Remote Exploration Using Cooperative Autonomous Agents*

Paper Tracking Number 446

Fredrik Lööw
Fredrik Mihlberg
Kungl Tekniska Högskolan
SE-100 44 Stockholm
Sweden
{d98-fmi,d98-flo}@nada.kth.se

Munehiro Fukuda
Computing and Software Systems
University of Washington
Bothell, WA 98011, USA
mfukuda@u.washington.edu

Luis Miguel Campos†
Samuel Irvine
Information and Computer Science
University of California, Irvine
Irvine, CA 92697, USA
{lcampos,sirvine}@ics.uci.edu

ABSTRACT
Multi-agent simulations have gained popularity as a new paradigm for analyzing problems in areas such as remote space exploration, social behavior, and biological modeling. We have developed an environment for large-scale multi-agent applications that provides support for thousands of autonomous agents, each with their own goal, behavior, and coordination scheme. The platform, known as M++, was developed to allow for high performance simulations while maintaining a simple programming interface to allow for rapid development of simulations. M++'s self-migrating thread structure allows a simulation to scale from small to large, while requiring few changes to the initial implementation.

This paper describes the results of a simulation taken from the area of remote exploration, where multiple agents are required to cooperate in order to collect samples from unknown locations in an environment containing static obstacles.

Keywords
multiagent-based simulation, mobile agents, cooperation, agent architectures

1. INTRODUCTION
The multi-agent paradigm views applications as the interaction between a collection of agents. Each agent has its own goals, behaviors, and local information in a synthetic world [8]. In recent years practical use of multi-agent systems has gained popularity in areas such as remote space exploration, social behavior, biological modeling and other areas that traditional mathematical techniques cannot always represent effectively. However two issues of paramount importance must be addressed before this new paradigm can be successfully applied to solve real world problems. The first issue is how to provide for high performance to large scale applications that often require millions of fine-grained agents. The second is how to facilitate the mapping of a problem description in terms of agents into a concrete software implementation (i.e ease of programmability).

Parallelization is one of the most promising techniques to address the issue of scalability and granularity. Two parallelizing schemes can be considered: cell-based and agent-based schemes [7].

While obtaining the desired effect of application parallelism, both schemes place the burden of parallel programming (processor allocation, message passing, and inter-agents synchronization) on the designer of the model. For instance, the cell-based scheme requires each processor to exchange the information of cells and agents that exist on the processor boundary. The agent-based scheme must implement an "interest manager" that keeps track of which agents are interested in communicating with each other. Without an intelligent interest manager [17], inter-agent messages would be broadcast over the system, leading to very inefficient use of the network and a subsequent degradation in system performance. In either scheme, an agent's functionality must address its behavioral characteristics relative to the simulation as well as methods for exploiting parallelism in the underlying system. As a result, the techniques used to parallelize multi-agent applications make it difficult for model designers to concentrate on agent programming.

To solve both the issue of high performance and programmability we have developed an environment for large-scale multi-agent applications that provides support for thousands of autonomous agents, each with their own goal, behavior, and coordination scheme [11, 20]. The platform, known as M++,

*Student Paper
†Corresponding Author

allows for high performance simulations while maintaining a simple programming interface to allow for rapid development of simulations. M++'s self-migrating thread structure allows a simulation to scale from small to large, while requiring few changes to the initial implementation.

To analyze M++'s performance and programmability we have programmed several algorithms for a remote exploration problem first posed by [19]. The remote exploration problem can be seen as one where multiple agents are required to cooperate in order to collect samples from unknown locations in an environment containing static obstacles. A more detailed description of the problem will be given in section 3.1.

The rest of the paper is organized as follows: Section 2 gives an overview of the M++ environment. Section 3 describes the problem in detail and gives a description of the different algorithms we implemented in order to solve it. Section 4 describes the procedure used during the experimental phase and analyzes the results obtained. Section 5 contrasts M++ with other approaches to mobile agents, thread migration and simulation systems, and shows why it is a superior approach. Finally, Section 6 concludes the paper and mentions future work.

2. M++ EXECUTION MODEL

In this section we describe M++'s architecture, components, basic primitives and mode of operation. For a more detailed description please refer to [11, 20, 10]

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 is composed of 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 IDs*, each local to the current and destination logical nodes respectively.

M++ threads are programmed in the M++ language, a super set of C++ that is preprocessed into C++ and compiled into executable code. Figure 2 shows the framework of M++ code. An M++ thread 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. The *main* method resumes a threads execution from the previous execution statement when initiating a thread migration to another node (whether local or physical.) Except for the *main* method, all C++ inheritance and polymorphism rules are applicable to M++ threads. During migration M++

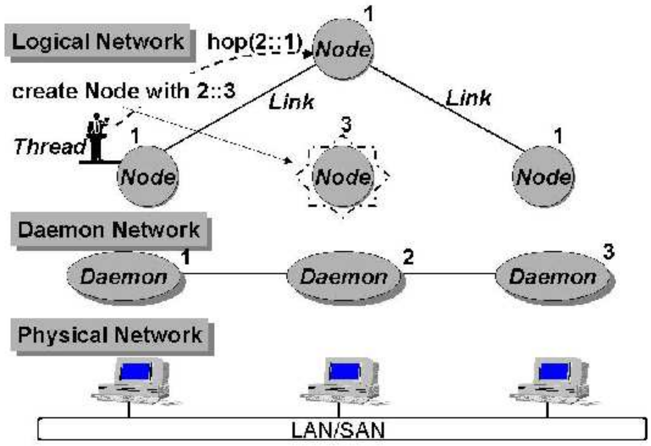

Figure 1: M++ network architecture

threads carry their execution and data status, while their code is dynamically loaded at their destination with NFS support.

```
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

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 network 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 that inform M++ threads of their network status as summarized in Table 1. The sharing mechanisms provided by these network objects can be considered 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

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

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

to sleep, wake up one and wake up all the others at the current *daemon* or *node*.

At system start-up 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 the principle of *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. This allows model designers to describe an agent's behavior from the perspective of the agent, rather than the that of the simulator system. However, strong migration requires each thread to carry all its state information and thus incurs an expensive overhead. In addition, migration always entails thread generation, destruction, and context switch.

3. SIMULATION OVERVIEW

3.1 Problem Description

The scenario we propose to analyze in this paper is an adaptation of a problem first proposed in [19] and can be described as:

> The goal is to explore a remote planet and collect samples of a particular rock. The location of rock samples is not known in advance, but it is known that they are typically clustered in

> certain spots. A number of autonomous agents are responsible for collecting the samples and returning them to their home base. No map of the planet is available, although it is known that the terrain is full of obstacles which prevents direct communication between agents.

As an aside, the original problem was developed to show the limitations of logic-based agents [12] in many practical problems and that reactive agents, in particular the subsumption architecture developed by Rodney Brooks is a better approach [2, 3, 4].

3.2 The Exploration Environment

We can think of our exploration environment $\mathcal{E}$ as an undirected graph with $n \times m$ nodes. The nodes are connected by edges in a grid pattern and the agents roam $\mathcal{E}$ by traveling along the edges from one node to another. The nodes are labeled by a pair of coordinates (x, y). With this in mind we can describe $\mathcal{E}$ as a 5-tuple

$$\mathcal{E} = (n, m, p_{items}, (low, high), p_{obstacle}).$$

The explanation of the parameters used in $\mathcal{E}$ is as follows:

n — the vertical dimension;

m — the horizontal dimension;

p_{items} — the probability that a node has items;

(max, min) — the number of items that can reside on a node;

$p_{obstacle}$ — the probability that an edge is blocked.

Of course, if an edge is blocked an agent cannot travel on that edge.

3.3 Agent Basic Design

Our design objective has been to create simple agents. By simple we mean the following: First, the agents should have none or very limited memory (i.e. a few bytes in our case). Second, the execution of the agents should be done according to the subsumption architecture [2]. According to the subsumption architecture an agent can be thought of as a set of rules where each rule has a unique priority. A rule is of the type

$$IF\ (condition)\ THEN\ (action).$$

Furthermore, we say that a rule *fires* if the condition is true and the action is performed accordingly. During the execution of an agent, an "infinite" loop scans through the rules in prioritized order. If a rule fires, the execution flow will start over again checking for the rule with highest priority.

3.4 A Simulation Run

Let us now more formally define what we mean by running a simulation. We say that we run a *simulation* with a constellation $\mathcal{C}$ of agents, when we do the following experiment.

1. Inject $\mathcal{C}$ at the node labeled by $(\lceil n/2 \rceil, \lceil m/2 \rceil)$ (i.e. home base).
2. Start the execution of $\mathcal{C}$. This means that all the agents in $\mathcal{C}$ respectively will start their execution according to their individual set of rules. Note that an item is considered collected when it is dropped off at $(\lceil n/2 \rceil, \lceil m/2 \rceil)$.
3. Count the number of execution steps $\mathcal{S}$ by $\mathcal{C}$. An execution step is when an agent travels on an edge from one node to another.
4. When all the items are collected, the experiment is terminated.

We have not put any restrictions on $\mathcal{C}$ other than that it should consist of a "reasonable" number of agents (of any kind).

It is clear that we wish to minimize the number of execution steps. We note that in this setting it would make sense to think of a simulation as a probabilistic function $sim(\mathcal{E}, \mathcal{C}) = \mathcal{S}$. Then our goal for a given $\mathcal{E}$ is to find a $\mathcal{C}$ that minimizes $E[sim(\mathcal{E}, \mathcal{C})]$, where $E[\,\cdot\,]$ is the expectation operator. However, for this to be meaningful we need to introduce a more rigorous formalization which is beyond the scope of this paper.

3.5 Agent Definition

So far, we have loosely specified what an agent is. It is time for a formal definition of an agent in our context. In our context an agent is a pair

$$(\prod, load)$$

where $\prod$ is the set of rules labeled with priority relative the other rules and *load* is the maximum number of items an agent can carry. Furthermore, some of the agents are endowed with the ability to drop crumbs at each edge in effect creating "trails". These crumbs are directional, that is they point to a particular direction. The agents are assumed to have infinitely many crumbs. In some cases a crumb can be detected by every agent and in other cases only the agent that dropped the crumb can detect it. Similarly, in some cases the agents know the way to $(\lceil n/2 \rceil, \lceil m/2 \rceil)$, and in other cases they do not. Note that this does not necessarily contradict the assumption that the agents should have a very limited memory capacity; for instance, in a "real world" scenario the agents could be guided to $(\lceil n/2 \rceil, \lceil m/2 \rceil)$ by an electronic signal broadcast. By dropping a crumb that leads backwards, we mean dropping a crumb, with the opposite direction of movement, at the edge which the agent is about to travel.

The following subsection describes the behavior of the different agents we have implemented.

3.6 Types of Agents

Steel's Agent

This agent is similar to Steel's agent proposed in [19]. The rules are as follows:

1. *IF* encounter an obstacle *THEN*
   change direction.
2. *IF* carrying items AND at home base *THEN*
   drop items.
3. *IF* carrying items AND not at home base *THEN*
   drop two crumbs backwards AND go to home base.
4. *IF* detect items AND not at home base *THEN*
   pick up items.
5. *IF* detect crumb(s) *THEN*
   pick one crumb up AND follow one (random) trail.
6. *IF* true *THEN*
   move randomly.

Egotistic Agent

This type of agent drops one individual crumb, only recognized by the agent itself, when carrying items. If however there is already a crumb in the current edge the agent will not drop another one. The agent always removes personal crumbs when encounters one. It will follow crumb trails for 3 or less step in a sequence that leads closer to the home base. This agent is very similar to the previous one. The difference is that it can only detect its own trails. In other words, we have removed the ability to cooperate with other agents.

1. *IF* encounter an obstacle *THEN*
   change direction.
2. *IF* carrying items AND at home base *THEN*
   drop items.
3. *IF* carrying items AND not at home base *THEN*
   drop a crumb leading backwards AND go to base.
4. *IF* detect items AND not at home base *THEN*
   pick up items.
5. *IF* detect personal crumb(s) *THEN*
   pick up one crumb AND follow one (random) item.
6. *IF* true *THEN*
   move randomly.

Tricky Agent

This type of agent has the particularity that only the first agent that finds the items drop one single crumb. In addition, an agent following a trail of crumbs without finding any items at the end will also remove the crumb. Note that this agent will only follow crumbs leading closer to the home base iff it is already following crumbs. The previous agent did unnecessary work. The Tricky Agent will minimize the overhead by using a boolean variable $pickedLastItem$. The idea is that only the first agent that finds items should make a trail, and the last one should remove it.

1. *IF* encounter an obstacle *THEN*
   change direction.
2. *IF* carrying items AND at home base *THEN*
   drop items AND $pickedLastItem \leftarrow FALSE$.
3. *IF* carrying items AND detect only one crumb AND $pickedLastItem = TRUE$ *THEN*
   follow the trail in opposite direction AND pick up crumb.
4. *IF* carrying items AND detect more than one crumb AND $pickedLastItem = TRUE$ *THEN*
   follow a (random) trail in opposite direction AND $pickedLastItem \leftarrow FALSE$.
5. *IF* carrying items AND $pickedLastItem = FALSE$ AND detect trails in opposite directions *THEN*
   follow a (random) trail in the opposite direction.
6. *IF* carrying items AND $pickedLastItem = FALSE$ AND detect no trail in opposite direction *THEN*
   go to home base AND drop crumb leading backwards.
7. *IF* carrying items AND $pickedLastItem = TRUE$ AND detect no trail in opposite direction *THEN*
   go to home base AND $pickedLastItem \leftarrow TRUE$.
8. *IF* detect items AND not at home base AND #items $\leq$ *load* *THEN*
   pick up items AND $pickedLastItem \leftarrow TRUE$.
9. *IF* detect items AND not at home base AND #items $>$ *load* *THEN*
   pick up items.
10. *IF* detect trails *THEN*
    follow one (random) direction.
11. *IF* true *THEN*
    move randomly.

Team of Agents

Two types of agents compose the team: Explorers and Loader.

Loader agents do not search. They simply wait at the home base for crumbs. When a crumb is detected in a edge adjacent to the home base node, a loader follows the trail and collect the items at the end of the trail. It will collect crumbs on the way back to the home base only if the last item sample was taken, or it followed a crumb that did not lead to any item sample.

Explorers search randomly. An explorer agent picks up samples only if no trail of crumbs leads into the node containing the item samples. On the way to the home base it drops one crumb only if no crumb is found at the edge.

The explorer has the following rules:

1. *IF* encounter an obstacle *THEN*
   change direction.
2. *IF* carrying items AND at home base *THEN*
   drop items.
3. *IF* carrying items AND not at home base *THEN*
   drop a crumb leading backwards AND go to home base.
4. *IF* detect items AND detect no trails *THEN*
   pick up the items.

5. *IF* true *THEN*
   move randomly.

The loader has the following rules:

1. *IF* carrying items AND at home base *THEN*
   drop items AND $pickedLastItem \leftarrow FALSE$.
2. *IF* carrying items AND not at home base AND
   $pickedLastItem = FALSE$ *THEN*
   go to home base.
3. *IF* carrying items AND not at home base AND
   $pickedLastItem = TRUE$ AND detect one trail *THEN*
   go opposite direction of trail AND pick up crumb .
4. *IF* detect items AND items $\leq$ *load* *THEN*
   pick up the items AND $pickedLastItem \leftarrow TRUE$.
5. *IF* detect items AND items $>$ *load* *THEN*
   pick up the items.
6. *IF* true *THEN*
   go to home base.

Comments

We would like to remark that some rare degenerated situations can occur in which some of the agents as described above do not work well. For instance, they might end up locked in a repetitive pattern. In such cases, we apply ad hoc techniques to break the deadlock. We must remember that we want to implement the basic behavior of an agent to measure its performance. Of course, from a theoretical point of view it is highly desirable to have a universal set of rules that can be applied to all situations. This requires very careful analysis if the behavior is non trivial, and we are presently working on it.

4. EXPERIMENTAL RESULTS

The results of the simulations completed until now are shown in Figures 3 and 4. Some parameters were kept constant across all simulations. In particular we used a grid size of 50×50 and a Uniform Probability Distribution. In the case of the simulation involving the Team of Agents we used a ratio of 9:1 for Explorers versus Loaders.
All data points represent an average of several runs with the same characteristics except for the initial seed used for the random generator. This procedure mitigates the effect of unrepresentative outcomes. Additionally the best and worst results for each data point were discarded.
For sake of completeness we would like to mention that all simulations were ran at Tsukuba's M++ cluster. More information about the cluster can be found in [11]

Figure 3 compares the performance of the four algorithms being tested in terms of the number of steps needed for the collection of agents to complete the task of collecting all rock samples and return them to the home base.

The results seem quite surprising at first. We observed that Steel's algorithm showed a decrease in the number of steps needed to complete the task as the number of agents involved increased, as one would expect. However, the Team of Agents algorithm showed exactly the opposite behavior. This can be explained if we consider that the total number of steps measured is the sum of the steps taken by all agents involved. In the case of the Team of Agents, as we increased the number of Explorers by a factor of 9 compared to the number of Loaders we end up with many extra and extraneous moves by the Explorers as the Loaders finish their job. Keep in mind that even as the Loaders carry samples back to the home base, the Explorers blissfully explore the grid unaware that there are no more samples to be found. This indicates that the constant ratio used in our simulations needs to be adjusted, perhaps as a function of the grid size. The remaining two algorithms show little variation in the number of steps taken when the number of agents increases. This is a very interesting result since it seems to indicate that for a given grid size there is a (relatively) fixed number of agents required to finish the task in constant *work* (if we consider that the number of steps taken is correlated to the amount of *work* performed by the entire population of agents).
We should naturally study the effect on each algorithm of a much larger population of agents and larger grid size (and number of samples to be collected).

Figure 4 shows the actual execution time taken by the agents for different degrees of parallelism (i.e. number of daemons involved). We chose Steel's algorithm to produce the results.

As expected we observed a decrease in the total running time required to complete the task as we increased the number of daemons/processors involved in the simulation. Also as expected, the largest improvement in performance was obtained when the number of agents involved was largest (in our case 80 agents). For this scenario we measured a decrease in running time in the order of 60% when we increased the number of daemons involved from 1 to 6. For remaining cases we observed a decrease between 30% and 40%. These preliminary results show unequivocally that M++ scales well especially when many agents are involved (as it is the case any real world application). However to fully justify our scalability claim, we need to run simulations involving several thousands of agents and dozens of machines/daemons. We are currently running such simulations.

A visualization tool (Java applet) and the entire database of simulation runs will be made available soon at:
`ftp://ftp.ics.uci.edu/pub/lcampos`

5. RELATED WORK

We are fully aware that *M++* is not the only option available to researchers interested in multi-agent simulations. In this section 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 [8] 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 interact with the user via a window system. Most systems are not expected to inject a large number of mobile agents interacting with one another simultaneously. From our experiment in [20], IBM Aglets [15] for instance falls into a congested situation with only eight aglets distributed among four workstations. This is not suitable for high performance simulations without a considerable effort on the part of the user to exploit parallelism.

- *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 [5] does not provide threads with navigational autonomy; Nomadic Thread [13] requires threads to decide their navigation at compile time; Nexus [9] permits only weak form of migration; and PM2 [18] implements inter-threads communication only via message passing. Users are forced to make a considerable programming effort to use their threads as agents. Accordingly, this approach is 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 [6], and Echo enabled various genotype agents to interact over a lattice of sites constructed on Sun [14]. Among them, the most generalized and popular system is Swarm [16] where a collection of agents form a swarm, which can be regarded as another agent in a higher-level swarm. These systems place their main emphasis on the construction of simulation environments rather than on parallel processing. Both Echo and Swarm can be parallelized using cell-based and agent-based schemes respectively (as discussed in Section 1). However this must be implemented at user level. This is not suitable for high performance simulation without considerable parallelization effort from the user.

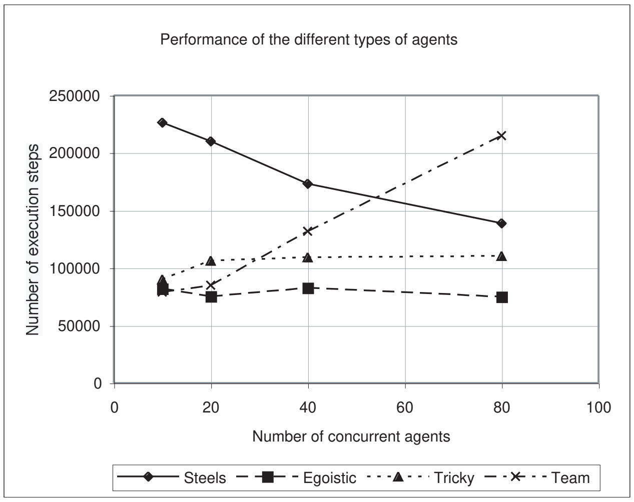

Figure 3: Algorithms Performance

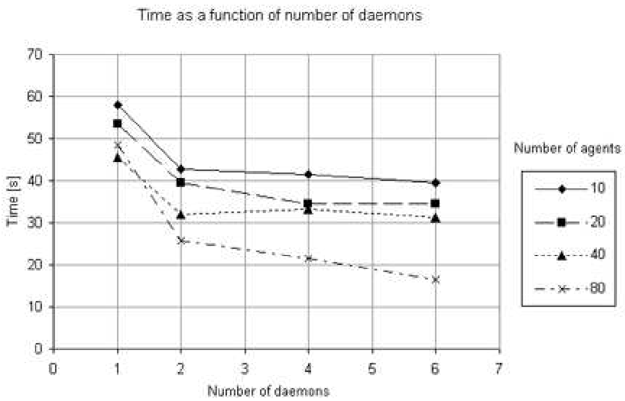

Figure 4: Scalability for Steel's algorithm

Although the systems listed above can be applied to multi-agent applications, they cannot match the programmability and efficiency of M++ when parallelizing these type of applications. M++ permits hundreds of thousands of agents to coexist simultaneously, migrate over the network autonomously, and interact with each other, while requiring a minimal programming effort.

6. CONCLUSIONS

In this paper we have described a platform, known as M++, which provides support for large scale multi-agent applications. We have shown, albeit only to a small degree, that the platform provides good performance for simulating large real world problems. The focus of this paper is however on the issue of programmability. We have shown that it is extremly easy to program autonomous multi-agent applications due to the mapping of the agent concept into M++'s self-migrating threads. We are currently obtaining new data that shows M++'s scalability (both in terms of number of deamons and problem size). We expect to add it to the final version of the paper.

7. REFERENCES

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

[2] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, Vol.2(No.1):14–23, 1986.

[3] R. A. Brooks. *Designing Autonomous Agents*, chapter Elephants do not play chess, pages 3–15. The MIT Press, Cambridge, MA, 1990.

[4] R. A. Brooks. Intelligence without reason. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 569–595, Sydney, Australia, 1991.

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

[6] 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.

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

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

[9] 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.

[10] M. Fukuda. *M++ User's Manual*, June 2001. http://faculty.washington.edu/mfukuda/m++.

[11] M. Fukuda, N. Suzuki, L. M. Campos, and S. Kobayashi. Programmability and performance of m++ self-migrating threads. In *2001 IEEE International Conference on Cluster Computing*, pages 331–340, Newport Beach, CA, USA, October 2001. IEEE Computer Society Press, USA.

[12] M. R. Genesereth and N. Nilsson. *Logic Foundations of Artificial Intelligence*. Morgan Kaufmann, San Mateo, CA, 1987.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] L. Steels. Cooperation between distributed agents through self organization. In *Decentralized AI - Proceedings of the First European Workshop on Modelling Autonmomous Agents in a Multi-Agent World*, pages 175–196, Cambridge, UK, August 1989. Elsevier Science Publishers.

[20] 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.