Self-Migrating Threads for Multi-Agent Applications

Naoya Suzuki Munehiro Fukuda
Institute of Information Sciences and Electronics
University of Tsukuba, Tsukuba, Ibaraki 305-8573, JAPAN
e-mail: {nas, fukuda}@is.tsukuba.ac.jp

Lubomir F. Bic
Department of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
e-mail: bic@ics.uci.edu

Abstract

We propose "self-migrating threads" as a new cluster computing paradigm for multi-agent applications, which can be viewed as the interactions among autonomous computing entities, each having its own objectives, behavior, and local information in a synthetic world. Self-migrating threads have both navigational autonomy of mobile agents and fine computation granularity of threads. They are also given the capability to construct system-wide logical networks, representing synthetic worlds. With those aspects, we expect that self-migrating threads provide multi-agent applications with good programmability and performance. We have designed the functionality of self-migrating threads and implemented a low-level migration library. In this paper, we discuss the feasibility of our design by considering the implementation techniques and basic migration performance.

1 Introduction

A recent programming trend in artificial intelligence and ecological simulations is taking the multi-agent approach, where applications are viewed as the interactions among autonomous objects, each having its own goals, behavior, and local information in a synthetic world [6]. Of interest is how to parallelize such applications for cluster computing.

The navigational autonomy of mobile agents have the potential to form a novel programming paradigm for multi-agent applications. We have demonstrated this potential with our MESSENGERS system where the mobile agents construct and roam over application-specific networks [1]. However, such applications sometimes demand finer granularity and larger scalability than is currently possible. For instance, some ecological simulations may need millions of entities, and some synthetic worlds may consist of very large numbers of network nodes and links. Currently, MESSENGERS demonstrates its competitive performance only in coarse- to medium-grained distributed computing due to the nature of interpretation [8]. Therefore, it is not feasible to use MESSENGERS directly for fine-grained parallel computing, now available on a cluster of PCs/workstations linked to high-speed network such as Myrinet [2].

Another software technology that migrates computations is based on threads. It migrates threads to remote processors in order to maintain good load balancing and reduce remote memory accesses. At a glance, threads, with their migration capability, may match fine-grained parallel computing of multi-agent applications. However, most existing systems have not aimed at such applications and thus do not support some important features of multi-agent applications: navigational and behavioral autonomy, dynamic construction of global synthetic worlds, and their scalability.

As a new cluster computing paradigm for multi-agent applications, we propose self-migrating threads that autonomously navigate over networks and resume their native-mode executions at each destination. We give them the same capability as MESSENGERS to construct system-wide logical networks at runtime. Therefore, these threads can behave as autonomous lightweight agents roaming over and changing their synthetic world. We have designed the functional specification of self-migrating threads that are coded using C++ and special navigational constructs. To study their performance, we have implemented a low-level thread migration library, had threads migrate among four workstations randomly, and obtained results competitive with those of similar computations using MESSENGERS, IBM Aglets [12], PVM, and MPI [10].

This paper demonstrates the feasibility of self-migrating threads for multi-agent applications from the programming and performance view points. Section 2 introduces our programming paradigm. Section 3 discusses some implementation techniques and shows preliminary performance. Section 4 differentiates our design from other thread migration systems in view of multi-agent applications. Finally, section 5 concludes the discussions.

2 Execution Model

The design of self-migrating threads is based on the following four principles: (1) preserving MESSENGERS' network architecture, (2) permitting users to describe threads and the network in C++, (3) providing flexible logical-to-physical network mapping schemes, and (4) eliminating overhead in logical network management.

2.1 Computational Network and Objects

We provide self-migrating threads with three levels of networks (see Figure 1.) The lowest level is the *physical network* (a SPARCstation cluster in our implementation), which constitutes the underlying computational resource. Superimposed on the physical layer is the *daemon network*, where each daemon is a UNIX process executing and exchanging threads with others. The *logical network* is an application-specific computation network dynamically constructed by self-migrating threads on top of the daemon network.

Computation involves four classes of C++ objects: *thread*, *daemon*, *node*, and *link*. The *thread* object is a self-migrating thread, which carries its data as it migrates over the *logical network*. The *daemon* object contains daemon node information, shared by all *thread* objects that are running on the same daemon process. The *node* object represents a logical network node, whose method and data members are accessed by *thread* objects residing on this node. The *link* object corresponds to a logical network link, capable of storing only data members and visible from threads residing on the both ends, (i.e., nodes) of this link.

The naming scheme of those objects is as follows: The *daemon* object is identified by a system-unique ID from all daemons. The *node* is identified by the concatenation of its *daemon* ID and daemon-local *node* ID, and thus referred to as *(daemon_ID/node_ID)*. The

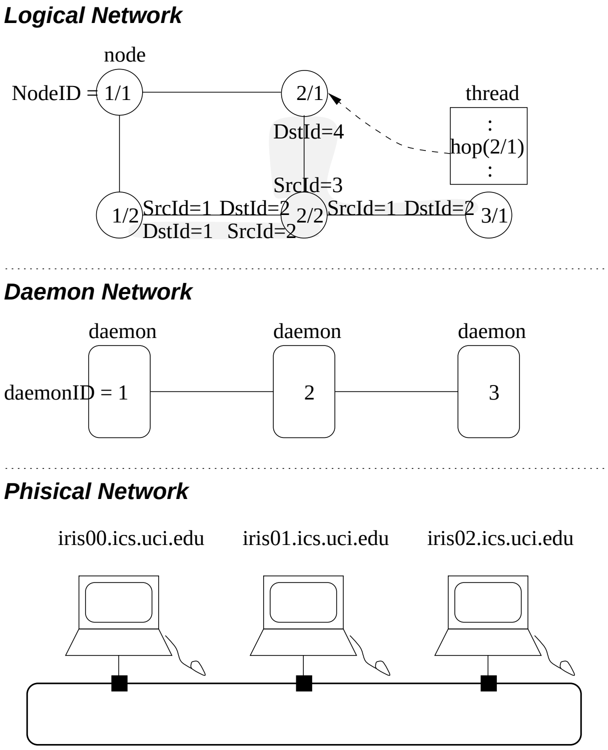


Figure 1. Computational network

link is recognized from both ends, (i.e., *nodes*), using a pair of source and destination *link* IDs. The source *link* ID is visible as the destination *link* ID at the other end, and vice versa. Figure 1 shows that the source and destination IDs of the *link* between two *nodes*, (1/2) and (2/2) are visible as 1 and 2 from *node*(1/2), but as 2 and 1 from the other *node*(2/2). Note that IDs are used for generation of and navigation to network objects but not for direct access to the object contents. A *thread* must first migrate to the object, whose contents are then accessible to it.

2.2 Principles of Operations

Figure 2 describes a C++ code framework which defines the *node*, *link* and *thread* classes as inheriting their base classes, provided by the system. Note that the daemon process provides the *daemon* class and the main() function, which do not need to be defined by a user. As shown in this figure, a user may define different *thread* classes derived from its base class. After initialization, new *thread* instances pass the control to the *body()* member function that is in charge of their autonomous network navigation.

When invoked, a daemon process instantiates a *daemon* object indicating its status, as well as two spe-

```
class Node : BaseNode { ...; };
class Link : BaseLink { ...; };
class Thread1 : BaseThread {
  public: body ( ); ...; };
class Thread2 : BaseThread {
  public: body ( ); ...; };
```

Figure 2. Network objects defined in C++

cial *nodes*, called *INIT* and *TRASH* respectively. New *threads* are injected from a user shell. Their executable images are then loaded into daemons if necessary. The *threads* start their navigation from the *INIT* node and terminate themselves by migrating to *TRASH*. *Threads* are capable of constructing and destructing other objects with *new* and *delete* operators during their execution. Those operators take a special form extended from the original C++ and return the ID of a given object on a successful completion. In the following, we will explain how a *thread* constructs and navigates over network objects.

Node Construction/Destruction: A *node* object is constructed and deleted as follows:

```
typedef int DaemonID, NodeID;
DaemonID daemonId;
NodeID localId;
localId = new Node(arguments)
          : ID(daemonId, localId);
localId = delete Node
          : ID(daemonId, localId);
```

The *new* operator creates and initializes a *node* with *arguments* only when the daemon with *daemonId* does not yet have the *node* with *localId*. Null *daemonId* and *localId* mean a *node* construction on the current daemon and a local node ID given by the system respectively. Similarly, the *delete* operator succeeds in a *node* deletion when such a *node* exists as a skeleton.

Link Construction/Deletion: A *link* object is constructed and deleted as follows:

```
typedef int LinkID;
LinkID sourceId, destId;
sourceId = new Link(arguments)
           : ID(daemonId, localId,
                sourceId, destId);
node.getLink();
sourceId = delete Link
           : ID(node.link[i]);
```

The *new* operator creates a *link* with *sourceId* and *destId* from the current node to the one with *daemonId* and *localId*, and initializes it with *arguments*, provided such a *link* does not already exist. Null *sourceId* and *destId* mean receiving this pair of IDs from the system. The current *node*'s *getLink()* member function generates a source ID list of all the emanating *links*, each of which is accessible with an index i and then deleted by the *delete* operator. Note that the distinction between *node* and *link* in creation or deletion is made with arguments given to the *ID()* function, which is overloaded accordingly.

Thread Navigation: The *thread* propagates along *links* and/or jumps directly to *nodes* using its member function *hop()*. Its execution is not interrupted by other *threads* residing on the same *node* unless it initiates a navigation.

```
node.getLink(sourceID, search, modify);
hop(node.link[i]);
```

The *node.getLink()* function, if given with *sourceID* and/or *search()*, generates a source ID list of the emanating *links* that either are identified with *sourceID* or satisfy the conditions provided by *search()*. It also modifies the contents of the selected *links* through a given *modify()*. The *thread* can then move itself along each of those *links* with *hop(node.link[i])*. The *hop()* is capable of multicasting the *thread*, depending on the variation of its arguments:

hop(node.link): propagates along all selected *links*.

hop(daemonID, localID): jumps to the *node* with *daemonID* and *nodeID*.

hop( ): jumps to the same current *node*; this is used to relinquish the CPU.

hop(TRASH): jumps to the *TRASH node*, thus terminating the *thread*.

As shown above, with only the *hop* method, our self-migrating threads can initiate the following actions necessary during their course of execution: network navigation, context switch, and termination.

2.3 Coding Multi-Agent Applications with Self-Migrating Threads

Figure 3 describes an ant evolution program, where ants search for food on a synthetic land. The application can be coded using two *thread* objects: *land* and *ant*. The *land thread* constructs a synthetic land, represented by an $N \times N$ toroidal grid of *nodes* on N daemons. The *ant threads* roam over the land by hopping from one *node* to another.

First, one *land thread* is injected in order to construct the grid. It creates a *node* on each daemon (line 10), establishes a *link* to each of those nodes (line 11), and thereafter, propagates itself along all the *link*s (lines 13-14). The *land thread* on each *node*, thus on each *daemon*, then starts constructing other $N - 1$ *node*s in parallel (lines 15-20), as it hops from one *node* to another in the increasing order of their logical node IDs (line 20).

In each iteration of *node* creation, the *land thread* first generates its eastern neighbor *node* on the local *daemon* and a horizontal *link* to the end (lines 16-17), unless the current *node* is at the east edge of the grid. If it is at the east edge, the *land thread* simply creates a *link* to the very first *node*, i.e., the node on the west edge of the grid.

Similarly, the *land thread* creates the southern neighbor *node* and a vertical *link* to the end (lines 18-19), unless its current *node* is at the south edge of the grid. If it is at the south edge, the *thread* simply creates a *link* to the node at the north edge of the grid on the same vertical axis.

Note that *node*s will be created by the *thread* working on the local *daemon* or the one working on the northern neighbor *daemon*, but never duplicated by both.

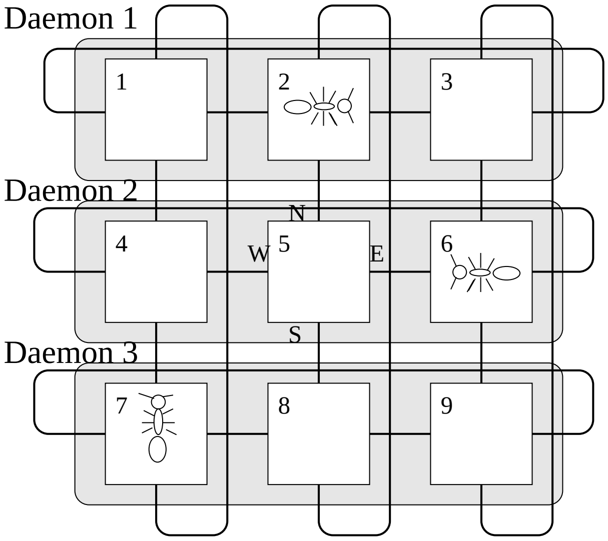
After the grid creation, a user may inject as many *ant thread*s as necessary. Upon injection, the *ant thread* goes into its life cycle (lines 30-33). In each interaction, it selects a *link* as the next moving direction (line 31), hops along it (line 32), and picks up any available food at the destination *node* (line 33).

There are two important advantages to coding multi-agent applications using our self-migrating threads: (1) a synthetic world is constructed in parallel with the propagation of self-migrating threads and flexibly mapped to daemons, and thus to actual workstations; and (2) the behavior of agents, (*ant thread*s in our example), can be programmed intuitively, which can therefore narrow the semantics gap between their abstract behavior and actual implementation.

3 System Implementation

3.1 Main Issues

The proposed execution model involves two functional concerns: (1) state capture and resumption during migration and (2) non-interruptible execution between any two network navigations. We also need to take care of two performance concerns: (1) frequent thread migrations and (2) scalable and fast logical net-




```
(1) enum direction {n, e, s, w};
(2) class Node : BaseNode {public: int food;}
(3) class Link : BaseLink { }
(4) class Land : BaseThread {
(5)  private:
(6)   int i, d;
(7)  public:
(8)   body( ) {
(9)    for (i = 1; i <= N; i++) {
(10)    new Node() :ID(i,(i-1)*N+1);
(11)    new Link() :ID(i,(i-1)*N+1,null,null);
(12)   } // node(1/1), (2/4) and (3/7) created
(13)   node.getLink( );
(14)   hop(node.link);
(15)   for (i= 1, d=daemon.id; i<=N; i++) {
(16)    new Node():ID(d,(d-1)*N+(i+1)%N);
(17)    new Link():ID(d,(d-1)*N+(i+1)%N,e,w);
(18)    new Node():ID((d+1)%N,(i+N)%(N*N));
(19)    new Link():ID((d+1)%N,(i+N)%(N*N),s,n);
(20)    hop(d, (d-1)*N+(i+1)%N);
(21) } }}
(22) class Ant : BaseThread {
(23)  private:
(24)   direction seed; int myFood;
(25)   direction compute ( ) { // random walk
(26)    return (srand(seed)%4+1);
(27)   }
(28)  public:
(29)   body( ) {
(30)    for (;;) {
(31)     node.getLink(compute( ),null,null);
(32)     hop(node.link);
(33)     myFood += node.food; node.food = 0;
(34) } }}
```

Figure 3. A example of multi-agent applications

work construction. The following discusses our solutions to those four concerns.

State Capture/Resumption: The thread state consists of its local variables, the CPU registers, its stack, any dynamically allocated heap memory, and open I/O connections. Among these, only the I/O connections are local to each machine and thus cannot be moved.

Dynamically allocated spaces are hard to carry, since pointers that specify dynamic spaces may be dangling at runtime. This dangling pointer problem has been well investigated in [5], which addressed the capability of carrying intra-thread heaped data. Instead of allowing general memory allocation and pointer use, we have proposed and implemented in MESSENGERS an abstract data type, termed *Dobj*, whose instances are created dynamically, are self-referenced, can be exchanged among mobile agents and their working places, and are carried with agents through library functions [9]. We introduce this *Dobj* abstract data type to our self-migrating threads so that they can construct and carry dynamic data structures with *Dobj* instances over the network.

For stack extraction and restoration, we can focus on the *body()* member function's stack contents. This is because *threads* are allowed to migrate only inside *body()*. The rest of the state to be captured includes the *thread*'s local variables and the CPU registers, which do not cause any problems except I/O-related register contents like interrupt masks.

The scenario of thread migration is as follows: The *thread.hop()* member function first salvages all *Dobj* dynamic objects attached to this *thread* and then calls setjmp() to save the register contents. The *body()*'s stack is saved using the stack base and top pointers, and the *thread* object itself is captured using *"this"* pointer and *sizeof* operator. From the program counter, *hop()* calculates the displacement from the *body()* function's entry point to the current execution address, (which we refer to as "execution displacement".) Thereafter, all captured states are sent to the destination daemon. It receives the object contents directly in a new space allocated for this *thread* object, and invokes the *body()* function which, at the beginning, overwrites its new stack with the one carried, computes the resumption point by adding the "execution displacement"to its entry point, and then calls longjmp() to resume its execution.

Non-interruptible Thread Execution: Non-interruptible execution is guaranteed by implementing the *node* as a monitor. At each *node*, only one of the arriving *threads* is allowed to resume its execution and access the *node*. The *thread.hop()* makes the current *thread* leave the *node* and selects another *thread* to execute. Therefore, there is no concurrency inside a *node*, while *threads* residing at different *nodes* may run concurrently. No deadlock occurs for *node* access, since there are no *threads* residing on two or more *nodes* at a time.

More complicated is the atomic *link* access. Such atomicity is guaranteed only inside the *modify()* that is passed as an argument to *node.getLink()*. The *node.getLink()* not only makes a list of the emanating *links* that satisfy a given condition but also passes each of those *links* to *modify()*. One side of the *link* has a monitor which the *modify()* needs to enter before modifying the *link* contents. Obviously, *modify()* issued at the other end of the *link* must be first dispatched to the monitor before its invocation, which guarantees deadlock-free atomicity.

Support for Frequent Thread Migration: It is anticipated that a large number of small threads (of various sizes) will migrate over the system frequently and irregularly. An appropriate communication library is necessary for the daemon process to handle this situation. Using high-level packages such as PVM and MPI eases our implementation but degrades performance, due to their message handling overhead or inflexible protocol to wrap the low-level communication, as reported in [3]. Thus, we designed our own communication module.

Our module employs polling and non-buffering communication policies, similar to Madeleine over TCP [3]. The difference is that our module embeds two **pthreads**, (which are distinguished from *thread* objects and thus invisible to users). The module establishes two socket connections with each remote daemon, one for read and the other for write. All the connections are asynchronous and kept open in order to avoid I/O blocking and redundant open/close system calls. The two **pthreads** watch ready socket connections with Unix **select** and work on *thread* objects transfers in parallel: one for receiving and the other for sending through the sockets. They write/read each migrating *thread* object with its size information directly from/to the allocated memory space through the socket. Therefore, no extra copying overhead occurs in the user level. They relinquish the CPU by **pthread_yield** whenever they find no available sockets and fail in non-blocking read/write calls. This scheme avoids interrupts to the daemon process, raises the CPU usage, and prevents the saturation/starvation of socket buffers.

Logical Network Management: The proposed execution model constructs the logical network independently from thread migration. The *node* and *link* objects are identified with an integer value rather than a string-oriented name. While they may include strings as their data members, nothing except the ID need be defined inside those objects. This scheme is therefore expected to make the logical network construction faster and more scalable than mobile-agents systems that are capable of generating logical networks or working places.

3.2 Preliminary Performance

Self-migrating threads may improve performance by reducing remote memory accesses. They however need more time to be transferred over the network than simple messages, due to the state capture/resumption and termination/creation of threads required at the source and destination sites. The smaller the threads and the more frequent their migrations, the more software overhead is incurred as compared with physical communication latency.

Therefore, we need to observe the performance of our inter-daemon communication module in terms of the transfer of frequent and small messages. In addition, we should also imitate the typical execution style of multi-agent applications where self-migrating threads, each with a small amount of computation, navigate over network frequently.

For those two performance evaluations, we used a network of four 170 MHz SPARCstations (96MB memory each) connected by a 100Mbps collision-free Hitachi switch. Using **pthread** library provided on Solaris 2.5, we have implemented self-migrating threads, which carry their states as described above.

Inter-daemon Communication Module: To asses the performance of our inter-daemon communication module, we coded the following test program: each of four processes running at a different machine repeatedly exchanges a message with the three others 1000 times. In each iteration, each process first sends all messages and thereafter receives the ones from the others, so that temporary congestion and I/O blocking are likely to occur. The test program itself does not include any flow control or any concurrent portion of code like a communication state machine. We used and compared three different communication packages for this program: our inter-daemon communication module, PVM, and MPI-CH.

Figure 4 shows the performance obtained when increasing the message size. Our inter-daemon communication module shows better performance than PVM and MPI-CH for any message size. Especially, for small messages below 500 bytes, the performance is 1.5 to 2 times better than the others. This is because

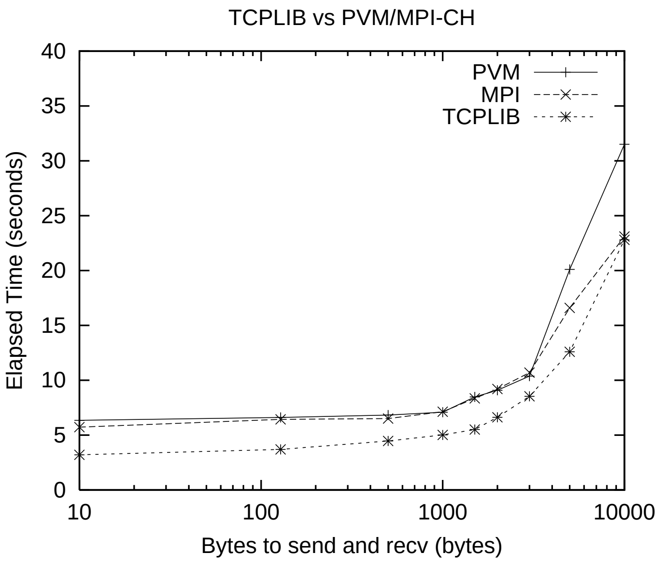


Figure 4. Performance of inter-daemon communication module

our scheme uses two threads working on sends and receives concurrently, while the other packages are single-threaded and thus handle I/O sequentially. Note that MPI's non-blocking communication feature does not maximize its efficiency unless it is combined with some sophisticated concurrent computation.

Threads' Random Walk: We modeled the multi-agent simulation as a simple random walk by self-migrating threads. In this program, 1,200 threads repeat 30 times a random walk among four daemon processes. For every migration, each thread performs a certain amount of dummy floating-point computation. Note that no logical network is involved and thus its management overhead is not considered. Again, to asses performance, we programmed five different versions: our self-migrating threads, PVM, MPI-CH, MESSENGERS and IBM Aglets. For PVM and MPI-CH, each process repeats the same number of messaging and performs the same amount of dummy computation for every message. For MESSENGERS and IBM Aglets, 1,200 mobile agents repeat the same operations.

Figure 5 shows the results. Our self-migrating threads demonstrate a better performance than all the other versions except MPI below 2000 floating-point computations. In particular, it demonstrates a better performance over PVM for any range of computation, due to frequent I/O blocking in PVM.

Self-migrating threads however show 10% to 15% performance slow-down as compared with MPI for any computation range. The main reason is that MPI's non-blocking communication performed effectively by having a user process proceed with the computation regardless of the completion of previous message sends. Unlike MPI, our self-migrating threads incur their creation and termination overhead whenever they navigate over the network.

As compared to MESSENGERS, self-migrating threads are faster 10% to 20%, below 2000 floating-point computation, while slowing down their performance beyond this threshold. This is because MESSENGERS incurs overhead by its frequent flips between interpretive and native mode executions in fine-grained computation but it gains by three implementation techniques in coarse-grained computation: (1) its automatic virtual-time synchronization mechanism, (2) the pool and reuse of Messengers terminated, and (3) the omission of interpretive code in their migration when the code has already been transferred to the same destination earlier.

The IBM-Aglets daemons are unable to exchange more than eight aglets among four workstations, because of the Java socket exception. Even in the case when we injected only two aglets on each workstation, for a total of eight aglets, they need 26.2 seconds for repeating the random walk 30 times. Thus, aglets are out of consideration for fine-grained parallel computing.

For our self-migrating threads, we expect further performance improvements with the same implementation techniques as MESSENGERS: virtual time and reuse of threads, the latter of which results in omitting the code in their transfers. Taking these into consideration, the basic performance we have obtained is convincing to use self-migrating threads for multi-agent applications.

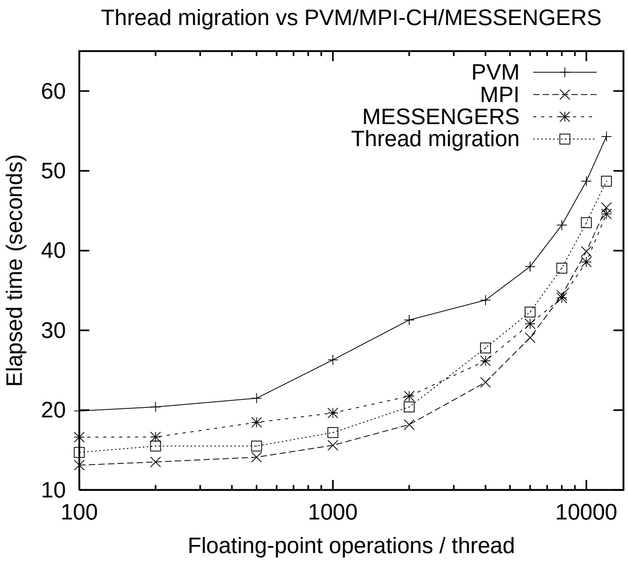


Figure 5. Performance of a random walk by 1200 self-migrating threads

4 Related Work

Thread migration has been realized by several distributed-memory-based systems for purposes of keeping a good load balance, providing efficient RPC or remote execution, and reducing remote memory accesses.

UPVM [4] is a system realizing dynamic load balancing by migrating threads to lightly loaded processors. A global thread scheduler is required for their migration, which in turn means that threads do not have navigational autonomy. Furthermore, UPVM limits the number of threads, due to its maintenance scheme for pointer contents transferred with each thread.

Nexus [7] provides an efficient way of remote execution where a given function is asynchronously executed by a remote thread that may be preallocated, created on demand, or reused from previous execution. Nexus may replace our logical network structure with its computation model of nodes, GPs and RSRs, and may emulate our self-migrating threads by relaying Nexus threads along RSR links. However, they must start from the beginning of their given function when spawned at a remote processor. Our self-migrating threads, on the other hand, resume their ongoing computation even after their migration.

Olden [15] and Nomadic Thread [11] migrate a thread to a remote machine when remote memory accesses have occurred frequently. However, such memory accesses are predicted and migration code is automatically inserted at compile time. In addition, their threads move over a globally fixed address space. Thus, they can hardly emulate autonomous agents, capable of dynamically constructing a synthetic world.

Ariadne [13] allows two processes to exchange a thread with each other through a sequence of library calls: *thread_pack/unpack()* and *send/recv()*. While thread migrations are fully controlled at user level, they are performed by processes but not directly by threads themselves. This implies that, in addition to thread programming, each application user needs to design his/her own daemon process that manages threads by themselves.

PM2 [14] provides threads with library calls to initiate state capture and migration in order to address load balancing, overlapping of communication by computation, and efficient cache utilization in data-parallel applications. However, inter-threads communication in PM2 is based on message passing, because of which system-wide data structures must be maintained at user level by each application.

In summary, it is possible to apply the above systems to multi-agent applications, however, users are forced to make considerable programming effort to resolve the above problems by themselves. On the other hand, our self-migrating threads focus on multi-agent applications from the beginning. They are thus capable of automatic state-capture, autonomous navigation, and dynamic construction of global computation spaces.

5 Conclusions

As a new cluster computing environment for multi-agent applications, we proposed self-migrating threads, which adopt the following three advantages: (1) navigational autonomy from mobile agents, (2) fine computation granularity from threads, and (3) logical network construction capability from MESSENGERS.

The trade-off for better performance is that native thread execution loses certain flexibilities of mobile agents, (e.g. self-modification). However, self-migrating threads support evolution in multi-agent applications by dynamically linking their executable images to the system on demand. In addition, our preliminary performance evaluation demonstrated convincingly the feasibility of migrating large numbers of fine-grained threads. Therefore, we expect that self-migrating threads will be a promising cluster-computing environment for multi-agent applications.

As the next step, we are planning to evaluate the performance of logical network constructions. Thereafter, we will fully implement the system, and develop multi-agent applications such as ant evolution and school of fish. Our final goal is porting the system to a Myrinet PC cluster we recently built, and assessing the appropriate granularity of our self-migrating threads on one of the best cluster computing environments.

References

[1] L. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, Vol.29(No.8):55–61, Aug. 1996.

[2] N. Boden, D. Cohen, R. Federman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. Myrinet: A gigabit per second local area network. *IEEE-Micro*, Vol.15(No.1):29–36, Feb. 1995.

[3] L. Bouge, J. Mehaut, and R. Namyst. MADELEINE: an efficient and portable communication interface for rpc-based multithreaded environments. In *Proc. of the 1998 Conf. on Parallel Architectures and Compilation Techniques, PACT'98*, pages 240–247, Paris, France, 1998.

[4] J. Casas, R. Konuru, S. Otto, R. Prouty, and J. Walpole. Adaptive load migration systems for pvm. In *Proc. of Supoercomputing '94*, pages 390–399, Washington D.C., 1994. IEEE.

[5] D. Cronk, M. Haines, and P. Mehrotra. Thread migration in the presnse of pointers. In *Proc. of the 30th Hawaii Int'l Conf. on Systems Sciences - HICCS'97*, pages 292–298, January 1997.

[6] J. Ferber. *Multi-Agent Systems An Introduction to Distributed Artificial Intelligence.* Addison-Wesley, 1999.

[7] I. Foster, C. Kesselman, and S. Tuecke. The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, Vol.37(No.1):70–82, Aug. 1996.

[8] M. Fukuda, L. F. Bic, and M. B. Dillencourt. Performance of the MESSENGERS autonomous-objects-based system. In *Proc. of the 1st Inte'l Conf. on Worldwide Computing and Its Applications, WWCA'07*, pages 43–57, Tsukuba, Ibaraki, Japan, March 1997. Springer LNCS 1274.

[9] M. Fukuda, N. Suzuki, and L. F. Bic. Introducing dynamic data structure to mobile agents. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA'99*, pages 1854–1860, Las Vegas, NV, 1999.

[10] W. Gropp and E. Lusk. User's guide for MPICH, a portable implementation of MPI. Version 1.1.2, Mathematics and Computer Science Division, Argonne National Laboratory, February 1999.

[11] S. Jenks and J.-L. Gaudiot. Nomadic Threads: A migrating multithreaded approach to remote memory accesses in multiprocessors. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques, PACT'96*, pages 2–11, Boston, MA, 1996.

[12] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison Wesley Longman, Reading, MA, 1998.

[13] E. Mascaarenhas and V. Rego. Ariadne: architecture of a portable threads system supporting thread migration. *Software - Practice and Experience*, Vol.26(No.3):327–356, March 1996.

[14] R. Namyst and J. Mehaut. PM2: Parallel multithreaded machine. a computing environment for distributed architectures. In *Proc. of ParCo'95*, pages 279–285. Elsevier Science Publishers, September 1995.

[15] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM TOPLAS*, Vol.17(No.2):233–263, 1995.