Distributed Objects: an Approach based on Replication and Migration

Antonio J. Nebro, Ernesto Pimentel, José M. Troya

Dpto. de Lenguajes y Ciencias de la Computación.
Universidad de Málaga.
Campus de Teatinos. 29071 Málaga. Spain

{antonio, ernesto, troya}@lcc.uma.es

Abstract

The achievement of efficient implementations is still considered an open problem in the distributed object-oriented programming languages field. In this paper we present an approach to implement in a reasonable efficient way distributed objects. This approach is based on the use of a protocol that allows objects to be migrated and replicated. The basic idea is to reduce the overhead of remote invocations among objects making them local. The effectiveness of our scheme requires a strict programming model, where object invocations must be enclosed between pairs of acquire and release operations, and operations are classified in commands or queries. Details of a thread based implementation are presented. Experiments with a parallel matrix multiply program and a parallel branch and bound program, executed in three different architectures, demonstrate that significant speedups can be obtained.

1. Introduction

Concurrent object-oriented programming is a design methodology which tries to combine the advantages of two paradigms: concurrent programming and object-oriented programming [Mey93]. The objective of concurrent programming is to obtain benefits from parallel architectures and increase expressiveness when describing algorithms, whereas object-oriented programming offers a set of techniques that support code reuse and modularity. From the point of view of a concurrent object-oriented methodology, parallel programs are collections of concurrent objects that communicate and synchronize by invoking the operations they define in their interfaces. When concurrent objects execute in distributed systems, they are called distributed objects.

In order to implement distributed objects in an efficient way it is necessary to reduce the cost of remote invocations. The overhead of method invocation is due to the costs of name translation, message creation and formatting, transmission, reception, buffering, scheduling and dispatch. There have been a number of approaches to reduce these costs in the case of local invocations [CKP+93][TMY94]. These approaches apply advanced compiler techniques, efficient runtime systems, or combination of both. Thus, if the compiler can detect that two objects will always be local, then several steps of the general invocation scheme can be avoided. For example, a method invocation can be reduced to a simple procedure call, or the method code can even be inlined.

However, these approaches have not paid much attention to the reduction of the invocation cost between distributed objects. One possibility is to employ migration and replication schemes. These techniques are widely employed in fields such as distributed shared memory [PTM96] and distributed programming languages [BKT92], but are rarely used in the field of concurrent object-oriented languages. Migrating and replicating distributed objects impose several requirements that must be considered. For example, how to deal with the inconsistency problem when one replica of a distributed object is modified, or how to know whether an operation modifies or not the state of an object.

In this paper, we propose a migration and replication approach to be applied in the context of distributed objects. Both techniques try to reduce the cost of remote invocation between objects by making them local. Migration moves an object from one node of the system to another one, and replication makes several copies of the same object in different nodes. Our proposal is also compatible with techniques to reduce the cost of local invocations.

The paper is organized as follows. Issues related to migration and replication are briefly commented on in Section 2. In the next section we describe our proposal, explaining how migration and replication may be combined in this context, and its consequences on the programming model. Implementation details and performance results are shown in Sections 4 and 5. Finally, Section 6 summarizes the conclusions.

2. Migration and Replication. Background

The main objective of migrating and replicating objects is to transform a remote invocation into a local one. A common characteristic of these techniques is that they must not modify the semantics of the program, i.e., they must be orthogonal to it. The behavior of a parallel program must remain the same even if some objects are replicated or migrated. Only performance benefits should be observed. Furthermore, they have in common that its applicability only depends on the use of the object instead of intrinsic characteristics of the object itself. This means that replication/migration are features of objects intead of classes. For example, if we define a class of matrix objects, we cannot establish in the class specification whether the instances will be replicable or migrable. It has to be determined specifically for each object. Thus, for instance, if two matrix objects are to be multiplied in parallel, then they can be replicable, because they are going to receive only read operations. However, the result matrix is not a good candidate to be replicable, because it will be mostly written. We assume that the decision of wether an object is replicable or migrable must be taken at the time of the object creation, and be specified by the programmer.

The decision to do an object migrable or replicable can be a complex task, in particular if the candidate object invokse operations on other objects. To simplify our study, the following discussion only will be applied to those objects that act as pure server objects, that is, objects not invoking operations on other objects. Nevertheless, static analysis could be applied on a program to get relevant information allowing these decisions to be taken by the compiler.

As mentioned before, a migrable object can reside, during its life, in different nodes of the system. There are two advantages of this mechanism: reduction of the number of remote invocation among objects and, consequently, load balancing.

If an object is going to be accessed by other objects following a high degree of locality, then it can be considered migrable. Only the cost of migration, that directly depends on the size of its state variables, has to be considered. If an object issues an operation on a migrable object, then the latter is moved to the same node as the former, and the invocation can be performed locally. The main inconvenience of this technique is when the degree of locality is not high. In that case, a migrable object can be continuously moving between nodes, and performance will be negatively affected.

At the implementation level, the main problem is the location of the mobile object. Three basic strategies are: to broadcast a request message to all the nodes; to have a location server; or to use a chain protocol based on sending a request message to the last known location.

On the other hand, replication techniques try to reduce the cost of remote invocations by replicating objects in different nodes. If an object is going to be accessed by other objects that reside in different nodes, and the most of the invoked methods do not modify the state of the object, then it can be candidate to be replicable. As mentioned, only the cost of replication has to be taken into account regarding migration.

A problem of replication occurs when one of the replicas of an object is modified. Then, the object becomes inconsistent, i.e., we have several copies of the same object, and the state variables of the copies do not have the same values. In these circumstances, the establishment of a memory consistency model is necessary, which will define the behavior of replicated object in these cases [Mos93]. The choice of the memory consistency model is important, because the stronger the model is, the greater the number of messages generated in the implementation. Weak consistency models allow the use of techniques to increase performance (buffering, pipelining of messages, etc.). The counterpart is that programming becomes more difficult, because the behavior of memory (replicated objects) is not intuitive.

Most replication schemes are based on invalidation-based or update-based protocols. When using an invalidation-based protocol, a write operation on a replicated object causes all the replicas to be invalidated and the write operation is performed only on a special copy, called the main copy. If the local copy has been invalidatedm, a read operation has to request a copy from the node where the main copy resides. With update-based protocols, the writing is performed on all of the replicas, so the concept of invalid copy does not exist.

3. A Proposal: Combining Replication and Migration

Our proposal is based on an invalidation-based protocol that combines replication and migration. Objects are divided into replicable and non-replicable, and migrable objects are a special case of replicable objects.

An outline of the algorithm is explained below. For each replicated object, there is always a main copy and, possibly, several secondary copies (see Figure 1). Only the main copy can be modified. In that case, the secondary replicas must be invalidated. If the object in which the operation is invoked executes in a different node than the invoker object, then it must move to the processor of this last one. In the case of a read operation, if the local replica has been previously invalidated, an updating is requested to the main copy.

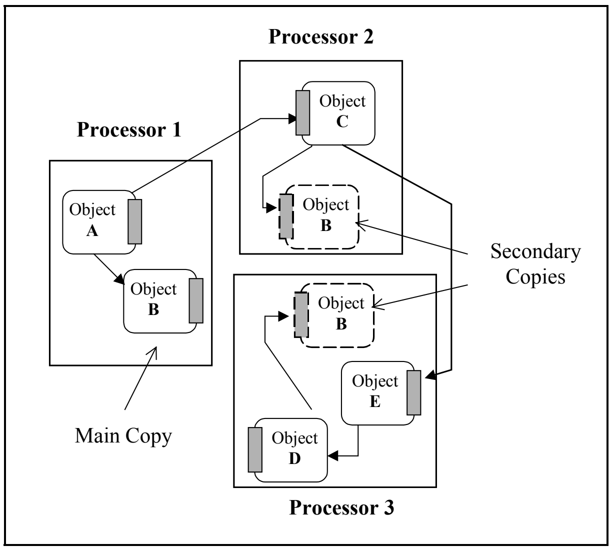

Figure 1.- *Example of Execution of a Distributed Program.*

Under this algorithm, all the accesses to replicable objects are local, including write accesses. If an object exhibits a high locality of reference (i.e., it is migrable), the overhead is only the cost of invalidating the replicas once (and only when the access produces a state change). If the object is not replicable nor migrable, the main copy is static and secondary copies do not exist.

An important aspect to be considered when using replication algorithms is the consistency protocol. Our algorithm is compatible with an entry consistency memory model [BZ91]. Under this model, each replicated object must be associated with a lock that must be explictly acquired and released. If a replicated object is accessed between a pair of acquire and release operations, then the object is said to be consistent. Furthermore, acquire operations can be classified in exclusive and non-exclusive, allowing replicated objects to be accessed following a multiple-reader/single-writer scheme. An advantage of entry consistency is that no messages are needed when releasing is done. The updating of an invalid replica is carried out when an acquire operation is performed.

The use of an entry consistency scheme to define the behavior of replicated objects introduces two problems. First, replicated objects are used in a different way that non replicated ones (the former must be acquired and released), and this fact makes the programming model more complex, due to the asymmetric treatment of both kind of objects. Second, if the programmer has to decide if an object is replicable, then he or she must know if a given operation modifies or not the state of the object, and this requires knowing how the operation is implemented.

We solve the first question requiring that non-replicable objects be acquired and released too. This is a special case of the replication/migration algorithm, where all the operations are always sent to the main and unique replica, which is not migrable in this case. The other open issue is the matter of how the programmer knows if a given operation modifies the internal state of the object or not. A solution is to classify operations in two categories: commands and queries [Mey93]. A *command* is an operation that modifies the state of the object, but does not return information about it. A *query* is an operation that returns some information about the object, without modifying its state. This distinction has the advantage of eliminating side effects: queries imply pure read accesses and commands imply pure write accesses. Furthermore, to enhance performance, the behavior of commands and queries is different: queries are synchronous, following an RPC scheme, while commands are asynchronous operations.

Taking into account the distinction between command and query operations, the classical example of a bounded buffer could be written as follows, using a C++ like notation:

```
ConcurrentObject BoundedBuffer
{
  int *buffer ;
  int dim ;
  int in, out, count ;

public:
  IntBuffer(int size) ;
  void put(int);      // Inserts an element
  void delete();      // Deletes first element
  int  head() ;       // Returns first element
  bool is_full() ;    // Returns TRUE if buffer is full
  bool is_empty() ;   // Returns TRUE if buffer is empty
} // BoundedBuffer
```

The inferface of a buffer object is composed of two commands (`put` and `delete`) and three queries (`head`, `is_full` and `is_empty`). It is not possible to define a `get` operation, because it is not a query nor a command. Instead, a pair of `head` and `delete` operations must be invoked after the object has been acquired in exclusive mode.

The necessity of explictly acquiring and releasing objects imposes a strict programming style, but it yields an important compensation: the knowledge of when and how an object is going to be used. This is the kind of information that is required to obtain the performance benefits of the entry consistency memory model.

4. Implementation

We have implemented a prototype that has been coded in C++ and consists of a runtime system and a library of base classes that must be inherited. The implementation is based on threads, asigning a thread per object. The prototype runs on three different architectures:

- 16 Sun UltraSPARC workstations interconnected through an ATM network (a distributed system composed of workstations).
- 4 Digital AlphaServer (4 processors each) interconnected through Memory Channel (a distributed system of multiprocessors).

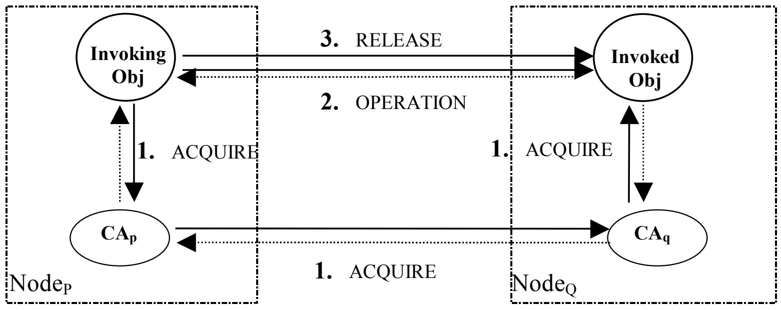

Figure 3.- *General Communication Scheme.*

- A 16 node Silicon Graphics Origin-2000 (a cache coherent shared memory multiprocessor)

The operating systems are, respectively, Solaris 2.5, Digital UNIX 4.0 and IRIX 6.4. We have used the Solaris thread package and the native pthreads packages available in IRIX and Digital UNIX. Interprocessor communications use the socket interface, although we have also coded a PVM version. The library and the programs have been compiled using the native C++ compilers.

The software was initially developed on the UltraSPARC architecture and ported later to the other two architectures. The use of two different threads packages was not a big problem because we have used the minimun set of thread funcions possible, and, at this level, every Solaris thread primitive has a pthread equivalent.

4.1 Implementation details

The architecture of the current implementation is shown in Figure 2. In each node a process exists including the runtime system and the distributed objects. The runtime system is composed of the *object table*, the *communication agent* (CA) and a set of data structures, as the *node identifier* and the *host table*. Each entry in the object table is an *object handler*, a data structure that holds object related data, such as the thread identifier, object identifier, object status, the incoming request queue, a pointer to the state variables and a pointer to the code of the operations. An object is referenced by its object identifier, but is always accessed through its object handler. When an object is referenced for the first time in a node, the runtime system creates an object handler in that node. If the object is replicable, a replica is also created.

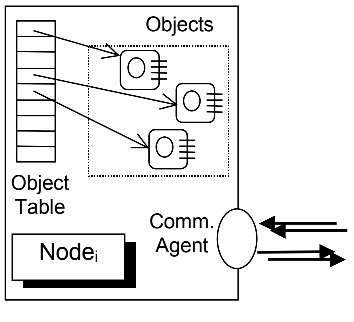

Figure 2.- *Architecture of the Implementation.*

The general communication scheme is shown in Figure 3. The messages for acquiring an object are sent to the CA. When the acquisition is performed, the invoking object directly communicates with the invoked object, without the intervention of the CA. This scheme reduces the number of messages per invocation. Protocols for object replication are carried out by the objects themselves. The CA plays a passive role: it only receives and delivers messages among objects. If objects are not replicated and are sited in the same node, the acquisition is carried out without the CA.

When communication among objects is local (for example, replicated objects are always accessed locally), the implementation allows the general communication scheme to be bypassed. An object can call the operation code of another object (method inlining), or even directly access to its state variables by means of its object handler. As a consequence, the overhead of local invocations is reduced substantially.

4.2 Using Threads

All the threads used in the implementation are kernel-level threads. A priori, user-level threads are more efficient and less resource consuming, but they are scheduled following a non-preemptive policy. We have studied this problem in a previous work [NPT97]. The conclussion is that using kernel-level, in this contex, is preferable to user-level threads, because the overhead of kernel management is negligible and the preemptive scheduling policy increases the concurrency level.

As the socket library is thread-safe on the three operating systems, we have used a kernel thread per connection into the CA. Each thread receives messages using a blocking *read* system call. The employment of kernel threads allows active threads to be executed while other ones are blocked.

When PVM is used instead of sockets, a different solution must be adopted because PVM is not thread-safe. In this case, a lock must guard the invocations to PVM routines. There is only a CA, and it cannot remain blocked in a receive primitive (no other object could issue remote messages then), so it must do a busy wait loop, using a PVM primitive to check if there are incoming messages. For this reason, this implementation is less efficient, and we will only show the results obtained using sockets.

The use of PVM would allow us to get an heterogeneous and interoperable distributed system, but at this moment, the Digital UNIX version of PVM is not interoperable with other versions.

5. Performance

To measure the performance of the current implementation, we have coded two parallel programs: MM, a simple program that multiplies square matrices of reals; and BB, a program that solves traveling salesman problem (TSP) instances using a parallel branch and bound algorithm.

The same programs are recompiled and executed in the three architectures. We must note that the number of replicas of a replicated object depends on the number of machines of the system, instead of the number of processors. Thus, up to sixteen copies can be made in the UltraSPARC ATM network, up to four in the AlphaServer system, and none replica is generated in the Origin-2000.

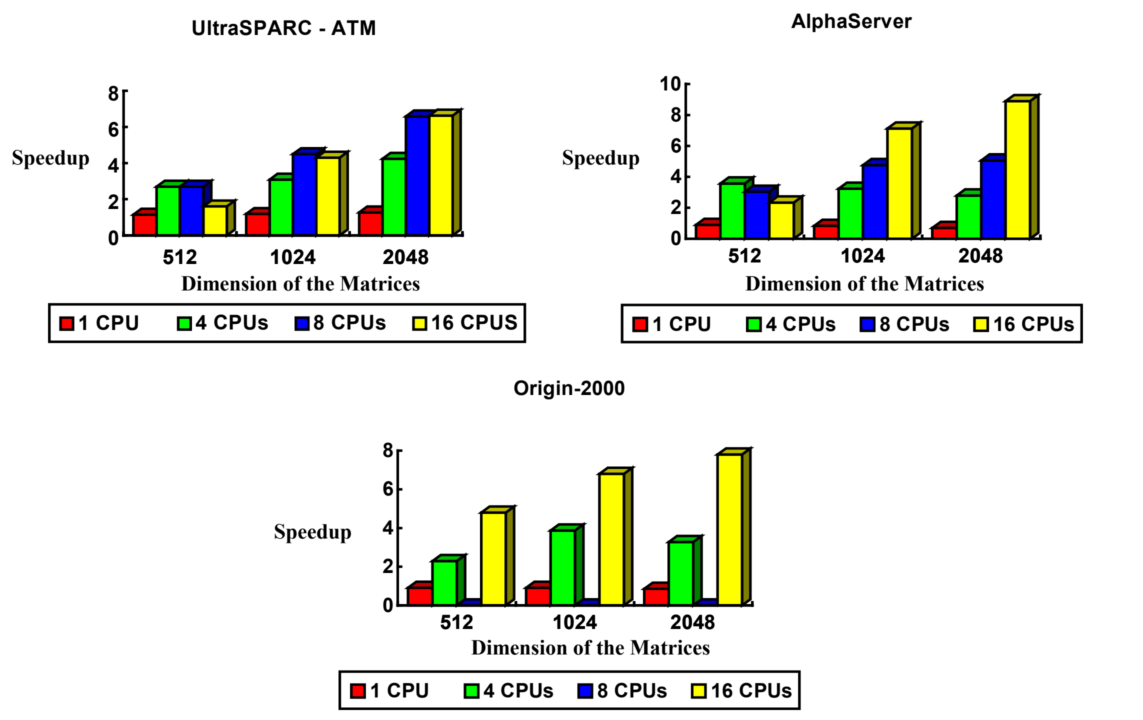

Figure 4.- *Speedups of parallel matrix multiply program.*

5.1 Matrix Multiply

The idea under MM is to measure the impact of replicating large objects, because the two matrices to be multiplied are fully replicated. The algorithm is based on dividing the result matrix (non replicable) in 4^N square submatrices, and computing them in parallel. The program is composed of three classes of distributed objects: matrix, multiplier and master. There is a multiplier object per processor, and each of them computes a submatrix. The master object reads the matrices from a file, creates the rest of the objects, starts the computation, detects the termination of the program, and displays the result matrix.

The multiplier objects acquire the matrices to multiply in non-exclusive mode, which implies that those matrices are transferred into a node when the first acquire access is performed. As the matrices to be multiplied are replicated, they are always local, and their state variables can be accessed directly.

In Figure 4, we present the results of multiplying square matrices with a size ranging from 512 to 2048. The speedups are calculated dividing the time obtained by a classical sequential matrix multiply program by the time obtained by the parallel program. Time is measured after reading the matrices, and when the computation termination has been detected.

The results are conditioned by the grain of the computation assigned to each multiplier object, that depends both on the size of the problem and the number of processors. For that reason, the speedups obtained multiplying 512-matrices are only close to linear when using four processors. On the contrary, with eight and sixteen processors, the speedups tend to increase when the size of the matrices is higher.

5.2 Branch and Bound

The branch and bound algorithm is a typical benchmarck for evaluating distributed languages. It is an enumerative technique that traverses an imaginary search tree whose nodes are subproblems of the original problem. The goal is to find the leaf-node with the best value of an objective function. In general, the parallelizaton of this algorithm follows some kind of replicated worker scheme, where the worker processes compute a search subtree.

The BB program is composed of four classes of distributed objects: worker, upperbound, buffer and master. There is a worker object per processor, and each of them computes a local search subtree. The global upperbound is an integer object that is invoked by the workers. As the upperbound is frequently read and rarely written (only when a new solution is found), it is replicated. The buffer object is used for load balancing and termination detection. The aim of the master object is to create the rest of objects, start the computation and print the final result.

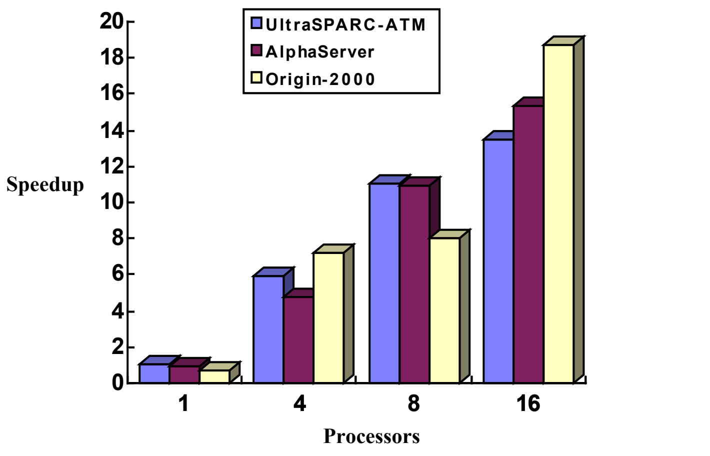

Figure 5.- *Speedups of parallel branch and bound program when solving a 100-city TSP.*

As the matrices to be multiplied in the MM program, the worker objects directly access to the state of the upperbound object to improve performance. In the case of a writting, this is possible because the exclusive acquire operation migrates the main copy to the node of the worker object, after invalidating all the replicas.

In Figure 5, we show the speedups obtained running the BB program to solve a 100-city TSP. To analyze these results, we must consider two issues. First, it is well-known that parallel branch and bound algorithms suffer anomalies that lead to superlinear speedups. This is due, basically, to the fact that the parallel program analyzes less nodes than the sequential one. Second, the presence of large caches in the processors can produce an important increase of performance if partitioned data fits into the caches. For example, each pair of processors in the Origin-2000 share a 4MB cache. As can be observed, we obtain a speedup close to 20 with sixteen processors in the Origin-2000.

6. Conclusions

We have presented an approach that allow us to replicate and migrate objects in the context of a concurrent object-oriented programming model. The approach is characterized by the necessity of acquiring and releasing objects and the distinction between query and command operations. This requirement imposes a strict programming style, but it yields information about when and how an object is going to be used. This information allow us to reduce the number of messages among nodes in the implementation by defining an entry consistency memory model. Objects can be replicated following a protocol that involves a migration scheme.

Details of a prototype have been discussed. The implementation is thread based, assigning a thread per object. It runs on three different architectures, each with sixteen processors: a network of workstations, a network of multiprocessors, and a multiprocessor. Portability has been achieved using C++, threads and sockets.

Performance measurements have been carried out using a matrix multiply program and a branch and bound program. The replicated objects are, in the first program, large matrices of float numbers, and, in the second one, an integer object. The results obtained from running the two programs show that significative speedups can be obtained with our approach in the three systems.

References

[BKT92] Bal, H. E., Kaashoek, M. F., Tanenbaum, A. S., Orca: A Language for Parallel Programming of Distributed Systems. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. Vol. 18, No. 3, March 1992.

[BZ91] Bershard, B. N., Zekauskas, M. J., Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. TECH. REPORT CMU-CS-91-170. 1991.

[CKP+93] Chien, A., Karamcheti, V., Plevyak, J., The Concert System - Efficient Runtime Support for Concurrent Object-Oriented Programming Languages on Stock Hardware. TECH. REP. UIUCDCS-R-93-1815, DEPARTAMENT OF COMPUTER SCIENCE, UNIVERSITY OF ILLINOIS, URBANA, Illinois, June 1993.

[Mey93] Meyer, B., Systematic Concurrent Object-Oriented Programming. COMMUNICATIONS OF THE ACM, vol. 36, no. 9. September 1993.

[Mos93] Mosberger, D., Memory Consistency Models. OPERATING SYSTEMS REVIEW, ACM PRESS. Vol. 27, No. 1. January 1993.

[NPT97] Nebro, A. J., Pimentel, E., Troya, J. M., Applying Distributed Shared Memory Techniques for Implementing Distributed Objects. ECOOP'97 WORKSHOP ON OBJECT ORIENTATION AND OPERATING SYSTEMS. Jyväskylä (Finland). June 1997.

[PTM96] Protic, J., Tomasevic, M., Milutinovic, V., Distributed Shared Memory: Concepts and Systems. IEEE PARALLEL AND DISTRIBUTED TECHNOLOGY. Summer 1996.

[TMY94] Taura, K., Matsuoka, S., Yonezawa, A., StackThreads: An abstract machine for scheduling fine-grain threads on stock CPUs. PROCEEDINGS OF WORKSHOP ON THEORY AND PRACTICE OF PARALLEL PROGRAMMING, number 907 in Lecture Notes on Computer Science, pp. 121-136. Springer-Verlag, 1994.