Dynamic Agent Allocation for Large-Scale Multi-Agent Applications

Myeong-Wuk Jang and Gul Agha

Department of Computer Science
University of Illinois at Urbana-Champaign,
Urbana IL 61801, USA
{mjang, agha}@uiuc.edu

Abstract. Although distributed computing is necessary to execute large-scale multi-agent applications, the distribution of agents is challenging especially when the communication pattern among agents is continuously changing. This paper proposes two dynamic agent allocation mechanisms for large-scale multi-agent applications. The aim of one mechanism is to minimize agent communication cost, while that of the other mechanism is to prevent overloaded computer nodes from negatively affecting overall performance. In this paper, we synthesize these two mechanisms in a multi-agent framework called *Adaptive Actor Architecture* (*AAA*). In AAA, each agent platform monitors the workload of its computer node and the communication pattern of agents executing on it. An agent platform periodically reallocates agents according to their communication localities. When an agent platform is overloaded, the agent platform migrates a set of agents, which have more intra-group communication than inter-group or inter-node communication, to a relatively underloaded agent platform. These agent allocation mechanisms are developed as fully distributed algorithms, and they may move the selected agents as a group. In order to evaluate these mechanisms, preliminary experimental results with large-scale micro UAV (Unmanned Aerial Vehicle) simulations are described.

1 Introduction

Large-scale multi-agent simulations have recently been carried out [8, 12]. These large-scale applications may be executed on a cluster of computers to benefit from distributed computing. When agents participating in a large-scale application communicate intensively with each other, the distribution of agents on the cluster may significantly affect the performance of multi-agent systems: overloaded computer nodes may become the bottleneck for concurrent execution, or inter-node communication may considerably delay computation.

Many load balancing and task assignment algorithms have been developed to assign tasks on distributed computer nodes [13]. These algorithms mainly use information about the amount of computation and the inter-process communication cost; a task requires a small amount of computational time to finish, and the

communication cost of tasks is known *a priori*. However, in many multi-agent applications, agents do not cease from execution until their system finishes the entire operation [5]. Furthermore, since the communication pattern among cooperative agents is continuously changing during execution, it may be infeasible to estimate the inter-agent communication cost for a certain time period. Therefore, task-based load balancing algorithms may not be applicable to multi-agent applications.

This paper proposes two agent allocation mechanisms to handle the dynamic change of the communication pattern of agents participating in a large-scale multi-agent application and to move agents on overloaded computer nodes to relatively underloaded computer nodes. *Adaptive Actor Architecture* (*AAA*), the extended multi-agent framework of *Actor Architecture* [9], monitors the status of computer nodes and the communication pattern of agents, and migrates agents to collocate intensively communicating agents on a single computer node. In order to move agents to another computer node, an agent platform on a single node manages virtual agent groups whose member agents have more intra-group communication than inter-group or inter-node communication. In order to evaluate our approach, large-scale micro UAV (Unmanned Aerial Vehicle) simulations including 10,000 agents were tested.

This paper is organized as follows. Section 2 introduces the overall architecture of our agent system. Section 3 explains in details two dynamic agent allocation mechanisms of our agent system. Section 4 shows the preliminary experimental results to evaluate these allocation mechanisms, and Section 5 describes related work. Finally, Section 6 concludes this paper with our future work.

2 Adaptive Actor Architecture

AAA provides a light-weight implementation of agents as active objects or actors [1]; agents in AAA are implemented as threads instead of processes, and they communicate using object messages instead of string messages. The actor model provides the fundamental behavior for a variety of agents; they are social and reactive, but they are not explicitly required to be "autonomous" in the sense of being proactive [16]. However, autonomous actors may be implemented in AAA, and many of the applications used in our experimental studies require proactive actors. Although the term agent has been used to mean proactive actors, for our purposes the distinction is not critical. In this paper, we use the terms 'agent' and 'actor' as synonyms.

Adaptive Actor Architecture consists of two main parts:

- *AAA platforms* which provide the system environment in which agents exist and interact with other agents. In order to execute agents, each computer node must have one AAA platform. AAA platforms provide agent state management, agent communication, agent migration, agent monitoring, and middle agent services.

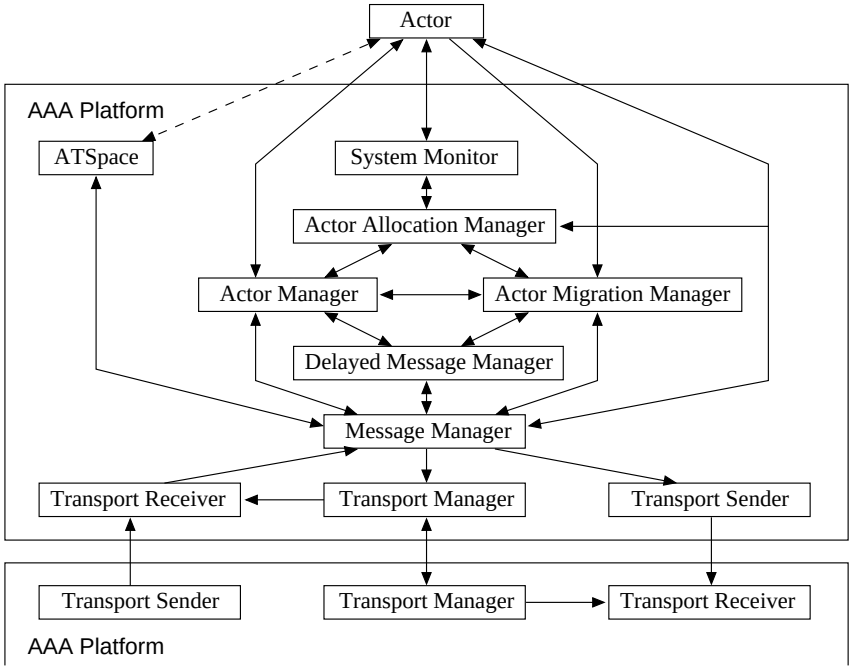

Fig. 1. Architecture of an AAA Platform

– *Actor library* which is a set of APIs that facilitate the development of agents on the AAA platforms by providing the user with a high level abstraction of service primitives. At execution time, the actor library works as the interface between agents and their respective AAA platforms.

An AAA platform consists of ten components (see Fig. 1): Message Manager, Transport Manager, Transport Sender, Transport Receiver, Delayed Message Manager, Actor Manager, Actor Migration Manager, Actor Allocation Manager, System Monitor, and ATSpace.

The *Message Manager* (MM) handles message passing between agents. Every message passes through at least one Message Manager. If the receiver agent of a message exists on the same AAA platform as the sender agent, the MM of the platform directly delivers the message to the receiver agent. However, if the receiver agent is not on the same AAA platform, this MM delivers the message to the MM of the platform where the receiver currently resides, and finally the MM delivers the message to the receiver. The *Transport Manager* (TM) maintains a public port for message passing between different AAA platforms. When a sender agent sends a message to a receiver agent on a different AAA platform, the *Transport Sender* (TS) residing on the same platform as the sender receives the message from the MM of the sender agent and delivers it to the *Transport Receiver* (TR) on the AAA platform of the receiver. If there is no

built-in connection between these two AAA platforms, the TS contacts the TM of the AAA platform of the receiver agent to open a connection so that the TM creates a TR for the new connection. Finally, the TR receives the message and delivers it to the MM on the same platform.

The *Delayed Message Manager* (DMM) temporarily holds messages for mobile agents while they are moving from their AAA platforms to other AAA platforms. The *Actor Manager* (AM) manages states of the agents that are currently executing and the locations of the mobile agents created on the AAA platform. The *Actor Migration Manager* (AMM) manages agent migration.

The *System Monitor* (SM) periodically checks the workload of its computer node and an *Actor Allocation Manager* (AAM) analyzes the communication pattern of agents. With the collected information, the AAM makes decisions for either agents or agent groups to deliver to other AAA platforms with the help of the Actor Migration Manager. The AAM negotiates with other AAMs to check the feasibility of migrations before starting agent migration.

The *ATSpace* provides middle agent services, such as matchmaking and brokering services. Unlike other system components, the ATSpace is implemented as an agent. Therefore, any agent can create an ATSpace, and hence, an AAA platform may have more than one ATSpaces. The ATSpace created by an AAA platform is called the *default ATSpace* of the platform, and all agents can obtain the agent names of default ATSpaces. Once an agent has the name of an ATSpace, the agent may send the ATSpace messages in order to use its services, and the messages are delivered through the Message Manager.

3 Dynamic Agent Allocation

In order to develop large-scale distributed multi-agent applications, the multi-agent systems must be scalable. This scalability may be achieved if the application or the infrastructure does not include centralized components which can become a bottleneck. Moreover, the scalability requires relatively balanced workload on computer nodes. Otherwise, the slowest node may become a bottleneck. However, balancing the workload between computer nodes requires significant overhead: the relevant global state information needs to be gathered, and agents have to transferred sufficiently frequently between computer nodes. Therefore, when the number of computer nodes and/or the number of agents is very large, load balancing is difficult to achieve. AAA uses the *load sharing approach* in which agents on an overloaded agent platform are moved to other underloaded agent platforms, but balanced workload among computer nodes is not required.

The third important factor for the scalability of multi-agent systems is the communication overhead. When agents on separate computer nodes communicate intensively with each other, this factor may significantly affect the performance of multi-agent systems. Even though the speed of local networks has increased considerably, the intra-node communication speed for agent message passing is much faster than inter-node communication. Therefore, if we can collocate together agents which communicate intensively with each other, com-

munication time significantly decreases. It is not generally feasible for a user to distribute agents based on their communication pattern, because the communication pattern among agents may change over time in unpredictable ways. Therefore, agents should be reallocated dynamically according to their communication patterns, and this procedure should be managed by a middleware system, such as agent platforms. Each agent platform in AAA monitors the status of its computer node and the communication pattern of agents on it, and the platform dynamically reallocates agents according to the information gathered.

3.1 Agent Allocation for Communication Locality

An agent allocation mechanism used in AAA handles the dynamic change of the communication pattern among agents. This mechanism consists of four phases: *monitoring*, *agent allocating*, *negotiation*, and *agent migration* (see Fig 2).

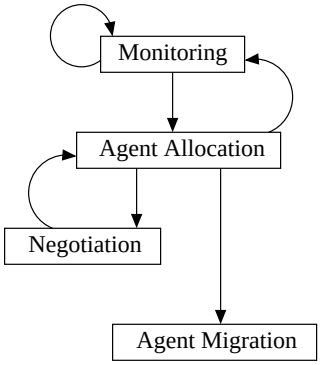

Fig. 2. Four Phases for Basic Dynamic Agent Allocation

Monitoring Phase The Actor Allocation Manager checks the communication pattern of agents under the support of the Message Manager. The Actor Allocation Manager makes a log with information about both the sender agent and the agent platform of the receiver agent of each message. Therefore, each agent element in the Actor Allocation Manager has variables representing all agent platforms communicating with this agent; M_{ij} is the number of messages sent from agent i to agent platform j.

Periodically or when requested by a system agent, the Actor Allocation Manager updates the communication pattern between agents and agent platforms with the following equation:

$$C_{ij}(t) = \alpha \left(\frac{M_{ij}(t)}{\sum_k M_{ik}(t)} \right) + (1 - \alpha) C_{ij}(t-1)$$

where $C_{ij}(t)$ is the communication dependency between agent i and agent platform j at the time step t; $M_{ij}(t)$ is the number of messages sent from agent i to agent platform j during the t-th time step; and α is a coefficient for the relative importance between recent information and old information.

For analyzing the communication pattern of agents, agents in AAA are classified into two types: *stationary* and *movable*. Any agent in AAA can move itself according to its decision, even though it is either stationary or movable. However, the Actor Allocation Manager does not consider stationary agents as candidates for agent allocation; an agent platform can migrate only movable agents. These types of agents are initially decided by agent programmers, and may be changed during execution by the agents, but not by agent platforms.

Agent Allocation Phase After a certain number of repeated monitoring phases, the Actor Allocation Manager computes the communication dependency ratio of an agent between its current agent platform and another agent platform:

$$R_{ij} = \frac{C_{ij}}{C_{in}}, \quad j \neq n$$

where R_{ij} is the communication dependency ratio of agent i between its current agent platform n and agent platform j.

When the maximum ratio of an agent is larger than a predefined threshold, the Actor Allocation Manager assigns this agent to a virtual agent group that represents the remote agent platform:

$$\max(R_{ij}) > \theta \quad \rightarrow \quad a_i \in G_j$$

where θ is the threshold for agent migration, a_i represents agent i, and G_j means agent group j.

When the Actor Allocation Manager has checked all agents and assigned some of them to agent groups, the Actor Allocation Manager starts the negotiation phase. After the agent allocation phase, information about the communication dependency of agents is reset.

Negotiation Phase Before an agent platform migrates the agents that are in an agent group to another agent platform, the Actor Allocation Manager of the sender agent platform communicates with that of the destination agent platform to check its current status. If the destination agent platform has enough space and available computational resources for new agents, its Actor Allocation Manager accepts the request for the agent group migration. Otherwise, the destination agent platform sends the number of agents that it can accept. The granularity of this negotiation between agent platforms is an agent. When the Actor Allocation Manager receives a reply from the destination agent platform, the Actor Allocation Manager sends as many agents to the destination agent platform as the number of agents recorded in the reply message. When the number in the reply message is less than the number of agents in the virtual

group, the agents to be migrated are selected according to their communication dependency ratios.

Agent Migration Phase When the destination agent platform can accept new agents, the Actor Allocation Manager of the sender agent platform initiates the migration of agents in the selected agent groups. After the current operation of a selected agent finishes, the Actor Migration Manager moves the agent to the destination agent platform decided by the Actor Allocation Manager. After the agent is migrated, the agent may restart its remaining operations.

3.2 Agent Allocation for Load Sharing

With the previous agent allocation mechanism, AAA handles the dynamic change of the communication pattern of agents. However, this mechanism may increase the workload of certain agent platforms. Therefore, our agent allocation has been extended. When an agent platform is overloaded, the System Monitor detects this and activates the agent reallocation procedure. Since agents had been assigned to their current agent platforms according to their communication pattern, choosing agents randomly to migrate to underloaded agent platforms might result in moving them back to their original agent platforms by the Actor Allocation Managers of their new agent platforms. This is because the moved agents may still have a high communication with their previous agent platform. This agent allocation mechanism consists of five phases: *monitoring*, *agent grouping*, *group allocation*, *negotiation*, and *agent migration* (see Fig 3).

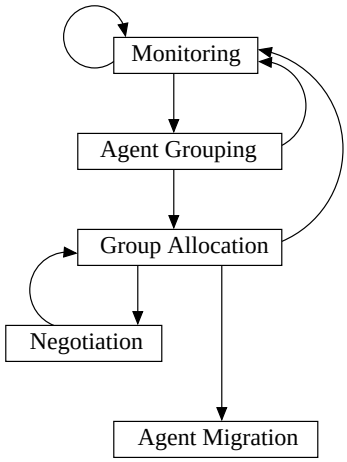

Fig. 3. Five Phases for Extended Dynamic Agent Allocation

Monitoring Phase In the second agent allocation mechanism, the System Monitor periodically checks the state of its agent platform; the System Monitor gathers information about the current processor usage and the memory usage of its computer node. When the System Monitor decides that its agent platform is overloaded, it activates the agent allocation procedure. When the Actor Allocation Manager is notified by the System Monitor, it starts monitoring the local communication pattern among agents and classifies them to agent groups. If an agent belonged to an agent group, it is assigned to this agent group; if an agent did not belong to any agent group, it is randomly assigned to an agent group. The number of agent groups that exist on an agent platform is predefined.

For checking the communication pattern of agents, the Actor Allocation Manager makes a log with information about the sender agent, the agent platform of the receiver agent, and the agent group of the receiver agent of each message. In addition to the number M_{ij} of messages sent from agent i to agent platform j, the number m_{ik} of messages sent from agent i to agent group k is updated when the receiver agent exists on the same agent platform. The summation of all m variables of an agent is equal to the number of messages sent by the agent to its current agent platform: $\sum_k m_{ik} = M_{in}$ where the index of the current agent platform is n.

After a predetermined time interval, or in response to a request from a system agent, the Actor Allocation Manager updates the communication pattern between agents and agent groups on the same agent platform with the following equation:

$$c_{ij}(t) = \beta \left(\frac{m_{ij}(t)}{\sum_k m_{ik}(t)} \right) + (1 - \beta) c_{ij}(t-1)$$

where $c_{ij}(t)$ is the communication dependency between agent i and agent group j at the time step t; $m_{ij}(t)$ is the number of messages sent from agent i to agents in agent group j during the t-th time step; and β is a coefficient for deciding the relative importance between recent information and old information.

Agent Grouping Phase After a certain number of repeated monitoring phases, each agent i is assigned to an agent group whose index is decided by $\arg\max_j (c_{ij}(t))$; this group has the maximum value of the communication localities $c_{ij}(t)$ of agent i. Since the initial group assignment of agents may not be well organized, the monitoring and agent grouping phases are repeated.

After each agent grouping phase, information about the communication dependency of agents is reset. During the agent grouping phase, the number of agent groups can be changed. When two groups have much smaller populations than others, these two groups may be merged into one group. When one group has a much larger population than others, the agent group may be split into two groups. The minimum number and maximum number of agent groups are predefined.

Group Allocation Phase After a certain number of repeated monitoring and agent grouping phases, the Actor Allocation Manager makes a decision to move an agent group to another agent platform. The group selection is based on the communication dependency between agent groups and agent platforms; the communication dependency D_{ij} between agent group i and agent platform j is decided by the summation of the communication dependency between agents in the agent group and the agent platform:

$$D_{ij} = \sum_{k} C_{kj}(t) \quad where \quad a_k \in A_i$$

where A_i represents the agent group i, and a_k is a member agent of the agent group A_i.

The agent group which has the least dependency to the current agent platform is selected; the index of the group is decided by the following equation:

$$\arg\max_{j} \left(\frac{\sum_{j,j \neq n} D_{ij}}{D_{in}} \right)$$

where n is the index of the current agent platform. The destination agent platform of the selected agent group i is decided by the communication dependency between the agent group and agent platforms; the index of the destination platform is $\arg\max_{j} (D_{ij})$.

Negotiation Phase If one agent group and its destination agent platform are decided, the Actor Allocation Manager communicates with that of the destination agent platform. If the destination agent platform accepts all agents in the group, the Actor Allocation Manager of the sender agent platform starts the migration phase. Otherwise, this Actor Allocation Manager communicates with that of the second best destination platform until it finds an available destination agent platform or checks the possibility of all other agent platforms.

This phase of the second agent allocation mechanism is similar to that of the previous agent allocation mechanism, but there are some differences. One important difference between these two negotiation phases is the granularity of negotiation. If the destination agent platform has enough space and available computation power for all agents in the selected agent group, the Actor Allocation Manager of the destination agent platform accepts the request for the agent group migration. Otherwise, the destination agent platform refuses the request. The granularity of this negotiation between agent platforms is an agent group; the destination agent platform cannot accept part of an agent group.

Agent Migration Phase When the sender agent platform receives the acceptance reply from the destination agent platform, the Actor Allocation Manager of the sender agent platform initiates the migration of agents in the selected agent group. The procedure for the following phase in the second agent allocation mechanism is the same as that of the previous agent allocation mechanism.

3.3 Characteristics

Transparent Distributed Algorithm These agent allocation mechanisms are developed as fully distributed algorithms; each agent platform independently performs its agent allocation mechanism according to information about its workload and the communication pattern of agents on it. There are no centralized components to manage the overall procedure of agent allocation. These mechanisms are transparent to multi-agent applications. The only requirement for application developers is to declare candidate agents for agent allocation as movable.

Load Balancing vs. Load Sharing The second agent allocation mechanism is not a load balancing mechanism but a load sharing mechanism; it does not try to balance the workload of computer nodes participating in an application. The goal of our multi-agent system is to reduce the turnaround time of applications with optimized agent allocation. Therefore, only overloaded agent platforms perform the second agent distribution mechanism, and agents are moved from overloaded agent platforms to underloaded agent platforms.

Individual Agent-based Allocation vs. Agent Group-based Allocation With the agent group-based allocation mechanism, some communication locality problems may be solved. First, when two agents on the same agent platform communicate intensively with each other but not with other agents on the same platform, these agents may continuously stay on the current agent platform even though they have a large amount of communication with agents on another agent platform. If these two agents can move together to the remote agent platform, the overall performance can be improved. However, an individual agent-based allocation mechanism does not handle this situation. Second, individual agent allocation may require much platform-level message passing among agent platforms for the negotiation. For example, in order to send agents to other agent platforms, agent platforms should negotiate with each other to avoid sending too many agents to a certain agent platform, thus overloading the agent platform. However, if an agent platform sends a set of agents at one time, the agent platforms may reduce negotiation messages and negotiation time.

Stop-and-Repartitioning vs. Implicit Agent Allocation Some object reallocation systems require the global synchronization. This kind approach is called the stop-and-repartitioning [2]. Our agent distribution mechanisms are executed in parallel with applications. The monitoring and agent allocation phases do not interrupt the execution of application agents.

Size of a Time Step In the monitoring phase, the size of each time step may be fixed. However, this step size may be adjusted by an agent application. For example, in multi-agent based simulations, this size may be the same as the size

of a simulation time step. Thus, the size of time steps may be flexible according to the workload of each simulation step and the processor power. To use dynamic step size, our agent system has a *reflective mechanism*; agents in applications are affected by multi-agent platform services, and the services of the multi-agent platform may be controlled by agents in applications.

4 Experimental Results

For the purpose of evaluation, we provide experimental results related to micro UAV (Unmanned Aerial Vehicle) simulations. These simulations include from 2,000 to 10,000 agents; half of them are UAVs, and the others are targets. Micro UAVs perform a surveillance mission on a mission area to detect and serve moving targets. During the mission time, these UAVs communicate with their neighboring UAVs to perform the mission together. The size of a simulation time step is one half second, and the total simulation time is around 37 minutes. The runtime of each simulation depends on the number of agents and the collaboration policy among agents. For these experiments, we have used four computers (3.4 GHz Intel CPU and 2 GB main memory) with a Giga-bit switch.

For UAV simulations, the *agent-environment interaction* model has been used [3]; all UAVs and targets are implemented as intelligent agents, and the navigation space and radar censors of all UAVs are implemented as environment agents.

To remove centralized components in distributed computing, each environment agent on a single computer node takes charge of a certain navigation area. UAVs communicate directly with each other and indirectly with neighboring UAVs and targets through environment agents. Environment agents provide application agent-oriented brokering services with the ATSpace [10]. During simulation, UAVs and targets move from one divided area to another, and UAVs and targets communicate intensively either directly or indirectly.

Fig. 4 depicts the difference of runtimes in two cases: dynamic agent allocation, and static agent allocation. Fig. 5 shows the ratio of runtimes in both cases. These two figures show the potential performance benefit of dynamic agent allocation. In our particular example, as the number of agents is increased, the ratio also generally increases. With 10,000 agents, the simulation using the dynamic agent allocation is more than five times faster than the simulation with a static agent allocation.

5 Related Work

The mechanisms used in dynamic load balancing may be compared to those in AAA. *Zoltan* [7], *PREMA/ILB* [2], and *Charm++* [4] support dynamic load balancing with object migration. Zoltan uses a loosely coupled approach between applications and load balancing algorithms using an object-oriented callback function interface [7]. However, this library-based load balancing approach depends on information given by applications, and applications activate object

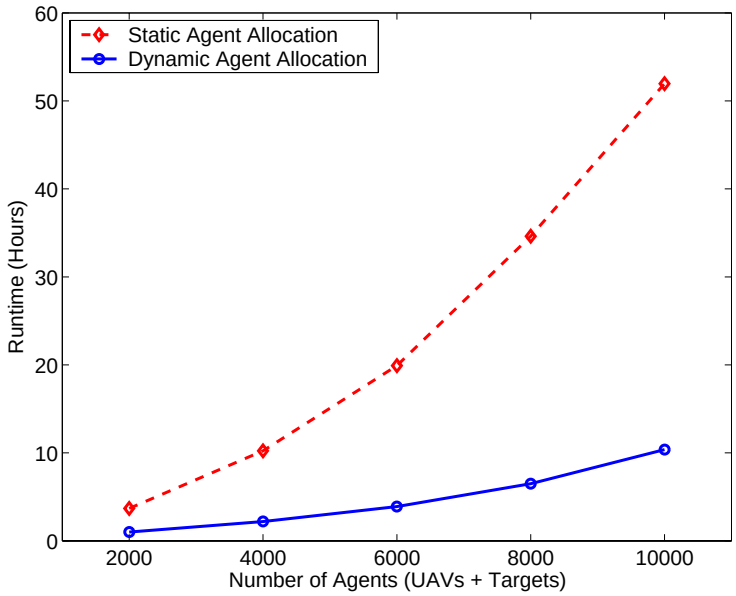

Fig. 4. Runtime for Static and Dynamic Agent Allocation

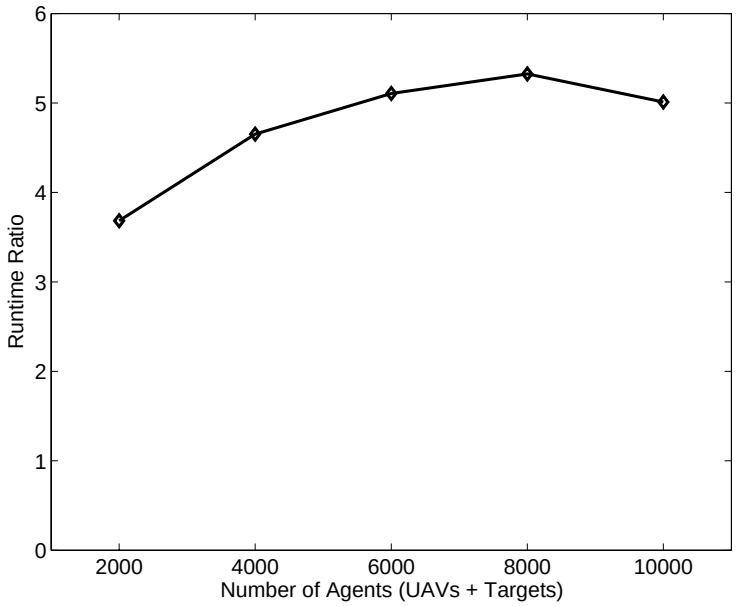

Fig. 5. Runtime Ratio of Static-to-Dynamic Agent Allocation

decomposition. Therefore, without developers' through analysis about applications, the change of dynamic access patterns of objects may not correctly be detected, and object decomposition may not be performed at the proper time. The ILB of PREMA also interacts with objects using callback routines to collect information to be used for the load balancing decision making, and to pack and unpack objects [2]. Charm++ uses the *Converse* runtime system to maintain message passing among objects, and hence, the runtime system may collect information to analyze communication dependencies among objects [4]. However, this system also requires callback methods for packing and unpacking objects as others do. In AAA, the Actor Allocation Manager does not interact with agents, but it receives information from the Message Manager and the System Monitor to analyze the communication patterns of agents and the workload of its agent platform. Also, developers do not need to define any callback method for load balancing.

J-Orchestra [15], *Addistant* [14], and *JavaParty* [11] are automatic application partitioning systems for Java applications. They transform input Java applications into distributed applications using a bytecode rewriting technique. They can migrate Java objects to take advantage of locality. However, they differ from AAA in two ways. First, while they move objects to take advantage of data locality, AAA migrates agents to take advantage of communication locality. Second, the access pattern of an object differs from the communication pattern of an agent. For example, although a data object may be moved whenever it is accessed by other objects on different platforms, an agent cannot be migrated whenever it communicates with other agents on different platforms. This is because an object is accessed by another single object, but an agent communicates with other multiple agents at the same time.

The *Comet* algorithm assigns agents to computer nodes according to their credit [5]. The credit of an agent is decided by its computation load, intra-communication load, and inter-communication load. Chow and Kwok have emphasized the importance of the relationship between intra-communication and inter-communication of each agent. However, there are some important differences. The authors' system includes a centralized component to make decisions for agent assignment, and their experiments include a small number of agents, i.e., 120 agents. AAA uses fully distributed algorithm, and experiments include 10,000 agents. Because of the large number of agents, the Actor Allocation Manager cannot analyze the communication dependency among all individual agents, but only that between agents and agent platforms and that between agent groups and agent platforms.

The *IO* of *SALSA* [6] provides various load balancing mechanisms for multi-agent applications. The IO also analyzes the communication pattern among individual agents. Therefore, it may not be applied to large-scale multi-agent applications because of the large computational overhead.

6 Conclusion and Future Work

This paper has explained two dynamic agent allocation mechanisms used in our multi-agent middleware called Adaptive Actor Architecture; these agent allocation mechanisms distributes agents according to their communication localities and the workload of computer nodes participating in large-scale multi-agent applications. The main contribution of this paper is to provide agent allocation mechanisms to handle a large number of agents which communicate intensively with each other and change their communication localities. Because of the large number of agents, these agent allocation mechanisms focus on the communication dependencies between agents and agent platforms and the dependencies between agent groups and agent platforms, instead of the communication dependencies among individual agents. Our experimental results show that micro UAV simulations using the dynamic agent allocation are approximately five times faster than those with a static agent allocation.

Our experiments suggest that increased load does not necessarily result in a decrease in the performance of multi-agent applications. If agents are properly located according to their communication pattern, the processor usage of their agent platforms is quite high. Adding more computer nodes can increase the turnaround time of the entire computation; when the number of agent platforms for an application exceeds a certain limit, the inter-node communication cost becomes larger than the benefit of distributed computing. Therefore, we plan to develop algorithms to determine the appropriate number of agent platforms for a large-scale multi-agent application.

Acknowledgements

This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. *IEEE Transactions on Parallel and Distributed Systems*, 15(2):183–192, February 2004.
3. M. Bouzid, V. Chevrier, S. Vialle, and F. Charpillet. Parallel Simulation of a Stochastic Agent/Environment Interaction. *Integrated Computer-Aided Engineering*, 8(3):189–203, 2001.
4. R.K. Brunner and L.V. Kalé. Adaptive to Load on Workstation Clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112, February 1999.
5. K. Chow and Y. Kwok. On Load Balancing for Distributed Multiagent Computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(8):787–801, August 2002.

6. T. Desell, K. El Maghraoui, and C. Varela. Load Balancing of Autonomous Actors over Dynamic Networks. In *Hawaii International Conference on System Sciences HICSS-37 Software Technology Track*, Hawaii, January 2004.
7. K. Devine, B. Hendrickson, E. Boman, M. St. Jhon, and C. Vaughan. Design of Dynamic Load-Balancing Tools for Parallel Applications. In *Proceedings of the International Conference on Supercomputing*, pages 110–118, Santa Fe, 2000.
8. L. Gasser and K. Kakugawa. MACE3J: Fast Flexible Distributed Simulation of Large, Large-Grain Multi-Agent Systems. In *Proceedings of the First International Conference on Autonomous Agents & Multiagent Systems (AAMAS)*, pages 745–752, Bologna, Italy, July 2002.
9. M. Jang and G. Agha. On Efficient Communication and Service Agent Discovery in Multi-agent Systems. In *Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS '04)*, pages 27–33, Edinburgh, Scotland, May 24-25 2004.
10. M. Jang, A. Abdel Momen, and G. Agha. ATSpace: A Middle Agent to Support Application-Oriented Matchmaking and Brokering Services. In *IEEE/WIC/ACM IAT(Intelligent Agent Technology)-2004*, pages 393–396, Beijing, China, September 20-24 2004.
11. M. Philippsen and M. Zenger. JavaParty - Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.
12. K. Popov, V. Vlassov, M. Rafea, F. Holmgren, P. Brand, and S. Haridi. Parallel Agent-Based Simulation on a Cluster of Workstations. *Parallel Processing Letters*, 13(4):629–641, 2003.
13. P.K. Sinha. Chapter 7. Resource Management. In *Distributed Operating Systems*, pages 347–380. IEEE Press, 1997.
14. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of 'Legacy' Java Software. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 236–255, Budapest, June 2001.
15. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, Malaga, June 2002. http://j-orchestra.org/.
16. M. Wooldridge. *An Introduction to MultiAgent Systems*. John Wiley & Sons, Ltd, 2002.