Preserving Message Integrity in Dynamic Process Migration

E. Heymann, F. Tinetti, E. Luque
Universidad Autónoma de Barcelona
Departamento de Informática
08193 - Bellaterra, Barcelona, Spain
e-mail: e.heymann@cc.uab.es

Abstract [1]

Processor and network management have a great impact on the performance of Distributed Memory Parallel Computers. Dynamic Process Migration allows load balancing and communication balancing at execution time. Managing the communications involving the migrating process is one of the problems that Dynamic Process Migration implies. To study this problem, which we have called the Message Integrity Problem, six algorithms have been analysed. These algorithms have been studied by sequential simulation, and have also been implemented in a parallel machine for different user process patterns in the presence of dynamic migration. To compare the algorithms, different performance parameters have been considered. The results obtained have given preliminary information about the algorithms' behaviour, and have allowed us to perform an initial comparative evaluation among them.

1. Introduction

Parallelism is a challenging computer technology. The main promise of this technology is to obtain high performance by replicating the computer hardware. A Distributed Memory Parallel Computer (DMPC) is organised as a set of nodes that communicate over an interconnection network, each node containing a processor and some memory. Interprocess communication can be achieved via explicit message passing or via virtual shared memory. In DMPC high performance is combined with scalability. With such machines, we have considered a message passing programming model. The unit of parallelism is the *process*, which is a logical entity that executes code sequentially. A *channel* allows communication and synchronisation among processes.

The performance on a DMPC can be improved if the use of its resources is balanced. This means balancing the computation load (load balancing) and balancing the network utilisation (communication balancing). It is difficult to achieve these goals statically because the behaviour of the parallel application normally cannot be determined in a static way [1]. When some processes, with high computation needs, have to be executed on a single processor, it would be convenient to separate those processes, that is, to place them on different processors, thereby balancing the load of the parallel application. In the same way, if a processor contains some processes so that their communication needs are greater than the network bandwidth at this node, a hot spot is produced. In such cases it would be also useful to separate those processes, thereby balancing the communication load. Therefore, we expect to obtain better performance dynamically, via Dynamic Process Migration. With a process migration support, the processes-processors mapping can be changed dynamically, at execution time [2].

In the same way as the address translation mechanisms and the dynamic data movement through the memory hierarchy (virtual/cache memory) have prevailed as essential characteristics of any sequential computer, it will be necessary to include a process migration mechanism in the hardware, thereby improving the performance of parallel computers.

Dynamic process migration implies the following problems:

- To evaluate the processors and network load, and how to use this information in deciding which process must migrate, where it will migrate and when (Dynamic Load Balancing Policies);

[1] This work was supported by the Spanish Comisión Interministerial de Ciencia y Tecnología (CICYT) under contract number TIC 95/0868, and was partially sponsored by the EU's Copernicus Program under Contracts number CIPA-C1-93-0251 and CIPA-CP-93-5383.

- How to migrate (physically), this includes the de-allocation of the process in the source processor and its allocation in the destination processor (Process Migration Mechanisms); and
- To manage the communications involving the migrating process, which must continue receiving messages independently of its network location (Message Integrity Management).

Our work is focused on the last problem, which we have called *the message integrity problem* (MIP). Given an initial mapping, in which each process S1, S2, ... , Sk communicating with process R, knows R's physical address, that is, the processor identification Pi, in a message passing machine, the MIP consists of keeping these connections, once process R migrates from processor Pi to processor Pj, as it is shown in figure 1.

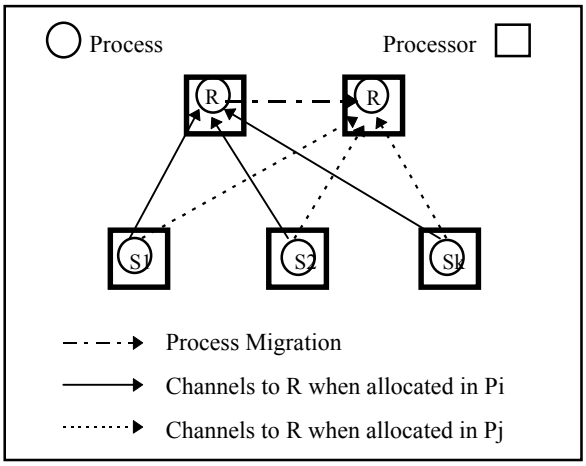

Figure 1. The Message Integrity Problem

Related work can be found in [3] and [4]; and a description of the generic work carried out about Dynamic Process Migration is presented in [5].

Another situation in which the MIP appears is the following: process *p* has many copies of itself on different processors, but only one of them is active at any given time. In this case, a message integrity algorithm is needed to assure that the messages directed to *p* arrive at the location where the active copy of *p* is found. It is interesting to note the similarity between the MIP and the cache access problem in a shared memory multiprocessor [6].

The MIP implies considering: which actions are taken by (1) the migrating process, say R, and by (2) the processes sending messages to R. To manage the MIP we have analysed six algorithms: *Centralised Scheme*, *Send First to Home Address*, *Migration Mailbox*, *Follow Me*, *Message Rejection* and *Full Protocol*. All these algorithms present different characteristics. In order to perform a comparison among them, we have studied them by sequential simulation, and we have also implemented all the algorithms in a parallel machine, and executed them for different user process pattern when dynamic process migration is allowed. The performance parameters we have considered to evaluate and compare the algorithms are: message latency, the extra load produced, the reaction time, the scalability, and how the algorithms modify the communication pattern over the network. For simplicity, but without loss of generality, we only consider static creation of user processes and communication channels.

The rest of the paper is organised as follows: The algorithms which solve the message integrity problem are described in section 2; section 3 presents the performance parameters and experimentation done with these algorithms, whose results show us the behaviour of each algorithm. These results are presented in section 4; section 5 contains the analysis of the results of the previous section, and in section 6 the conclusions and future work are exposed.

2. Algorithms Description

To describe the algorithms which offer different alternatives to the Message Integrity Problem, the following convention has been used: let R be the message receiver process that will migrate from processor Pi to processor Pj, and let S1, S2, ... , Sk be the processes communicating with R through static channels and sending messages to it.

The communication overhead produced by each algorithm is made up of: (1) Retransmission Messages, which represent the messages that arrived at a wrong destination and must be re-sent in order to reach their true destination, due to the fact that the destination process R had migrated; and (2) Control Messages, which represent the management messages needed to implement the algorithm.

- **Centralised Scheme:** There is a Central Server Process (CS) which provides physical address (processor identification) of the destination process, through its process-processor allocation table. This CS process has to be notified of any migration, that is, when a process migrates it must send its new address to the CS. When a process Si wants to send a message to process R it must:

(1) Ask R's location from the Central Server.

(2) Wait for the CS's response -for example Pi -. This is the last known address of R.

(3) Send the desired message to Pi.

It is possible to send a message to a wrong address. In such case a NACK signal is received. This situation can be produced when the answer to Si leaves just before the Central Server is being informed about R's migration. In this case, steps 1-3 are repeated.

Each send operation implies, in most cases, two control messages, one for asking the CS for the address of the destination process R, and one to receive its answer. If R has migrated but the CS has not still been notified, a message destined to R will arrive to a wrong destination and a retransmission will be carried out. In this case, more control messages are produced. Each migration operation produces one control message in order to notify to the CS the new location of the migrating process.

- **Send First to Home Address:** The *home processor* is the initial processor where each process is assigned. Suppose that process R has been initially assigned to processor Pi (R's home processor), then any message destined to process R will be sent to processor Pi, no matter if R is still there. Once the message arrives to Pi, if process R has migrated, the message is retransmitted to R's new location, say Pj. This implies that the home processor must always know the location of its initial processes. When a process migrates, it must send its new location to its home processor. In this way the home processor will be able to retransmit the messages to R. It is possible that some messages have been sent to Pi for process R before the home processor has been notified about R's new location. These messages will not find process R in Pi, so a NACK signal will be sent to the home processor. Once it receives the new location of R, it will retransmit the message to R. In this last case, extra control messages are produced.

Each migration operation produces a control message destined to R's home processor notifying its new address. Once a process has migrated, all the messages directed to it will be sent to its home processor and then retransmitted. Additional retransmissions will be done when process R migrates, its home processor has not still been notified, and a message to process R arrives.

- **Migration Mailbox:** The idea in this case is based on modifying the semantic of the *receive* primitive. When a receive is executed by process R, a message request is sent to a predefined address, called the *migration mailbox* of process R. Any process Si that wants to communicate to the process R, sends the message to R's migration mailbox, no matter if R has migrated or not. This implies that no action is taken when a process migrates, that is, nobody has to be notified about R's migration. This scheme has the following restriction: if a process must migrate, it must wait to complete all pending receives operations before the migration is allowed to be made.

The migration mailbox of a process R is located in the same processor as this process R is initially mapped, say Pi. Therefore all the messages destined to process R will be sent to processor Pi. When process R and its Migration Mailbox are in different processors, that is, once R has migrated, each receive operation will produce a control message and a retransmission.

- **Follow Me:** When process R migrates, its new location (migration address) is recorded in the processor it leaves. In this way each process builds a path to follow by the messages sent to it. When process Si wants to send a message to process R, the message is sent to the processor where R was initially assigned, say Pi. If R has migrated, the migration address left in Pi is used to follow process R. This step is performed as many times as needed to reach R.

On the one hand this algorithm does not generate any control message, but on the other hand, once a message is injected into the processor network, it will be retransmitted until reaching process R, that is, as many times as R migrates, unless R migrates to a processor in which it was located before. In this last case the path is reduced.

- **Message Rejection:** When a process migrates from processor Pi to processor Pj its new location is recorded in the processor it leaves. When a process Si sends a message to R it uses its last information about R's location, which is the location of R when Si sent its last message to it. If a processor Pi receives a message to R and R is not in the processor because it has migrated, it sends a NACK signal to Si's processor, attaching R's (last) known address. Si's processor then updates its own R's location using the received address, and retransmits the message. These steps are repeated until the message finally reaches R.

Each time a message is rejected, both a control message notifying the known address of process R, and a retransmission to the received address, are generated.

• **Full Protocol:** Process R will not be allowed to migrate until:(1) all process S1, S2, ... , Sk communicating with it know its new location; and (2) R receive all the message already sent to it.

When process R migrates, it performs a full-information exchange protocol for all the processes S1, S2, ... ,Sk. This protocol consists of the following steps:

(1) A signal to processes S1, S2, ... , Sk is sent making them to stop sending messages to R;

(2) these processes send R the number of messages they have sent to R;

(3) R waits for the arrival of all the pending messages;

(4) when all the pending messages have arrived, then R migrates to Pj;

(5) R notifies its new address to processes S1,S2,... , Sk, and allows them to continue sending messages to it.

This algorithm does not generate any retransmission, but each time a process R migrates, 3 * k control messages are generated, with k representing the number of processes communicating with R.

A variation of this algorithm allows process R to migrate after step 2. The pending messages will be retransmitted from Pi to Pj.

3. Performance Parameters and Experimentation

The proposed algorithms have been evaluated and compared through the following performance parameters:

• *Latency*: This is the average time of sending a message from one process to another, that is, the time elapsed from when a message leaves the source process to when it reaches the destination process;

• *Load Overhead*: This represents the extra load over the network that each algorithm adds, which consists of the average number of message retransmissions per message, plus the average number of control messages. This parameter points out the bandwidth an algorithm consumes;

• *Reaction time*: This is the time elapsed from when a process is selected to migrate, and the time in which this process can be migrated, preserving the reception of its messages;

• How an algorithm modifies the *communication pattern* between processors. Focusing the communication pattern is necessary to evaluate if the algorithm would be able to perform communication balancing; and

• *Scalability*: How the algorithm behaves when adding processes and processors.

To evaluate these performance parameters the following experimentation, which includes simulation and implementation, has been carried out:

• Modelling and sequential simulation of the proposed algorithms. This simulation attempts to analyse the intrinsic behaviour of each algorithm, without the influence of external elements like the load over the communication network or the message sending pattern of the application.

• Implementation of the different algorithms which solve the MIP in a parallel machine. In this implementation the application, that is the user processes, is simulated by means of a synthetic program which simulates computation and, sends and receives real messages. The processes migration is also simulated, but like the communication messages, it generates real traffic over the communication network. The synthetic program allows to perform executions modelling different situations. The latency is affected by the communication network load. Messages between user processes, control messages, retransmission messages, and the migrating user processes are transmitted using the communication network, and all of them constitute network load.

For each one of the six algorithms we have carried out simulations and executions varying different input parameters. The number of processors studied were 8, 32 and 64 processors for the simulations and 8, 16, and 32 for the executions. This parameter allows to study the scalability of the algorithms. The process/processors ratios considered were 5 and 20 processes per processor. This last parameter has not been relevant in the results. Finally, different percentages of migration have been studied: 0%, 1%, 5% and 10%. The 0% percentage of migration is used as reference, and represents the case when dynamic process migration is supported but no process migrates. Each time a send operation is produced a migration occurs with probability *p*, which represents the percentage of migration. Both simulation and execution takes place until an average of 150 messages per process are generated.

It is very important to point out that all the processes are connected among themselved, that is, process R can receive messages from any other process; and when adding processes, the connectivity is scaled. This fact will affect only the full protocol algorithm, in which a process at migration time communicates with all the processes communicating with it, without having a multicast mechanism. In order to evaluate the effect of the number of connections a process has, the full protocol algorithm has been simulated with the following processes graphs: a complete graph, where all the processes are connected with each other; a bi-directional pipe, where each process is connected with another two, and the number of

connections has a linear growth when scaling; and the hypercube topology, where in a 2^n hypercube each process is connected with *n* processes, and the number of links has a logarithmic growth when scaling.

For each algorithm a total of 24 simulations and executions were carried out, also the Full Protocol algorithm was simulated with the different topologies described above.

3.1 The Sequential Simulation

The sequential simulators were implemented using a discrete event driven simulation model. The latency we have considered is the minimum under a processor network, because we are not considering in this first approximation the effect in the latency produced by the load over the network each algorithm adds. As we have handled the latency and the load over the network independently, although the overhead produced by the algorithms described increases the network load, it will not affect the latency.

The architecture model considered in the sequential simulation supports the communication between each pair of processes, independently of their location in the processor network [7]. The communication time between each pair of processes is a variation under an exponential function around a uniform value, provided by a two-step routing. The first step is a random routing, and in the second step the message is routed to its destination processor. The average process communication time is given by the average distance between a pair of processors in an hypercube topology, which presents logarithmic growth, and doubled, thereby simulating the two-step routing. In this way, the communication time is given for an exponential function around the following values: 2.8 in an 8 processor hypercube, 5.0 in a 32 processor hypercube and 6.0 in a 64 processor hypercube. Accordingly, the latency for a particular message is proportional to the number of hops that it takes the message to arrive to its destination.

3.2 The Parallel Implementation

The general parallel implementation can be described in terms of the processes shown in fig. 2, which is modified for each proposed algorithm. The synthetic model of the application is composed of user processes, whose execution is simulated. The synthetic application's processes are represented with a discontinuous line in figure 2. The rest of the process, that is, the implementation processes for the algorithms are represented with a continuos line.

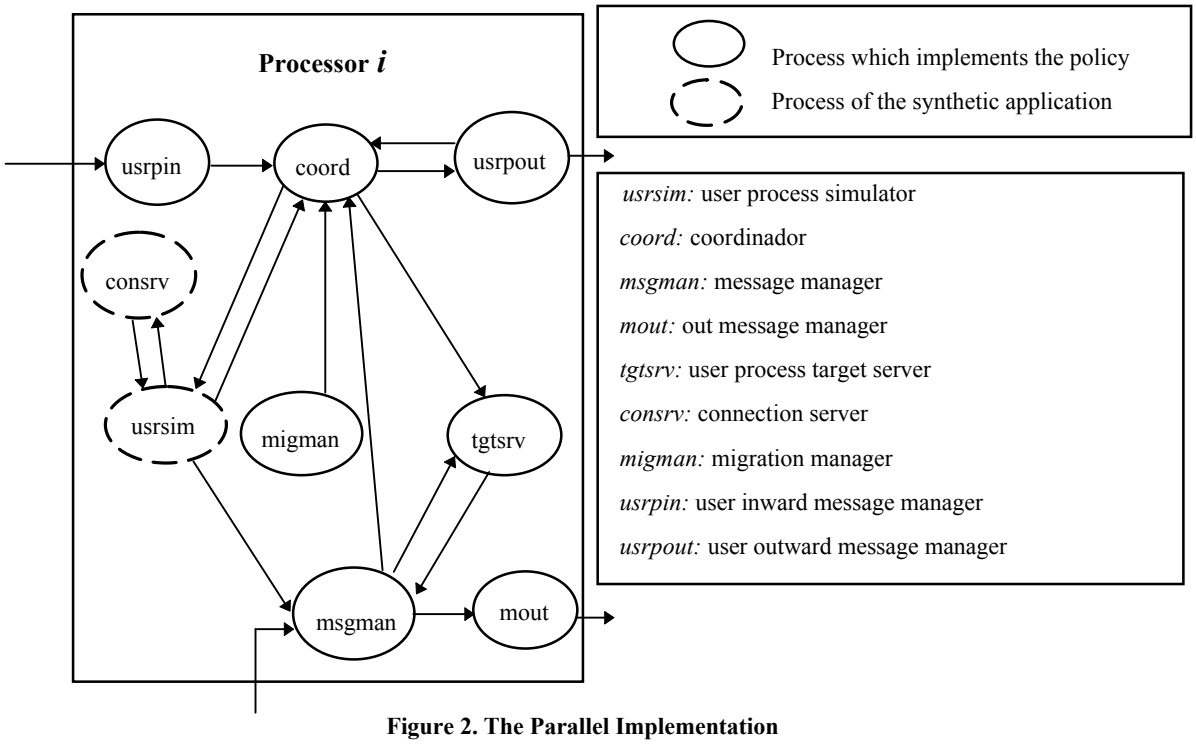

Figure 2. The Parallel Implementation

Process *usrsim* (user simulator process) simulates the behaviour of the user process time slice, as in multiprocessing systems. It receives a user description from process *coord* (the probability of sending a message and the computing value),

and simulates the user process in a time slice. If the user process does not communicate in the simulated time slice, then process *usrsim* consumes the time slice and requests the process *coord* for another user process to simulate. If the user process communicates in the simulated time slice, process *usrsim* asks process *consrv* for the communicating partner and performs a message request to process *msgman.*

Process *coord* (coordinator process) keeps the user processes running and also migrates user processes. User processes are sent to simulate its execution to process *usrsim*. When an outward migration request is received from process *migman* a user process is sent to another processor (both chosen randomly) via the process *usrpout.* Incoming user processes are received from process *usrpin*. Process *tgtsrv* is updated in each migration.

Process *msgman* (message manager process) handles message requests. These requests come from other processors or from the process *usrsim* running in the same processor. For each request, process *msgman* asks process *tgtsrv* the location (processor) of the message destination. If the destination user process is running locally, then the message is consumed, otherwise the request is handled according to the policy for solving the MIP. When a request has to be sent to another processor, process *msgman* sends it to process *mout.*

Process *mout* (message out manager process) is dedicated to distributing messages to other processors.

Process *tgtsrv* (target user process location server process) serves requests from process *msgman* for user process location and remains updated at least for the location of user processes running locally. It receives information from process *coord* about incoming and outgoing user processes.

Process *consrv* (user interconnections server process) knows the interconnection graph between user processes. Once a user process has decided to send a message, process *usrsim* asks process *consrv* for the destination user process. Given a source user process for a message, the process *consrv* selects a destination user process randomly.

Process *migman* (migration manager process) generates outward migration requests for process *coord.* Process *migman* implements the migration policy for user processes.

Processes *usrpin* and *usrpout* (user processes inward and outward migration processes) handle the physical migration (to and from the processor) of user processes.

Process *mout* communicates with process *msgman* in all the other processors, and process *usrpout* communicates with all the other processes *usrpin.* Two separated data circuits are defined: one for message data (*mout-msgman*), and another for process migration data (*usrpin-usrpout*). This general structure is set up by loaders.

The simulation of user defined applications can be defined according to the user processes (1) number; (2) computing and communication behaviour; (3) interconnection pattern; (4) probability of migrating; and (5) migration policy. The algorithms are implemented on a real processor network. In the parallel machine it is not possible to know the global state of the system (e.g. there is not a global clock), and some local measures are taken to calculate the message latency. Three measures made in each processor are used to calculate message latency: (1) Queue length of the message request when it came to the process *msgman*,(2) Hop latency (elapsed time to send a request from a process *msgman* to another), and (3)Number of hops a message takes to reach its destination. These three measures are taken locally at each processor without knowledge of the parallel computer state.

4. Results

In this section some significant results for the sequential simulation and the parallel implementation are shown. Figures 3, 4 and 5 show the results of the sequential simulation for all the algorithms, considering latency, retransmissions, and control messages for 8 and 64 processors. The X-axis represents the percentage of migration. In the latency's graphs, the Y-axis represents the time; in the retransmissions' graphs, the Y-axis represents the average of retransmissions per process, normalised with respect to the average number of messages generated by each process, that is 150; and in the control messages' graphs, the Y-axis represents the average control messages generated per process, also normalised with respect to the average number of messages generated by each process. As the user processes interconnection strongly affects the number of control messages generated by the Full Protocol policy, Figure 6 shows the number of control messages produced by this policy for user processes connected in a full, pipe, and hypercube graph. The complete results of the sequential simulation can be found in [8].

Figures 7, 8, and 9 show the results obtained in the parallel implementation for 8 and 32 processors (the limit of the parallel computer). Measures and graphical axes are organised as explained for the sequential simulation.

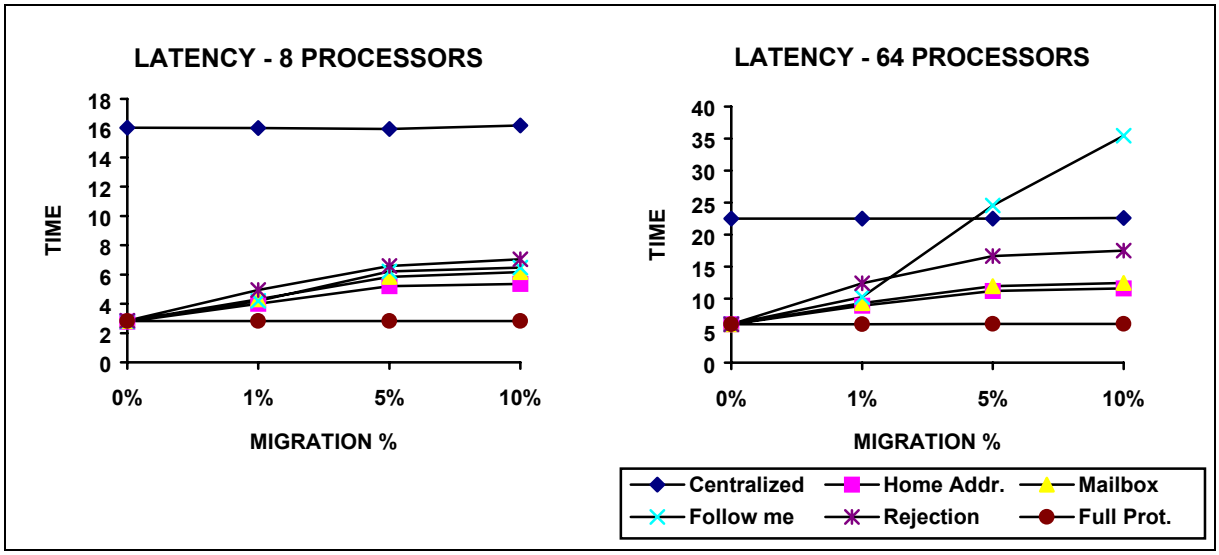

Figure 3. Sequential Simulation: Message Latency.

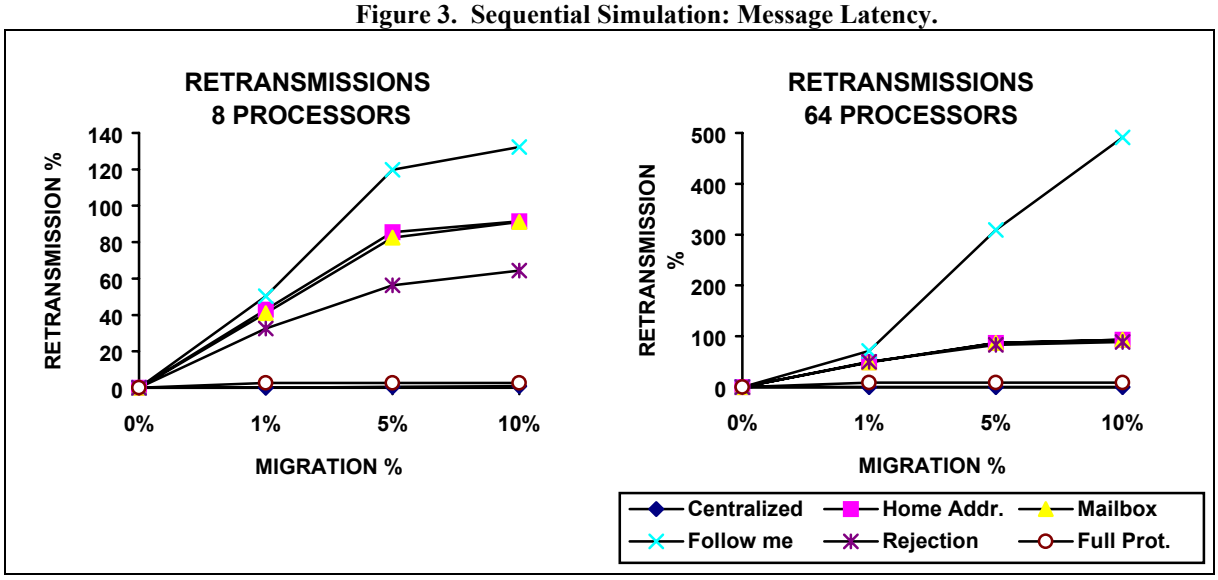

Figure 4. Sequential Simulation: Message Retransmissions.

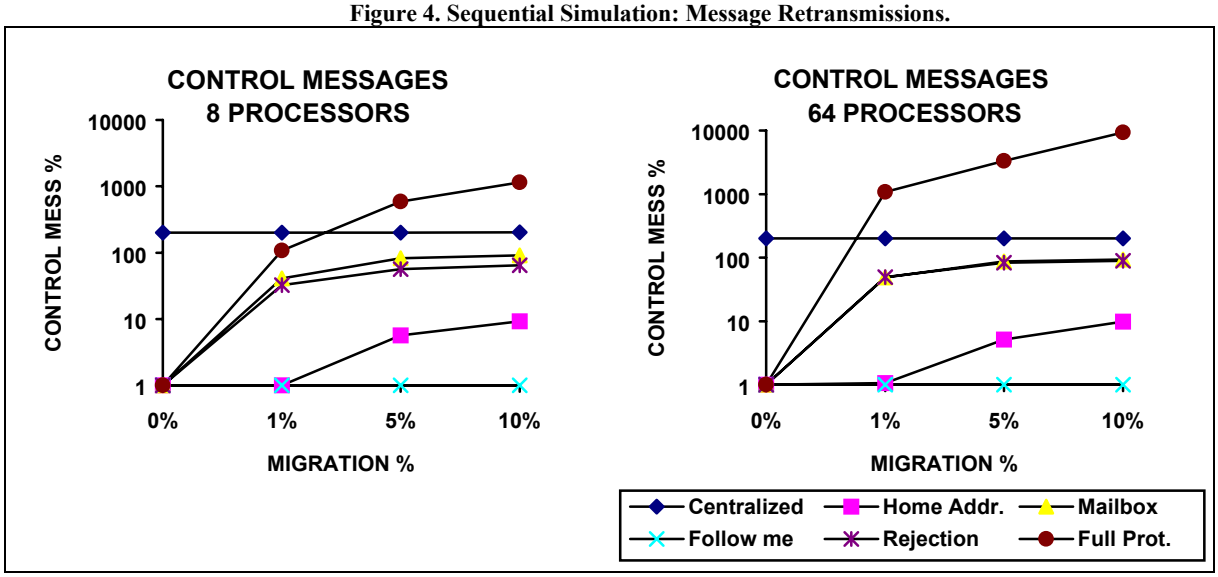

Figure 5. Sequential Simulation: Control messages

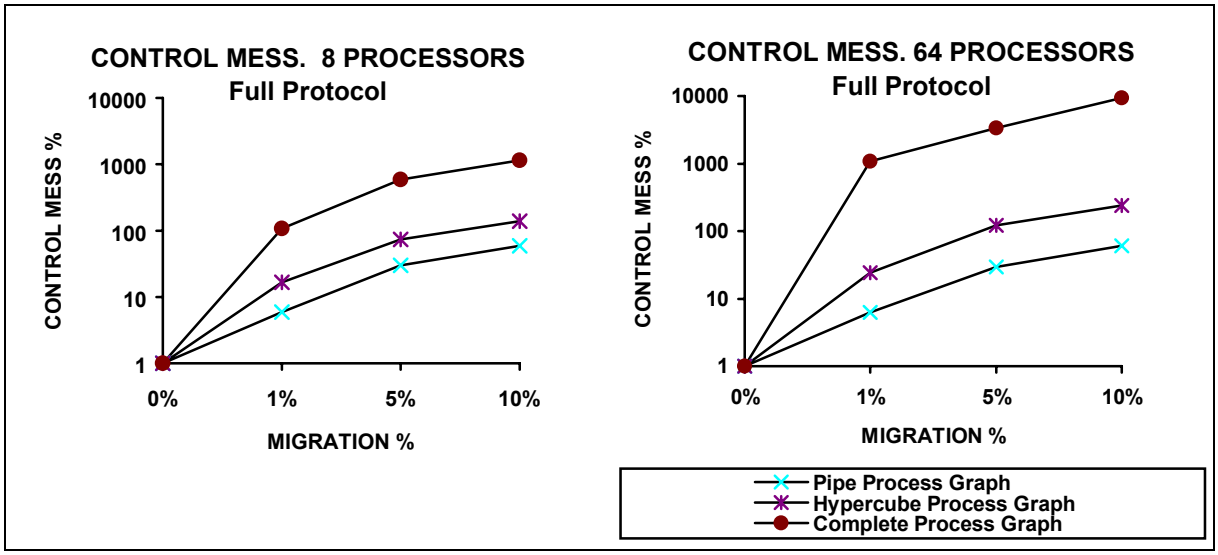

Figure 6. Sequential Simulation. Full protocol algorithm: Control messages.

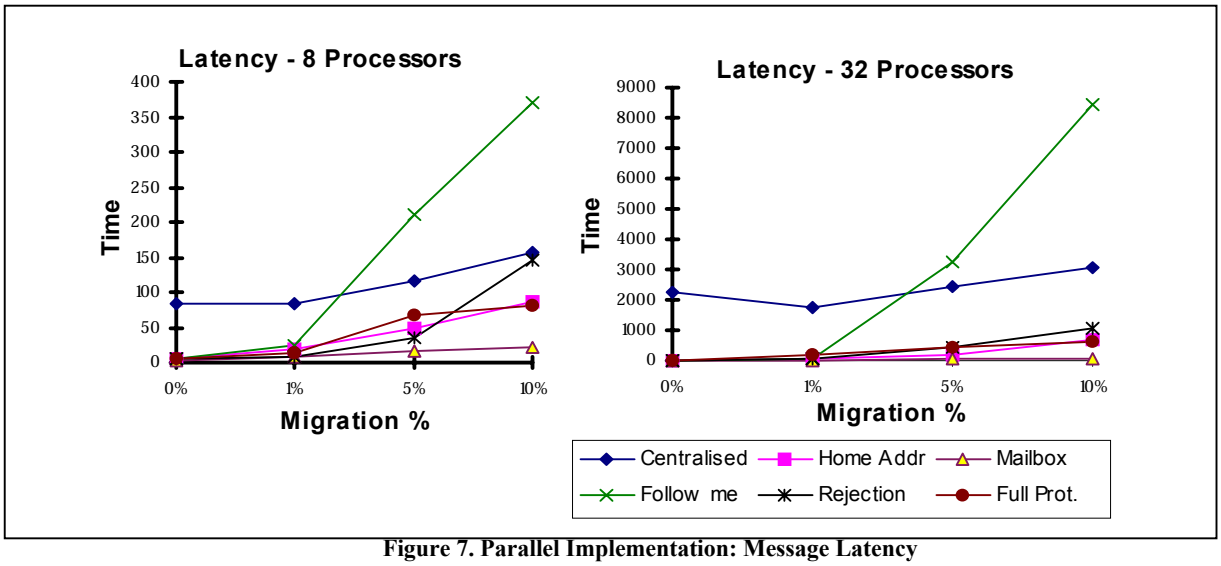

Figure 7. Parallel Implementation: Message Latency

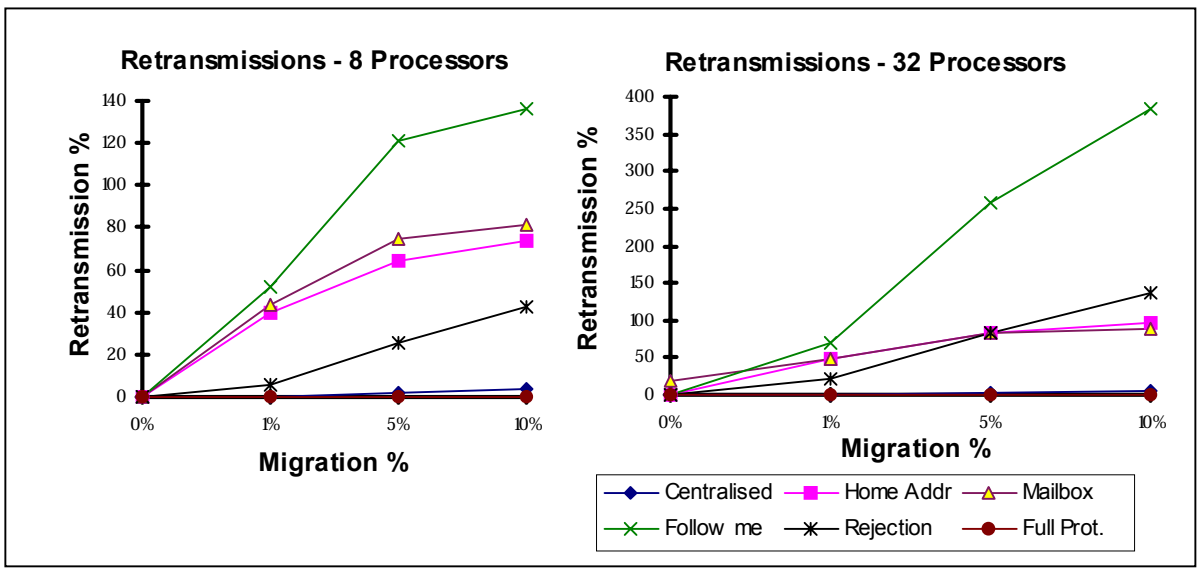

Figure 8. Parallel Implementation: Message Retransmissions

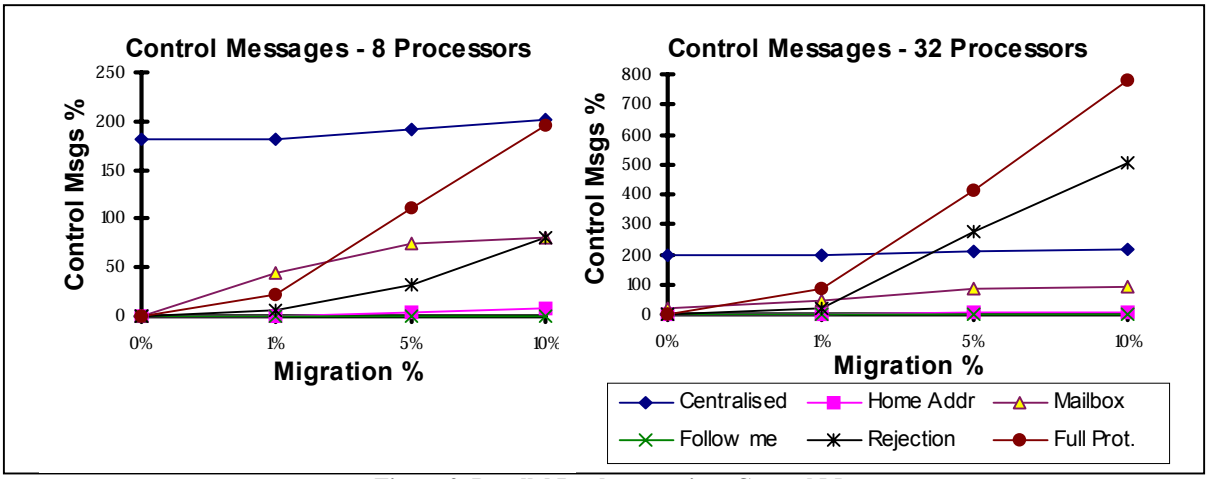

Figure 9. Parallel Implementation: Control Messages

5. Analysis of Results

With respect to the message latency, four different types of behaviour were obtained in the sequential simulation: The Full Protocol algorithm presents a latency equal to the communication time of sending a message from one process to another; the Centralised Scheme presents a constant latency, independently of the percentage of migration, which is comparatively high in the case of 8 processors, but the difference decreases when adding processors; the latency of the Follow Me algorithm gets worse when adding processors and when increasing the percentage of migration; and the Send First to Home Address, Message Rejection and Migration Mailbox algorithms have a similar behaviour, which consists of having a low latency when the percentage of migration is low, and a saturation point, after which the latency increases.

In the parallel implementation the message latency is strongly affected by the network load. As message latency depends on message retransmissions, the Follow Me algorithm results are the worst in almost all situations. The sequential simulation as well as the parallel implementation show an increment in the latency when processors are added (fig. 3 and 7). However, the scalability of the policies in the parallel implementation is worse than the observed in the sequential simulation. Taking as departure point the measures with 8 processors, the message latency when using 4 times more processors in the parallel implementation is increased by a factor of between 10 and 15 depending on the policy. On the other hand, the sequential simulation shows that using 8 times more processors decreases performance in the message latency by a factor of between 1.5 and 2.

Looking at the network load produced by each policy, there is a trade-off between retransmissions and control messages. For a relatively low number of processors, the number of control messages generated in the Centralised Scheme is initially higher than the number of control messages in any other policy. However, even in the case of a low number of processors and user processes, the number of control messages generated by the Full Protocol tends to grow exponentially in relation to the migration rate (fig. 5 and 9). On the other hand, the Full Protocol policy does not generate any retransmission (fig. 4 and 8). As the number of control messages generated in the Full Protocol depends strongly on the user processes interconnection, figure 6 shows this relation for different process-interconnection graphs.

Good scalability is found for message retransmissions in all algorithms. Message retransmissions increase in the parallel implementation for 32 processors compared with the values obtained for 8 processors, but this increase is not relevant. On the contrary, the number of control messages for the Full Protocol policy increases exponentially with the number of processors. Different behaviour is found in the Reject Message policy for 8 and 32 processors in the parallel implementation.

The Full Protocol has the highest reaction time. Depending on the implementation, within the Centralised Scheme and the Send First to Home Address, the new location of the migrating process has to be notified before it can be migrated. Otherwise, the new location is sent after the migration. In the Migration Mailbox, the migrating user process must wait for the messages it has asked its migration mailbox for. Usually, there is at most only one pending reception, and then one message to wait before the migration can be done. The user processes in the Follow Me and the Message Rejection policies react immediately. Many steps have to be performed since a user process is chosen for migration until it is sent to another processor in the Full Protocol policy. Measures taken in the parallel implementation show that the reaction time for the Full Protocol varies between the time taken for 20 and 30 user messages.

If an algorithm does not modify the network communication pattern adequately, it will not be able to perform communication balancing. The Full Protocol algorithm, after the information exchange step in which a depending-on-the-connection number of messages are generated, modifies the communication pattern. The Message Rejection algorithm changes the communication pattern too. On the other hand, the communication pattern remains nearly the same in the Follow Me policy. Independently of the location of a migrated user process, the Send First to Home Address and Migration Mailbox policies maintain the load around the home processor and migration mailbox processor respectively. The Centralised Scheme produces a hot spot around the Central Server.

6. Conclusions and Future Work

Dynamic load balancing and communication balancing in DMPC implies having a process migration mechanism. An algorithm which guarantees that processes receive messages independently of its network location must be provided. We have studied by sequential simulation and implemented in a parallel machine, six algorithms with different characteristics to handle this problem, which we have called the Message Integrity Problem.

The sequential simulation gave us an idea of the advantages and disadvantages of each intrinsic algorithm in different situations with respect to the performance parameters we have considered.

The parallel implementation allowed us to consider more parameters, such as network load, and to obtain more accurate results, by considering real conditions of migration and message traffic.

None of the basic algorithms has had good enough results for all possible situations. Nevertheless we initially discarded some algorithms like Full Protocol, Follow Me and Centralised Scheme. New algorithms combining the advantages of the proposed algorithms must be developed and studied. Indexes that will allow us to better characterise the behaviour of the algorithms are currently being developed. Furthermore, we are developing a formal model for all the different algorithms that will allow us to predict results beyond simulation.

7. References

[1] Luque, E; Ripoll, A; Cortès, A; Margalef, T. **A Distributed Diffusion Method for Dynamic Load Balancing on Parallel Computers.** Proceedings of the EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society, January, 1995.

[2] Smith,J.**A survey of Process Migration Mechanisms.** Operating Systems Review, ACM SIGOPS, July 1988.

[3] Zhu, W.; Goscinski,A.; Gerrity,G. **Process Migration in RHODOS.** Australian Defense Force Academy, Canberra, 1990.

[4] Casas, J.; Clark, D.; Konuru, R.; Otto, S. **MPVM: A migration Transparent Version of PVM.** Computing Systems, vol. 8, no 2, pp. 171-216, Spring 1995.

[5]Mascarenhas, E.; Rego, V. **Ariadne: Architecture of a Portable Threads System Supporting Thread Migration.** Software Practice and Experience, vol. 26 pp. 327-356. March 1996.

[6] Lenoski, D; Weber, W. **Scalable Shared-Memory Multiprocessing.** Morgan Kaufmann Publishers, 1995.

[7] Luque, E.; Franco, D.; Heymann, E.; Moure, J.C. **TransComm: A Communication Microkernel for Transputers.** Proc. of the Fourth Euromicro Workshop on Parallel and Distributed Processing. IEEE Computer Society Press, January 22-26, 1996, pp. 147-153.

[8] Heymann, E. **Support for Process Migration in Distributed Memory Parallel Computers.** Master Thesis. Universitat Autònoma de Barcelona, 1995.