



PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE  
ESCUELA DE INGENIERIA

# Distributed Mutual Exclusion and Thread Migration on DSM Systems

**FEDERICO MEZA MONTOYA**

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Doctor of Science in Engineering

Advisor:

**YADRAN ETEROVIC S. (ALVARO E. CAMPOS+)**

Santiago de Chile, March, 2007



PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE  
ESCUELA DE INGENIERIA  
Departamento de Ciencia de la Computación

# Distributed Mutual Exclusion and Thread Migration on DSM Systems

**FEDERICO MEZA MONTOYA**

Members of the Committee:

**YADRAN ETEROVIC S.**

**DAVID FULLER P.**

**MARCELO ARENAS**

**JOSÉ M. PIQUER**

**SELIM G. AKL**

**JOSÉ E. FERNÁNDEZ L.**

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Doctor of Science in Engineering

Santiago de Chile, March, 2007

*To Alvaro*

## ACKNOWLEDGMENTS

I am greatly indebted to my friend and former advisor, Alvaro Campos. His example as a teacher and researcher, but most important as a person, will live in everyone of his students. This work is dedicated to him, and although he could not see it finished, I am sure he is happy and proud.

I would like to thank the people of the GSS (Systems Software Group) at the PUC, specially to Cristian Ruz, who was beside me until I finished my work.

Thanks to my family, for their unconditional love and for the sacrifices they made to let me make the most of my opportunities. Thanks to my wife, Claudia (Caro), for the faith she putted in me, and for being there when I needed her.

I wish to thank all people in the DCC at the PUC, faculty, graduate students and staff, they all gave me a warm welcome to Chile and a pleasant stay. Thanks Yadran, for helping me finishing my work and for your encouraging words. Thanks Sole for all your help, but specially for your friendship.

Thanks to my colleagues at the Universidad de Talca, for their support and for covering for me while I was working on my thesis.

Finally, I wish to thank professor Selim Akl, for his valuable help as a reader of this thesis.

This work was supported by Fondecyt under Grant 2990074.

Federico Meza

## TABLE OF CONTENTS

	Page
<b>DEDICATORY</b> . . . . .	<b>I</b>
<b>ACKNOWLEDGMENTS</b> . . . . .	<b>II</b>
<b>TABLE OF CONTENTS</b> . . . . .	<b>III</b>
<b>LIST OF FIGURES</b> . . . . .	<b>V</b>
<b>LIST OF TABLES</b> . . . . .	<b>VI</b>
<b>RESUMEN</b> . . . . .	<b>VII</b>
<b>ABSTRACT</b> . . . . .	<b>IX</b>
<b>I. INTRODUCTION</b> . . . . .	<b>11</b>
I.1. Background and related work . . . . .	15
I.1.1. Distributed shared memory systems . . . . .	15
I.1.2. Multithreading and thread migration . . . . .	17
I.1.3. Distributed mutual exclusion . . . . .	19
I.2. DSM-PEPE Related Work . . . . .	20
I.3. Structure of the document . . . . .	21
<b>II. DSM SYSTEMS: DESIGN AND IMPLEMENTATION ISSUES</b> <b>23</b>	
II.1. Introduction . . . . .	23
II.2. Design Issues . . . . .	24
II.2.1. Pages and Consistency . . . . .	24
II.2.2. Underlying Message Passing . . . . .	26
II.2.3. Application View and Memory Regions . . . . .	27
II.2.4. Multithreading and Migration . . . . .	28
II.2.5. Synchronization . . . . .	30
II.2.6. Layered Architecture . . . . .	30
II.3. Implementation Issues . . . . .	32

II.3.1. Portable Modules . . . . .	32
II.3.2. Non-portable Modules . . . . .	34
II.4. Preliminary Results . . . . .	35
II.5. Concluding Remarks . . . . .	37
<b>III. MULTITHREADED DISTRIBUTED MUTUAL EXCLUSION</b>	<b>39</b>
III.1. Introduction . . . . .	39
III.2. The algorithm . . . . .	40
III.2.1. System model . . . . .	40
III.2.2. Brief description . . . . .	41
III.2.3. Detailed description . . . . .	43
III.2.4. A variant on the proposed algorithm . . . . .	47
III.3. Proof outline . . . . .	50
III.4. Performance . . . . .	51
III.4.1. Simulation model . . . . .	51
III.4.2. Results . . . . .	52
III.5. Related work . . . . .	56
III.6. Concluding Remarks . . . . .	56
<b>IV. THREAD MIGRATION TO EXPLOIT DATA LOCALITY</b>	<b>58</b>
IV.1. Introduction . . . . .	58
IV.2. Multithreading and Thread Migration Issues . . . . .	59
IV.3. DSM-PEPE Thread Migration Mechanism . . . . .	60
IV.4. Experiments and Results . . . . .	62
IV.5. Related work . . . . .	69
IV.6. Concluding remarks . . . . .	70
<b>V. CONCLUSIONS</b>	<b>71</b>
<b>BIBLIOGRAPHY</b>	<b>73</b>

## LIST OF FIGURES

	Page
Figure I.1. Matrix Multiplication on a Shared-Memory Parallel Environment . . . . .	12
Figure I.2. Matrix Multiplication on a Message-Passing Environment . .	13
Figure II.1. DSM system architecture . . . . .	29
Figure II.2. Objects inside a program . . . . .	31
Figure II.3. Speedup for two problem sizes running on Linux and Windows	36
Figure III.1. Behavior of the algorithm when servicing several requests . .	42
Figure III.2. Behavior of the modified algorithm when servicing several requests . . . . .	49
Figure III.3. Performance comparison of the 3 algorithms with a single thread per node. . . . .	53
Figure III.4. Performance comparison of the 3 algorithms with 5 threads per node. . . . .	55
Figure III.5. Performance comparison of the 3 algorithms with 10 threads per node. . . . .	55
Figure IV.1. DSM-PEPE thread structure . . . . .	61
Figure IV.2. Thread suspension and activation during a migration . . . .	62
Figure IV.3. Sequential N-body: Execution time grows as $n^2$ . . . . .	64
Figure IV.4. Parallel and Migratory-Threads outperform Sequential N-body with speedups 3,77 and 3,99, respectively . . . . .	65
Figure IV.5. Speedup for the Parallel and Migratory-Threads Applications	66
Figure IV.6. Messages and data exchanged when computing 4 steps for $2^{17}$ particles . . . . .	68
Figure IV.7. Messages sent by each processor when computing 4 steps for $2^{17}$ particles . . . . .	68

## LIST OF TABLES

	Page
Table IV.1. Time involved in message transmission . . . . .	67
Table IV.2. Time involved in thread migration . . . . .	67

PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE  
ESCUELA DE INGENIERIA  
DEPARTAMENTO DE CIENCIA DE LA COMPUTACION

---

## EXCLUSIÓN MUTUA DISTRIBUIDA Y MIGRACIÓN DE HEBRAS EN SISTEMAS DE MEMORIA COMPARTIDA DISTRIBUIDA

Tesis enviada a la Dirección de Investigación y Postgrado en cumplimiento parcial de los requisitos para el grado de Doctor en Ciencias de la Ingeniería por

**FEDERICO MEZA MONTOYA**

### RESUMEN

Los sistemas de *Memoria Compartida Distribuida* (MDC) proveen un ambiente paralelo para aprovechar el poder computacional disponible en las redes de computadores. Ofrecen un espacio virtual de memoria compartida sobre un sistema de memoria distribuida, explotando la escalabilidad y bajo costo de un multicomputador, y la facilidad de programación de un multiprocesador de memoria compartida.

En este trabajo presentamos DSM-PEPE, un sistema de MCD multihebra que se ejecuta sobre dos plataformas diferentes. DSM-PEPE fue diseñado como una plataforma de experimentación para aspectos relacionados con MCD, y como un ambiente para computación paralela. Nuestro sistema implementa coherencia de memoria mediante dos protocolos diferentes: *consistencia secuencial* y *consistencia de entrada*. Las aplicaciones pueden ser multihebra, utilizando una biblioteca de nivel de usuario. Dos características destacables de DSM-PEPE son: (1) un mecanismo para garantizar exclusión mutua distribuida, y (2) un mecanismo de migración de hebras diseñado para mejorar el desempeño de las aplicaciones explotando la localidad de los datos.

Para garantizar un acceso sincronizado a los recursos compartidos se requiere un mecanismo de exclusión mutua distribuida. Sin embargo, la mayoría de los algoritmos que se han propuesto no son apropiados para aplicaciones multihebra, en las que puede generarse más de una solicitud de exclusión mutua desde un mismo nodo. En esta tesis presentamos un nuevo algoritmo basado en *token*, para proveer exclusión mutua distribuida en sistemas multihebra. Nuestro algoritmo se basa en hebras de propósito

general, llamadas *alien threads*, que se ejecutan en un procesador en representación de hebras que se están ejecutando en otros procesadores. El algoritmo utiliza un árbol para dirigir las peticiones por el *token*. Los resultados obtenidos mediante un estudio basado en simulación muestran que nuestro algoritmo tiene un desempeño muy bueno en la presencia de una alta carga de peticiones, y un desempeño aceptable cuando la carga es baja.

Los sistemas de MCD se basan en migración de datos para hacer que los datos estén disponibles para las hebras en ejecución. El mecanismo de migración de hebras de DSM-PEPE fue diseñado como una alternativa para este paradigma de migración de datos. A las hebras se les permite migrar de un nodo a otro, como lo requiera la computación. Los resultados de los experimentos efectuados muestran la factibilidad del mecanismo de migración de hebras de DSM-PEPE, como una alternativa para mejorar el desempeño de las aplicaciones aumentando la localidad espacial de los datos.

**Miembros de la Comisión de Tesis Doctoral:**

**Yadran Eterovic S.**

**David Fuller P.**

**Marcelo Arenas**

**José M. Piquer**

**Selim G. Akl**

**José E. Fernández L.**

**Santiago de Chile, March, 2007**

**DISTRIBUTED MUTUAL EXCLUSION AND THREAD  
MIGRATION ON DSM SYSTEMS**

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Doctor in Engineering Sciences by

**FEDERICO MEZA MONTOYA**

**ABSTRACT**

*Distributed Shared Memory* (DSM) systems provide a parallel environment to harness the available computing power of computer networks. DSM systems offer a virtual shared memory space on top of a distributed-memory multicomputer, featuring the scalability and low cost of a multicomputer, and the ease of programming of a shared-memory multiprocessor.

We present DSM-PEPE, a multithreaded DSM system that runs on two different environments. DSM-PEPE was designed as an experimentation platform for DSM related topics and as an environment for parallel computing. Our system implements memory coherence through two different protocols: *sequential consistency* and *entry consistency*. Application-level multithreading is implemented through a user-level library. Two remarkable features of DSM-PEPE are: (1) a mechanism for multithreaded distributed mutual exclusion, and (2) a thread migration mechanism designed to improve system performance by exploiting data locality.

To provide synchronized access to shared resources, a distributed mutual exclusion mechanism is required. However, most of the algorithms that have been proposed are not suitable for multithreaded applications where more than one mutual exclusion request could be issued from a single node. We present a novel token-based algorithm to provide distributed mutual exclusion on multithreaded systems. Our algorithm relies on special-purpose *alien threads* running at host processors on behalf of threads running at other processors. The algorithm uses a tree to route requests for the token.

Results from a simulation study show that our algorithm performs very well under a high load of requests while delivering acceptable performance under a light load.

DSM systems rely on data migration to make data available to running threads. The thread migration mechanism of DSM-PEPE was designed as an alternative to this data migration paradigm. Threads are allowed to migrate from one node to another, as needed by the computation. We show by experimentation the feasibility of the thread migration mechanism of DSM-PEPE as an alternative to improve application performance by enhancing spatial locality.

**Members of the Doctoral Thesis Committee:**

**Yadran Eterovic S.**

**David Fuller P.**

**Marcelo Arenas**

**José M. Piquer**

**Selim G. Akl**

**José E. Fernández L.**

**Santiago de Chile, Marzo, 2007**

## I. INTRODUCTION

Parallel computing aims to reduce execution time for applications involving high computational needs. This is accomplished by distributing computation among several processors working in parallel.

Multiprocessors are built from several processing units sharing a common memory. Processors communicate with each other by writing on and reading from this shared memory. Synchronization is achieved by shared variable-based mechanisms –like semaphores– commonly implemented in hardware.

Multicomputers are loosely coupled collections of processors, in the sense they have a distributed memory. Processors communicate and synchronize by sending messages through a network. Programs request the shared data they need and send the data requested to them. Several well-known libraries help programmers to write distributed message-passing programs. PVM (Geist et al., 1994) is one of such libraries, while MPI (Pacheco, 1997) has become a message-passing standard, in several platforms.

Multiprocessors are expensive and offer low scalability, but it is easy to write programs for them due to their simple semantics for sharing data. Multicomputers are built using conventional hardware and show better scalability, but communication is slow and programming is harder, since programs must partition the shared data and manage communication between the distributed memories.

Consider the parallel program shown in Figure I.1 to compute the multiplication of two  $n \times n$  matrices on a shared-memory parallel environment. We use the notation for concurrent programming proposed by Andrews (Andrews, 1991). Each process

```

var A[1:n,1:n]:int, B[1:n,1:n]:int, C[1:n,1:n]:int := ([n,n]0)

Process[pid:1..n]::
  var i:int, j:int

  if (pid = 1)
    call init();
  fi

  barrier(pid);

  fa i:=1 to n, j:=1 to n
    C[pid,i] := C[pid,i] + A[pid,j] * B[j,pid]
  af

```

Figure I.1 Matrix Multiplication on a Shared-Memory Parallel Environment

computes one row of the result. Hence, we need  $n$  processes working in parallel. Note that communication is accomplished by reading and writing the shared matrices. No synchronization is required because there are no data race conditions involved.

On the other hand, consider the message-passing-based program shown in Figure I.2 to solve the same problem. In this case, the program –specifically Process 1– must deal with data distribution and results collection. Also, processes 1 through  $n$  must deal with receiving initial data and sending final results.

*Distributed Shared Memory* (DSM) systems provide the scalability and low cost of a multicomputer, and the ease of programming of a shared-memory multiprocessor (Li and Hudak, 1989). DSM systems offer a virtual shared memory space on top of a

```

chan channel[1:n](int)
Process[1]::
  var A[1:n,1:n]:int, B[1:n,1:n]:int, C[1:n,1:n]:int := ([n,n]0)
  var i:int, j:int, pid:int

  call init();
  fa pid:=2 to n, j:=1 to n
    send channel[pid](A[pid,j])
    fa i:=1 to n
      send channel[pid](B[i,j]);
    af
  af
  fa i:=1 to n, j := 1 to n
    C[1,i] := C[1,i] + A[1,j] * B[j,1]
  af
  fa pid:=2 to n, j:=1 to n
    receive channel[pid](C[pid,j])
  af
Process[pid:2..n]::
  var A[1:n]:int, B[1:n,1:n]:int, C[1:n]:int := ([n]0)
  var i:int, j:int, k:int

  fa j:=1 to n
    receive channel[pid](A[j])
    fa i:=1 to n
      receive channel[pid](B[i,j])
    af
  af
  fa i:=1 to n, j:=1 to n
    C[i] := C[i] + A[j] * B[j,pid]
    send channel[1](C[i])
  af

```

Figure I.2 Matrix Multiplication on a Message-Passing Environment

distributed-memory multicomputer. This virtual space enables programs on different computers to share memory, even though the processors do not share memory at all. All the communication involved is accomplished by the underlying DSM system so it is hidden from application programs. Despite the overhead introduced by the DSM layer, experimentation shows that applications still perform within acceptable bounds (Lu et al., 1997).

In this thesis we present a software DSM system called DSM-PEPE. This name stands for *Distributed Shared Memory – Parallel Environment for Program Execution*. One of the main issues considered in the design of the system was portability. DSM-PEPE runs on top of GNU/Linux and MS-Windows boxes, sharing most of the code. Only a few assembly language lines, and some specific instructions to handle exceptions depend on the operating system underneath. DSM-PEPE implements two consistency protocols: sequential consistency (Lamport, 1979), and entry consistency (Bershad and Zekauskas, 1991). Application multithreading is implemented through a user-level library. Threads are allowed to migrate from one node to another, as needed by the computation.

To provide synchronized access to shared resources, a thread-level distributed mutual exclusion mechanism is required. DSM-PEPE implements a novel algorithm for mutual exclusion (Meza et al., 2005). Several per-node requests for a single token could be issued by threads running at each node. Our algorithm relies on special-purpose *alien threads* running at host processors on behalf of threads running at other processors. The algorithm uses a tree to route requests for the token. We present a performance simulation study comparing two versions of our algorithm with a known algorithm. Results show that our algorithm performs very well under a high load of requests while delivering acceptable performance under a light load.

We show by experimentation the feasibility of the thread migration mechanism as a means to improve application performance by enhancing spatial locality.

## I.1 Background and related work

### I.1.1 Distributed shared memory systems

DSM systems emulate shared-memory programming semantics on top of a message-passing distributed environment. We are interested in DSM systems that are implemented exclusively at the software level, excluding those systems that require special network or caching hardware. Some software DSM systems may require modifying the compiler to generate sharing and coherence code. These systems are not considered in this study because they lack portability. We focus on systems supported exclusively by user-level software, using the virtual-memory management primitives of the operating system (Lo, 1994). As a consequence, sharing granularity is handled at the page level.

The original model proposed by Li and its first implementation –Ivy– were page-based (Li, 1986, Li and Hudak, 1989). Other systems are based on shared variables, *e.g.*, Munin (Bennett et al., 1990), or based on shared objects, *e.g.*, Linda (Carriero and Gelernter, 1989). Page-based DSM systems suffer from *false sharing* (Carter et al., 1997). However, their implementation can take advantage of the paging hardware, reducing the overhead induced by the middleware.

Several techniques have been used in order to improve the performance of DSM systems (Carter et al., 1997). Relaxed consistency models reduce communication by delaying the propagation of updates (Hutto and Ahamad, 1990). To avoid false

sharing produced by page sharing granularity, multiwriter protocols can be used (Keleher, 1996). Multithreading helps to reduce communication latency by keeping the processors busy while servicing remote requests (Thitikamol and Keleher, 1998). Prefetching is another technique aimed to reduce communication latency by predicting the future memory accesses over a page, and getting that page in advance, anticipating the actual data access and hence avoiding page faults (Bianchini et al., 1998, Pinto et al., 2003, Ruz and Piquer, 2005). Finally, thread migration has been proposed to improve data locality (Thitikamol and Keleher, 1999), (Itzkovitz et al., 1998). Some of these approaches are studied as a part of this work.

Page-based DSM system maintain cached copies of pages to reduce the effect of remote accesses. These replicas introduce a memory coherence problem that must be addressed by a consistency protocol enforcing a consistency model. The consistency model states the behavior of shared-memory updates, that is, when and how these updates become visible to other processors in the system. In particular, the model specifies the values that may be returned by read operations executed on the DSM system. Early systems –like Ivy– feature a strict consistency model known as *sequential consistency* (Lamport, 1979). Under this model, updates to the shared memory become visible to other processors in the exact same order. Consistency protocols introduce some degree of overhead due to the messages that must be sent to implement coherence as well as to delays produced by the execution of the protocol (Lo, 1994).

Relaxed consistency models help to improve system performance by reducing the number of constraints so the number of update and invalidate messages reduces considerably. Several models have been proposed. *PRAM consistency* serializes writes made by each processor, so updates made by one processor are made visible to other

processor in the same order (Lipton and Sandberg, 1988). However, updates from different processors may be seen in distinct orders at different processors. *Causal consistency* makes updates visible accordingly to a causality relation among memory references (Hutto and Ahamad, 1990). *Entry consistency* associates variables with locks and delays updates made within a critical section protected by a lock, until the next time the lock is acquired by another processor (Bershad and Zekauskas, 1991). *Release consistency* propagates memory updates at the time of release of a lock (Gharachorloo et al., 1990). *Lazy release consistency* is a lazy implementation of release consistency that delays updates a little further (Keleher et al., 1992). *Scope consistency* is based on the concept of *consistency scopes*; updates made in a scope are made visible only within that scope (Iftode et al., 1996).

Other well known DSM systems are Treadmarks (Keleher, 1997), distributed under a commercial license; CVM (Keleher, 1997), that runs under several versions of Unix; and Brazos (Speight and Bennett, 1997), that runs under MS-Windows.

DSM-PEPE implements sequential consistency and entry consistency. Applications associate a consistency protocol to regions in the distributed shared memory space.

### **I.1.2 Multithreading and thread migration**

Multithreading helps to improve system performance by overlapping communication and computation. Every time a thread is blocked because it issued an access request to a remote location, another thread may progress at the local processor. Even though the cost involved in the context-switch, processing and transmission of the request and reply messages introduce the higher costs (Thitikamol and Keleher, 1998).

Thread migration allows load balancing and dynamic reconfiguration. In a DSM system, it can be used to enhance data locality by moving threads to the data they need (Thitikamol and Keleher, 1999, Jenks and Gaudiot, 2002). Multithreading and thread migration in DSM systems has been studied in the past. DSM-Threads is a runtime system supporting an API for POSIX Threads (Mueller, 1997). DSM-Threads supports several memory consistency models and uses decentralized algorithms for consistency and synchronization. There is no support for thread migration.

Some systems, like Emerald (Jul et al., 1988), Arachne (Dimitrov and Rego, 1998), and Nomadic Threads (Jenks and Gaudiot, 2002, 2003, Jenks, 2004) rely on language and compiler support to implement the migration mechanism. Because of this, these systems lack portability and lie beyond the scope of this work.

Ariadne is a user-level thread library for multiprocessors and distributed memory systems (Mascarenhas and Rego, 1996). It supports interprocess thread migration to access remote data. When a thread migrates, its stack is inspected to identify and update missing pointers.

Amber is an object-oriented DSM system implementing thread migration. Object location is handled explicitly by the application and the system requires a large address space (Chase et al., 1989). MigThread uses preprocessing and run-time support to implement migration (Jiang and Chaudhary, 2002*a,b,c*). Nomad is a light-weight thread migration system that delays the sending of the complete stack (Milton, 1998).

Millipede is a DSM system for MS-Windows that implements multithreading at the kernel level. Its thread migration policy attempts to enhance data locality by evaluating the page access history at execution time (Itzkovitz et al., 1998, Friedman et al., 1997).

Our system –DSM-PEPE– is a DSM system for MS-Windows and GNU-Linux. Multithreading is implemented at application-level. Threads are allowed to migrate to enhance data locality.

### I.1.3 Distributed mutual exclusion

Synchronization is a key issue when programming a system with shared-memory semantics. There are only a few works addressing synchronization in a distributed multithreaded environment. Our main contribution in this area is an algorithm to provide multithreaded distributed mutual exclusion. This algorithm was used to implement thread-level locks in DSM-PEPE.

Mutual exclusion provides synchronized access to shared resources ensuring that, at any time, at most one process can be executing in its critical section. Distributed mutual exclusion algorithms focus on mutual exclusion on distributed environments lacking shared memory. Several algorithms address the distributed mutual exclusion problem when only one process is running at each processor. Multithreaded distributed systems allow the existence of several threads of execution within each process. Thus, we need to provide mutual exclusion to a large number of distributed threads.

Distributed mutual exclusion algorithms can be classified as *permission-based* or *token-based* (Raynal, 1991). We focus our study on distributed algorithms, excluding those algorithms that use a central coordinator.

When a permission-based algorithm is used, a process wishing to enter its critical section must obtain permission from a subset of previously defined processes. A process receiving such a request grants its permission immediately if it is not interested

in entering its critical section. Otherwise, some policy must be used to resolve the conflicting requests. The algorithms proposed by Ricart and Agrawala (Ricart and Agrawala, 1981), by Maekawa (Maekawa, 1985), and by Agrawala and El Abbadi (Agrawala and Abbadi, 1989) fall into this category.

Token-based algorithms rely on a unique token which must be acquired by a process wishing to enter its critical section. The token could be traveling from one process to another continuously or could be obtained by sending a request. The algorithms proposed by Raymond (Raymond, 1989), by Neilsen and Mizuno (Neilsen and Mizuno, 1991), by Banerjee and Chrysanthis (Banerjee and Chrysanthis, 1996), and by Naimi, *et al.* (Naimi et al., 1996) fall into this category.

Distributed mutual exclusion for multithreaded environments has not been studied extensively. Mueller presents the design and implementation of distributed synchronization primitives, focusing on the impact of multithreading on synchronization (Mueller, 2000). Distributed mutual exclusion relies on a token-passing mechanism, based on the algorithm described by Naimi, *et al.* (Naimi et al., 1996).

## **I.2 DSM-PEPE Related Work**

Several related works involving DSM-PEPE have recently been developed by people in our research group.

Cristian Ruz and José M. Piquer studied the effects on performance of a prefetching technique based on page-access histories on top of DSM-PEPE (Ruz and Piquer, 2005).

Jesus Figueroa and José M. Piquer used the migratory threads of DSM-PEPE to develop a parallel genetic algorithm following the multiple-population scheme using intelligent individuals (Figueroa and Piquer, 2005).

Jorge Pérez studied the development of algorithms for distributed mutual exclusion in environments allowing multiple requests per-node (Pérez, 2005). His work includes a simulation-based evaluation of the impact of multithreading on performance.

Jorge Pérez and Christian Orellana developed an algorithm for distributed mutual exclusion based on two known algorithms. Their algorithm benefits from the key features of the original algorithms while avoids most of their drawbacks (Pérez and Orellana, 2005).

Enrique Guadalupe developed a generic framework to extend existing distributed mutual exclusion algorithms to support multiple requests per-node (Guadalupe, 2005).

Karen Palma used an aspect-oriented approach aimed to improve the internal architecture of DSM-PEPE (Palma et al., 2004).

### **I.3 Structure of the document**

The rest of the document is organized as follows. Chapter II presents the main design and implementation issues involved in the construction of a DSM system. Specifically, the key features of DSM-PEPE are discussed, as well as some preliminary performance results. This chapter is based on a paper presented at the International Conference of Computational Science and its Applications – ICCSA’2003, and published in Lecture Notes in Computer Science (Meza et al., 2003).

Chapter III introduces an algorithm to implement distributed mutual exclusion on a multithreaded environments. A simulation-based study comparing our algorithm with two existing algorithms is presented. This chapter is based on a paper presented at the International Symposium and School on Advanced Distributed Systems – ISSADS’2005, and published in Lecture Notes in Computer Science (Meza et al., 2005).

Chapter IV presents the thread migration mechanism of DSM-PEPE and how this mechanism could be used to improve application performance by enhancing spatial locality. This chapter is based on a paper accepted at the International Conference on Algorithms and Architectures for Parallel Processing – ICA3PP’2007, to appear in Lecture Notes in Computer Science (Meza and Ruz, 2007).

Finally, Chapter V summarizes the results of our work and proposes future lines of research.

## II. DSM SYSTEMS: DESIGN AND IMPLEMENTATION ISSUES

### II.1 Introduction

*Distributed Shared Memory* (DSM) systems provide the scalability and low cost of a multicomputer, and the ease of programming of a shared-memory multiprocessor (Li and Hudak, 1989). They offer a virtual shared memory space on top of a distributed-memory multicomputer. This virtual space enables programs on different computers to share memory, even though the computers physically do not share memory at all. All the communication involved is accomplished by the underlying DSM system. Despite the overhead introduced by the DSM layer, applications still perform within acceptable bounds (Lu et al., 1997).

We are interested in DSM systems implemented exclusively by software, excluding those systems that require special network or caching hardware. Some software DSM systems may require modifying the compiler to generate coherence code, specially variable-based and object-based systems. These systems are not considered in this study because they lack portability. We focus on systems supported by user-level software, using the virtual-memory management primitives of the operating system (Lo, 1994). As a consequence, sharing granularity is handled at the page level.

In this paper, we outline the key issues involved in the design and implementation of a portable DSM system that runs on top of Linux and Windows. Special attention is given to the operating-system-independent aspects of the design. We claim that using a layered architecture it is possible to reduce the operating-system dependency to a minimum. We present DSM-PEPE, a DSM system aimed to use low-cost hardware

and conventional networks to provide a distributed parallel environment. Our system conforms a scalable, low-cost, distributed parallel machine that executes programs with shared-memory semantics. A few modules had to be rewritten, but most of the code remained the same between the different platforms. Preliminary results proved the feasibility of DSM-PEPE as a parallel machine.

## **II.2 Design Issues**

A software, page-based DSM system allows a collection of independent computers to share a single, paged, virtual address space. There are several issues that must be considered when designing such a DSM system. Some of them are particular to the underlying operating system, but most are platform independent.

In this paper, we show the design and implementation of a DSM system from this perspective. Modules are grouped in layers, in such a way that operating-system dependency is reduced. Our goal is to obtain a system that can be ported easily from one operating system to another, on top of the same hardware.

### **II.2.1 Pages and Consistency**

Pages are usually cached on several nodes, in order to increase performance. This replication introduces a coherence problem that is managed by a consistency protocol enforcing a particular consistency model. The consistency model states the memory behavior in the presence of conflicting accesses across processors (Adve and Gharachorloo, 1995). Relaxing the model allows better performance, under certain programming restrictions (Adve and Hill, 1990).

Consistency protocols must be kept separate from the page-management module. Otherwise, each time a protocol is included, it would be necessary to write page-management primitives for it. Some consistency protocols rely exclusively on the triggering of page-fault events, for example, sequential-consistency protocols. Protocols for relaxed consistency models usually take consistency actions when synchronization operations occur, for example, protocols for release and entry consistency.

Implementing a sequential-consistency protocol involves writing handlers for page-fault events on read and write operations and for serving requests from a remote processor. For example, a write fault triggers a local event that is managed by the local handler. In one approach, the handler obtains an up-to-date copy of the page and instructs the other processors to invalidate any cached copy. Invalidation notices and the request for the page are handled by the remote handler. Read faults produce similar scenarios.

Sharing granularity affects system performance (Torrellas et al., 1994). Multiwriter protocols reduce the impact of false sharing on performance when page granularity is used (Keleher, 1996), but they are difficult to implement. The simplest approach is to allow false sharing and to use an invalidation protocol as described above.

Page management is accomplished through the virtual-memory interface of the operating system. The system must be capable of changing the status of virtual pages in order to make them valid or invalid, as well as read-only or writable. A small set of operating-system-independent primitives must be available to the upper levels of the system, in order to make high-level layers portable.

Exception handling allows the DSM system to catch access faults to pages that require consistency management. Application programs run guarded by an exception

handler. Each time a page fault occurs, the DSM system evaluates the reasons and takes the necessary actions to resume the program.

### II.2.2 Underlying Message Passing

In a multicomputer, processors must communicate through message passing. It is possible to use PVM or a user-level library based on MPI, to avoid the use of low-level socket communication. When using sockets, it is desirable to hide operating-system dependencies introducing a higher-level layer. Hence, code built on top of this new layer can be used across platforms.

DSM systems exchange messages of two types. First, consistency-related messages, controlled by the consistency protocol. Second, synchronization-related messages to implement distributed locks and barriers. All data could be directed through a single socket, but this may produce interference between the two types of messages. A more portable approach involves the use of an abstraction built on top of sockets: a post-office object, which uses an exclusive socket to communicate with its peers. A few primitives are available through the post office, allowing sending and receiving point-to-point messages, as well as broadcasting.

Communication through sockets can be connectionless or connection oriented. Connectionless sockets are efficient for scattered communication. DSM systems show high rates of message exchange. Hence, connection-oriented sockets seem to be more suitable. However, connection setup introduces unwanted overhead; thus, it is desirable to keep connections open.

### II.2.3 Application View and Memory Regions

The DSM approach is more attractive than message passing to write applications for a multicomputer, since most programmers find shared memory easier to use. To sustain this fact, applications must see the global memory space, as they see the shared memory in a multiprocessor. The use of memory pointers is an easily understood and portable interface. Programmers are familiar with the use of dynamically allocated memory. This approach is suitable for implementing the DSM system, since addresses stored in pointers can be easily mapped from the process space to the system global space.

It is convenient to group related variables in regions. A region is a high-level abstraction used to provide flexibility to the application programmer. Memory allocated from a single region is managed by the same consistency protocol. However, pages contained in different regions can be handled by different protocols. When the consistency protocol does not match the data-access pattern of an application, performance can be degraded. It is desirable to have a set of protocols with particular characteristics and be able to use them as needed. Sometimes, it may be necessary to use more than one protocol in the same application. This facility can be useful when the access pattern of an application changes during its execution. Hence, it is possible to match dynamically these data-access patterns. In general, the use of regions allows specifying extra information for portions of the address space. For instance, the entry consistency model requires that programmers associate each shared variable with a lock (Bershad and Zekauskas, 1991). This association can be done by grouping together, in the same region, all related variables.

## II.2.4 Multithreading and Migration

Multithreading in DSM systems has been widely studied (Thitikamol and Keleher, 1999, Itzkovitz et al., 1998, Mueller, 1997). Application-level multithreading reduces remote latency in DSM systems, by overlapping communication and computation (Thitikamol and Keleher, 1998). Besides, multithreading helps to enhance program structure.

Threads can exist at kernel level or at user level. User-level threads are portable, flexible, and can be implemented efficiently. Only the thread-switching facility is actually dependent on the underlying hardware and operating-system platforms. In a DSM system there is a single address space. Also, there must be a single global thread space, and threads must be unique across processors. Synchronization should coordinate threads no matter where they are running.

Thread migration is another topic widely studied. In general, process migration allows load balancing and dynamic reconfiguration. In a DSM system, thread migration enhances data locality by moving threads to the data they need (Thitikamol and Keleher, 1999, Itzkovitz et al., 1998). This locality reduces consistency-related communication, but an effective migration policy must be implemented. In a DSM system, the primary goal is not load balancing, but to enhance data locality. In order to do so, data-access patterns of applications must be accounted for dynamically. Some DSM systems allow multithreading and thread migration at the application level, for example, Millipede (Friedman et al., 1997, Itzkovitz et al., 1998) and DSM-Threads (Mueller, 1997).

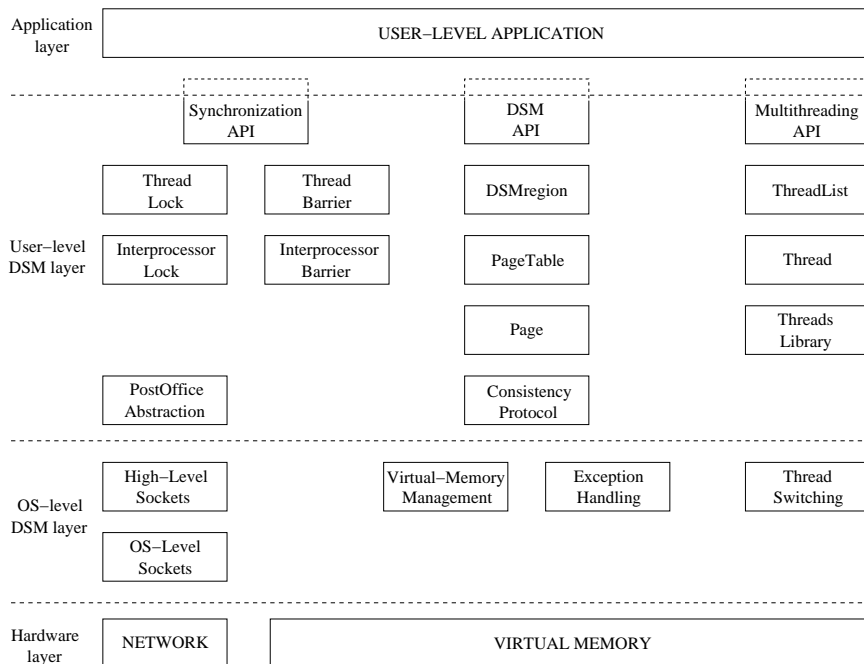


Figure II.1 DSM system architecture

Implementing thread migration requires considering several issues. We focus our study on computers having the same hardware architecture, running the same operating system. Hence, some problems do not arise, for example, data and address representation. Data-addresses correspondence is not a problem since shared data is stored in the global shared-memory space. Code-addresses correspondence is easily handled, by having all threads start at all processors, even if they are actually running in only one of them. Each running thread has a suspended peer in each of the other processors, waiting to receive it in the event of a migration. Stack migration can be easily accomplished, since threads are implemented at the user level. The threads library provides functionality to suspend a thread and recover its stack, as well as to resume a suspended thread with an updated stack.

## II.2.5 Synchronization

Synchronization is a key issue when programming a system with shared-memory semantics. At least two kinds of synchronization primitives are required: locks, to provide mutual exclusion, and barriers, to control interprocess progress.

In a multithreaded DSM environment, there are two levels of synchronization. At the bottom level, interprocessor synchronization allows processors to coordinate with each other. At the top level, and visible to the application program, thread synchronization allows global threads from the application program to coordinate with each other.

Thread synchronization can be built on top of interprocessor synchronization. For example, thread barriers must block arriving threads until all threads from all processors have arrived to the barrier. One possible implementation provides local barriers within each processor. When all local threads have arrived, the processor notifies a global interprocessor barrier about the arrival. The global barrier blocks the processor until all processors have notified their arrival. A release notice from the global barrier triggers the release of all blocked local threads in each processor.

## II.2.6 Layered Architecture

Fig. II.1 shows the suggested layered architecture for a portable DSM system. Interaction is accomplished through three *Application Programming Interfaces* (API). The synchronization API provides lock and barrier synchronization to global threads. These primitives can be implemented on top of the interprocessor synchronization functionality. The DSM API allows applications to create shared-memory regions

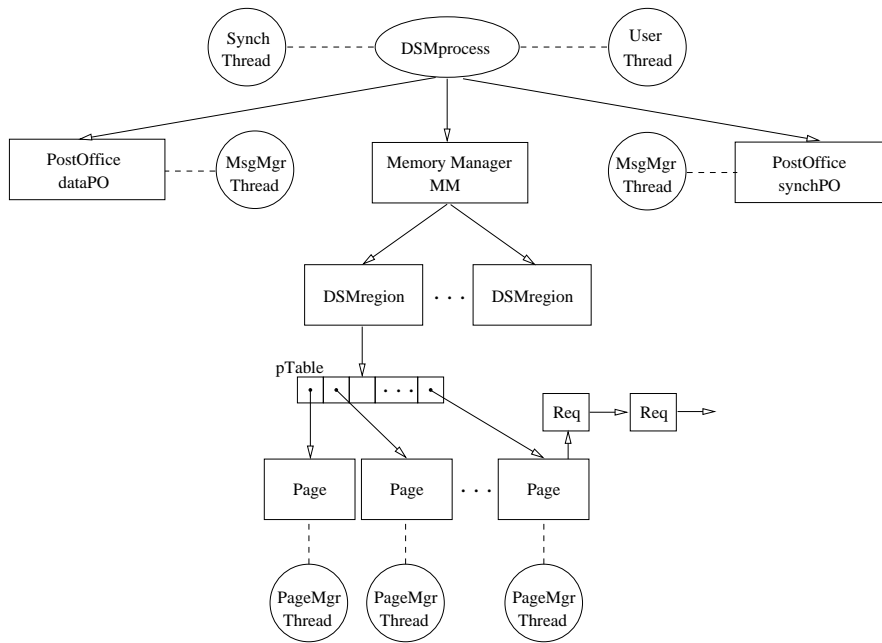


Figure II.2 Objects inside a program

and to allocate memory from them. Lower-layer modules implement page functionality and coherence. The consistency protocol is kept apart from all other modules, making it independent, easily replaceable, and allowing the coexistence of different protocols on the same user application. The multithreading API is implemented on top of a global threads layer, which, in turn, is implemented on top of a user-level threads library. Only the thread-switching component is operating-system dependent.

Being a distributed system, most of the functionality relies on the communication modules. The post-office abstraction hides communication details from upper-level layers. Portions of the socket layers need to be rewritten for each operating system.

DSM-PEPE was implemented using this layered architecture, making it possible to reuse most of the code on both, Linux and Windows, versions. Fig. II.2 shows the object instances that are present on a process running on DSM-PEPE. The core of the system is the **Memory Manager** object, which contains a **DSMregion** object for

each actual region. Pages are represented by instances of the class `Page`, linked to the region through a `PageTable`. Consistency for a page is handled by a dedicated thread, associated to the `Page` object. Synchronization management is accomplished by a single per-processor thread.

Two `PostOffice` objects are responsible for the communication, avoiding interference between synchronization-related and consistency-related messages. A thread is created for each `PostOffice` instance, in order to receive asynchronous messages.

## **II.3 Implementation Issues**

Most of the DSM-system components are operating-system independent. A few modules, namely, page management, exception handling, message passing, and thread switching, are dependent in some degree on the underlying platform. Our system is portable in the sense that only certain portions of these components are different between the ports for Linux and Windows.

### **II.3.1 Portable Modules**

Page management is the core of the DSM system. Application programs use the shared memory through regions. Consistency of a region is managed by a consistency protocol, specified when the region is created. The size of the region defines the number of pages it contains. The page object links a memory page to its region and to the consistency protocol. Consistency for a page is ensured by a specialized thread. Hence, message-handling threads are not blocked unnecessarily.

The DSM API includes functions to create regions, specifying its size and consistency protocol, and to allocate memory from a region, specifying the amount of memory needed.

Consistency protocols are implemented taking the page class as their basis. A new subclass must be derived from the page class, and handlers for the consistency events must be coded. Any additional data that the protocol may need is defined within the subclass. For example, a sequential-consistency protocol may require a hint of the probable owner of the page; this information would be stored in the page subclass.

Interprocessor barriers use a centralized coordinator. The coordinator receives messages from all processors arriving at the barrier, and broadcasts a release notice once it has received all of them. The implementation of interprocessor locks is distributed. Each node keeps a hint of the probable holder of the lock, and sends a request when it wants to acquire the lock. The lock holder queues the requests and delivers the queue when releasing the lock. Once the lock is released and sent to another processor, the former holder forwards any request it receives to the node to which it relinquished the lock.

Barriers at the thread level are implemented on top of interprocessor barriers. Thread locks are implemented using local queues and *alien threads*. An alien thread stands for a remote thread requesting the lock from another processor. The alien thread waits at the queue of the holder processor; once it gets the lock, it sends the lock to the remote thread it represents. When the lock migrates, the queue is transferred with it.

Threads are global entities, implemented on top of a user-level library (Cormack, 1988). A pool of threads is created in every processor when the application program

starts running. All threads start execution and block until they are needed. When the program forks a thread to execute some particular function, the thread is setup in all of the processors atomically, although it is run only at the location that did the fork. The other instances stay suspended, waiting for the thread to migrate in. Each thread is given a unique system-wide identifier, which can be used to join with other threads. Thread migration is supported through the suspended state; only suspended threads can migrate. The migration facility extracts the thread state, including its stack, and sends it to the target processor. Once there, the thread is resumed in the same state in which it was when it was suspended.

Communication is done exclusively using the post-office abstraction. Two global post offices exist in each system node. The first is used exclusively for consistency-related messages. The second is used for the exchange of synchronization-related messages. Reception of messages in each post office is handled by an independent thread.

### II.3.2 Non-portable Modules

Page management and exception handling are highly dependent on the operating system underneath. For page management, a few primitives can be abstracted to implement the DSM system on top of them. Basically, we need to be able to set pages as invalid, and to set page protection to read-only or read-write. In Linux this page manipulation is possible using the `mprotect` system call. Virtual-address correspondence is accomplished manipulating the heap. In Windows, we must use some functions from the virtual-memory interface: `VirtualAlloc`, `VirtualQuery` and `VirtualProtect`. Virtual-address correspondence is guaranteed using memory-mapped files.

Exception handling allows to catch faulting accesses to memory, in order to handle those faults related to the DSM management. Application programs must be protected by an exception handler. Windows provides a structured mechanism which is suitable for this purpose. It is possible to protect any code, in particular application code, with a handler for virtual memory exceptions. In Linux there is no such a structured way of handling exceptions, but we can use the Unix signal-handling mechanism instead. In particular, it is possible to install a handler for the `SIGSEGV` signal, raised on faulting memory accesses; the `signal` system call fulfills this purpose.

Communication is not isolated from portability issues. Although socket communication is relatively high level, Windows implementation of sockets differs from Linux and Unix implementations. These differences must be hidden in a software layer that provides a portable interface to the upper layers.

Finally, thread switching is the only component of the thread library that is not portable across operating systems. A few lines of assembly code accomplish this task, and had to be written for each operating system.

## II.4 Preliminary Results

DSM-PEPE runs on top of two of the most popular operating systems nowadays. We succeeded in implementing a system that works on both platforms with minor changes. This section presents results from executing a simple parallel application on both systems, and a comparison of performance between them. Also, we show speedups relative to a sequential, equivalent program.

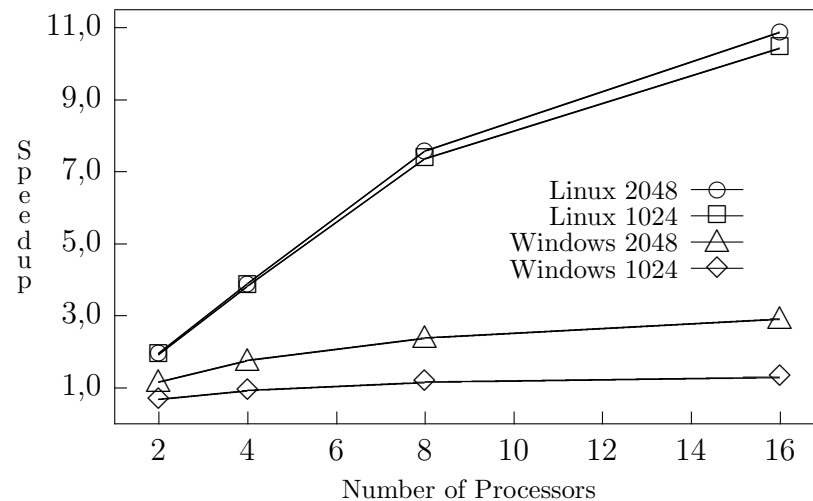


Figure II.3 Speedup for two problem sizes running on Linux and Windows

The testbed is a set of up to 16 personal computers with the same configuration: Intel Pentium 4 running at 1.66 GHz, 256 MB RAM, and 512 KB L2 cache; connections are switched at 100 Mbps. We used both Windows 2000 and RedHat Linux 8.0.

The application chosen is a simple integer square-matrix multiplication. Two different matrix sizes were used:  $1024 \times 1024$  and  $2048 \times 2048$ . No special optimization was done. Computation is distributed splitting the result matrix in slices of the same size, using a striped approach. Each processor is responsible for computing the values in the slice assigned to it. This scheme produces a highly parallel task; source matrices are accessed only for reading, and there are no race conditions, since the writing of the result is distributed among the processors. The turnaround time of the program was measured using 2, 4, 8, and 16 processors, for both matrix sizes. Also, a sequential version of the program was run in a single processor, in order to calculate speedups for the parallel versions. Measurements were taken five times to produce an average value; variance was negligible.

Fig. II.3 shows a comparison for the 16 experiments performed. Speedups are clearly higher for the application running on the Linux version of the system, producing a high degree of efficiency. Since most of the code is identical across versions, as well as is the hardware platform, the difference is due to operating-system-dependent components of the system. Preliminary evaluation shows that socket communication is the major responsible for the lower performance on Windows. Moreover, exception handling and virtual memory management mechanisms are at a higher level in Windows than in Linux, producing additional overhead in the Windows version of the system.

## II.5 Concluding Remarks

Portability is a key issue when designing and implementing a software DSM system. The layered design of DSM-PEPE allowed us to reuse most of the code for both the Linux and Windows versions of the system. Only those components responsible for low-level socket communication, exception handling, virtual-memory management, and thread switching needed to be rewritten across platforms.

System maintenance is easy, because of the modular design. A specially designed interface for the consistency-related events, allows to add new consistency protocols without affecting other parts of the system.

Our preliminary results confirm the potential of a software DSM system as a parallel computing environment. Programming for the system is easy, and the cost of building the virtual machine is low. In the future, we expect to perform an extensive set of experiments using different applications extracted from known benchmark suites.

Slightly heterogeneous distributed systems can be built using DSM-PEPE, that is, systems composed of computers running different operating systems on top of the same hardware. Hardware homogeneity allows an easy exchange of data between machines running different operating systems. An open area of study involves thread migration in such a slightly heterogeneous system.

Thread migration presents several problems in the context of relaxed consistency protocols. Currently, we are studying the relation between relaxed memory protocols, thread migration, and the data-access patterns of the programs.

### III. MULTITHREADED DISTRIBUTED MUTUAL EXCLUSION

#### III.1 Introduction

Mutual exclusion provides synchronized access to shared resources ensuring that, at any time, at most one process can be executing in its critical section. Distributed mutual exclusion algorithms focus on mutual exclusion on distributed environments lacking shared memory. Several algorithms address the distributed mutual exclusion problem for systems where only one process is running at each processor. Multithreaded distributed systems allow the existence of several threads of execution within each distributed process. Thus, there is a need to provide mutual exclusion to a number of distributed threads. We are particularly interested in DSM systems (Li and Hudak, 1989) with support for multithreading and thread migration.

In this work, we present an algorithm for distributed mutual exclusion in a multithreaded system. The algorithm is token-based, and it uses a tree to route requests issued to acquire the token. We rely on special-purpose *alien threads* running at host processors on behalf of threads running at other processors.

When a thread asks for permission to enter its critical section, if the token is not present at the processor it is running at, a remote alien thread is activated in order to obtain the token and send it to the requesting processor. Alien threads behave just like ordinary threads, and must compete for the token with other user threads.

We performed a simulation study comparing two versions of our algorithm with a previously proposed algorithm based on path reversal on trees. This algorithm is, to the best of our knowledge, the only documented implementation addressing the

same problem. Results obtained from the simulation are encouraging. Our algorithm performs very well under high load conditions, outperforming the other proposal as the number of threads per node increases.

Our algorithm was successfully implemented on *DSM-PEPE*, a multithreaded distributed system with support for thread migration (Meza et al., 2003).

## III.2 The algorithm

### III.2.1 System model

The system is a loosely-coupled network of computers, consisting of  $n$  processors:  $p_1, p_2, \dots, p_n$ . At any time, at each processor  $p_i$ , there are  $m_i$  threads running. Threads are allowed to migrate according to some system policy, for instance, pursuing load balancing or minimal message exchange.

Processors communicate through message passing. We assume that message delivery is guaranteed by the network. We also assume that two messages issued at one processor and addressed to the same node are received in the same order at the destination. This is the usual behavior of switched local area networks, where there is only one possible route between each pair of computers.

A thread wishing to enter its critical section must obtain permission by calling `Acquire()`. The thread could be delayed until mutual exclusion can be guaranteed. Once the thread leaves the critical section, it must notify the system by calling `Release()`. Mutual exclusion must be ensured between the call to `Acquire()` and the call to `Release()`.

### III.2.2 Brief description

Our algorithm is token-based. A thread wishing to enter its critical section must obtain a single system-wide token. Uniqueness of the token guarantees the mutual exclusion (Hélary et al., 1994). At the higher level, ownership of the token is not granted directly to threads but to processors. Once a processor owns the token, threads running at that processor can compete for it. Requests issued at each processor are stored in a local queue on that processor. Ownership of the token is granted to one of the processors during initialization. For the remaining processors a unique path must exist to allow them to reach the actual owner of the token. This is accomplished through a chain of *probable owners*, building up a tree rooted at the first owner.

Requests made by threads running at the processor currently owning the token are serialized and served according to their arrival time. A request issued by a thread running at a processor not owning the token involves sending a request to the probable owner. Our algorithm accomplished this task, by signaling a special-purpose thread running at the probable owner of the token.

At any processor, there are  $n - 1$  *alien threads*, each acting on behalf of one of the remaining  $n - 1$  processors in the system. Alien threads behave just like ordinary threads, but they are blocked most of the time. An alien thread at processor  $p_i$  is signaled when some thread at the home processor –that is, the processor for which the alien thread is working on behalf of– is requiring the token, and it is presumed that the token is held by processor  $p_i$ , that is, the probable owner at the home processor is set to  $p_i$ . The woken alien thread asks for the token at its host processor. Then, once the token is granted, the alien thread sends the token to the processor it is representing. Note that it is possible for a woken alien thread to find out that the

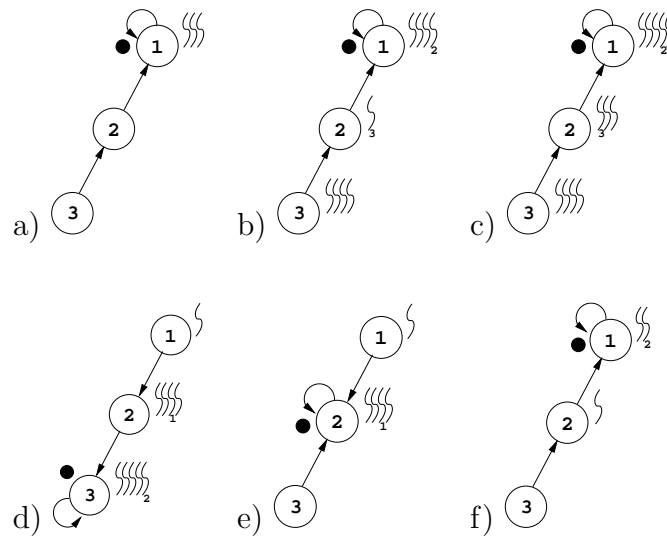


Figure III.1 Behavior of the algorithm when servicing several requests

token is no longer on its host processor. When this situation occurs, the alien thread acts like an ordinary thread requesting the token. The alien thread signals the alien thread on the processor where the token apparently went to, that is, the probable owner on its host processor. This scenario could appear several times, until the processor currently holding the token is reached. This kind of forwarding resembles the algorithm described by Raymond for distributed mutual exclusion of single-threaded processors (Raymond, 1989).

Figure III.1 shows the behavior of our algorithm in a system with 3 nodes. Initially, node 1 owns the token –represented as a filled circle– and several threads are blocked at that node, each waiting to enter its critical section (Figure III.1a). Note that node 1, the owner of the token, is at the root of the tree used to route the requests. At this moment, several requests are issued from threads running at node 3. The first of these requests produces a signal to an alien thread on node 2, to wake up and act on behalf of node 3 (Figure III.1b). However, since the token is not present at

2, the recently woken alien thread blocks, producing a signal to an alien thread on node 1, to wake up and act on behalf of node 2. This alien thread waits at the end of the local queue of node 1. After that, some threads issued new requests at nodes 1 and 2, producing their local enqueueing (Figure III.1c). At this point it is important to note the behavior of the alien threads currently active in the system. An alien thread is waiting for the token at node 1, the current owner of the token, on behalf of 2, and another alien thread is waiting for the token at node 2, on behalf of 3. The thread that issued the original request is waiting for the token at node 3. A node only sees a queue of requests, some issued by local threads and some issued by alien threads. Eventually, the alien thread running at node 1 on behalf of 2 obtains the token, and sends it to node 2. Since the queue was not empty at this time, that is, there are pending requests at node 1, an alien thread on node 2 is signaled to bring the token back to node 1. At the head of the queue at node 2 was the alien thread that acts on behalf of node 3, so it sends the token to node 3, and an alien thread on node 3 is signaled to bring the token back to node 2. Note that, at node 2 there are several pending requests, including one that will bring the token back to node 1 (Figure III.1d). Eventually, the token returns to node 2 (Figure III.1e) and to node 1 (Figure III.1f).

### III.2.3 Detailed description

Each processor must hold the following information:

- **probOwner**: process identifier `–pid–` of the processor last known as the token owner. Initially set in such a way that there is a single path, following the `probOwners` chain, from each node to the initial owner of the token. The first owner sets `probOwner` to its `pid`.

- `tokenRequested`: `true` if there are pending requests for the token issued from this processor, *i.e.*, a request for the token has been already sent. Initially `false` at every processor.
- `numLocal`: number of requests for the token that have been issued locally; initially 0 at every processor. Recall that only the first request actually makes an alien thread to be signaled.

Local mutual exclusion for the operations showed below is mandatory. However it has been omitted intentionally to illustrate the solution more clearly. Semantics of the *wait* and *signal* operations are consistent with those on conditions variables. A signal across processors involve sending a message to the target processor.

A thread acquiring the token must execute:

```

Acquire() {
    numLocal++;
    if (probOwner != pid) && (! tokenRequested) {
        // Not owner and not previously requested => Request token
        signal(alien thread on probOwner);
        tokenRequested = true;    // to avoid multiple requests
        wait(for signal from the alien thread);
        probOwner = pid;         // processor becomes owner
        tokenRequested = false;
    }
    else {
        // Processor owns token, or token has been requested already
        if (numLocal > 1) {
            wait(for signal from another local thread);
        }
    }
}

```

If the token is not currently held by the processor it must be requested, by signaling the alien thread on the probable owner. The thread blocks waiting for the token to arrive. If some other thread has previously called `Acquire`, we must prevent multiple requests. If the token is held by the processor, or it has been requested already, there is no need for remote requests. If the token is held but free the thread is allowed to enter its critical section.

A thread releasing the token must execute:

```
Release() {
    numLocal--;
    signal(local thread waiting for the token);
}
```

The alien thread executing on processor *host* on behalf of processor *home* must execute:

```
alienThread(host_pid, home_pid) {
    while(true) {
        wait(for signal from home processor);        // stay idle
        Acquire();                // acquire token on host processor
        signal(thread waiting for the token on the home processor);
        probOwner = home_pid; // update host-processor's probOwner
        numLocal--;
        if (numLocal > 0) {
            // Request the token on behalf of host processor
            tokenRequested = true; // to avoid multiple requests
            signal(alien thread on home processor on behalf of host);
        }
    }
}
```

An alien thread waits until signaled from its home processor. Then, it acquires the token, competing with local threads on the host processor, as well as with other alien threads trying to get the token on behalf of their homes. Once an alien thread

succeeded on acquiring the token, it signals its home processor, allowing a remote waiting thread to resume under mutual exclusion. It is possible to have additional threads left on the local queue when an alien thread acquires the token delivering it to its home processor. If this happens, the alien thread requests the token before turning idle. This is accomplished by signaling the alien thread that represents its host on its home processor. A simple improvement to the algorithm presented involves *piggybacking* this request on the same signaling message that delivers the token.

The behavior of an alien thread holding the token is slightly different from the behavior of a user thread. A user thread is expected to release the token once it leaves the critical section. However, an alien thread does not release the token, but delivers it directly to another thread at his home processor instead.

#### **III.2.4 A variant on the proposed algorithm**

An alien thread forwards requests when the token is not present at its host processor. This is done by signaling the alien thread on the probable owner, on behalf of its host processor. Thus, the token is forced to follow exactly the same path followed by the requests. This behavior is desirable under high requests load, because there will be always pending requests on the returning path of the token, avoiding the exchange of extra messages. However, under a light load, the token could be sent directly to the processor that issued the first request, avoiding the extra steps produced by the forwarding. Only the code executed by the alien threads must be modified in order to implement this variant.

An alien thread must execute the following code:

```

alienThread(host_pid, home_pid) {
    while(true) {
        wait(for signal from home processor);          // stay idle
        if ((probOwner != host_pid) && (numLocal == 0)) {
            signal(alien thread on behalf of home, on host's probOwner);
        }
        else {
            // Behaves like the original alien thread
            Acquire();          // acquire token on host processor
            signal(thread waiting for the token on the home processor);
            probOwner = home_pid; // update host-processor's probOwner
            numLocal--;
            if (numLocal > 0) {
                // Request the token on behalf of host processor
                tokenRequested = true; // to avoid multiple requests
                signal(alien thread on home processor on behalf of host);
            }
        }
    }
}

```

Instead of simply forwarding requests, the improved alien thread checks first if there are local requests at its host processor that justify acquiring the token. Otherwise, it signals an alien thread on behalf of its home processor, avoiding the passing of the token across its host processor.

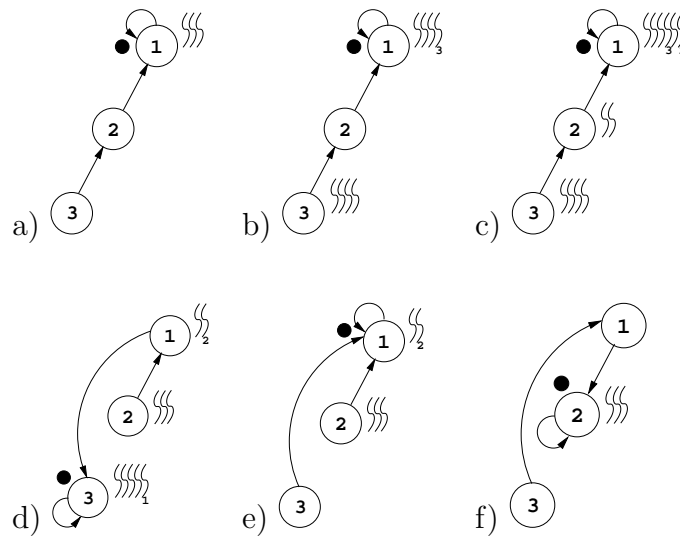


Figure III.2 Behavior of the modified algorithm when servicing several requests

Figure III.2 shows the behavior of the modified algorithm for the same requests sequence used in the example of the original algorithm in Section III.2.2. Note that, when the first request by a thread running at node 3 is issued, the alien thread at node 2 does not remain active because the queue at node 2 is empty. It just forwards the signal to an alien thread at node 1, to wake up and act on behalf of node 3 instead (Figure III.2b). Eventually, this alien thread obtains the token, and sends it directly to node 3 (Figure III.2d).

The path followed by the token back to the requester node is not necessarily the same path previously followed by the request on its way to the owner of the token. Note that several alien threads in the request path just forwarded the request, changing the topology of the tree.

### III.3 Proof outline

A mutual exclusion algorithm must satisfy several conditions. The following is an outline of the proof of correctness for three of these conditions, considering the original algorithm.

**Mutual exclusion:** It must be assured that, at any time, at most one thread can be executing in its critical section. Our algorithm is token-based: there is a single system-wide token, owned by the node having `probOwner == pid`. This condition is enforced during initialization. A thread asks for the token by executing `Acquire` and could be delayed on two conditions: (1) when the token is currently held by its host node but it is assigned to another thread, or (2) when the token is not locally present at the time. Either way, the thread is allowed to continue executing in its critical section only when the thread currently holding the token relinquished it by executing `Release`, or when an alien thread signals the blocked thread remotely. In the former, it is straightforward to verify that mutual exclusion is assured. In the latter, the signaling alien thread previously obtained the mutual exclusion by executing `Acquire` on its host processor. Since an alien thread does not have a critical section, it relinquishes the mutual exclusion on behalf of the thread that made the request on its home processor. This way, mutual exclusion among threads is assured.

**Deadlock freedom:** It is easy to verify that a deadlock can not occur under some reasonable restrictions. For a deadlock to occur there must be a circular-wait condition involving two or more threads in the system. Assuming that a thread executing in its critical section is not allowed to execute `Acquire` again, this condition will never occur.

**Starvation freedom:** If we assume the use of a fair policy for serving local requests at each node starvation will not occur. Recall that we have a single path from each node to the node currently holding the token. Note that a request issued at a node not owning the token results in an alien thread being queued at the node currently owning the token. If the local service policy is fair, the alien thread eventually obtains the token and allows the thread it is acting on behalf of, to enter its critical section.

### III.4 Performance

Analytic studies of distributed mutual exclusion algorithms are hard to perform, due to the rapid growth of the cardinality of the state space as the number of nodes increases (Chang, 1996). In multithreaded systems, the size of the state space grows even faster. For this reason, we choose a simulation approach to study the performance of our proposal.

#### III.4.1 Simulation model

We use a simulation model based on similar studies (Chang, 1996, Johnson, 1995). We assume that requests to enter the critical section arrive at each node according to a Poisson process with parameter  $\lambda$ . Thus, the time elapsed between requests behaves according to an exponential distribution. We assume that, at every node, requests are made by randomly-chosen user threads. Note that the simulation process remains under Poisson behavior as long as any running –not waiting– local thread exists in a node. When all the threads running at a node are waiting to enter to its critical section, the process stops until the first local thread completes its critical section.

The  $\lambda$  parameter will give us a notion of the load of the entire system. The time taken by a thread to execute its critical section is modeled as a constant  $C$ . The message propagation delay is a constant  $M$  multiplied by a random number having a uniform distribution between 0 and 1.

We are interested in two main measures: the average number of messages exchanged per critical section entry, and the average waiting time for the permission to enter the critical section.

To obtain statistically reliable results we made long-time simulations executing 100,000 critical section entries. On each experiment we use  $N = 31$  nodes, because we have a binary complete tree for the initial state. We simulate a variable number of threads per node. The parameter  $\lambda$  takes values in the  $[0, 1]$  interval. The parameter  $M$  was taken as 0,1 and the parameter  $C$  as 0,01. These values are consistent with those used in similar studies (Chang, 1996),(Johnson, 1995).

### III.4.2 Results

Figures III.3 through III.5 show the results of the simulations for the two versions of our algorithm –using piggybacking– as well as the algorithm proposed by Mueller (Mueller, 2000), using 1, 5 and 10 threads per node. In Figure III.3 there is a single thread per node. In this case, the first implementation of our algorithm resembles the algorithm by Raymond (Raymond, 1989). Results are consistent with that fact (Johnson, 1995).

Under a light load, the second version of our algorithm requires fewer messages than the first one, because the token is sent from the releaser node to the requester

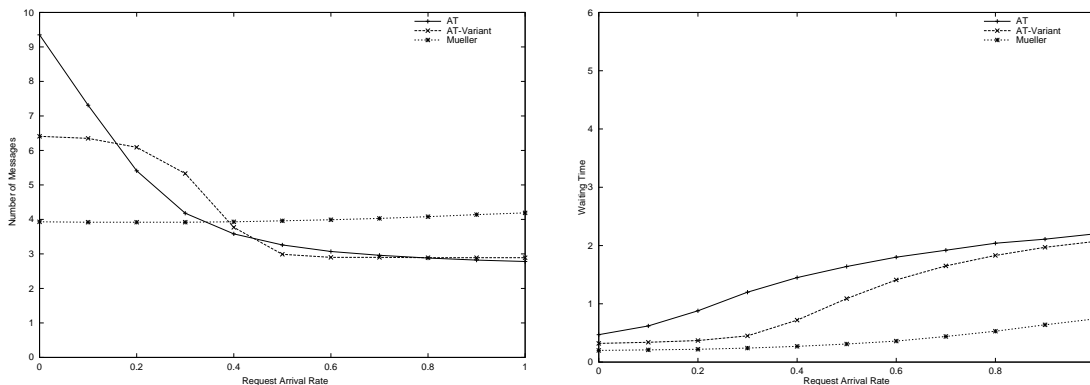


Figure III.3 Performance comparison of the 3 algorithms with a single thread per node.

directly. The alien thread that was waiting for the token at the releaser node, acts on behalf of the requester node. The first implementation enforces the token to travel along the tree structure to reach the requester node. This is so because several alien threads need to be signaled in the path previously followed by the requests. The algorithm proposed by Mueller has the best comparative performance under a light load, considering the number of messages exchanged. This is due to the *aggressive path compression* technique, characteristic of the path reversal approach (Naimi et al., 1996).

Considering the waiting time, the second version of our algorithm behaves better than the first one under all loads. Mueller's algorithm outperforms the other two.

Under a high load, both versions of our algorithm need almost the same number of messages per critical section. When the first alien thread associated to a request is signaled, it is most likely that the token was already requested on the host node. This will make both versions of the algorithm behave the same way. This situation can be easily observed in the code of the alien thread: both versions will execute the same code under high load conditions. This is also the cause of the very small number of

messages exchanged per critical section under high loads. Both implementations of the alien thread algorithm outperform Mueller's algorithm.

When we turn to multithreaded scenarios (Figures III.4 and III.5) the relative behavior among the two versions of our algorithm remains unchanged. Moreover, the waiting time is almost identical, for every number of threads per node. Besides, an important decrease in the number of messages exchanged per critical section entry is achieved under high loads as the number of per-node threads increases. Once a request has been sent from a node –that is, an alien thread has been signaled–, subsequent requests issued by threads on the home node do not involve the sending of additional messages. Thus, most of the requests issued by threads wishing to enter the critical section will be served without message exchange. The algorithm proposed by Mueller does not show any of these behaviors. The number of messages exchanged does not change significantly as the number of threads per node increases. Waiting time increases abruptly as the system load increases. The growing rate of the waiting time in the Mueller's algorithm, also increases as the number of threads per node increases. His algorithm is very sensitive to load growth on multithreaded scenarios.

The second version of the alien thread algorithm outperforms the initial version in all aspects. The algorithm proposed by Mueller showed better performance under a light load. Under high loads, the alien thread algorithm showed better results.

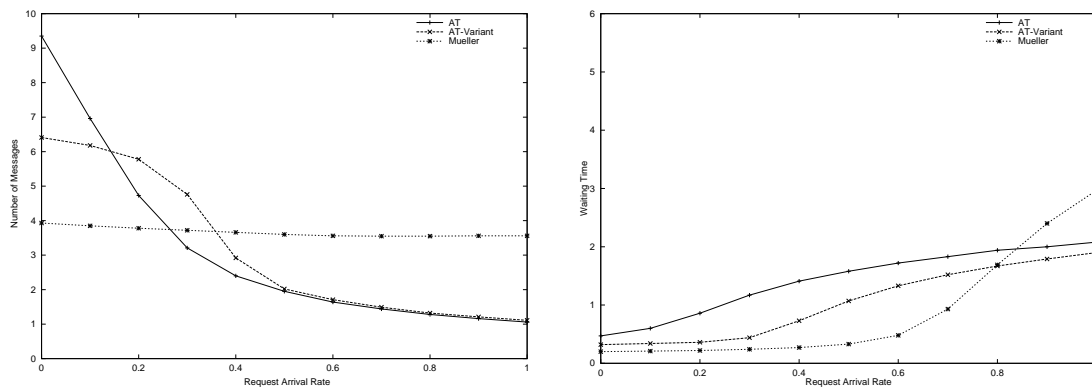


Figure III.4 Performance comparison of the 3 algorithms with 5 threads per node.

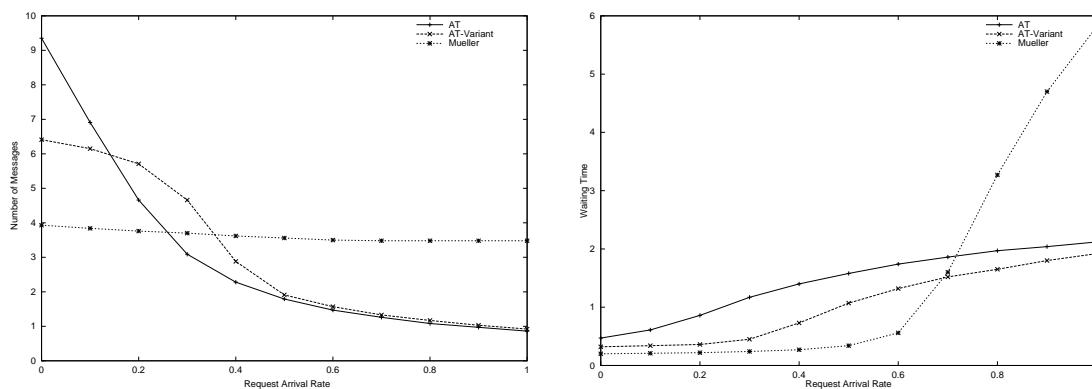


Figure III.5 Performance comparison of the 3 algorithms with 10 threads per node.

### III.5 Related work

Several distributed mutual exclusion algorithms have been proposed in the literature. They can be classified as *permission-based* or *token-based* (Raynal, 1991). We focus our study on token-based distributed algorithms, excluding those algorithms that use a central coordinator.

Token-based algorithms rely on a unique token which must be acquired by a process wishing to enter its critical section. The token could be traveling from one process to another continuously or could be obtained by sending a request. The algorithms proposed by Raymond (Raymond, 1989), by Neilsen and Mizuno (Neilsen and Mizuno, 1991), by Banerjee and Chrysanthis (Banerjee and Chrysanthis, 1996), and by Naimi, *et al.* (Naimi et al., 1996) fall into this category.

Distributed mutual exclusion for multithreaded environments has not been studied extensively. The design and implementation of distributed synchronization primitives are presented by Mueller, focusing on the impact of multithreading on synchronization (Mueller, 2000). Distributed mutual exclusion is based on a token-passing mechanism based on the algorithm described by Naimi, *et al.* (Naimi et al., 1996).

### III.6 Concluding Remarks

We presented a simple implementation of a token-based algorithm providing mutual exclusion to distributed threads running on a loosely-coupled system. This mechanism has been successfully implemented on a DSM system supporting thread migration.

We developed two versions of the algorithm and compare them to a known implementation of another algorithm, targeting to the same problem. A simulation of performance shows that both of our algorithms outperforms the other implementation under high load conditions. The difference increases as the number of threads per node increases. Under a light load, our algorithms still perform within reasonable limits.

The first version of our algorithm, limited to a single user thread per node, behaves just like a well-known single-threaded distributed mutual exclusion algorithm (Raymond, 1989). The third algorithm considered in our study (Mueller, 2000), was originally conceived as an extension of another single-threaded algorithm based on path reversal on trees (Naimi et al., 1996). Our intention is to extend the study to several single-threaded algorithms for distributed mutual exclusion, exploring the feasibility of extend them using the same ideas used to develop the alien-threads algorithm.

## IV. THREAD MIGRATION TO EXPLOIT DATA LOCALITY

### IV.1 Introduction

A large portion of the execution time of distributed applications is devoted to access remote data. Multithreading in a distributed system helps to reduce the impact of the latency produced by message exchange, by overlapping communication and computation (Thitikamol and Keleher, 1998). While waiting for a long-latency operation, the processor allows the progress of threads other than the one being blocked.

Thread migration has been proposed as a mechanism to improve performance by enhancing data locality (Thitikamol and Keleher, 1999, Jenks and Gaudiot, 2002). The idea is to move threads closer to the data they need, gathering at the same processor those threads using data stored in that location, instead of moving the data to the processors where the threads are running. However, there is a tradeoff between the mechanisms used to increase data locality and load balance. The former aims to reduce interprocessor communication while the latter attempts to increase processors utilization and hence the level of parallelism. Without restrictions, a system would exhibit high data locality at the cost of poor utilization of the processors.

In this paper we present the thread migration mechanism of DSM-PEPE, a DSM system with support for multithreading at the user-level. We show the potential of thread migration as an alternative to data migration to improve application performance by exploiting data locality. In particular, we present a series of experiments using an application with an access pattern that exhibits some degree of spatial locality. The application that uses our thread migration mechanism performed better

than the original parallel application used for comparison and showed more regular speedup patterns.

The rest of the document is structured as follows. Section IV.2 deals with the main issues involved in the implementation of multithreading and thread migration. In Section IV.3, the thread migration mechanism of DSM-PEPE is presented. Details about the experiments are covered in Section IV.4. Section IV.5 summarizes other works related to thread migration. Finally, Section IV.6 presents some concluding remarks and future lines of research.

## **IV.2 Multithreading and Thread Migration Issues**

Multithreading can be implemented at kernel level or at user level. In the former, system calls are issued for thread creation and context switches. The kernel is highly involved in thread management; thus, this approach lacks portability. User-level threads are more portable and easier to manage; the context switch has a lower cost because the operating system is not aware of the threads. However, when a user-level thread is blocked, it could block the entire process, reducing the benefits of the use of multithreading. Some mechanism must be implemented to avoid this drawback.

Thread migration involves the suspension of a thread at some point of its execution. While suspended, it is copied or moved to another processor, and resumed at the new location at the same point where its execution was suspended. The resumed thread must not be aware of the migration being carried out. To migrate a thread, all data defining the thread state must be copied, that is, its stack and the values stored in the processor registers.

Special attention must be given to the migration of the thread stack. It can contain local variables, activation registers, and pointers that could refer to memory addresses inside or outside the stack. If the stack is relocated at a different address in the destination processor, pointers to stack addresses would be outdated. Also, pointers to memory addresses that are not part of the DSM space would point to invalid addresses.

We are interested in thread migration in hardware homogeneous systems. Migration in heterogeneous environments introduces additional issues that must be considered and that are beyond the scope of this work.

### IV.3 DSM-PEPE Thread Migration Mechanism

Threads in DSM-PEPE are provided through a user-level library. A kernel timer is used to implement preemption and avoid blocking the entire process when a thread becomes blocked. The library runs on several processor architectures and operating systems. In particular, DSM-PEPE runs on top of MS-Windows and GNU/Linux, both on the Intel family of processors. Applications in DSM-PEPE follow a SPMD –Single Program Multiple Data– model (Meza et al., 2003).

A data structure called `thread` is used to store the information required to administer the threads. First, the library stores in this structure information about registers (*e.g.*, the stack pointer), administration data (*e.g.*, the thread *id*) and a pointer to the function that the thread is executing. Next, a fixed-size thread stack is stored, followed by the arguments passed to the thread function. Figure IV.1 shows this structure.

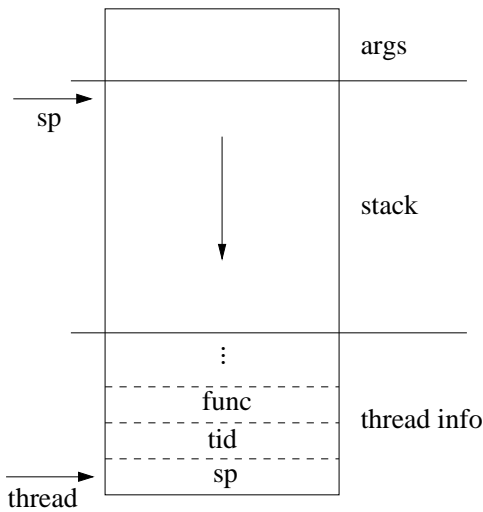


Figure IV.1 DSM-PEPE thread structure

In DSM-PEPE, a function from the API allows an application-level thread to migrate to a different location by providing the *thread id* and the target *processor id*. Migration is prohibited to threads holding system resources, like files or locks, to ease resource management. In order to ensure coherence when the thread moves to another location, thread data must be stored in the DSM space or in thread local variables which reside in the thread stack.

The migration mechanism is supported by the concept of replicated threads. Each application-level thread is replicated at each processor when created. However, the thread is activated only at the forking processor while it remains suspended at the rest of the processors. In this way, we guarantee that the thread stack is located at the same address at each processor, avoiding the problem of outdated references to variables within the stack during a migration. Besides the *ready* queue, containing the threads that are currently waiting for the processor to be assigned, a *suspended* queue is used for the threads that are about to migrate and the thread replicas that wait for a migration to come in. Figure IV.2 shows how a thread running at a

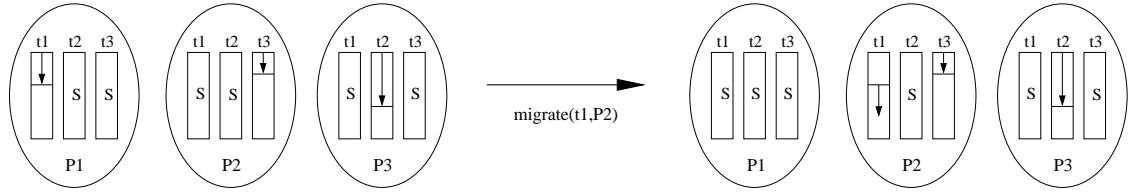


Figure IV.2 Thread suspension and activation during a migration

processor is migrated to another location. Thread  $t_1$  at P1 is migrated to P2 where it resumes execution on the suspended replicated thread that was created at P2 when  $t_1$  was originally forked. The thread that was running  $t_1$  on P1 now became suspended.

Migration is accomplished by sending a message containing the **thread** structure to the target processor. The size of this message will depend on the size of the stack assigned to the thread during creation. Usually the thread stack will be 1 KB long, but in special circumstances a larger stack will be needed. This is the case of the application shown in Section IV.4 where we store a large data structure –up to 32 KB– in the stack of each thread. Upon reception of a migration message, the system on the target processor copies the received structure to the suspended replicated thread. This is accomplished by copying the stack to the same address where it was located at the originating processor.

#### IV.4 Experiments and Results

In order to evaluate the effectiveness of the thread migration mechanism to improve performance by exploiting data locality, we ran a series of experiments with an application whose memory access pattern exhibits certain degree of spatial locality.

The application selected is 2D N-body, a simulation of the evolution of a system of  $n$  bodies or particles interacting under the influence of gravitational forces on a two-dimensional space. The force exerted on each body arises due to its interaction with all the other bodies in the system. Thus, at each step  $n - 1$  forces must be computed for each of the  $n$  particles. The computation needed grows as  $n^2$ . This application is referred to as *Sequential N-body*.

Parallelization was accomplished by distributing computation among 4 processors. Processor  $P0$  initializes an array on distributed shared memory containing the mass, initial coordinates and initial velocity for all particles in the system. At each step, each processor is responsible for the computation of the new coordinates and velocity of  $\frac{1}{4}$  of the total particles. To do this, the processor must read mass and coordinates data for all the particles in the system. The DSM system provides each process with data updated on other processors using the sequential consistency protocol. Barriers are used to synchronize progress. To reduce the false sharing induced by storing shared and not-shared data on the same array, a second array is used to store velocities and force accumulators. Hence, the array on DSM actually stores only truly-shared data: mass and coordinates. This application is referred to as *Parallel N-body*.

Thread migration was introduced in order to exploit data locality. At each step, each processor updates data for its own particles and reads data from other particles stored on the remaining processors. Local versus remote data exhibits a  $\frac{1}{3}$  ratio at each processor. Hence, instead of making data from each processor to be updated by the consistency protocol on the remaining processors, each processor sends a migratory thread to accomplish computation at the processor where the data is stored. Threads store in their stacks the accumulators needed by computation, which are moved

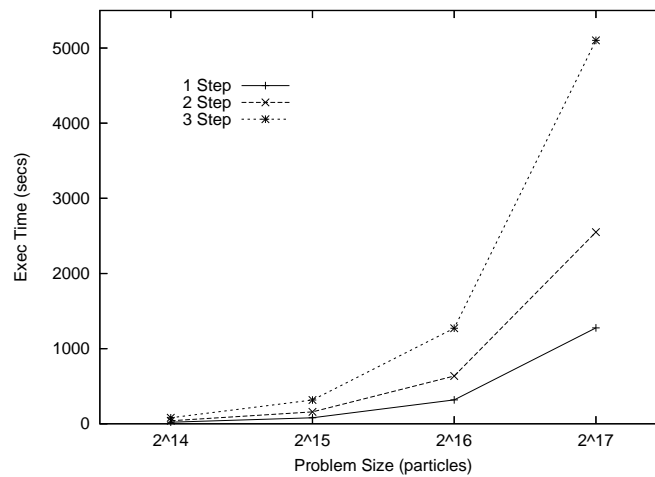


Figure IV.3 Sequential N-body: Execution time grows as  $n^2$

transparently as threads migrate. This application is referred to as *Migratory Threads N-body*.

Four different problem sizes were used in the experiments:  $2^{14} = 16384$ ,  $2^{15} = 32768$ ,  $2^{16} = 65536$ , and  $2^{17} = 131072$  particles. Each application was run to complete 1, 2, and 4 steps of computation, in order to lessen the overhead produced by the initial data distribution during the first iteration.

The testbed includes 4 computers with the same configuration: Intel Pentium IV processors running at 3 GHz, 256 MB RAM, 16 KB L1 cache, 2 MB L2 cache. The network link is an Ethernet switched at 100 Mbps. The operating system is GNU/Linux, kernel 2.6.15-23 (Ubuntu 6.06 LTS).

Figure IV.3 shows execution time of the sequential application as problem size increases. It can be seen that execution time grows as  $n^2$ . Figure IV.4 compares execution time for the three applications using the largest problem size:  $2^{17}$  particles. Both the

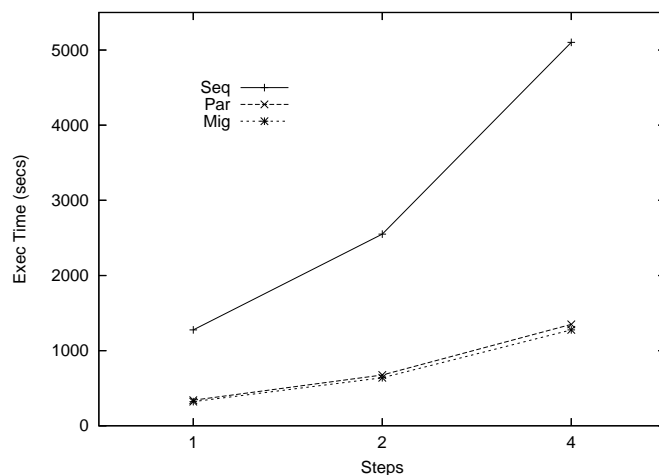


Figure IV.4 Parallel and Migratory-Threads outperform Sequential N-body with speedups 3,77 and 3,99, respectively

parallel and the migratory-threads applications outperform the sequential application. The speedup for the parallel program was 3,77, while for the migratory-threads program it was 3,99. Results for the remaining problem sizes show the same behavior.

Figure IV.5 shows the speedups for both, the parallel and the migratory-threads based applications, for 1, 2 and 4 computation steps. Speedups for the application using thread migration are clearly higher and show a regular trend, improving as the number of computational steps increases. This is due to the ad-hoc migration strategy used to solve the problem that improves data locality, reducing the number of page faults and the total data exchanged. When a thread migrates to another processor it carries its accumulators and uses only local data to perform computation. Results for the largest data set are better due to the higher level of parallelism with respect to the amount of data exchanged, that is, larger data sets involve coarser computation granularity.

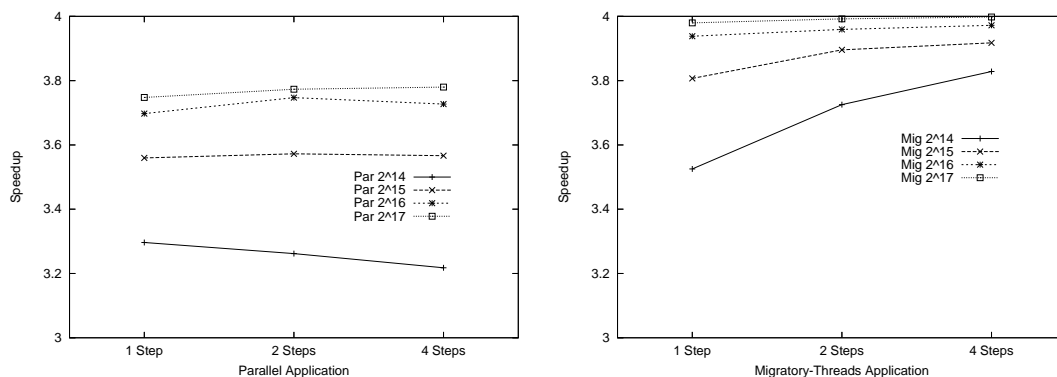


Figure IV.5 Speedup for the Parallel and Migratory-Threads Applications

The parallel application involves a large number of small-size messages, most of them due to memory consistency actions. The largest of these messages is a 4 KB message sent as a reply to a remote page fault. On the other hand, the migratory application involves less messages but half of them of a large size.

Table IV.1 shows the time involved in sending and delivering a message of different sizes, as measured in our testbed. Table IV.2 shows the time involved in migrating a thread using two different stack sizes. It can be seen that the time involved in a thread migration is slightly higher than the time needed to send a message of the same size. This is the expected behavior because the migration involves copying the stack in the originating and destination processors and some additional actions. Nevertheless, as consistency actions involve more than a single message (for example, to invalidate remote copies), it could be expected that an application that relies on migration to avoid page faults performs better and sends fewer messages.

For the largest problem size  $-2^{17}$  particles– and 4 steps of computation, the parallel application exchanged 47790 messages for a total of 49,75 MB, while the migratory application exchanged only 4302 messages for a total of 40,78 MB. Of the total

Table IV.1 Time involved in message transmission

Size (KB)	Time ( $\mu$ secs)
$\approx 0$	66
4	514
32	2971

Table IV.2 Time involved in thread migration

Stack size (KB)	Time ( $\mu$ secs)
4	547
32	3445

messages exchanged in the parallel application, 26 % are page-fault replies –4 KB– while the remaining 74 % are short messages, mostly related to consistency and synchronization. In the migratory application, 24 % of the total messages exchanged corresponds to large migration messages –36 KB– while 28 % are page-fault replies –4 KB– and 48 % are short messages. This comparison can be seen in Figure IV.6. Although the migratory application sends larger messages, caused by the large stack defined for the threads, the parallel application sends more cumulative data, mostly due to the large number of page faults involved.

Figure IV.7 compares the number of messages sent by each processor, when computing 4 steps of computation for the largest problem size. Because at the beginning of computation all data is stored at processor 0, a large amount of work is accomplished by that processor during the first step, in order to distribute data among the other processors. Afterwards the workload is uniformly distributed among all processors.

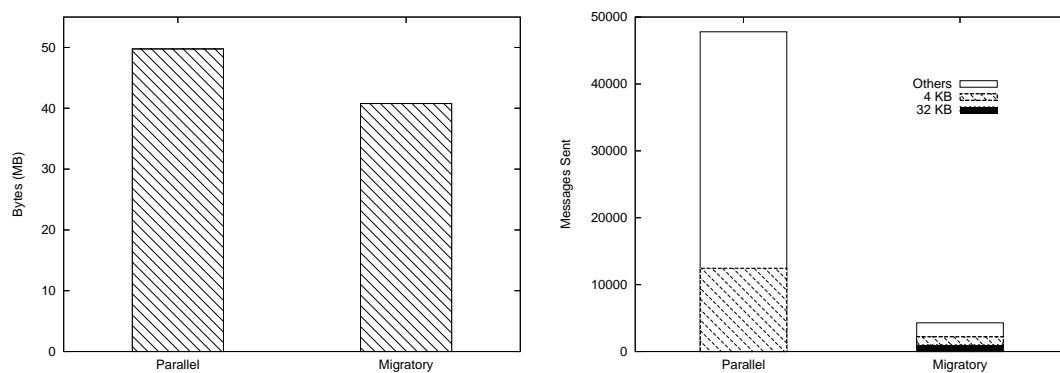


Figure IV.6 Messages and data exchanged when computing 4 steps for  $2^{17}$  particles

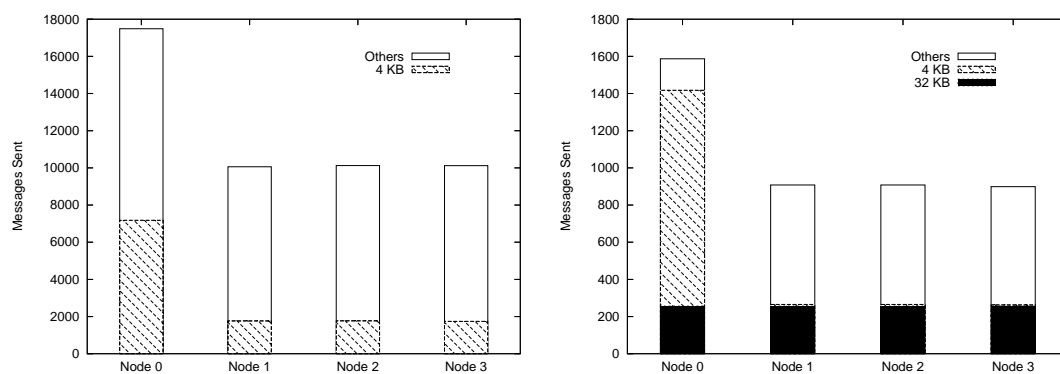


Figure IV.7 Messages sent by each processor when computing 4 steps for  $2^{17}$  particles

## IV.5 Related work

Three different strategies to deal with the problem of addresses stored in the stack are found in the literature. The first approach is used in systems that rely on programming language and compiler support to obtain information about the pointers in order to update them during a migration. In this category are systems like Emerald (Jul et al., 1988), Arachne (Dimitrov and Rego, 1998), and Nomadic Threads (Jenks and Gaudiot, 2002, 2003, Jenks, 2004). Emerald is an object-based system for the construction of distributed programs whose objects can move freely among processors. The compiler supports the translation of pointers during migration. Arachne provides thread migration across heterogeneous platforms by extending the C++ programming language. Nomadic Threads are compiler-generated fine-grain threads that migrate between processors to access data. They are supported by a runtime system that manages data and thread migrations. The approach followed by these systems lacks portability, because of their strong dependency on the compiler.

The second approach is to identify and update pointers stored in the stack at execution time, as it is done in Ariadne (Mascarenhas and Rego, 1996). When a thread is migrated, its stack is inspected to identify and update outdated pointers. However, there is no guarantee that all pointers will be identified.

The third approach is the one used in DSM-PEPE, and in systems like Millipede (Itzkovitz et al., 1998), Nomad (Milton, 1998), and Amber (Chase et al., 1989). Millipede is a DSM system for MS-Windows that implements multithreading at the kernel level and thread migration. Nomad is a light-weight thread migration system that delays the sending of the complete stack. Amber is an object-oriented DSM system implementing thread migration. Object location is handled explicitly by the application

and the system requires a large address space to be available. Data outside the stack, being referenced by pointers in the stack, are not migrated.

There are also mixed approaches, like MigThread (Jiang and Chaudhary, 2002*b,a,c*), that use preprocessing and run-time support to deal with the migration of the threads stacks.

#### **IV.6 Concluding remarks**

Distributed-memory applications implementing parallelism by data distribution among the processors usually benefits from a better cache utilization. This could be the case of the application used in this work and can explain the high speedups achieved.

The migratory application performs better than the parallel application, although the difference of speedups is not strong in relative terms. This can be explained because the parallel application was optimized to reduce the false sharing within the shared data structure. Also, the essence of the chosen problem involves a large number of pages that must be updated while the thread is migrating, causing a large stack to be moved along with each migration. In the future we will perform experimentation with other applications that may benefit from the enhanced data locality that thread migration can provide.

## V. CONCLUSIONS

We presented DSM-PEPE, a multithreaded DSM system that runs on two different environments: MS-Windows and GNU/Linux. DSM-PEPE has proven to be a suitable research and experimentation platform for DSM related topics. In fact, people from our research group have used DSM-PEPE to study prefetching techniques based on page-access histories, and to implement parallel genetic algorithms using the thread migration mechanism, among other works.

Also, DSM-PEPE has shown its potential as a platform for parallel computing, preserving the scalability and low cost of a multicomputer, while introducing the ease of programming of a shared-memory multiprocessor.

Most of the distributed mutual exclusion algorithms proposed are not suitable for multithreaded applications where more than one mutual exclusion request could be issued from a single node. We presented a token-based algorithm providing mutual exclusion to distributed threads running on a loosely-coupled system. This algorithm, called the *alien-threads algorithm*, has been successfully implemented on DSM-PEPE.

We performed a simulation study comparing two versions of the *alien-threads algorithm* with a previously proposed algorithm based on path reversal on trees. This algorithm is, to the best of our knowledge, the only documented implementation addressing the same problem. Results show that the *alien-threads algorithm* outperforms the other algorithm under high load conditions. The difference increases as the number of threads per node increases. Under a light load, our algorithm still performs within reasonable limits.

We showed the potential of the thread migration mechanism of DSM-PEPE as an alternative to data migration. Threads are allowed to migrate from one node to another, as needed by the computation. Preliminary results show that applications based on thread migration can achieve higher speedups by improving data locality.

However, at this time migration must be done by the application itself. An open line of research involves the study of migration policies that improve locality taking into account the application data access pattern and the tradeoff between level of parallelism and locality of reference.

Under the *entry consistency* model, updates are propagated when a lock is acquired. Only data associated with that lock are made consistent. Certain applications—exhibiting regular data access patterns—could benefit from a thread migration mechanism that make memory consistent by moving threads instead of data. We plan to implement a migration policy for DSM-PEPE driven by the coherence actions taken by our *entry consistency* protocol.

## BIBLIOGRAPHY

- ADVE, S. and GHARACHORLOO, K. (1995), Shared Memory Consistency Models: A Tutorial, Technical Report ECE-9512, Rice University, Houston, TX (USA).
- ADVE, S. and HILL, M. (1990), Weak Ordering: A New Definition, in *17th ACM Annual International Symposium on Computer Architecture*, pp. 2–14.
- AGRAWALA, D. and ABBADI, A. E. (1989), An Efficient Solution to the Distributed Mutual Exclusion Problem, in *Proc. 8th. ACM Symposium on PODC*, pp. 193–200.
- ANDREWS, G. R. (1991), *Concurrent Programming: Principles and Practice*, Addison Wesley, Menlo Park, California.
- BANERJEE, S. and CHRYSANTHIS, P. (1996), A New Token Passing Distributed Mutual Exclusion Algorithm, in *Proc. of the 16th. International Conference on Distributed Computing Systems (ICDCS 96)*, pp. 717–725.
- BENNETT, J., CARTER, J. and ZWAENEPOEL, W. (1990), Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence, in *Second ACM Symposium on Principles and Practice of Parallel Programming*, pp. 168–176.
- BERSHAD, B. N. and ZEKAUSKAS, M. J. (1991), Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, Technical Report CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA (USA).
- BIANCHINI, R., PINTO, R. and AMORIM, C. L. (1998), Data Prefetching for Software DSMs, in *ICS '98: Proceedings of the 12th International Conference on Supercomputing*, ACM Press, pp. 385–392.

CARRIERO, N. and GELERNTER, D. (1989), Linda in Context, *Communications of the ACM* **32**(4), 444–458.

CARTER, J. B., BENNETT, J. K. and ZWAENPOEL, W. (1997), Techniques for Reducing Consistency-Related Communication in Distributed Shared Memory Systems, *ACM Transactions on Computer Systems* **13**(3), 205–243.

CHANG, Y. (1996), A Simulation Study on Distributed Mutual Exclusion, *Journal of Parallel and Distributed Computing* **33**(2), 107–121.

CHASE, J. S., AMADOR, F. G., LAZOWSKA, E. D., LEVY, H. M. and LITTLEFIELD, R. J. (1989), The Amber System: Parallel Programming on a Network of Multiprocessors, in *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, Litchfield Park AZ USA, pp. 147–158.

CORMACK, G. V. (1988), A Micro-Kernel for Concurrency in C, *Software—Practice & Experience* **18**(5), 485–491.

DIMITROV, B. and REGO, V. (1998), Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms, *IEEE Transactions on Parallel and Distributed Systems* **9**(5), 459–469.

FIGUEROA, J. and PIQUER, J. M. (2005), Intelligent Individuals, Asynchronous Communication and No Generations in a Multiple-Population Parallel Genetic Algorithm, in *Advanced Distributed Systems*, Guadalajara Mexico.

FRIEDMAN, R., GOLDIN, M., ITZKOVITZ, A. and SCHUSTER, A. (1997), Millipede: Easy Parallel Programming in Available Distributed Environments, *Software: Practice and Experience* **27**(8), 929–965.

GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R. and SUNDERAM, V. (1994), *PVM, Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, Massachusetts.

GHARACHORLOO, K., LENOSKI, D., LAUDON, J., GIBBONS, P., GUPTA, A. and HENNESSY, J. L. (1990), Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, in *17th Annual International Symposium on Computer Architecture, ACM*, pp. 15–26.

GUADALUPE, E. (2005), *A Framework to Extend Distributed Mutual Exclusion Algorithms to support Multiple Requests Per-Node*, Master's Thesis, Departamento de Ciencia de la Computación, Universidad Católica de Chile, Santiago, Chile.

HÉLARY, J.-M., MOSTEFAOUI, A. and RAYNAL, M. (1994), A General Scheme for Token- and Tree-Based Distributed Mutual Exclusion Algorithms, *IEEE Transactions on Parallel and Distributed Systems* **5**(11), 1185–1196.

HUTTO, P. and AHAMAD, M. (1990), Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories, in *10th IEEE International Conference on Distributed Computing Systems*, pp. 302–311.

IFTODE, L., SINGH, J. P. and LI, K. (1996), Scope Consistency: A Bridge between Release Consistency and Entry Consistency, in *8th Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, pp. 277–287.

ITZKOVITZ, A., SCHUSTER, A. and SHALEV, L. (1998), Thread Migration and its Applications in Distributed Shared Memory Systems, *Journal of Systems and Software* **42**(1), 71–87.

JENKS, S. (2004), Multithreading and Thread Migration using MPI and Myrinet, in *Proceedings of the Parallel and Distributed Computing and Systems (PDCS'04)*.

JENKS, S. and GAUDIOT, J.-L. (2002), An Evaluation of Thread Migration for Exploiting Distributed Array Locality, in *Proceedings of the 16th Annual International Symposium on High Performance Computing Systems and Applications (HPCS'02)*, IEEE Computer Society, pp. 190–195.

JENKS, S. and GAUDIOT, J.-L. (2003), A Multithreaded Runtime System with Thread Migration for Distributed Memory Parallel Computing, in *Proceedings of High Performance Computing Symposium*.

JIANG, H. and CHAUDHARY, V. (2002a), Compile/Run-time Support for Thread Migration, in *16th International Parallel and Distributed Processing Symposium*, Fort Lauderdale, Florida.

JIANG, H. and CHAUDHARY, V. (2002b), MigThread: Thread Migration in DSM Systems, in *Proceedings of the ICPP Workshop on Compile/Runtime Techniques for Parallel Computing*.

JIANG, H. and CHAUDHARY, V. (2002c), On Improving Thread Migration: Safety and Performance, in *Proceedings, 9th International Conference on High Performance Computing — HiPC2002*, Vol. 2552 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 474–484.

JOHNSON, T. (1995), A Performance Comparison of Fast Distributed Mutual Exclusion Algorithms, in *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95)*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 258–264.

JUL, E., LEVY, H., HUTCHINSON, N. and BLACK, A. (1988), Fine-Grained Mobility in the Emerald System, *ACM Transactions on Computer Systems* **6**(1), 109–133.

KELEHER, P. (1997), CVM: The Coherent Virtual Machine, Technical Report, Department of Computer Science, University of Maryland.

KELEHER, P., COX, A. L. and ZWAENEPOEL, W. (1992), Lazy Release Consistency for Software Distributed Shared Memory, in *19th ACM Annual International Symposium on Computer Architecture*, pp. 13–21.

KELEHER, P. J. (1996), The Relative Importance of Concurrent Writers and Weak Consistency Models, in *16th International Conference on Distributed Computing Systems*, pp. 91–98.

LAMPORT, L. (1979), How to make a Multiprocessor Computer that correctly executes Multiprocess Programs, *IEEE Transactions on Computers* **C-28**(9), 690–691.

LI, K. (1986), *Shared Virtual Memory on Loosely Coupled Multiprocessors*, PhD Thesis, Yale University.

LI, K. and HUDAK, P. (1989), Memory Coherence in Shared Virtual Memory Systems, *ACM Transactions on Computer Systems* **7**(4), 321–359.

LIPTON, R. and SANDBERG, J. (1988), PRAM: A Scalable Shared Memory, Technical Report CS-TR-180-88, Department of Computer Science, Princeton University.

LO, V. (1994), Operating Systems Enhancements for Distributed Shared Memory, *Advances in Computers* **39**, 191–237.

LU, H., DWARKADAS, S., COX, A. L. and ZWAENEPOEL, W. (1997), Quantifying the Performance Differences between PVM and TreadMarks, *Journal of Parallel and Distributed Computing* **43**, 65–78.

MAEKAWA, M. (1985), A  $\sqrt{n}$  Algorithm for Mutual Exclusion in Decentralized Systems, *ACM Transactions on Computer Systems* **3**(2), 145–159.

MASCARENHAS, E. and REGO, V. (1996), Ariadne: Architecture of a Portable Threads System Supporting Thread Migration, *Software – Practice and Experience* **26**(3), 327–356.

MEZA, F., CAMPOS, A. E. and RUZ, C. (2003), On the Design and Implementation of a Portable DSM System for Low-Cost Multicomputers, in *Computational Science and its Applications*, Vol. 2667 of *Lecture Notes in Computer Science*, Springer, pp. 967–976.

MEZA, F., PÉREZ, J. and ETEROVIC, Y. (2005), Implementing Distributed Mutual Exclusion on Multithreaded Environments: The Alien-Threads Approach, in *Advanced Distributed Systems*, Vol. 3563 of *Lecture Notes in Computer Science*, Springer, pp. 51–62.

MEZA, F. and RUZ, C. (2007), The Thread Migration Mechanism of DSM-PEPE, in *Algorithms and Architectures for Parallel Processing*, Vol. 4494 of *Lecture Notes in Computer Science*, Springer, pp. 177–187.

MILTON, S. (1998), Thread Migration in Distributed Memory Multicomputers, Technical Report TR-CS-98-01, Dept of Comp Sci & Comp Sciences Lab, Australia National University, Canberra 0200 ACT, Australia.

MUELLER, F. (1997), Distributed Shared-Memory Threads: DSM-Threads, in *Workshop on Run-Time Systems for Parallel Programming*, pp. 31–40.

MUELLER, F. (2000), Decentralized Synchronization for Multithreaded DSM, in *Proc. of the 2nd. Workshop on Software Distributed Shared Memory (WSDSM 2000)*.

NAIMI, M., TREHEL, M. and ARNOLD, A. (1996), A  $\log(N)$  Distributed Mutual Exclusion Algorithm based on Path Reversal, *Journal of Parallel and Distributed Computing* **34**(1), 1–13.

NEILSEN, M. and MIZUNO, M. (1991), A DAG-Based Algorithm for Distributed Mutual Exclusion, in *Proc. of the 11th. International Conference on Distributed Computing Systems (ICDCS 96)*, pp. 354–360.

PACHECO, P. S. (1997), *Parallel Programming with MPI*, Morgan Kaufmann, San Francisco, California.

PALMA, K., RUZ, C., ETEROVIC, Y. and MURILLO, J. M. (2004), Aplicando Orientación a Aspectos para mejorar el Diseño de un Sistema de Memoria Compartida Distribuida, in *WSDP '2004*.

PÉREZ, J. (2005), *Extending Distributed Mutual Exclusion Algorithms to support Multithreading*, Master's Thesis, Departamento de Ciencia de la Computación, Universidad Católica de Chile, Santiago, Chile.

PÉREZ, J. and ORELLANA, C. F. (2005), Un Nuevo Algoritmo Distribuido de Exclusión Mutua que Minimiza el Intercambio de Mensajes, *Revista Facultad de Ingeniería – Universidad de Tarapacá* **13**(1), 89–98.

PINTO, R., BIANCHINI, R. and DE AMORIM, C. L. (2003), Comparing Latency-Tolerance Techniques for Software DSM Systems, *IEEE Transactions on Parallel and Distributed Systems* **14**(11), 1180–1190.

RAYMOND, K. (1989), A Tree-Based Algorithm for Distributed Mutual Exclusion, *ACM Transactions on Computer Systems* **7**(1), 61–77.

RAYNAL, M. (1991), A Simple Taxonomy for Distributed Mutual Exclusion Algorithms, *ACM SIGOPS Operating Systems Review* **25**(2), 47–50.

RICART, G. and AGRAWALA, A. K. (1981), An Optimal Algorithm for Mutual Exclusion in Computer Networks, *Communications of the ACM* **24**(1), 9–17.

RUZ, C. and PIQUER, J. M. (2005), Searching an Optimal History Size for History-Based Page Prefetching on Software DSM Systems, in *High Performance Computing and Communications*, Vol. 3726 of *Lecture Notes in Computer Science*, Springer, pp. 133–142.

SPEIGHT, E. and BENNETT, J. K. (1997), Brazos: A Third Generation DSM System, in *Proceedings of the First USENIX Windows/NT Workshop*.

THITIKAMOL, K. and KELEHER, P. (1998), Per-Node Multithreading and Remote Latency, *IEEE Transactions on Computers* **47**(4), 414–426.

THITIKAMOL, K. and KELEHER, P. (1999), Thread Migration and Communication Minimization in DSM Systems (invited paper), *Proceedings of the IEEE* **87**(3), 487–497.

TORRELLAS, J., LAM, M. S. and HENNESSY, J. L. (1994), False Sharing and Spatial Locality in Multiprocessor Caches, *IEEE Transactions on Computers* **43**(6), 651–663.