations. *Journal of Parallel and Distributed Computing*, vol. 14, pages 175–186, February 1992.

[5] L. Dikken, F. Van der Linden, J. Vesseur, and P. Sloot. Dynamic Load Balancing on Parallel Systems. *High Performance Computer Network*, pages 273–277, 1994. Lecture Notes in Computer Science 797.

[6] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software - Practice and Experience*, vol. 21(8), pages 757–785, 1991.

[7] D. L. Eager, E. D. Lazoska, and J. Zahorjan. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, vol. SE-12(5), pages 662–675, May 1986.

[8] E. N. Elnozahy and W. Zwaenepoel. On the Use and Implementation of Message Logging. *Proceedings of the 24th International Symposium on Fault-Tolerant Computing Systems*, pages 298–307, June 1994.

[9] G. A. Geist and V. S. Sunderam. Network Based Concurrent Computing on the PVM System. *Concurrency: Practice and Experience*, vol. 4(4), pages 293–311, June 1992.

[10] J. M. Hélary, A. Mostefaoui, R. H. B. Netzer, and M. Raynal. Preventing Useless Checkpoints in Distributed Computation. *Proceedings of the 27th International Symposium on Fault-Tolerant Computing Systems*, pages 68–77, June 1997.

[11] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - a Hunter of Idle Workstations. *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, January 1988.

[12] P. Sens and B. Folliot. The STAR Fault-Tolerant Manager for Distributed Operating Environments. *Software Practice and Experience*, vol. 28(10), pages 1079–1099, August 1998.

[13] G. C. Shoja, G. Clarke, and T. Taylor. REM: a Distributed Facility for Utilizing Idle Processing Power of Workstations. *Proceedings of the IFIP Conference on Distributed Processing*, October 1987.

[14] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: a Load-Sharing Facility for Large Heterogeneous Distributed Computing Systems. *Software - Practice and Experience*, vol. 23(2), pages 1305–1336, December 1993.

Index