A Design of Self-Migrating Threads in C++

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

Technical Report ISE-TR-99-160

May 7, 1999

Abstract

The navigational autonomy of mobile agents has the potential to form a novel programming paradigm for certain parallel applications such as individual-entity-based simulations and some graph problems. However, for finer granularity and better scalability of computations, mobile agents cannot perform well if they are interpreted. Thread migration is another software technology that moves threads to remote processors in order to reduce their remote memory accesses. But this migration is passive and requires a shared memory infrastructure. We propose self-migrating threads that navigate autonomously over a network and resume their native-mode executions at the destination. By giving them the capability to construct system-wide logical networks, we expect that self-migrating threads can support entity- and graph-based applications at any granularity and scalability. 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.

Contents

1 Introduction

Mobile agents are script entities that autonomously navigate over a network and perform various tasks at each node they visit [2]. Of interest is their inherent parallelism, obtained by the dissemination of the agents through the network. With their navigational autonomy and task-coordinating capability, mobile agents have the potential to form a novel programming paradigm for some types of parallel and distributed computing.

We have developed MESSENGERS, one of a few systems that realized distributed computing through the parallel execution of and interactions among mobile agents roaming over application-specific networks [1]. This computation paradigm is well suited for entity-based simulations and certain graph problems [8, 6]. 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 graph analysis tools, like Petri-nets, may generate very large numbers of network nodes. Currently, MESSENGERS demonstrates its competitive performance only in coarse- to medium-grained distributed computing due to the nature of interpretation [7]. Therefore, it is not feasible to use MESSENGERS directly for massively parallel computations.

Another software technology that migrates computations is based on threads. It migrates threads to remote processors if these contain frequently accessed data and thus reduces remote memory accesses. Although threads are the finest-grained computing entities, they are generally based on shared memory programming, which requires distributed shared memory or similar infrastructures to be implemented over the system. In addition, most thread migrations take a passive form, where threads are moved by their underlying language systems or operating systems.

As a new approach to entity- and graph-based parallel computations, 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 light-weight entities roaming in a dynamic computation space, rather than passive computations running on a shared memory infrastructure.

We have designed the functional specification of self-migrating threads that are coded using C++ and use special navigational constructs. To study the performance feasibility, 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 from two view

points: our implementation techniques and basic performance.

The rest of this paper is organized as follows: Section 2 briefly introduces our programming paradigm using self-migrating threads; Section 3 discusses some implementation techniques and shows preliminary performance; Section 4 differentiates our design from other related works; and Section 5 concludes the discussions.

2 Execution Model

Our design of self-migrating threads is based on the following four principles: (1) preserving MESSENGERS' network architecture, (2) permitting users to describe threads and 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 as in MESSENGERS (see Figure 1.) The lowest level is the *physical network* (a LAN or processor interconnection network), 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 created on top of the daemon network.

A *daemon network* topology is initially defined in a user's configuration file by enumerating all participating physical nodes, (i.e., daemon nodes) and listing the neighbors for each node. The *logical network* is dynamically constructed by self-migrating threads, either independently of or with regard to the underlying *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 members 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 either directly with a system-unique ID from all daemons or indirectly with a relative ID from its neighbor. The system-unique ID takes a positive integer, while the relative

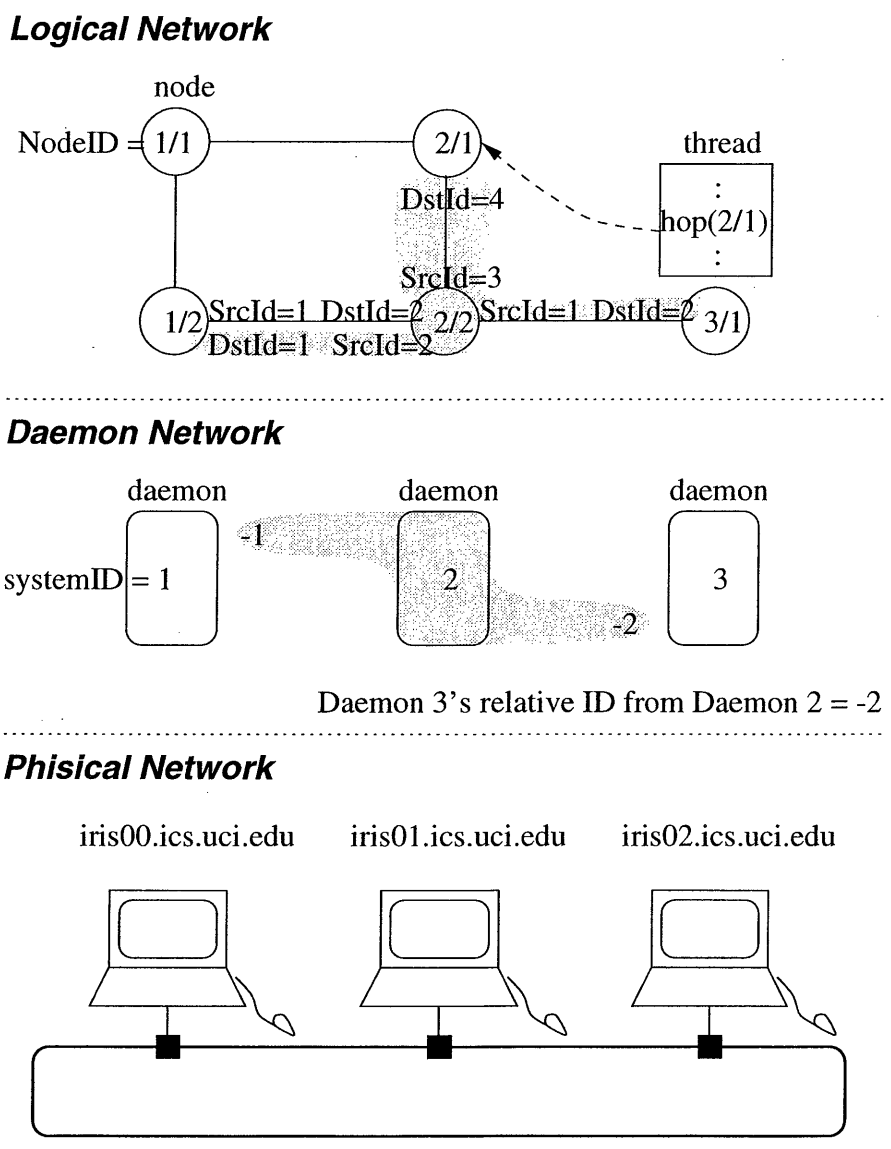

Figure 1: Computational network and objects

one uses a negative integer. In Figure 1 for instance, the *daemon* running on the *iris02* workstation is identified either with the system-unique ID ♯3 from all *daemons* or with the relative ID ♯-2 from the neighbor running on *iris01*. The *node* is identified with the concatenation of its *daemon* ID and daemon-local *node* ID, and thus referred to as *(daemon_ID/node_ID)*. The *link* is recognized from both ends, (i.e., *node*s), 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 *node*s, (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).

It is expected that *thread*s, with the ID-oriented naming, can migrate to network objects faster than most string-oriented mobile agents. However, 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 move itself 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.

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

Figure 2: Network objects defined in C++

When invoked, a daemon process instantiates a *daemon* object indicating its status, as well as two *node*s, called *INIT* and *TRASH* respectively. New *thread*s are injected from a user shell, start their navigation from the *INIT* node and terminate themselves by migrating to *TRASH*. *Thread*s are capable of constructing and destructing other objects with *new* and *delete* operators during their execution. 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 a *node* and invokes its constructor with *arguments* when the daemon specified with *daemonId* does not yet have the *node* with *localId*. Otherwise it fails in the node construction. Null *daemonId* and *localId* mean a *node* construction on the current daemon and a local node ID given by the system respectively. Upon a successful creation, *new* returns the *local ID* of this *node*. Similarly, the *delete* operator succeeds in a *node* deletion when such a *node* exists as a skeleton and returns the deleted *node*'s daemon-local ID.

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

```
DaemonId daemonId;
NodeID localId;
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 then passes *arguments* to its constructor, 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 *link*s, each of which is accessible with an index i and then deleted by the *delete* operator. Both the *new* and *delete* operators return the source ID of the created and deleted *link*s respectively upon a successful completion. 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 *link*s and/or jumps directly to *node*s using its member function *hop()*. Its execution is not interrupted by other *thread*s residing on the same *node* unless it initiates a navigation.

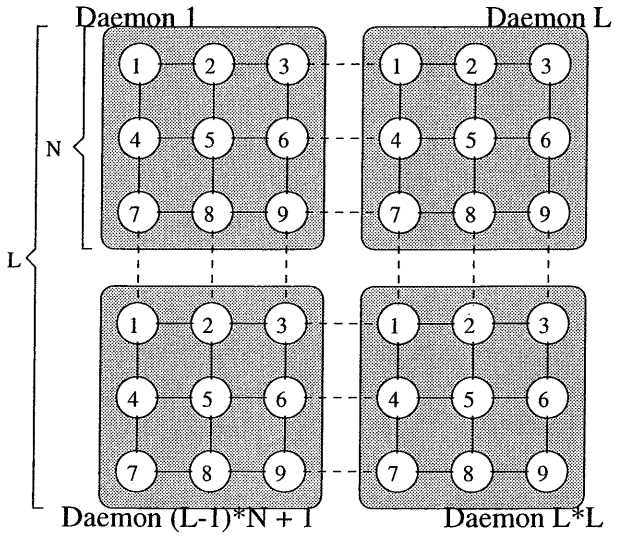

Figure 3: A Mesh Construction

```
node.getLink(searchFunc, modifyFunc);
hop(node.link[i]);
```

The current *node*'s *getLink()* member function, if given with *searchFunc()*, generates a source ID list of the emanating *link*s that satisfy the conditions provided by *searchFunc()*. It also modifies the contents of the selected *link*s, using a given *modifyFunc()*. Null *searchFunc* passed to *getLink()* means selecting all the emanating *link*s. The *thread* can then move itself along each of those *link*s with *hop(node.link[i])*. The *hop()* is capable of multicasting the *thread*, depending on the variation of its arguments:

hop(node.link): propagates the *thread* along all selected *link*s.

hop(daemonID, localID): makes the *thread* jump to the *node* with *daemonID* and *nodeID*.

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

hop(TRASH): makes the *thread* jump to the *TRASH node*, which terminates it.

Figure 3 shows an example of constructing an $NL \times NL$ mesh of *nodes* on an $L \times L$ mesh of *daemon* so that an $N \times N$ sub-mesh of *node*s are mapped to each *daemon*. The *thread* object that performs such a network construction is coded in Figure 4. Upon being injected on the *INIT node* of a *daemon* and initialized by the default constructor, the *thread* object invokes the *body()* member function. It first

creates a *node* with daemon-local ID ♯1 on each daemon (line 12) and establishes a *link* to each of those nodes created (line 13). Thereafter, the *thread* propagates itself along all the *link*s (lines 15-16). It then starts constructing an $N \times N$ mesh of *node*s on each *daemon* in parallel (lines 17-35), as it hops from one *node* to another in the increasing order of their local node IDs (line 34).

In each iteration of *node* creation, the *thread* first generates a horizontal *link* and a new *node* at the end (lines 18-25). Assuming that it is residing on *node*(null/i), where *null* means the current daemon and i satisfies $1 \leq i \leq N \times N$, it creates its eastern neighbor *node*(null/i+1) on the local *daemon* and a horizontal *link* to the end (lines 19-20), unless *node*(null/i) is at the east edge of the sub-mesh (line 18). If it is at the east edge of the sub-mesh but not of the whole mesh (line 21), the *thread* creates *node*(daemon.id+1/i-N+1) on its eastern neighbor, the *daemon* (daemon.id+1), and a horizontal *link* to it across the daemon boundary (lines 22-24).

Similarly, the *thread* creates a vertical *link* and a new *node* at the end (lines 26-33). It creates the southern neighbor *node*(null/i+N) locally and a vertical *link* to the end (lines 27-28), unless *node*(null/i) is at the south edge of the sub-mesh (line 26). If it is at the south edge of sub-mesh but not of the whole mesh (line 29), the *thread* creates *node*(daemon.id+L/i-N(N-1)) on the south neighboring *daemon* and a horizontal *link* to it (lines 30-32).

Note that *node*s at the north and west edge of each sub-mesh will be created by the *thread* working on the local *daemon* or the one working on its neighboring *daemon*, but never duplicated by both.

3 System Implementation

3.1 Main Issues

The proposed execution model involves two functional concerns: (1) state capturing and resumption 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 network construction. The following discusses our solutions to those four concerns.

State Capturing/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.

```
(1) enum direction {north, east, south, west};
(2) class Node : BaseNode { };
(3) class Link : BaseLink { };
(4) class Thread : BaseThread {
(5)   public:
(6)     body( );
(7)   private:
(8)     int i;
(9) }
(10) Thread::body( ) {
(11)   for (i = 1; i <= L * L; i++) {
(12)     new Node() : ID(i, 1);
(13)     new Link() : ID(i, 1, null, null);
(14)   }
(15)   node.getLink( );
(16)   hop(node.link);
(17)   for (i = 1; i < N * N; i++) {
(18)     if (i % N != 0) {  // ! at right vertical
(19)       new Node() : ID(null, i+1);
(20)       new Link() : ID(null, i+1, east, west);
(21)     } else if (daemon.id % L != 0) {
(22)       new Node() : ID(daemon.id+1, i-N+1);
(23)       new Link() : ID(daemon.id+1, i-N+1,
(24)                       east, west);
(25)     }
(26)     if ((i-1)/N != N-1) {//! at lower horizon
(27)       new Node() :ID(null, i+N);
(28)       new Link() :ID(null, i+N, south,north);
(29)     } else if ((daemon.id-1) / L != L-1) {
(30)       new Node() :ID(daemon.id+L, i-N*(N-1));
(31)       new Link() :ID(daemon.id+L, i-N*(N-1)),
(32)                       south, north);
(33)     }
(34)     hop(null, i+1);
(35) } }
```

Figure 4: A code example of thread object

Dynamically allocated spaces are hard to carry, since there is no information in the compiled code to detect pointers that specify those dynamic spaces. 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. If *threads* are allowed to migrate only inside *body()*, it must not be called recursively. 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 copies the object contents to 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 resuming 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 *modifyFunc()* 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 *modifyFunc()*. One side of the *link* has a monitor which the *modifyFunc()* needs to enter before modifying the *link* contents. Obviously, *modifyFunc()* 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 will frequently migrate over the system. To handle this situation, we have developed a TCP/IP-based communication module that the daemon process uses. This 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. Two threads, (which are different from *thread* objects and thus invisible to users), watch ready socket connections and work on *thread* objects transfers in parallel: one for receiving and the other for sending through the connections. They relinquish the CPU whenever they find no available sockets and fail in read/write system calls. This scheme not only raises the CPU usage but also avoids 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 network or working places.

3.2 Preliminary Performance

Self-migrating threads bring the possibility of performance improvement by reducing remote memory accesses. They however need more time to be transfered over the network than simple messages, due to the state capturing/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 performance degradation is incurred.

Therefore, we need to observe the performance of the inter-daemon communication module we developed especially for the transfer of frequent and small messages. In addition, we should also imitate the typical execution style of individual-based simulations in that 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.

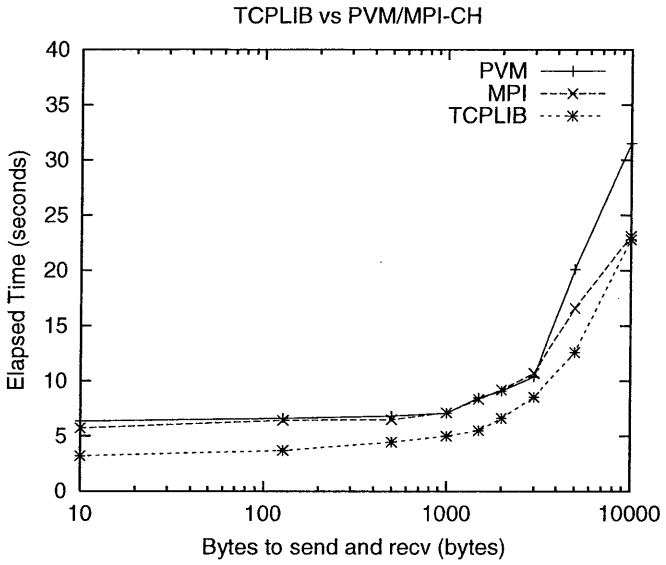

Figure 5: Performance of inter-daemon communication module

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 is likely to occur. We used and compared three different communication packages for this program: our inter-daemon communication module, PVM, and MPI-CH[10].

Figure 5 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 our scheme uses two threads working on send and receive concurrently, while the other packages are single-threaded and thus may be sometimes blocked on I/O.

Threads' Random Walk: We modeled the individual-based 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 6 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. This is because PVM processes make the CPU idle when blocked by I/O, while our self-migrating threads resume their computations as soon as two threads in charge of communication wait for I/O.

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 provides non-blocking communication so that a user process can 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 incur 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, (2) the pool and reuse of Messengers terminated, and (3) the omission of interpretive code in their migration when the code has already been transfered 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. Therefore, we injected only two aglets on each workstation, for a total of eight aglets. Even in such a small scale computation, 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 individual-based simulations.

4 Related Works

Thread migration has been employed in several distributed-memory-based systems: UPVM [4], Ariadne [13], and Emerald [14]. However, this form of migration typically

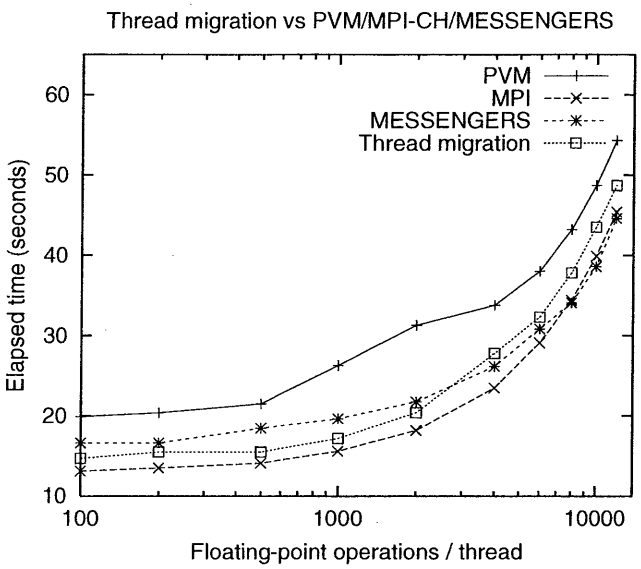

Figure 6: Performance of a random walk by 1200 self-migrating threads

takes on a passive form. In other words, the threads are being moved by some other object or the underlying operating system. On the other hand, the self-migrating threads have navigational autonomy and move as the result of their own actions.

Several other systems have realized self-migrating threads: Obliq [3], Nomadic Threads [11], and Olden [15]. In Obliq, agents run as threads. Each however executes interpretive agent code and thus degrades performance. Nomadic Threads is a system that generates and migrates a thread to a remote machine when remote memory accesses have occurred frequently. Such memory accesses are predicted and migration code is automatically inserted at compile time. Similarly, Olden dispatches a thread to a remote host. However the thread carries only the current activation stack, while leaving behind the previous stacks, into which it passes the return value when terminating the current function. Threads in both Nomadic Threads and Olden use a procedural language and move over a globally fixed address space. In addition, the time and direction of their navigation is determined by their compiler, not by a user, although the threads themselves initiate navigation.

Our self-migrating threads dynamically construct and migrate over a system-wide logical network, use C++, and permit a user to control migration. In addition, they can carry and exchange dynamic data structures among them, while some of the other systems partially addressed the capability of carrying heaped data [5].

5 Conclusions

We have focused on scalability and granularity in entity-based and graph-based applications, for which mobile network objects improve programmability. For this purpose, we have designed and are now implementing self-migrating threads whose behavior is described in C++. This paper discussed the design feasibility from two view points: implementation schemes and basic performance. The experiments in our low-level communication module and thread migration demonstrated convincingly the feasibility of migrating a large number of threads that include small data and small computations.

As the next step, we are planning to evaluate the performance of logical network constructions. Thereafter, we will fully implement our design of self-migrating threads and develop some applications, such as Petri-net reachability graph generation, shortest path/minimum spanning tree search, and ecological/molecular-level entity-based simulations.

References

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

[2] Lubomir F. Bic, Michael B. Dillencourt, and Munehiro Fukuda. Mobile network objects. In *Encyclopedia of Electrical and Electronics Engineering*, page to appear. John Wiley & Sons, Inc., 1998.

[3] L. Cardelli. Obliq: A language with distributed scope. *Computing Systems*, 8(1):27–59, Winter 1995.

[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., November 1994. ACM/IEEE.

[5] David Cronk, Matthew Haines, and Piyush Mehrotra. Thread migration in the presnse of pointers. In H. El-Rewini and Y. N. Patt, editors, *Proc. of the 30th Hawaii Int'l Conf. on Systems Sciences - HICCS'97*, pages 292–298. IEEE Computer Society Press, 7-10 January 1997.

[6] Michael B. Dillencourt, Lubomir F. Bic, and Fehmina Merchant. Load balancing in individual-based spatial applications. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques, PACT'98*, pages 350–357, Paris, France, October 1998.

[7] Munehiro Fukuda, Lubomir F. Bic, and Michael 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.

[8] Munehiro Fukuda, Lubomir F. Bic, and Michael B. Dillencourt. Global virtual time support for individual-based simulations. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications, PDPTA'98*, pages 9–16, Las Vegas, NV, July 1998. CSREA Press.

[9] Munehiro Fukuda, Naoya Suzuki, and Lubomir 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*, page to appear, Las Vegas, NV, June 1999. CSREA Press.

[10] William Gropp and Ewing 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] Stephen Jenks and Jean-Luc 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, Massachusetts, 21-23 Octobr 1996.

[12] Danny Lange and Mitsuru 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.K. Raj, E Tempero, H.M. Levy, A.P. Black, et al. Emerald - a general-purpose programming language. *Software-Practice & Experience*, 21(1), Jan. 1991.

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