The Thread Migration Mechanism of DSM-PEPE

Federico Meza[1] and Cristian Ruz[2]

[1] Depto. de Ciencia de la Computación, Universidad de Talca
Camino Los Niches Km. 1, Curicó – CHILE
fmeza@utalca.cl
[2] Escuela de Ingeniería Informática, Universidad Diego Portales
Ejército 441, Santiago – CHILE
cruz@inf.udp.cl

Abstract. In this paper we present the thread migration mechanism of DSM-PEPE, a multithreaded distributed shared memory system. DSM systems like DSM-PEPE 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.
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 perfomance by enhancing spatial locality.

Keywords. Thread migration, distributed shared memory, multithreading, spatial locality.

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 [1]. 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 [2, 3]. The main idea is to move threads closer to the data they need, that is, to gather at the same processor those threads using the 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

Federico Meza was supported by Fondecyt under grant 2990074.

interprocessor communication while the latter attempts to increase utilization of the processors and hence the level of parallelism. If there are no 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 2 deals with the main issues involved in the implementation of multithreading and thread migration. In Section 3, the thread migration mechanism of DSM-PEPE is presented. Details about the experiments are covered in Section 4. Section 5 summarizes other works related to thread migration. Finally, Section 6 presents some concluding remarks and future lines of research.

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 in the target processor.

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.

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 arquitectures 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 [4].

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 1 shows the fields of the structure.

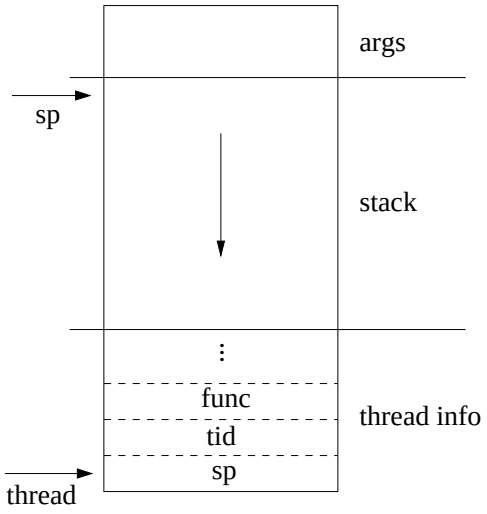

Fig. 1. DSM-PEPE thread structure

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 2 shows how a thread running at a processor is migrated to another location. Thread `t1` at `P1` is migrated to `P2` where it resumes execution on the suspended replicated thread that was created at `P2` when `t1` was originally forked. The thread that was running `t1` on `P1` now became suspended.

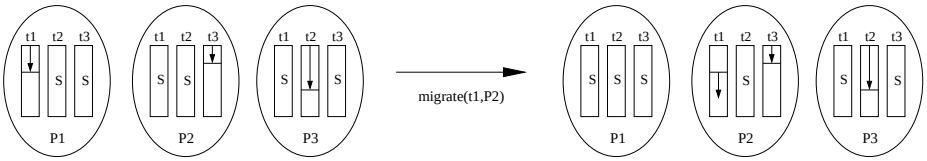

Fig. 2. Thread suspension and activation during a migration

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 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.

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 refered 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 refered 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 $\frac{1}{4}$ 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 transparently as the threads migrate. This application is refered 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 is composed of 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).

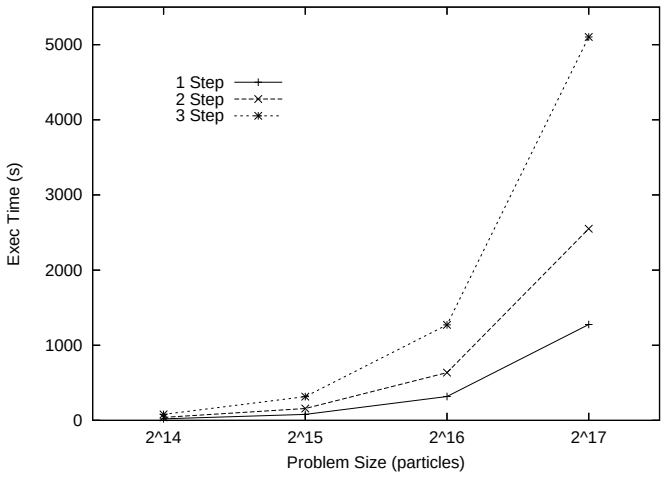

Fig. 3. Sequential N-body: Execution time grows as n^2

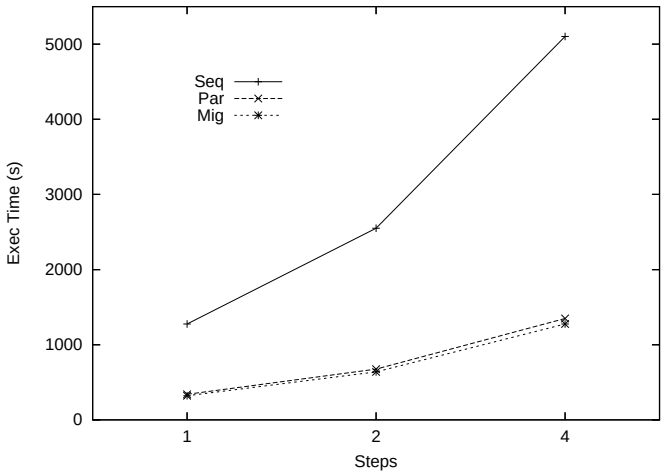

Fig. 4. Parallel and Migratory-Threads outperform Sequential N-body with speedups 3.77 and 3.99, respectively

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

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 1 shows the time involved in sending and delivering a message of different sizes, as measured in our testbed. Table 2 shows the time involved in migrating a thread using two different stack sizes, as measured in our testbed. It can be seen that the time involved in a thread migration is sligthly higher

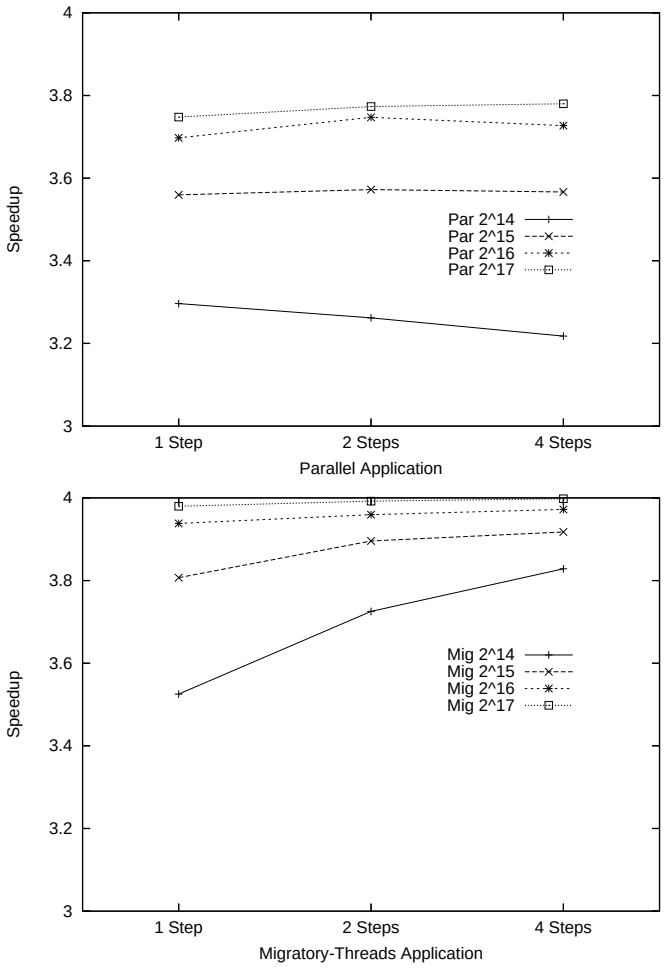

Fig. 5. Speedup for the Parallel and Migratory-Threads Applications

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.

Table 1. Time involved in message transmission

Size (KB)	Time (μsecs)
≈ 0	66
4	514
32	2971

Table 2. Time involved in thread migration

Stack size (KB)	Time (μsecs)
4	547
32	3445

For the largest problem size –2^{17} particles– and 4 steps of computation, the parallel application exchaged 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 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 –32 KB– while 28% are page-fault replies –4 KB– and 48% are short messages. This comparison can be seen in Figure 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 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.

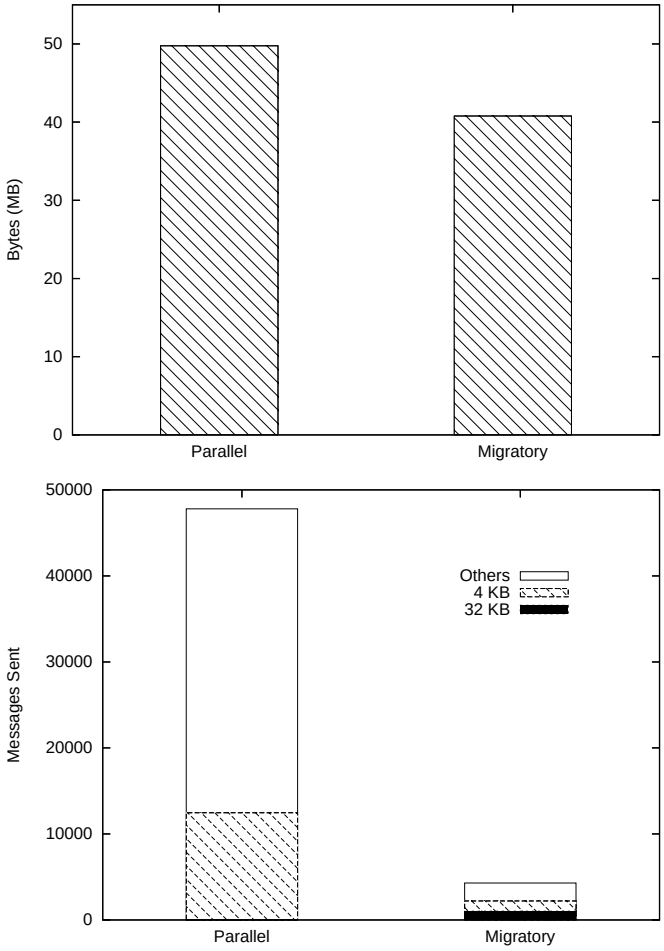

Fig. 6. Messages and data exchanged when computing 4 steps for 2^{17} particles

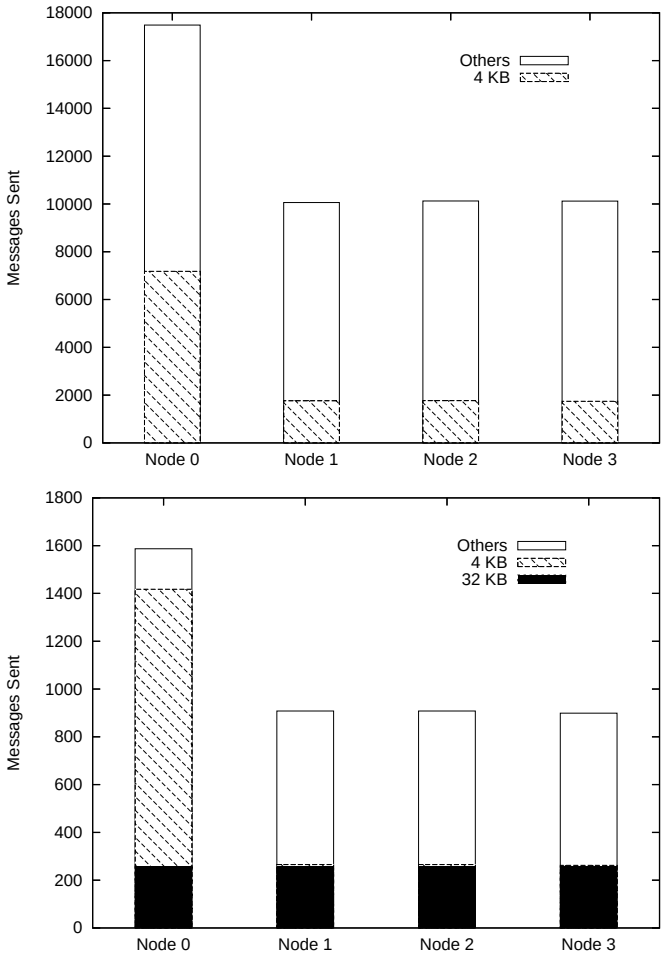

Fig. 7. Messages sent by each processor when computing 4 steps for 2^{17} particles

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 [5], Arachne [6], and Nomadic Threads [3, 7, 8]. 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 [9]. This is a user-level thread library. When a thread is migrated, its stack is inspectioned 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 [10], Nomad [11], and Amber [12]. 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 [13–15], that use preprocessing and run-time support to deal with the migration of the threads stacks.

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 escence 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.

References

1. Thitikamol, K., Keleher, P.: Per-Node Multithreading and Remote Latency. IEEE Transactions on Computers **47** (1998) 414–426
2. Thitikamol, K., Keleher, P.: Thread migration and communication minimization in DSM systems (invited paper). Proceedings of the IEEE **87** (1999) 487–497
3. Jenks, S., Gaudiot, J.L.: An evaluation of thread migration for exploiting distributed array locality. In: Proceedings of the 16th Annual Internation Symposium on High Performance Computing Systems and Applications (HPCS'02), IEEE Computer Society (2002) 190
4. Meza, et al.: On the Design and Implementation of a Portable DSM System for Low-Cost Multicomputers. In: Computational Science and its Applications. Volume 2667 of Lecture Notes in Computer Science., Springer (2003) 967–976
5. Jul, et al.: Fine-Grained Mobility in the Emerald System. ACM Transactions on Computer Systems **6** (1988) 109–133
6. Dimitrov, B., Rego, V.: Arachne: A Portable Threads System Supporting Migrant Threads on Heterogeneous Network Farms. IEEE Transactions on Parallel and Distributed Systems **9** (1998) 459–??
7. Jenks, S., Gaudiot, J.L.: A multithreaded runtime system with thread migration for distributed memory parallel computing. In: Proceedings of High Performance Computing Symposium. (2003)
8. Jenks, S.: Multithreading and thread migration using mpi and myrinet. In: Proceedings of the Parallel and Distributed Computing and Systems (PDCS'04). (2004)
9. Mascarenhas, E., Rego, V.: Ariadne: Architecture of a Portable Threads System Supporting Thread Migration. Software – Practice and Experience **26** (1996) 327–356
10. Itzkovitz, et al.: Thread Migration and its Applications in Distributed Shared Memory Systems. Journal of Systems and Software **42** (1998) 71–87
11. Milton, S.: 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 (1998)
12. Chase, et al.: 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 (1989) 147–158
13. Jiang, H., Chaudhary, V.: MigThread: Thread Migration in DSM Systems. In: Proceedings of the ICPP Workshop on Compile/Runtime Techniques for Parallel Computing. (2002)
14. Jiang, H., Chaudhary, V.: Compile/Run-time Support for Thread Migration. In: 16th International Parallel and Distributed Processing Symposium, Fort Lauderdale, Florida (2002)
15. Jiang, H., Chaudhary, V.: On Improving Thread Migration: Safety and Performance. In: Proceedings, 9th International Conference on High Performance Computing — HiPC2002. Volume 2552 of Lecture Notes in Computer Science., Springer-Verlag (2002) 474–484