12th International Parallel Processing Symposium and 9th Symposium on Parallel and Distributed Processing, March 1998, Orlando, Florida

Efficient Fine-Grain Thread Migration with Active Threads

Boris Weissman*, Benedict Gomes*, Jürgen W. Quittek†, Michael Holtkamp‡
{borisv, gomes, quittek, holtkamp}@icsi.berkeley.edu

* University of California at Berkeley and International Computer Science Institute
† International Computer Science Institute, currently at CCRL NEC Europe Ltd.
‡ Technical University of Hamburg-Harburg and International Computer Science Institute

Abstract

Thread migration is established as a mechanism for achieving dynamic load sharing. However, fine-grained migration has not been used due to the high thread and messaging overheads. This paper describes a fine-grained thread migration system whose extensible event mechanism permits an efficient interface between threads and communications without compromising the modularity and performance of either. Migration is supported by user level primitives based on which applications may implement different migration policies. The system is portable and can be used directly or serve as a compilation target for parallel languages. The system runs on a cluster of SMPs and observed performance is orders of magnitude better than other reported measurements.

1 Introduction

In recent years, the multi-threaded programming model has grown increasingly popular, propelled in part by the availability of SMPs. Programming languages make use of threads both for expressiveness (particularly in GUI) and to exploit the parallelism of SMPs. The convenience of the model arises from the ease of sharing data, and automatic load balancing since threads, once blocked, may resume on any processor.

In a hybrid distributed system with SMPs as nodes, it is necessary to either provide a mixture of programming styles or to extend a traditional distributed or shared memory programming style. Given these choices, extending a simple and powerful thread programming model to a distributed system is appealing. To achieve this it is necessary to support a) a form of distributed shared memory to permit the sharing of data and b) thread migration to achieve the relatively transparent load balancing. A limited form of migration is remote thread creation. However, migration after creation is necessary for load balancing, particularly with long running threads.

Why not existing threads? Performance! While modern thread packages are adequate for medium-grained concurrency, the overheads are unacceptable for the fine-grained case. To lower thread overheads, many have turned to simpler, non-blocking threadlets (Filaments [8], Cilk [2], Multipol [14]).

Why not threadlets? Though non-blocking threadlets do provide high performance, they are not adequate in their expressiveness for modern explicitly parallel object-oriented languages (including Java, Ada, and locally designed Sather [12]). Using threadlets requires shifting to a radically different programming style, such as continuation passing or implicitly parallel functional programming.

Why is thread performance poor? There are several reasons for the poor performance of commercial thread packages. Kernel level threads, as found in NT involve expensive kernel traps. Even the user level commercial thread packages have high thread creation costs targeted primarily at user interfaces.

Why is migration performance poor? Many migration systems make use of kernel traps for either the thread [7] or communication system [10][11]. The degree of integration between the two is also a factor. For instance, Nexus [5] presents an interface to a combined thread and communication system; however, it makes use of off-the-shelf components.

Improving Performance. We demonstrate a user level thread package, Active Threads, with thread creation overheads comparable to threadlets. We also utilize user level messaging, with support for concurrent access to the network interface. A novel feature is an extensible event mechanism that permits integration between the thread and message passing systems. This paper makes the following contributions:

- An architecture for a high performance fine-grained thread migration system for clusters of SMPs.
- An extensible thread event mechanism that permits efficient integration of thread and communication systems.
- Detailed performance metrics for migration. Migration is completely at user level, permitting a better cost analysis.

2 The Migration Interface

Thread migration is triggered by the following calls into the thread package. The return value indicates success or failure. The first argument refers to the thread bundle (Section 4). The general calls have variants that specify exactly which thread to migrate. A thread may elect to either push itself or any available (i.e. runnable) thread to the destination node. The steal calls may elect to either steal any available work or to steal data from a particular other node. In addition, asynchronous versions of all calls are also provided.

```
int steal(bundle_t b);
  Steal a thread from any other node.
int steal_from(bundle_t b, node_t from);
  Steal a thread from the node 'from'.
int push(bundle_t b, node_t to);
  Push any other blocked thread to the node 'to'
int push_self(bundle_t b, node_t to);
  Push the current thread to the node 'to'
```

3 Transferring State

In order to move a thread to a different processor, it is necessary for the transferred thread to correctly access all related resources. Data local to the thread (i. e. stack and thread-local heap) may be copied to the destination. However, since the addresses on the target processor may be different from the original addresses, internal pointers may no longer be valid.

Two approaches may be used to handle local pointers. In the first approach, pointer values may change after migration and must be updated. The second approach is to partition the address space such that local pointers maintain the same values.

In order to update pointer values it is necessary to register all stack pointers. If registration is left to the programmer, pernicious bugs can arise. A more basic issue is that it is frequently impossible to identify pointers cached in registers, unless a great deal of compiler and language level support is provided. Ariadne [10] requires user registration but provides no solution to handling registers as pointed out in [7]. Emerald [9] is a full fledged system, with compiler and even kernel level support. Even with such support, updating pointers exacts a run-time penalty over and above the cost of moving the thread data.

Since we were interested in a portable, high performance system, we chose the approach preserving pointer values. A pre-defined area of the virtual memory space is reserved for each thread stack on all nodes. A similar approach was used by Millipede [7], Amber [3], and UPVM [4]. Indeed, Amber, a successor to Emerald, chose this approach for similar reasons [3]. In this solution, the total number of threads is limited by the single node address space. However, with the increasing availability of 64 bit address spaces, or on smaller sized clusters, this limitation is not a serious issue.

In addition to thread local data, a thread may also access data shared by multiple threads such as synchronization objects. After migration, if pointers to this shared state are to still be valid, it is necessary for them to indicate the nodes on which the data resides. A simple solution is to partition the global address space. References to non-local data result in a trap, which can either forward the reference to the proper location or transfer the required data. Another approach is to use global pointers, which create the illusion of a shared address space.

In the parallel language Sather (which uses Active Threads as a compilation target), the node location is encoded in the unused high bits of the address. In the C++ based library [6], we implemented a set of classes encapsulating global pointers.

Though the migration system requires a shared address space, it does not depend on any particular implementation.

4 The Thread System

Active Threads is a general-purpose light-weight thread package for that can handle millions of threads on a variety of platforms. More details and API are given in [15]. We concentrate on the aspects that enable us to integrate threads with communications and implement fast thread migration.

Groups of logically related threads with common properties are organized into *bundles*. All threads in a bundle share the same scheduling policy. An application may create bundles with different scheduling policies such as FIFO, LIFO, priority, etc.

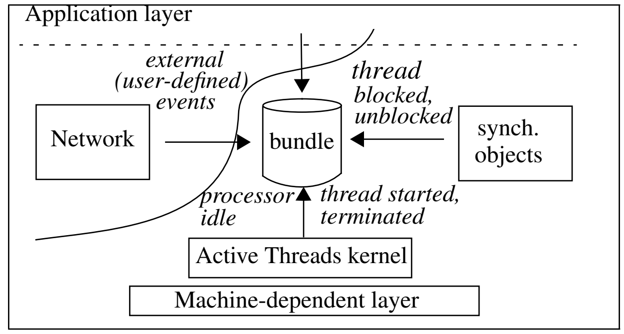

Figure 1: Scheduling events in Active Threads

Active Threads supports a general scheduling event mechanism upon which different scheduling policies are built. Different susbsytems communicate with bundles by vectoring *scheduling events*. Bundles encapsulate all aspects of scheduling by providing scheduling event handlers. The direction of the event flow is shown in Figure 1.

All events are partitioned into two groups: internal Active Threads events and external (user-defined) events. Internal events deal with common thread operations: thread creation, termination, blocking, unblocking, dispatching by the processor, etc. There is a total of 8 internal events [15].

A user can also define events logically originating outside Active Threads and register event handlers for these events. For instance, such events can be used to check the network for incoming messages whenever a processor becomes available or to perform a unit of garbage collection when all node processors are idle. This functionality is also used to implement thread migration transparently to the rest of the thread system.

5 Communications

Aside from thread overheads, the other major overhead involved in thread migration arises from messaging systems. Much of it is caused by repeated copying of messages to and from buffers in the OS space. The Active Message (AM) approach [13] eliminates this overhead by handing the delivered message directly to a user-level handler. Although our thread migration mechanism does not rely exclusively on the communications style of AM, user-level communications is indispensable for low migration latencies. We have implemented a variant of the Berkeley AM that supports the general AM communications style, while extending it in some important ways.

The AM system, as originally conceived and available in the current Berkeley distribution, was not designed to run on multi-threaded SMPs. Multi-threading raises the possibility of contention for the network device; consequently, all accesses to the messaging system must be properly protected.

Another extension of AM functionality deals with the kinds of operations allowed in the remote handlers. Active Message handlers are non-blocking and complex protocols must be implemented in applications to achieve the desired functionality. Other researchers have pointed out similar problems [5].

Our communication system retains the Active Messages

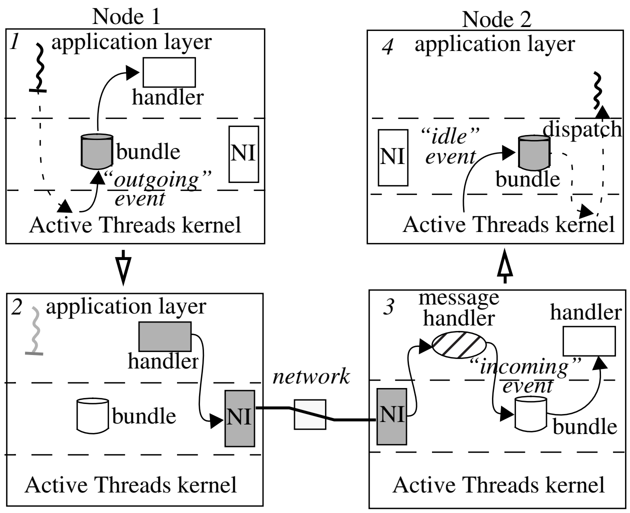

Figure 2: Events and thread migration.

programming style and high performance. In addition, it supports SMPs, and enables remote thread creation and blocking message handlers.

6 Thread Migration Mechanism

We view migration as a natural extension of thread scheduling to a distributed environment. Migration is implemented by defining new bundles that are extended with special user-defined migration events. Migration happens transparently to other thread activities and bundles and threads that do not support migration need not be concerned with it. Different migration policies are implemented by defining new bundles.

We consider an example of a thread relocating itself to a different node. To enable any existing bundle to support this functionality, only two new user-defined events are added:

1. *outgoing* event; generated by the Active Threads kernel on behalf of the user thread after it is stopped and its context is saved. The event is handled by the *outgoing* event handler which is responsible for all transfer operations.
2. *incoming* event; generated externally to the thread system by the network message handler on message arrival. The incoming event handler unpacks the received thread.

Figure 2 captures the essence of the underlying mechanism. At time 1, a user thread executing on node 1 calls *push_self.* The Active Threads kernel stops the calling thread and saves its registers at the top of its stack. It then dispatches an *outgoing* thread event on behalf of the stopped thread to its bundle.

At time 2, the *outgoing* event handler catches the event vectored at time 1. As an optimization, the handler packs all the pieces of the thread's state such as the thread control block (TCB), thread-local storage (TLS), and the thread stack into a contiguous area. The handler then initiates the network transfer by handing the message over to the network interface (NI).

At time 3, the incoming message handler of the destination node pulls the thread's state off the network and dispatches an *incoming* event to the destination bundle. The *incoming* event handler unpacks the message, obtains a new TCB from the thread system, initializes it with the proper values and adds a new thread to the bundle's internal scheduling data structures.

Finally, at time 4, the Active Threads kernel detects that one of the processors of node 2 is idle and dispatches a *processor idle* event. At this point, the newly arrived thread is indistinguishable from other threads and is dispatched for execution.

The other primitives in the migration interface are handled analogously. For *push()*, the first step is simplified since the migrating thread is already stopped. Variants of *steal()*, initiate migration by sending a message to the thread supplier node first. The subsequent steps are the same as for *push()*.

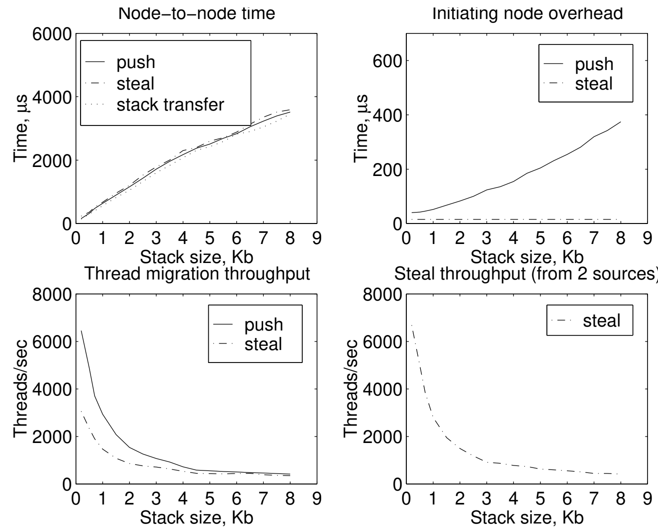

Figure 3: Migration latency and overhead.

7 Microbenchmarks

Our main migration platform is a cluster of SPARCstation-10 connected by Myrinet [1]. Each workstation has four 50 Mhz HyperSPARC processors. The Myrinet NI is an I/O card that plugs into the standard SBus. It contains a 32-bit RISC "LANai" network processor, DMA engines, and local memory (SRAM). The NI's memory is mapped into the user process address space and can be accessed through load/stores to mapped main memory addresses. The Myrinet network consists of crossbar switches with eight bidirectional ports that can be linked into arbitrary topologies. A one way latency for an AM message of 5 words is 17µs. A bulk transfer of 1K takes 560µs and is mostly constrained by the host I/O bus speed.

The costs of thread migration operations for our configuration are presented in Figure 3 (average over 1,000 migrations)

Point-to-point time for a push includes blocking a thread on a source node and resuming at the destination. It was measured by having a thread ping-pong between two nodes. A similar measurement for steal involves thread chains: a stolen thread initiates the next steal upon arrival. Both costs are dominated by the stack transfer time. Steal is slightly more expensive than push because of an extra initiating message. Migrating a null thread (a stack of 112 bytes for SPARC v8 and gcc 2.7.1) to a remote node takes 150µs, while pushing a thread with an active stack of 1Kb takes 630µs.

Many load balancing strategies avoid paying the full price by initiating migration early while the processors are still busy. Thus, the cost of push can be reduced to a small initial overhead. The source node resumes the computation while the transfer is still in progress. This overhead is dominated by the copying of the thread state to NI's SRAM. This copy overhead is, in turn, determined by the I/O bus speed and grows linear as the active thread stack size increases. For instance, the push overhead for a null thread is 40µs and the overhead for a thread with an active stack size of 1Kb is only 52µs.

The steal overhead consists of two parts - the initiation on the requesting node and the service overhead on the remote node. The former is fixed and fairly small - only 8µs on our system. The latter is exactly the thread push overhead.

For applications with small tasks, but dynamic loads, frequent migration is needed and throughput rather than latency is the key factor. Active Threads achieves a fairly high throughput of thousands of threads per second.

The last graph of Figure 3 demonstrates the scalability of our implementation. The throughput for small threads is doubled when a node prefetches from two sources simultaneously.

It is hard to directly compare the performance of thread migration systems because of hardware differences such as processor and network speeds and the memory hierarchy. Reported performance parameters vary. Still, we find it instructive to list the performance characteristics of several recent systems, along with their hardware characteristics.

System		null	1K	2K
Ariadne (SS5/Ethernet)			11ms	14ms
Millipede (Pentium/fast Ethernet)		2ms		?70ms?
PM^2 (ALPHA/single processor)				2.2ms
ActiveThreads (SS-10/Myrinet)	(overhead)	40µs	52µs	68µs
	(latency)	150µs	630µs	1.1ms

Table 1: Migration cost as function of stack size.

The numbers in Table 1 warrant some explanation. In terms of raw processing power, Active Threads run on a modest platform - all the other systems employ hardware with the clock speed several times faster. Unlike previous systems, threads are migrated between SMPs with additional protection cost.

Millipede [7] reports the "average latency" of migration of 70ms; however, the stack size is not specified. In our experiments, the average active stack size was between 1K and 2K.

The PM^2 [11] latencies are for thread migration between processes on a single processor DEC ALPHA workstation using PVM for interprocess communications.

Both overhead and point-to-point latencies for Active Threads are presented in Table 1. In contrast to the other systems, we provide user-level threads and communication. This allows us to keep overheads small, relative to the full operation latencies. The overhead on other systems, although not reported, must account for most of the point-to-point latencies: Ariadne [10] traps to the OS kernel for TCP/IP; Millipede is based on kernel-level threads and traps to the kernel for thread operations; PM^2 traps to the kernel for interprocess communication. As a result, Active Threads latencies are at least an order of magnitude lower than those of the other systems while the migration overhead can be several orders of magnitude lower.

8 Application Studies

Because of space limitations, we present a single application, a threaded version of "grep", a Unix utility for regular expression search. Other data can be found in [16][6]. "at_grep" is based on a version by Ron Winacott[1]. It extends the standard grep with new features such as the recursive directory search.

The basic structure of at_grep is simple[2]: it parses the input regular expression and then starts a recursive search over the directory tree. There are two types of threads: *search* and *cascade*. A search thread is created for non-directory files found while descending the directory tree. Such a thread searches a single file. A cascade thread is created whenever the search comes across a new directory. Cascade threads are essentially producer threads - they call into the OS to obtain the directory information and then create new search and cascade threads.

Steal-based migration policy. We recompiled our application for the distributed platform. Search threads were designated migratable and desirable migration points were marked. In the absence of user specifications, a default migration policy enables an idle processor to steal work from any remote node. These minimal changes resulted in a fairly competitive distributed implementation of grep. Figure 4 presents the speedup for the command "*at_grep mutex*" invoked on the Sather runtimes sources (400 files in 50 directories, includes Active Threads).

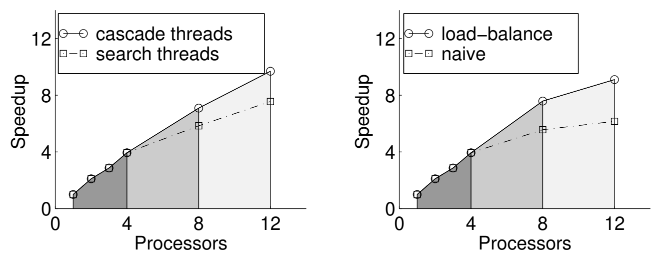

Figure 4: at_grep steal (left) and push (right)

The program was then modified to designate cascade threads as migratable. Although this means potentially greater load imbalance, all our runs resulted in better performance.

Push-based migration. We next investigated direct thread migration via explicit push calls. Two heuristics were tried:

1. *naive round-robin*. A single producer node finds files, creates threads and pushes them to other nodes.
2. *load-balancing*. Similar to the above, but the producer node keeps track of the load of other nodes by recording

1. The original sources are available from Sun Microsystems at http://www.sun.com/workshop/sig/threads/apps.html

2. The original version used relatively expensive Solaris threads and a complex work-bag structure to avoid thread creation. With more light-weight Active Threads, this complexity was no longer necessary and was eliminated

the file sizes for all previously pushed search threads.

While the load-balancing heuristic resulted in performance close to that of the best stealing heuristics, no further improvement was obtained. Instead of exploring more complex load-balancing policies, we have chosen to exploit the file location information to guide thread migration.

Locality-guided migration. All SMPs in our network have local disks. While searching through large repositories, grep routinely examines remote files. In fact, most applications of grep in our networking environment follow this pattern.

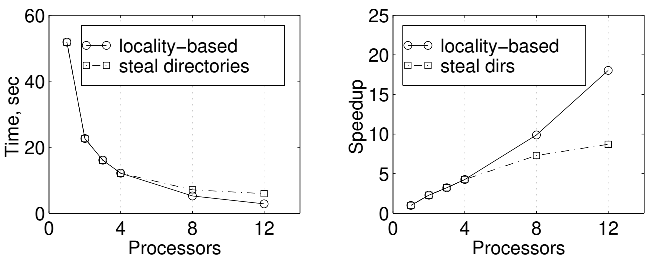

Figure 5: Effects of locality-guided migration.

We have modified the push-based version to exploit file location information. If after start-up a search or cascade thread determines that data resides on a remote workstation, it pushes itself to that machine. Figure 5 compares performance of the locality guided policy with that of the best stealing policy. We observed super-linear speedups for large distributed repositories. This is possible since, in the single node case most data is remote. A combination of thread migration with file location information allowed us to keep the grep response time interactive even for fairly extensive searches. In our experiments, single processor execution takes almost a minute, while a distributed locality-guided execution takes 2 seconds for the same inputs and patterns.

Thread granularity and migration. Figure 5 shows the thread lifetime distribution for two cases: a) local data, b) remote data. The threads are quite fine-grained in both cases, (remote median 9.97 ms, local 3.98 ms). The migration cost must be significantly lower than the average thread life-time to be worthwhile. With the other thread migration systems, thread migration may not produce useful results for this example. Since the granularity at which thread migration is useful is determined by the thread migration overhead, high-performance migration has a qualitative effect on the programming style.

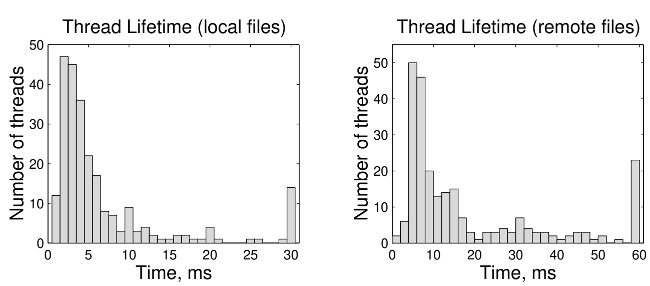

Figure 6: Thread lifetime distribution for grep.

9 Conclusions

We have demonstrated a portable thread migration system which achieves high performance by closely integrating threads with messaging based on an extensible event mechanism. The migration system is implemented as a set of library calls independent of any specific migration policy.

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. in IEEE Micro, 15(1), Feb. 1995.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and J. Zhou. Cilk: An Effcient Multithreaded Runtime System, in Journal of Parallel and Distributed Computing, Vol. 37. No 1, August 1996. pp 55-69.

[3] J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy, R. J. Littlefield. The Amber System: Parallel Programming on a Network of Multiprocessors, in ACM Symposium on Operating System Principles, December 1989

[4] J. Casa, R. Konuru, S. W. Otto, R. Prouty, J. Walpole. Adaptive Migration systems for PVM, in Supercomputing ‘94, pp 390 - 399, Washington D.C., November 1994

[5] I. Foster, C. Kesselman, S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication, in Journal of Parallel and Distributed Computing, Vol. 37. No 1, August 1996. pp 70-82.

[6] M. Holtkamp. Thread Migration with Active Threads. International Computer Science Institute, Technical Report TR-97-038.

[7] A. Itzkovitz, A. Schuster, L. Wolfovich. Thread Migration and its Applications in Distributed Shared Memory Systems, to appear in: The Journal of Systems and Software, 1997

[8] D. K. Lowenthal, V. W. Freeh, G. R. Andrews. Using Fine-Grain Threads and Run-Time Decision Making in Parallel Computing, in Journal of Parallel and Distributed Computing, Vol. 37. No 1, August 1996. pp 41-54

[9] E. Jul, H. Levy, N. Hutchinson, A. Black. Fine-Grained Mobility in the Emerald System, in ACM Transactions on Computer Systems, Vol. 6, No. 1, pp 109 - 133, Feb. 1988

[10] E. Mascarenhas, V. Rego. Ariadne: Architecture of a Portable Threads System Supporting Thread Migration, in Software - Practice and Experience, Vol 26(3), pp 327 - 356, March 1996

[11] R. Namyst and J. Mehaut. PM2: Parallel Multithreaded Machine. A computing environment for distributed architectures. www.fifl.fr/~nemyst/pm2.html

[12] D. Stoutamire, S. Omohundro. The Sather 1.1 Specification. ICSI Technical Report TR-96-012.

[13] T. von Eichen, D. Culler, S. Golstein and E. Schauser. Active Messages: a mechanism for integrated communication and computation, in Proceedings of the 19th International Conference on Computer Architecture 1992.

[14] C.-P. Wen, S. Chakrabarti, E. Deprit, A. Krishnamurthy, K. Yelick. Runtime Support for Portable Distributed Data Structures, in Workshop on Languages, Compilers, and Runtime Systems for Scalable Computers, May 1995.

[15] B. Weissman. Active Threads: an Extensible and Portable Light-Weight Thread System. ICSI TR-97-036, Oct. 1997

[16] B. Weissman, B. Gomes, J. W. Quittek, M. Holtkamp, A Performance Evaluation of Fine-Grain Thread Migration with Active Threads, ICSI Tech. Report TR-97-054, Dec. 1997.