Programmability and Performance of M++ Self-Migrating Threads

Munehiro Fukuda [1,2] Naoya Suzuki [2] Luis Miguel Campos [3] Shinya Kobayashi [4]

[1] Computing and Software Systems, University of Washington, Bothell, WA 98011, USA
[2] Information Sciences and Electronics, University of Tsukuba, Tsukuba, Ibaraki 305, Japan
{fukuda, nas}@is.tsukuba.ac.jp
[3] Information and Computer Science, University of California, Irvine, CA 92697, USA
lcampos@ics.uci.edu
[4] Department of Computer Science, Ehime University, Matsuyama, Ehime 790, Japan
kob@cs.ehime-u.ac.jp

Abstract

M++ self-migrating threads are C++ compiled objects, capable of constructing computational networks and autonomously migrating themselves over a cluster system. The main goal of M++ is to provide a cluster computing paradigm for multi-agent simulation that has gained popularity in the areas of artificial life and ecological simulations, and biological modeling. This type of simulation usually requires autonomous objects, each having its own goals and behaviors, to interact with each other in a synthetic world. The M++ self-migrating threads match the characteristics of such objects in multi-agent simulations, and can thus describe them with a narrow semantics gap. Fine granularity and large scalability are two fundamental characteristics of this type of simulations that require fast thread context switch and migration. With the scheduling policy we have designed, M++ manages its threads in pin-downed pages, so that Myrinet's network interface card can directly access and transfer those threads without unnecessary copying operations. In this paper, we present the M++ threads' programmability and study their performance using two multi-agent simulation programs: ant farm simulation and growing neural network.

1 Introduction

The multi-agent approach views applications as the interaction among agents, each having its own goal, behavior, and local information in a synthetic world [7]. This approach has gained popularity in areas such as artificial life, ecological simulation, and biological modeling, in which traditional mathematical techniques cannot always be applied effectively. However, the question of how to provide high performance to such applications is a main concern since they often require to simulate interactions between millions of fine-grained agents.

Parallelization is one of the promising techniques to address such scalability and granularity requirements. Two parallelizing schemes can be considered: cell-based and agent-based schemes [6]. The former divides cellular automata or multi-dimensional array elements, each representing a region of space, and maps them to different processors. In every simulation cycle, a processor scans all the cells allocated to it, computes the behavior of agents residing on each cell, and exchanges agents and cell variables with logically-neighboring processors. The latter, forms multiple groups of agents, and assigns those groups to different processors. Each processor works as a proxy to its own agents, funnels their accesses to remote agents, and exchanges data with remote proxies, (i.e., remote processors.)

While obtaining the desired effect, (i.e., application parallelism), both schemes places the burden of parallel programming, such as processor allocation, message passing, and inter-agents synchronization onto the model designer. From the model designer's view point, for instance, the cell-based scheme requires each processor to exchange the information of cells and agents that exist on the processor's boundary. The agent-based scheme must implement a so-called "interest manager" that keeps track of which agents are interested in communicating with each other. Without an intelligent interest manager [15], inter-agent messages would be broadcast over the system, leading to a very inefficient use of the network and consequently to a degradation in overall performance. In either scheme, an agent functionality is divided among several methods that are called not only for behavioral simulation but also to deal with the intricacies of parallel programming. As a result, the techniques used to parallelize multi-agent applications make it difficult for model designers to concentrate on agent programming.

We propose an alternative approach to multi-agent applications using *M++* self-migrating threads as a new cluster computing paradigm for multi-agent applications. M++ self-migrating threads are C++ compiled objects, capable of constructing logical computational networks and autonomously migrate themselves over a cluster system. They represent a natural mechanism for implementing multi-agent applications: each individual agent can be represented as a distinct member of a collection of self-migrating threads, moving through a logical network representing a synthetic world and interacting with other agents. Of utmost importance is their competitive performance which requires fast thread context switch and migration. Using the scheduling policy we have designed, M++ manages its threads in pin-downed pages, so that Myrinet's network interface card can directly access and transfer those threads without unnecessary copying operations.

To demonstrate the superiority of M++ self-migrating threads' programmability and performance, we have programmed two multi-agent simulations in both M++ and MPI: ant farm simulation [4] and growing neural network [17]. In the former, hundreds of ant agents search for food in a farm space by tracing other ants' pheromone, and thus focus on the interaction among M++ threads. The latter, dynamically creates a neural network and shapes its connectivity, therefore illustrating network construction and signal diffusion with M++ threads. By comparing the M++ and MPI implementations, we have confirmed that M++ allows for elegant solutions to problems that would be difficult to program using MPI, while showing only a slightly slower performance than MPI.

The rest of the paper is organized as follows: Section 2 gives an overview of the M++ environment. Section 3 illustrates the M++ and MPI programs for both ant farm and growing neural network and compares them from a programmability view point. Section 4 compares M++ and MPI's performance. Section 5 contrasts M++ with other approaches such as mobile agents, thread migration and simulation systems, and shows why it is a superior approach. Finally, Section 6 concludes the paper and mentions possible future work.

2 Execution Model

2.1 Principle of Operations

M++ is a cluster-computing environment that supports the development and use of multi-agent applications structured as collections of self-migrating threads, simply called M++ threads. The system can be seen as being composed by three network layers as shown in Figure 1. The lowest layer is the *physical network* (a Myrinet cluster of eight PCs in our implementation), which constitutes the underlying computational nodes. Superimposed on the physical layer is the *daemon network*, where each daemon is a Unix process executing and exchanging M++ threads with others. It's processor mapping and system unique ID, termed *daemon ID*, are predefined on a user basis via a profile. The *logical network* is an application-specific computation network dynamically constructed by M++ threads on top of the daemon network. Each logical node has a *node ID* local to the corresponding daemon, while a logical link maintains a set of *source* and *destination ID*s, each local to the current and destination logical nodes respectively.

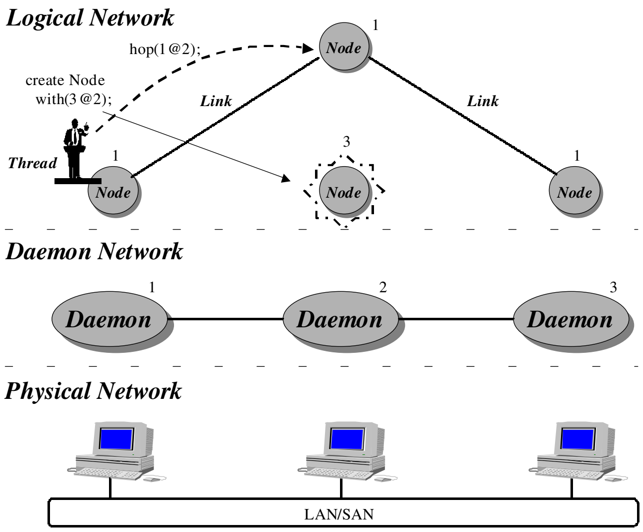

Figure 1. M++ network architecture

M++ threads are programmed in the M++ language, a superset of C++, preprocessed into C++ code, and are fully compiled into executable code. Figure 2 shows the framework of M++ code. Their definition is distinguished from ordinary C++ classes by replacing the *class* keyword with *thread*. The M++ threads not only include public, protected, and private members, as in C++, but also the definition of their autonomous behavior in the *main* method which resumes execution from the previous execution statement when initiating a thread migration to another node (whether local or physical.) Except for this *main* method, C++ inheritance and polymorphism are applicable to M++ threads. Upon migration, M++ threads carry their execution and data status, while their code is dynamically loaded at their destination with NFS support.

M++ threads distinguish three classes of network objects: *daemon*, *node*, and *link*. The *daemon* object contains daemon node information as well as user-defined data, shared by all M++ threads that are running on the same daemon process. The *node* object represents a logical net-

```
class Node{
  public:
};
class Link{
  public:
};
thread Thread {
  public:
    Thread(): daemonId(0){ }
    main(){
      create node<Node> with( 1@daemonId );
      hop( 1@daemonId );
    }
  private:
    int daemonId;
};
```

Figure 2. M++ frame work

methods	return values
int daemon.id( )	the current daemon ID
string daemon.name( )	the current host name
int daemon.total( )	the total number of daemons
int node.id( )	the current node ID
string node.name( )	the current node class name
int node.thr_num( )	the number of threads on this node
int link.count( )	the number of links incident to this node
int link.max( )	the maximal link ID
bool link.exists(int linkId)	the existence check of link with *linkId*

Table 1. Daemon, node, and link's methods

work node, whose method and data members are accessed by M++ threads residing on this node. The *link* corresponds to a logical network link and makes its methods and data members visible to threads residing on the both ends, (i.e., nodes) of the link. These *daemon*, *node*, and *link* objects provide methods to inform M++ threads of their network status as summarized in Table 1. The sharing mechanisms provided by these network objects can be considered as a form of inter-thread communication, however this does not mean that M++ threads are provided with a complete view of a distributed shared memory since prior to accessing a given object, they must migrate themselves to it. Similarly, M++ thread synchronization is permitted on the same daemon or logical node via the predefined statements: *sleep*, *wakeup*, and *wakeupall*, each causing respectively, an M++ thread to sleep, wake up one and wake up all the others at the current *daemon* or *node*.

At system startup, a single logical node, named *INIT*, is created on every daemon node. Any M++ thread may be injected (from the shell or by another M++ thread) into any of the INIT nodes. It may then start creating new logical nodes and links on the current or any other daemon, and may thereafter jump directly or along the links to one of the destination nodes. These navigational operations are performed using the following four statements: [e]create, [e]destroy, [e]hop, and [e]fork. (Note that the statements starting with an 'e' return an error code, otherwise they return nothing for performance reasons.)

The *create* statement: This statement permits an M++ thread to create a new logical node or a new logical link. It is also used to launch a new thread.

- *[e]create node<NodeClassName>(args...) with (NodeId[@DaemonID]);*
  instantiate a new logical node from the *NodeClassName* class, assign *NodeId* to it, and map it to the daemon given by *DaemonId*. *Arguments* are given to this node constructor. Omitting *DaemonId* creates the node on the current daemon. (This syntax is similarly applied to the statements below.)
- *[e]create link<LinkClassName>(args...) with(SourceId) to(NodeId[@DaemonId]) with(DestinationId);*
  instantiate a new logical link from the *LinkClassName* class, assign *SourceId* and *DestinationId* to it, and connect it from the current node to the one with *NodeId* maintained by the daemon with *DaemonId*.
- *[e]create thread<ThreadName>(args...) [with(NodeId@DaemonId)];*
  launch a new M++ thread from the *ThreadName* description on the node with *NodeId* maintained by the daemon with *DaemonId*. Unlike node and link IDs, thread IDs are determined by the system.

The *destroy* statement: This statement permits an M++ thread to destroy a new logical node or a new logical link. The current implementation enables an M++ thread to be destroyed only at the termination of its own *main* method not by other threads.

- *[e]destroy node(NodeId[@DaemonId]);*
  remove the logical node with *NodeId* managed by the daemon *DaemonId*
- *[e]destroy link(linkId);*
  remove the logical link accessible with *linkId* from the current logical node.

The *hop* statement: An M++ thread can navigate itself over the logical network by invoking this statement. Execution resumes from the statement following the *hop* statement.

- *[e]hop([NodeId[@DaemonId]]);*
  allow an M++ thread to jump directly to the node with *NodeId* managed by the daemon with *DaemonId*.

- *[e]hopalong(LinkId);*
  navigate an M++ thread along the link visible with the source *LinkId* from the current node.

The ***fork*** **statement:** It allows an M++ thread to spawn its copy on a specified destination and have it start from the statement following the *fork* statement. The original thread continues its execution without waiting for the duplicated thread's termination.

- *[e]fork([NodeId[@DaemonId]]);*
  fork a child thread on the node with *NodeId* managed by the daemon with *DaemonId*.

- *[e]forkalong(LinkId);*
  fork a child thread on the node that can be traced to along the link visible with the source *LinkId* from the current node.

These navigational operations are based on so-called *strong migration* [1], in which each M++ thread resumes its execution exactly from where it has invoked a migration statement. With this migration scheme, the thread behavior can be programmed exactly according to its scenario and executed sequentially, which allows for model designers to describe their agent's behavior from each agent's view point and not from the simulator system's perspective. However, in return for this elegance, strong migration requires each thread to carry all its state, which thus incurs expensive overheads. In addition, migration always entails thread generation, destruction, and context switch. In the following, we describe our implementation techniques to mitigate these overheads.

2.2 System Implementation

From a performance view point, we focused on the following five implementation techniques in M++: (1) thread migration, (2) underlying thread library, (3) inter-daemon communication, (4) logical network construction, and (5) non-interruptible execution.

Thread Migration

Strong migration is the highest degree of mobility and thus requires the underlying system to transfer a migrating thread's code with all its state: its local variables, the CPU registers, its stack, dynamically allocated data, and open I/O connections. However, our focus is on the reactive school of agents [7] that does not claim for their overall intelligent behavior but instead their simple mechanisms to react to other agents and the local environment. Therefore, our migration strategy focuses on how to slim down the state each thread must carry with it for better performance, while maintaining its scenario-oriented and reactive behavior. Based on the following observations, M++ threads carry only their local variables, the CPU registers, and its main method's stack.

First, we assume NFS to be installed in most clusters where a migrating thread code can be found at the destination and thus need not be transferred. Second, as far as all thread methods are in-lined in the *main( )* method, a thread can still migrate at any point in any thread methods by maintaining only its *main( )*'s stack. The major drawback is recursive calls. Third, we provide M++ threads with a system-predefined class that can be dynamically instantiated, as achieved in the MESSENGERS mobile-agent system [10]. This relieves the daemon's devastating pointer maintenance and thus mitigates performance deterioration. And finally, forwarding I/O connections can be substituted with having threads return where they have established I/O.

The thread migration in M++ takes a straightforward implementation that suspends a thread using setjmp( ), transfers in a packet its state including the execution displacement from its *main( )*'s entry to the suspended point, and resumes the execution at the destination using longjmp( ). A careful attention must be paid to maintain the M++ thread's polymorphism, for which the daemon first creates a new thread instance and then reinitializes it with the incoming packet, using the copy constructor that however does not corrupt its new virtual function table with the packet contents.

Underlying Thread Library

The first implementation of the M++ daemon used the *Pthread* library to instantiate its threads. This exempted the daemon from having to deal with low-level thread management, thus having enabled our quick prototyping [19]. However, *Pthreads* did not cope with a large problem size. The main reasons are their page-based stack allocation and user-to-kernel thread mapping scheme, which unnecessarily allocates large memory space, runs user threads that are even relinquishing CPU, and increases kernel interactions. To address these problems, we have designed our original thread library, termed *Sthread*. It allocates adjustable thread stacks in user heap regions, maps all user threads to only one kernel thread, and employs a non-preemptive scheduling policy that allows a yielding thread to directly handle the ready thread queue and to resume the next ready thread.

With the *Sthread* library, the M++ daemon successively runs all threads in its ready queue without having them interact with it. Even in the case the daemon has a few threads to execute, it waits until they repeat a given number of thread context switches. This policy fits well the typical simulation where thousands of fine-grained agents repeat simple reactions to one another.

Inter-Daemon Communication

M++ implements its inter-daemon communication using both TCP and Myrinet's GM library. The TCP implementation employs polling and non-buffering communication policies, which remove expensive interrupt handling and unnecessary copying operations. The GM implementation pursues further performance improvement by migrating threads with zero-copy communication. Similar communication policies and improvement strategies have been realized by Madeleine [2] over TCP and BIP [18]. The difference is that our communication module gathers in one packet multiple threads that migrate to the same destination, and scatters them from the packet onto the user space at their destination. This *gather and scatter* implementation drastically reduces the number of read and write system calls or GM library calls. While we inevitably allow threads to be copied even using the GM library, mitigating Myrinet's messages can sufficiently compensate such memory-coping overheads. This is because thread migration in M++ is characterized as simultaneous transfers of small threads. For better performance, we are currently modifying our communication module, so that the daemon will immediately resume a group of threads received in a pin-downed memory region rather than scatter them onto the user space.

Logical Network

Upon encountering a *create* statement, an M++ thread internally sends a request message to the daemon where a new node or link should be created, (whether local or remote.) From our observations, such requests are issued much less frequently to a remote daemon than to a local one, in which case spooling request messages does not contribute to the system performance. To make matters worse, since the logical network is generally constructed by extending branches from its leaf nodes, message spooling may result in delaying further network construction. For these reasons, M++ threads sends each single request independently as soon as it needs to create a network component.

M++ prevents threads from duplicating a logical node with the same node ID on the same daemon, which relieves each thread from negotiating with other threads for exclusive node/link creation. In particular, for a link access, the thread internally sends a lock message to one end point of the link rather than migrating to both ends to acquire its lock at user level. This reduces not only the number of thread migrations but also the amount of data to be transferred.

Non-Interruptible Execution

Exclusive 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 hop and sleep statements make the current thread relinquish CPU and select another thread to execute. Hence, while there is no concurrency inside a node, threads residing at different nodes may run concurrently.

3 Programmability

This section illustrates both M++ and MPI programs for ant farm and growing neural network, and compares their programmability.

3.1 Ant Farm

Ant farm was developed by R.J. Collins and D.R. Jefferson [4] to find the gene of the most efficient ant forager. Although the original algorithm uniformly distributed food over a 16×16-meshed farm and simulated the behavior of only 128 ants, we modified it to place food close to each of four mesh corners and permit any number of ants to follow other ants' pheromone. This modification will be useful to observe M++'s performance under more sever conditions where hundreds of M++ threads migrate and communicate with each other very frequently. In this modification, our ant behavior is characterized by repeating the following actions at each simulation cycle: (1) pick up food if it finds any, (2) drop off food in the nest or on route to the nest accidentally, (3) follow the strongest pheromone, (4) decide the next direction if it feels no pheromone, (5) secrete some pheromone and walk to the next cell. To give more reality to each action, the gene of each ant characterizes its average number of food particles to pick up and drop off, average pheromone amount to emit, pheromone sensitivity, and probability to walk straight.

Figure 3 shows an MPI implementation of ant farm. Each processor repeats a cycle of simulation where it scans an array of local cells (line 2), simulates the behavior of ants for each cell (line 4), moves the ants and the *cell[i].phero* amount of cell pheromone to their next destination processor queues denoted by *p[processor_id].aList* and *p[processor_id].pList* respectively (lines 5-7), sends those ants and pheromone values to the neighboring processors (lines 8-12), and finally delivers to appropriate cells the new ants and pheromone it has received (lines 13-21).

The corresponding M++ version of this application is shown in Figure 4. Each *ant* thread repeats a cycle of simulation (lines 6-10) where it decides the next behavior (line 7), launches a *gvt* (global virtual time) thread if it has a specific id (lines 8-9), and goes to sleep until it is awaken by the *gvt* thread (line 10). The *gvt* thread repeats visiting each daemon node (line 20) and summing up the number of sleeping *ant*s (line 21) until this number becomes equal to the total number of *ant*s over the system (line 22). It then increments the simulation time by hopping to each daemon (line 25) and wakes up all the sleeping *ant*s (line 26).

From a glance, the M++ template code is longer than the corresponding MPI code. However, it is much closer to a final implementation than the MPI code is. In fact, all the M++ code needs, before compilation is possible, is

```
for (t = 0; t < timeMax; t++) {
 for (i = 0; i < cellMax; i++) {
  while((ant = cell[i].aList.deq()) != NULL){
   ant->behavior();
   p[ant->toCPU].aList.enq();}
  if (cell[i].isOnBorder)
   p[cell[i].toCPU].pList.enq(cell[i].phero);}
 for (i = 0; i < procMax; i++)
  if (p[i].isNeighbor) {
   packet = pack(p[i].aList, p[i].pList);
   MPI_Send(packet.size, 1, ...);
   MPI_Send(packet, packet.size, ...); }
 for (i = 0; i < procMax; i++) {
  if (p[i].isNeighbor) {
   MPI_Recv(packet.size, 1, ...);
   MPI_Recv(packet, packet.size, ...);
   unpack(packet, p[i].aList, p[i].pList);
   while((phero = p[i].pList.deq()) != NULL)
    cell[phero.toCell].phero=phero.value;}
  while((ant = p[i].aList.deq()) != NULL)
   cell[ant->toNode].aList.enq(ant);}
}
```

Figure 3. Ant farm using MPI

simply substantiating the *behavior( )* method. On the other hand, the MPI code still needs a complete implementation of queuing, packing and unpacking messages as well as computing their destinations. Such M++ superiority obviously arises from its system-level implementation of automatic data capturing and transfer. However, this is not necessarily its major advantage. Additionally M++ offers the following two programming merits:

- *Relative Node Addressing:* In M++, once logical nodes are created and connected with links, each node is relatively reachable from threads that hop along a link from one node to another. On the other hand, each cell in the MPI code is accessible only by its absolute address, (i.e., a pair of processor ID and cell index), which requires agents to grasp how their synthetic space has been mapped over a cluster. This means that space re-mapping in MPI results in having to modify the agent's code.

- *Encapsulation of Agent Behavior and Migration:* M++ encapsulates the behavior and status of each agent in a thread. The agent can migrate to another cell space while executing its *behavior()* method. A different type of ant can be programmed in an independent thread file and injected to the system at any point during execution time, thus facilitating open-ended simulations. On the other hand, the MPI code does not permit agent migration to be embedded in its behavior. The reason is that MPI assumes two-side communication where a pair of sender and receiver must agree with the number of agents and the size of each agent to be sent prior to the actual transfers. Therefore, the MPI code must separate the behavioral simulation from the agent transfer.

```
thread ant {
private:
 void behavior();
public:
 void main() {
  while (daemon.time < timeMax ) {
   behavior();
   if (id() == daemon.specificId) {
    create thread<gvt>() with( INIT @ 0 );
    daemonvar.sleepyAnts++; sleep daemon;
} } } };
thread gvt {
private:
 int sleepyAnts, i;
public:
 void main() {
  do {
   sleepyAnts = 0;
   for (i = 0; i < daemon.total(); i++) {
    hop( INIT @ i );
    sleepyAnts += daemonvar.sleepyAnts; }
  } while(daemonvar.totalAnts > sleepyAnts);
  for (i = 0; i < daemon.total(); i++) {
   hop( INIT @ i );
   daemonvar.time++; daemonvar.sleepyAnts=0;
   wakeupall daemon;
} } };
```

Figure 4. Ant farm using M++

As a result, M++ makes it easier than MPI for model designers to code their agents from each agent's view point. However, its disadvantage is explicit inter-agents synchronization at user level, which must be achieved by the *gvt* thread in Figure 4. In MPI, on the other hand, the transfer of agents and pheromones implies inter-processor synchronization at each simulation cycle.

3.2 Growing Neural Network

The Codi-1bit model was developed by the Advanced Telecommunication Research Institute in Japan in order to build, grow and evolve cellular-automata-based neural network for solving various problems [17]. One of the problems tackled was to obtain a neural network to emit its signal close enough to the $sin\theta$ curve $(0 \leq \theta \leq 2\pi)$. The model prepares 30 different pieces of genetic code, for each of which it builds a $24 \times 24 \times 18$ array of cells, initializes each cell, simulates the growth of neural network for the first 100 cycles, and evolves the network for the following 330 cycles. Thereafter, the model computes the next generation of 30 genes through the processes of selection, crossover, and mutation. The original program was coded in C to find the best genetic code through the repetition of 10 generations.

Figure 5 shows an MPI implementation to simulate the growth of the neural network for 100 cycles denoted as *GrowthSteps*. We have divided the cell array crosswise and mapped each partition to a different processor. Each processor updates the state of its own cells every simulation cycle (lines 1-5). All the cells are initially blank. As the neural network grows, each cell becomes one of three neural components: (1) a neuron, (2) an axon, or (3) a dendrite upon detecting the respective conditions being satisfied: (1) if the cell includes a neuron seed in its *Chromo* variable (lines 6-8), (2) if it receives some level of signal from a given number of neighboring axon cells (lines 9-10), or (3) if it has been signaled from a certain number of dendrite neighbors (lines 11-12). Thereafter, the cell emits an axon or a dendrite signal to its array of six *sig* elements, each corresponding to a different neighboring cell. Those signals, emitted from processor-boundary cells, must be exchanged among processors. The *WrappedKicking* function takes charge of this signal exchange (line 16), whose MPI implementation resembles that of the ant farm previously described.

```
 (1) for (s = 0; s < GrowthSteps; s++)
 (2)  for (z = 0; z < SpaceSizeZ; z++)
 (3)   for (y = 0; y < SpaceSizeY; y++)
 (4)    for (x = 0; x < SpaceSizeX; x++)
 (5)     if (inRange(CA = &CASpace[x][y][z])) {
 (6)      if (CA->type == BLANK) {
 (7)       if (CA->Chromo == NEURONSEED )
 (8)        CA->type = NEURON;
 (9)       else if(inComingSig(CA->sig)==AXON)
(10)        CA->type = AXON;
(11)       else if(inComingSig(CA->sig)==DEND)
(12)        CA->type = DEND; }
(13)      for (i = 0; i < 6; i++)
(14)       CA->sig[i] = outGoingSig(CA, i);
(15)     }
(16) WrappedKicking(CASpace);
```

Figure 5. Codi-1bit using MPI

Figure 6 shows an M++ implementation for the growth phase. The network growth starts by injecting the same number of *neuron* threads as neuron seeds. Each thread creates a different neuron cell, (i.e., node) at cycle 0 (lines 10-12) and thereafter reiterates extending axon and dendrite branches from the neuron node for the following 99 cycles. To be more specific, for each cycle the *neuron* thread examines each of six directions outgoing from the node (line 13), creates an axon or a dendrite node according to the direction state (lines 14-19), extends a new link to and propagates itself to the new node (lines 20-21). The original thread terminates itself (lines 23-24), while all the other new copies synchronize at the next simulation cycle and repeat the growth phase (line 25).

```
 (1) thread neuron {
 (2) private:
 (3)  int state, t, did, nid, x, y, i;
 (4)  neuron( int argc, const char** argv):
 (5)   x(atoi(argv[4])), y(atoi(argv[5]))
 (6)   nid(atoi(argv[6])), did(atoi(argv[7])){ }
 (7) public:
 (8)  void main() {
 (9)   for (t = 0; t < GrowthSteps; t++) {
(10)    if (t == 0) {
(11)     create node<Neuron>(x, y) with(nid@did);
(12)     continue; }
(13)    for (i = 0; i < 6; i++) {
(14)     if (node<Base>.state(i) == AXON)
(15)      create node<Axon>(x, y) with(nid@did);
(16)     else if (node<Base>.state(i) == DEND)
(17)      create node<Dend>(x, y) with(nid@did);
(18)     else
(19)      continue;
(20)     create link<Link>() with(i) to(nid@did);
(21)     if((state=forkalong(i))==Errno::forkChild)
(22)      break; }
(23)    if ( state != Errno::forkChild ) //parent
(24)     break;
(25)   synchronize();
(26) } } };
```

Figure 6. Codi-1bit using M++

The Codi-1bit neural network can be categorized as graph-based computation where a node extends new branches or relays a signal to its neighbors. From such a graph-based computing view point, M++ has the following two superiorities over MPI:

- *Flexible Network Construction:* Although a $24 \times 24 \times 18$ cube is still a maintainable array size in the MPI code, the use of a fixed array wastes memory when the MPI code must assume a large cube to construct a neural network of unknown size. Using a dynamic array is one possible solution to solve this problem. This solution however places the burden of cell-to-node remapping at run time on the programmer. On the other hand, M++ creates only as many nodes and links as an application requires dynamically, making it therefore scalable in terms of memory space.

- *Narrower Semantics Gap:* In the MPI code, each cell must repeatedly propagate a neural signal to its six neighboring cells, whether or not some neighbors have already changed into being a part of neural network, (i.e., a neuron, an axon or a dendrite cell.) The original code could be modified to stop transmitting a signal to those cells, which would however bring complication to the code especially when neighboring cells are on a remote processor. In any case, MPI incurs the wide semantics gap between the original algorithm and the actual implementation. Contrary to MPI, the corresponding M++ code devotes neuron threads to network construction at each leaf node, which thus narrows the semantics gap.

While simulating both graph growth and signal propagation in a more natural fashion than MPI does, M++ has to deal with a more complicated problem of synchronization. This is because the number of neuron threads varies from time to time. Some are forked from their parents, others simply terminate. Therefore, every spawned or terminating neuron thread is supposed to increment or decrement the number of neuron threads respectively. The rest of the work is similar to ant farm, (i.e., circulating a *gvt* thread to check if those live neuron threads have completed their cyclic simulation.) However, such distributed termination detection is quite irrelevant to the original application, and thus should be implemented at system level. For this reason, we are planning to implement in M++ a system-level logical clock maintenance mechanism as developed in MESSENGERS [9].

Features	Descriptions
Network	Myrinet(1.27Gbps)
# Computing Elements	8
Computing Element	Intel Pentium II 450MHz, 120MB memory
OS	Solaris-7

Table 2. Cluster specification for experiments

Test#	Descriptions	Elapsed Time (seconds)	Speed Up (magnitude)
Test 1	*Thread creation/termination*		
	(a) 1024 Pthreads	303.043	1.00
	(b) 1024 Sthreads	6.365	47.61
Test 2	*Random context switch*		
	(a) 1024 Pthreads	5.402	1.00
	(b) 1024 Sthreads	0.989	5.46
Test 3	*2000 threads' random migration*		
	(a) TCP	6.492	1.00
	(b) Zero-copy communication	4.864	1.33

Table 3. Basic Performance

4 Performance

In general, thread migration is much more expensive than simple message passing, due to the transfer of both data and execution state. Thread migration also incurs thread creation, termination, and context switch overheads. However, M++ was designed to mitigate such overheads with the *Sthread* library and a zero-copy migration mechanism. Table 3 shows the comparison between Sthreads and Pthreads in terms of creation, termination, and context switch, and between the zero-copy mechanism and TCP for M++ thread migration over a Myrinet cluster as specified in Table 2. The results show that the use of Sthreads and the zero-copy communication mechanism contributed to an improvement in M++ performance as expected.

The question we attempt to answer in this section is whether these software techniques make M++ competitive with MPI. For this evaluation, we have compared M++ with MPI using the two applications discussed in Section 3. In the following, we will show the comparison results. (Note that we have patched both MPI and Myrinet's GM library, so that we have evaluated the MPI performance under zero-copy communication.)

4.1 Ant Farm

In the original algorithm, 128 ants simultaneously start from the central cell of a square farm where food is uniformly distributed. In order to balance each processor load and observe how M++ scales with the number of agents (i.e. ants), we distributed food close to the four corners of a 200×200 farm and repeated the simulation run by increasing the number of ants from 75 to 300.

Figure 7 shows the M++ and MPI performance for the ant farm simulations. For a run with 75 ants, M++ performs 1.1 to 4.5 times faster than MPI, although it does not demonstrate the effect of parallelism. Since each ant thread includes only a few floating-point operations, the run with 75 ants is still not large enough to compute in parallel and to overshadow migration overheads.

For a run with 300 ants, the M++ performance is comparable to MPI except for a run with 8 processors. Since MPI scans all the cells and for each cell simulates ants and computes the pheromone amount, its performance strongly depends on the scalability in space rather than the number of agents. Obviously, MPI would perform better than M++ once the number of agents exceeded 300. However, it also implies that M++ demonstrates its competitive performance even in such pessimistic situations where each agent needs only a few floating-point operations.

4.2 Growing Neural Network

Figure 8 compares the performance of M++ versus MPI implementations when computing the first four generations of the Codi-1bit neural network. The MPI implementation achieves a 3.95 speed-up factor with eight processors. Although MPI must always exchange neural signals emitted from every single cell on processor boundary, each signal itself is represented in a simple character data. Consequently,

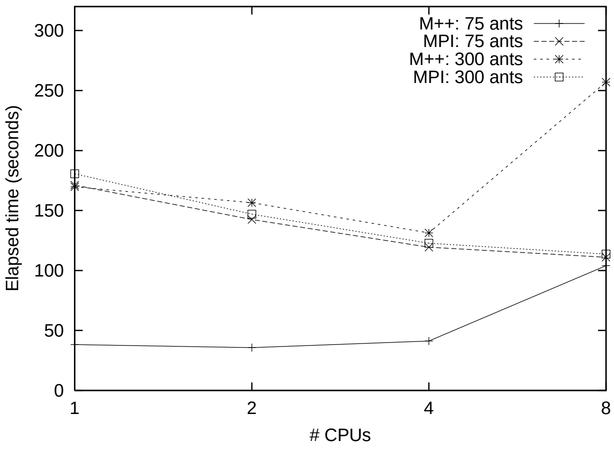

Figure 7. Ant farm performance

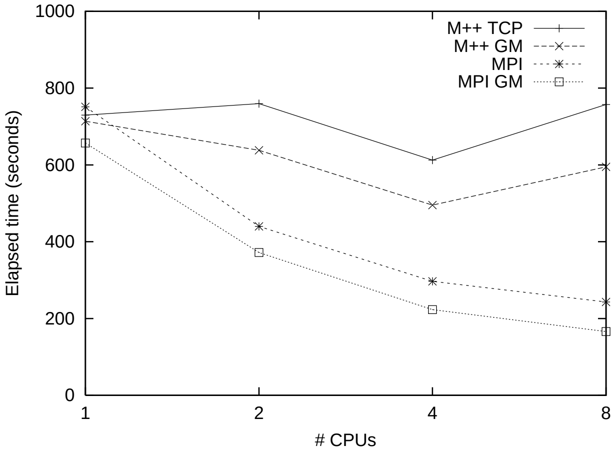

Figure 8. Codi-1bit performance

the total amount of exchanged data still remains of negligible size for cluster computing.

On the other hand, using the zero-copy migration library M++ showed only a 1.4 speed-up factor with four processors and even slowed down with eight processors. Contrary to MPI, M++ computes the diffusion of only meaningful neural signals, however the simulation space is not large enough to be divided into multiple sub-spaces. This causes most signal threads to repeatedly migrate to remote nodes rather than stay longer at the same local sub-space, and thus incurs migration overheads.

In summary, for ant farm and growing neural network simulations, both demanding fine-grained computation, M++ showed the effect of parallel execution using its zero-copy migration and Sthread. However, the overhead of thread migration still affects the M++ performance. In particular, under conditions such that a large number of agents are packed in a small space, M++ cannot catch up to the MPI performance. Our next task is to identify better and larger application examples as well as to further improve our zero-copy migration and Sthread libraries.

5 Related Work

M++ is obviously not the only option available to researchers interested in multi-agent simulations. In here we describe why M++ is a superior environment when compared to mobile agents, thread migration systems, and agent-based simulators.

- *Mobile Agents:* They are classified as cognitive agents [7] that self-contain all knowledge, intelligence, and behavior necessary to independently achieve a sequence of network tasks such as electronic commerce and information retrieval. They are interpreted, coarse-grained, and interacting with the user via a window system. Most systems are thus not expected to inject a large number of mobile agents interacting among one another simultaneously. From our experiment in [19], IBM Aglets [13] for instance falls into a congested situation with only eight aglets distributed among four workstations. Not suitable for simulations requiring a large number of agents that must interact with each other frequently.

- *Thread Migration:* Several distributed-memory-based systems have implemented thread migration for purposes of balancing processor loads, providing efficient RPC and reducing remote memory accesses. It is possible to apply the traditional thread migration systems to multi-agent applications, however doing so has several shortcomings. For instance, UPVM [3] does not provide threads with navigational autonomy; Nomadic Thread [11] requires threads to decide their navigation at compile time; Nexus [8] permits only weak form of migration; and PM2 [16] implements inter-threads communication only via message passing. Users are hence forced to make a considerable programming effort to use their threads as agents. Not suitable for problems requiring strong navigational autonomy and high level interaction between agents.

- *Agent-Based Simulators:* Various agent-based simulators have been made available to the public domain. For instance, Manta provided an ant colony simulation platform on Windows [5], and Echo enabled various genotype agents to interact over a lattice of sites constructed on top of Sun workstations [12]. Among them, the most generalized and popular system is Swarm [14] where a collection of agents form a swarm, which can be regarded as another agent in a higher-level swarm. Their main emphasis is on the construction of simulation environments not on parallel processing. Both Echo and Swarm can be parallelized using cell-based and agent-based schemes respectively (as discussed in Section 1), which must be however implemented at user level. Not suitable for high performance simulation without considerable parallelization effort from the user.

Consequently, the above systems can be applied to multi-agent applications, however they cannot match the programmability and efficiency of M++ when parallelizing these type of applications. M++ allows hundreds of thousands of agents to coexist simultaneously, migrate over the network autonomously, and interact with each other, all requiring less user-level programming effort.

6 Conclusions

In this paper we propose M++ self-migrating threads, each behaving as a fine-grained reactive agent, as a new cluster computing paradigm for multi-agent applications. While they represent a natural mechanism to program multi-agent applications, M++ threads must pay considerable migration overheads. However, our zero-copy migration and thread scheduling implementations succeeded in improving the M++ performance up to 1.85 times better than the first prototype [19]. What is most remarkable is that such minimization has been achieved in the most pessimistic case where each agent performs only a few floating-point computations before migrating, (i.e., very fine computational granularity.) Therefore, we expect M++ to become a promising cluster-computing paradigm for multi-agent applications.

The M++ system is currently being used by users in University of Tsukuba, UC Irvine, and Ehime University. The next step includes porting M++ from Solaris to Linux platforms, tuning it up for scalable computation on an AP3000 cluster, and implementing a system-level clock management and a debugging environment. Once that is completed, we will release the system to the public domain.

Acknowledgment

This research was supported by Grant-in-Aid for Scientific Research No. 11780187, Japanese Society for the Promotion of Science.

References

[1] J. Baumann, F. Hohl, K. Rothermel, and M. Straβer. Mole - concenpts of a mobile agent system. In *Mobility Processes, Computers, and Agents*, pages 536–554. Addison-Wesley, 1999.

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

[3] 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.

[4] R. Collins and D. Jeffereson. Antfarm. In *Artificial Life II*, pages 579–601. Addison-Welsey, 1992.

[5] A. Drogoul, B. Corbara, and F. D. Manta: New experimental results on the emergence of artificial ant societies. In *Artificial Societies: the computer simulation of social life*, London, 1995.

[6] J. M. Epstein and R. Axtell. *Growing Aritificial Societies: Social Science from the Bottom Up*. MIT Press, 1996.

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

[8] 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.

[9] M. Fukuda, L. F. Bic, and M. B. Dillencourt. Global virtual time support for individual-based simulations. In *Proc. of PDPTA'98*, pages 9–16, Las Vegas, NV, July 1998.

[10] M. Fukuda, N. Suzuki, and L. F. Bic. Introducing dynamic data structure to mobile agents. In *Proc. of PDPTA'99*, pages 1854–1860, Las Vegas, NV, 1999.

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

[12] T. Jones and S. Forrest. An introduction to sfi echo. Technical report, Santa Fe Institute, 1600 Old Pecos Trail, Suit A., Santa Fe NM 87501, November 1993.

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

[14] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The swarm simulation system: A toolkit for building multi-agent simulations. Technical report, Argonne National Laboratory, University of Chicago, June 1996.

[15] K. L. Morse. An adaptive, distributed algorithm for interest management. Ph.d. dissertation, Dept. of Information and Computer Science, UC Irvine, CA 92697, Feb. 2000.

[16] 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.

[17] Norberto, E. Nawa, M. Korkin, and H. de Garis. Atr's cambrain project: The evolution of large-scale recurrent neural network modules. In *Proc. of PDPTA'98*, pages 1087–1094, Las Vegas, NV, July 1998.

[18] L. Prylli and B. Tourancheau. Bip: a new protocol designed for high performance networking on myrinet. In *Proc. of PC-NOW'98*, pages 472–485. Springer LNCS 1388, 1998.

[19] N. Suzuki, M. Fukuda, and L. F. Bic. Self migrating threads for multi-agent applications. In *Proc. of the 1st IEEE Int'l Workshop on Cluster Computing - IWCC'99*, pages 221–228, Melbourne, Australia, December 2-3 1999.