Adaptive Placement of Parallel Java Agents in a Scalable Computing Cluster

Arie Keren and Amnon Barak*
Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel

Abstract

This paper describes a framework for parallel computing in a locally confined, scalable computing cluster (SCC) using Java agents. The framework consists of a programming model with agents and asynchronous invocations, and a scheme for adaptive placement of multiple agents in an SCC. Our scheme is geared to improve the overall performance by a dynamic match between the available cluster resources and the execution requirements of the agents. This is accomplished by agent migrations among the nodes using on-line algorithms for load leveling and reduction of the inter agent communication overhead. Simulations of several examples show that our scheme outperforms a static agent placement scheme by as much as 40% for the test cases.

1 Introduction

Agent-based software engineering has become a popular methodology for the development of advanced software systems. Java is the main language for implementing WWW agents [7]. This is due to its ability to support code mobility. Other advantages of Java are its simplicity, object-orientation, portability and multi-threading. This make it ideal for the development of parallel applications [5]. Such applications could be implemented by using multiple, collaborating agents, called "parallel agents". Parallel agents can be useful for the execution of demanding applications, e.g., data mining and high-performance computing.

A possible hardware configuration of a web-server for the execution of parallel agents is a Scalable Computing Cluster (SCC) [2], consisting of a collection of commodity workstations that are connected by a high speed LAN. This is a cost-effective high performance environment that offers opportunities for parallel processing, parallel I/O and high availability.

Several projects use Java as a language for parallel computation in a locally confined computing cluster, in which all the machines are located in close proximity and belong to the same organization, e.g., a multiprocessor web server. For example, JavaParty [11] is geared to support parallel computing on a cluster of workstations by providing a semi-transparent DSM, in which remote objects and threads are explicitly identified by means of the `remote` keyword. Object migration is supported and can be used by the runtime system to balance the load and to enhance locality. Parallel Java [6], another project similar to ours, extends Java with remote objects, one-way asynchronous invocations and object groups. A simple static load-balance strategy creates remote objects on a randomly chosen node. Migration is not supported.

This paper presents a framework for agent-based parallel computing in a locally confined computing cluster. Our framework consists of a programming model that uses Java agents and asynchronous invocation, and a scheme for adaptive placement of multiple agents in an SCC. This scheme is geared to improve

*E-mail:{ariek,amnon}@cs.huji.ac.il

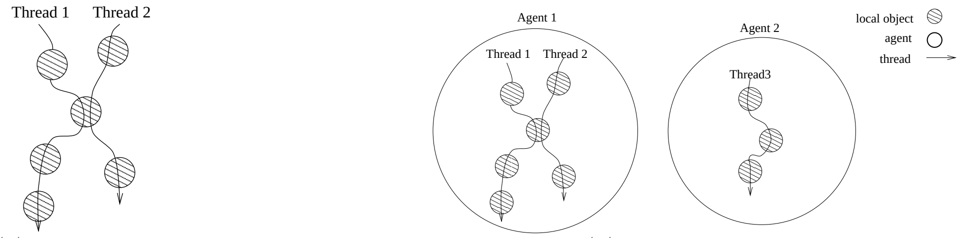
the overall performance by migrating agents among the nodes for load leveling and for reductions of the communication overhead between the agents. The scheme is scalable and based on decentralized control. The execution requirements of the agents are monitored and dynamically matched against the overall, available cluster resources.

2 Agent-based parallel computing in Java

One possible approach for Java-based parallel programming in distributed-memory multicomputers, is to use Distributed Shared Memory (DSM) in order to provide the shared-memory illusion for threads [9]. However, DSM can suffer from poor performance due to the overhead of the messages that preserve the consistency of shared objects. Therefore, in practical DSM systems the programmer is responsible to annotate the application objects with the appropriate consistency protocol, or even to implement some application-specific protocol, in order to reduce the number of such messages [4]. Note that these annotations may impair the simplicity and the portability of a parallel application.

An alternative approach, is to extend the Java object model towards distributed memory architectures, using a variation of the Actor model [1], which is the basis for many parallel object-oriented systems. In the Actor model each object has its own memory, objects are active, and they cooperate by exchanging messages.

Our framework adopts an Actor-based approach and extends Java with *agents* and *asynchronous method invocations*. An *agent* is an active object with several threads of control. Each agent has its own memory, which is shared by the agent's threads. This memory is composed of regular Java objects, that are the agent's attributes, called *local*. An agent object is different from a local object, in that it defines a sharp boundary for execution of threads. While in pure Java, threads are allowed to act across many local objects, as shown in Figure 1a, in our model the activity of a thread is bounded by the agent which created the thread, see Figure 1b.



(a) Java threads (b) Java with agents

Figure 1: Agents, local object and threads.

Cooperation between agents is realized by means of *Asynchronous Method Invocations* (AMI). The semantics of an AMI is as follows: (a) the invoking thread continues its execution without waiting for the completion of the method; (b) the invoking thread can supply a *continuation* object to the AMI, that can latter be used for synchronizing with the method's completion, or retrieving the return value, or checking for exceptions; (c) the invoked agent creates a thread which executes the method; and (d) local objects that are parameters of an AMI, are transfered by value. Continuation objects are the primary mechanism for synchronization between agents and various synchronization schemes can be provided by them.

3 Adaptive placement of agents in SCC

This section presents a scheme for adaptive placement of multiple agents in an SCC. The goal of this scheme is to provide better utilization of the resources of the SCC, such that the overall performance of the agents is improved. This goal is achieved by:

1. Migrating agents from over-loaded to less loaded nodes, in order to distribute the load evenly among the nodes.
2. Migrating intensively communicating agents to common nodes, subject to load leveling constrains, in order to benefit from shared memory to minimize the communication overhead.

The scheme includes algorithms for on-line measurement of the resource utilization, dessimination of load information among the nodes, and decision on migration of agents based on the collected information. The algorithms are scalable in the sense that they are based on decentralized control, the migration decisions involve at most two nodes, and the number of exchanged messages is logarithmically proportional to cluster size.

3.1 Measurement and dessimination of the load index

The load index of a node is based on the node's *utilization*. This index is sent to other nodes using two simultaneous dessimination schemes. First, each node sends its load index by attaching it to messages that are sent by its agents to other nodes. Load indexes are also sent to randomly chosen nodes, using a probabilistic load exchange algorithm [3], in order to support a dynamic configuration. The net result is that at each time unit, each node has information about a subset of nodes in the cluster. This subset is called the node's *information window*.

We note that in addition to the load index, each node measures parameters of its agents that are used locally. These parameters are the agent's load, which is the fraction of the agent's CPU usage; and the agent's communication to other agents in the local node and in other nodes.

3.2 The migration decision algorithm

The migration decision algorithm is executed periodically by each node, in an asynchronous manner. First, the algorithm determines if migrating some of its agents to another node will result in a performance gain, which is a function of the resulting change in the load and the inter-node communication. This is done for each node in the local information window. Then, if a substantial gain, i.e., above a threshold value, is obtained, the migration follows. We note that before the migration takes place, a reservation message is sent, to avoid concurrent migrations.

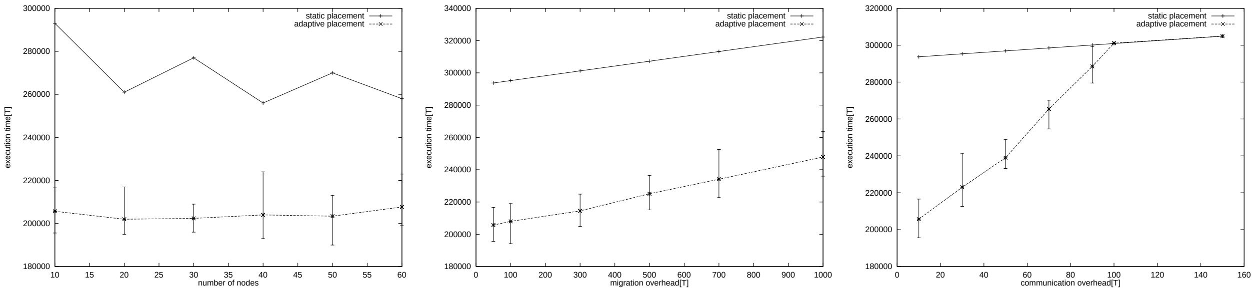
4 Performance evaluation

The performance of the adaptive placement scheme was evaluated by simulations of two agent-based applications under static and adaptive placement schemes. This section describes the applications and the results of the simulations.

4.1 Simulation of client-server application

We first present a simulation of an agent-based, scalable Web server in which the data is distributed among the nodes. Each node has a server agent, that is responsible to retrieve local data. Client agents arrive to the Web server to process information. Each client invoke the server agent of a specific node to retrieve the required data. Initially the clients are placed at the nodes which have their data, i.e., the initial assignment does not accrue any communication overhead. The simulation includes a large number of clients, which are distributed among the nodes uniformly. We note that in spite the even distribution of the clients among the nodes, the loads of the nodes are not balanced due to the different amounts of work of the clients. This uneven load distribution could further be amplified in web servers in which many clients attempt to access the same data.

The results of the simulations are presented in Figure 2. The value of the time unit, T, is $100\mu sec$. Figure 2(a) presents the execution times of the clients vs. the number of nodes. The figure shows the advantage of the adaptive placement, which is 20-30% better than the static placement. Next, we checked the sensitivity of the adaptive placement scheme to the migration overhead parameter (m) for 10 nodes. In the simulation, m was gradually increased from $50T$ to $1000T$. The results of these simulations are given in Figure 2(b), showing that in spite of the migration overhead incurred by the adaptive scheme, its performance was consistently better than that of the static scheme. A comparison of the performance of the two schemes, when the communication overhead (o) is increased is presented in Figure 2(c). From the figure it can be seen that as long as the communication overhead is below $100T$, which is the mean amount of work of an agent, the adaptive scheme outperforms the static scheme. Then, for larger values of this overhead, the two schemes perform identically. This can be explained by the fact that migration algorithm accounts for the communication overhead between agents. Thus, in spite of load imbalances among the nodes, migration of clients were discouraged when the communication overhead was equal or greater than the agent's work.



(a) Execution time vs. P (b) Execution time vs. m (c) Execution time vs. o

Figure 2: Static vs. adaptive execution times of the web server application.

4.2 Simulation of a parallel scientific application

This example is a simulation of the Jacobi iterative method for solving a two-dimensional, first-order finite difference equation that results from the Laplace's equation. The parallel agent-based formulation of Jacobi is carried out by partitioning the two-dimensional grid into square regions and assigning each square to an agent. The agents work in parallel, such that each agent computes the kth iteration and then exchanges the computed values with its neighbors. To simulate time-sharing environment, the Jacobi agents are executed in parallel with sequential client agents that arrive to the cluster at arbitrary times, and are placed in a round-robin fashion on 8 nodes.

The execution times of the Jacobi application were measured for the adaptive and the static placement schemes on an increasing number of nodes. The results of the simulations are shown in Figure 3. These results show that the adaptive scheme outperforms the static scheme by 17%– 40%, when the number of nodes is increased from 8 to 32.

5 Summary and future work

This paper presented a framework for parallel programming by means of Java agents. The framework makes it possible to construct mobile parallel applications that can be executed on any general purpose computing

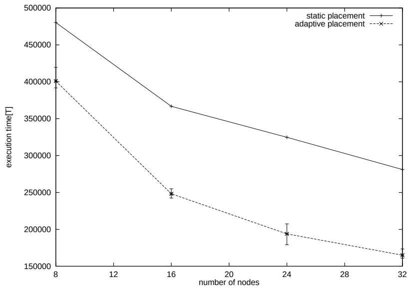


Figure 3: Simulation results for the Jacobi application.

cluster. It also contains a scheme for an adaptive placement of multiple agents in a scalable computing cluster.

We plan to implement the described framework in our scalable computing cluster that consists of over 80 Pentium based PC's connected by the Myrinet LAN. The implementation involves two concurrent efforts. The first one will use an existing Java agent system which offers communication and mobility mechanisms, e.g. Voyager [10]. We will add a management layer that will implement the proposed load balancing scheme. The second effort emphases an efficient implementation of the basic communication and mobility mechanisms in the JVM, or above it. Current implementations of distributed objects in Java, e.g. RMI, suffer from poor performance which limit their possible usage in parallel applications. On the other hand, recent works showed that multi-threading and asynchronous models of communication can be efficiently implemented on high-speed networks [8]. It is an open question if these methods can be used for an implementation of low-latency communication in Java.

We are also developing a new scheme, that is based on theoretically proven competitive algorithms, for on-line assignment of agents in a scalable computing cluster. The scheme minimizes the maximum slowdown among agents with CPU, memory and communication requirements. It can be easily extended to handle additional resources as well.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[2] T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for NOW. *IEEE Micro*, 15(1):54–64, February 1995.

[3] A. Barak, S. Guday, and R. G. Wheeler. *The MOSIX Distributed Operating System, Load Balancing for UNIX*, chapter 8. LNCS 672. Springer-Verlag, 1993.

[4] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[5] G. Fox and W. Furmanski. Overview of Java for parallel computing and as a general language for scientific and engineering simulation and modelling. *Concurrency: Practice and Experience*, 9(6), June 1997.

[6] L. V. Kale, M. Bhandarkar, and T. Wilmarth. Design and implementation of parallel Java with global object space. In *PDPTA'97*, July 1997.

[7] J. Kiniry and D. Zimmerman. A hands-on look at Java mobile agents. *IEEE Internet Computing*, 1(4):21–30, July 1997.

[8] K. Langendoen, R. Bhoedjang, and H. Bal. Models for asynchronous message handling. *IEEE Concurrency*, 5(2):28–38, April 1997.

[9] F. Mueller. Distributed shared-memory threads: DSM-Threads. In *Workshop on Runtime Systems for Parallel Programming (RTSPP), IPPS'97*, April 1997.

[10] ObjectSpace. *ObjectSpace Voyager, Technical Overview*, September 1997.

[11] M. Philippsen and M. Zenger. JavaParty - transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.