Performance of the MESSENGERS Autonomous-Objects-Based System

Munehiro Fukuda
Lubomir F. Bic
Michael B. Dillencourt

Department of Information and Computer Science
University of California, Irvine

Abstract. MESSENGERS is a system for general-purpose distributed computing based on the concept of autonomous mobile objects, capable of navigating through a network and invoking native-mode C functions in the nodes they visit. We present performance measurements using three specific applications to illustrate the cost of using this paradigm for developing and using distributed applications.

1 Introduction

Autonomous objects are mobile entities, capable of navigating autonomously through the underlying computational network and performing various tasks at the nodes they visit. The autonomous-object approach to distributed computation has several functional advantages over the conventional message-passing approach. Autonomous objects can be programmed independently of the network topology and the hardware architecture. Computations based on autonomous objects can continue to function correctly even under faulty conditions, since objects can migrate or be dispatched to correctly functioning nodes. Algorithms can be encapsulated inside autonomous objects, thus narrowing the semantic gap between distributed algorithms and their implementation [1].

Various aspects of autonomous objects have been explored in recent years [2, 3, 4, 5, 6, 7]. Autonomous objects have two potential performance advantages: (1) network interactions can be reduced in the client-server model, because autonomous objects can do their computing at the server's site on behalf of their clients; and (2) inherent parallelism can be exploited by propagating autonomous objects over the network. Up to now, most applications of autonomous objects have focused on network service tasks, such as information retrieval and electronic commerce over WAN's (the Internet). In these application domains, the granularity of the computation is coarse, and the communication overhead and the speed of human interaction overshadow the cost of interpretation associated with running autonomous objects.

In the present paper, we explore the performance issues that arise when applying autonomous objects to perform *general-purpose* distributed computing on a LAN. As we demonstrated in [1], computing with autonomous objects has considerable advantages as a programming metaphor compared to conventional message-passing approaches, but there is some additional overhead associated with autonomous objects, due to the cost of interpretation. By measuring the performance relationships between autonomous objects and message-passing we hope to gain a better sense of the tradeoffs between these two distributed computing paradigms. The interpretation cost can be ameliorated by performing most of the computation in native-mode executable modules (referred to henceforth as native code). As the computational granularity increases, the interpretation overhead decreases as a fraction of the total computation time. Hence one relationship to be investigated is how big the computational granularity must be to make interpretation overhead acceptable.

Our performance study is done in the context of the MESSENGERS system[1], which has been previously described in [8]. We give a brief overview of the MESSENGERS system in Section 2, and we present the main results of our performance study in Section 3.

2 Overview of MESSENGERS

2.1 Principles of Operation

The MESSENGERS system involves three levels of networks as illustrated in Figure 1. The lowest level is the *physical network* (a LAN or WAN), which constitutes the underlying computational resource (Sun Workstations in our present implementation). Superimposed on the physical layer is the *daemon network*, where each daemon is a UNIX process running a MESSENGERS language interpreter. This provides a virtual environment to isolate the application from physical network details. The *logical network* is an application-specific computation network created on top of the daemon network. Multiple logical network nodes may be created on the same daemon network nodes, thus running on the same physical node, and they may be interconnected by logical links into an arbitrary topology. The logical network is the application's organization of the data and function space and is used by individual *Messengers* to navigate through this space and to coordinate the invocation of native-mode C functions.

Messenger programs, referred to as *Messenger scripts*, are written in a subset of C and are compiled into a low-level interpretive language for more efficient transport and parsing. Each script is carried in its entirety by the Messenger as it propagates through the network and is replicated each time the Messenger needs to follow more than one logical link.

Messengers scripts distinguish three types of variables. *Messenger variables* are private to and carried by each Messenger as it propagates through the logical computational network. *Node variables* are resident in nodes of the logical network and shared by all Messengers visiting the same logical node. *Network variables* are predefined at each logical node and give each Messenger access to the network information local to the current node.

[1] Lowercase (Messengers) is used to denote the individual autonomous objects while upper case (MESSENGERS) denotes the system as a whole.

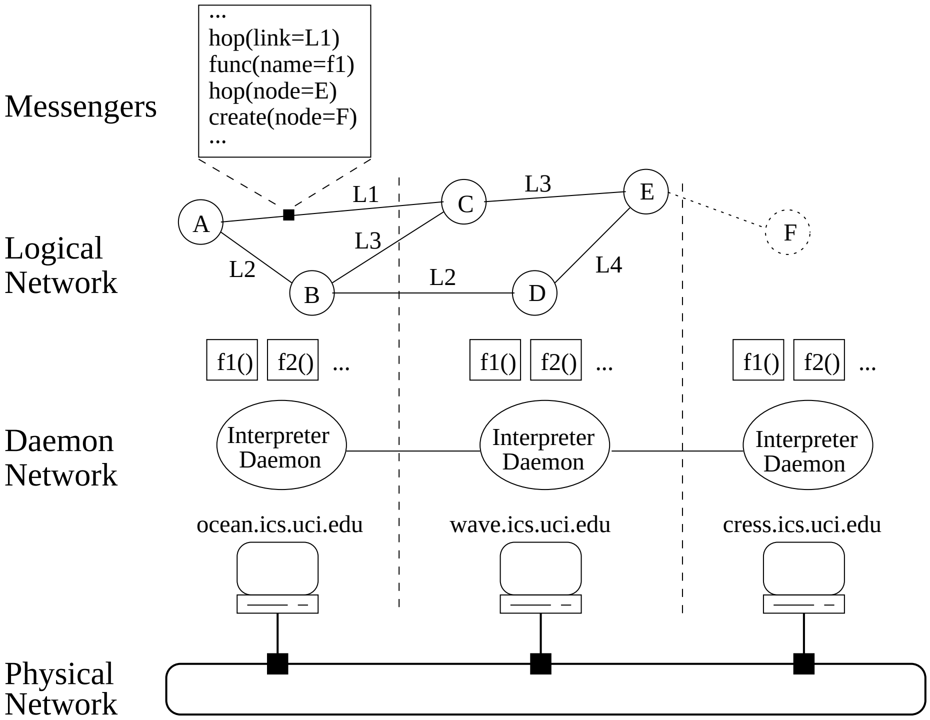

Fig. 1. Network Layers

A Messenger script consists of variable declarations followed by a sequence of statements. *Computation statements* include arbitrary assignment expressions and the common control constructs supported in C. *Navigational statements*, expressed as predefined C functions, permit a Messenger to create new logical links and nodes, delete existing ones, and move arbitrarily through the network by following links or jumping to specific nodes. A number of optional parameters may be specified as part of the navigational statements, including the specification of particular nodes, links, or link weights, or the use of wild cards for partial matching. The Messenger is propagated to all destinations that match the navigational specification. Finally, *task-coordinating statements* allow a Messenger to load and execute ordinary precompiled C functions. This is accomplished using an *exec* statement, which spawns a separate concurrent process for the invoked function, or using a *func* statement, which invokes the function as part of the current Messenger's behavior and returns its results to it.

System library functions are also provided for a variety of supporting functions, including explicit creation and destruction of Messengers, inter-Messengers communication, and virtual time management.

To illustrate the above principles graphically, consider Figure 1, which shows a Messenger in transit between two logical nodes, A and C. The Messenger's script fragment shows "hop(link=L1)" as the current statement that effected the transfer of the Messenger from the current node A along link L1. At the destination node, C, the Messenger's next statement, "func(f1)", will invoke the node-resident function f1. Upon returning from the function call, the next statement, "hop(node=E)", causes the Messenger to transfer to the new destination node E. From node E, the Messenger's next step is a *create statement*, which creates a new node, F, connected to the current node E. The link is unnamed and the physical node is unspecified, which causes the creation of the new node on a physical node chosen by the system. This allows the logical network to spread throughout the daemon network using system defaults or mapping information supplied in the create statement.

2.2 Advantages of Autonomous Objects

The use of autonomous objects as a general coordination paradigm for the development and operation of general-purpose distributed applications has the following major advantages [9]:

- *Dynamic Composition.* Autonomous objects paradigms are capable of supporting open-ended applications, i.e. those where the necessary functionalities cannot be determined *a priori.* This typically includes interactive simulations or control applications where the human operator is an integral component of the system. With conventional message-based approaches, the system's capabilities are limited by the predefined types of message is it able to respond to. Autonomous objects, on the other hand, are interpreted and hence it is possible to introduce arbitrary new behaviors or functionalities at run time. This property will be illustrated by the Toxicology simulation of Section 3.3.1.
- *Topology Invariance.* Some autonomous-objects-based systems, such as MESSENGERS, provide a daemon network as an abstraction of the physical network, and a logical network for structuring the application's data and function space. This allows applications to be written without any knowledge of the physical network and, consequently, may be run on networks with arbitrary sizes or topologies without any modifications. This property will be illustrated by the divide-and-conquer application of Section 3.3.2. The absence of any application-specific node programs also permits computations within dynamically changing network topologies, since the interpreter daemon may be started automatically when a node restarts, thus automatically integrating it within the current computational network.
- *Proximity Correspondence.* Many applications require only near-neighbor communication at the logical level. For example, the complex behavior of a group of individuals results from simple interactions among individual neighbors [10]. When proximity in the logical (simulated) space is correlated with proximity in the physical (network) space, most communications will take place within the same processor or between only a small number of physically adjacent processors. Unlike conventional systems where process migration must be explicitly programmed for, autonomous objects (each

representing an individual) automatically hop between processors whenever they cross the boundary of the logical cell mapped onto the processor. This property will be illustrated by the individual-based ecology simulation of Section 3.3.3.

It is in this broader context of general-purpose distributed computations that we investigate the performance trade-offs of our autonomous-objects-based systems, MESSENGERS.

3 Performance Evaluation

We have evaluated MESSENGERS on a network of Sun SPARC Station ELC's (16MB memory each) connected by a 10Mbps Ethernet. The interpreter daemon is implemented as a single Unix process with two concurrent threads (one performing send and the other receive operations) for communication with other daemons over the network.

3.1 Interpreter Performance

The first series of experiments investigates the overhead incurred by the interpreter. It compares the performance of basic operations running MESSENGERS on a *single* machine and the analogous operations executed by a conventional C program using the Unix `lwp` (lightweight process) multi-threading library. Four tests, as summarized in Table 1, were performed. Test 1 shows that interpreting arithmetic operations in MESSENGERS scripts incurs an order-of-magnitude overhead compared to native code execution. Test 2 shows that MESSENGERS function call is two orders of magnitude more costly than a C function call. Creating a Messenger requires approximately three times as much as spawning a `lwp` thread (but much less than a Unix process), as shown by Test 3. Finally, a Messenger context switch is comparable to a `lwp` thread context switch (Test 4).

The above shows that interpretation of MESSENGERS scripts results in considerable overhead in same cases. It is important to note, however, that applications will not necessarily run an order of magnitude or more slower than native code. The reason is that every Messenger is a composition of interpreted script code and native-mode C functions, as explained in Section 2, where the latter executes with no additional overhead. Furthermore, communication is typically the most important source of overhead in distributed applications and hence the cost of interpretation is only one small factor of the overall performance. The following section considers the cost of communication.

3.2 Communication Performance

Since communication is the most important feature determining the performance of distributed applications, we first compared the performance of a Messenger *hop* operation with Unix socket communication. Table 2 shows the result of

Test #	programs	times of slow down
Test 1	*Arithmetic interpretation vs execution*	
	(a) Sequential C program	1.0
	(b) Messengers	12.8
Test 2	*Func statement vs ordinary function call*	
	(a) Sequential C program	1.0
	(b) Messengers	102.5
Test 3	*Creation: Messenger vs thread/process*	
	(a) Thread	1.0
	(b) Messengers	2.9
	(c) Unix process	146.0
Test 4	*Context switch: Messenger vs thread*	
	(a) Thread	1.0
	(b) Messengers	1.1

Table 1. Interpreter Performance

sending 800 bytes of data between two remote threads using sockets versus the migration of a comparable size Messenger between two physical nodes.

The second test, shown in Figure 2, compares three simple programs. The first is a sequential C program that performs $2 * j$ floating point operations in a loop. The second involves two `lwp` threads, A and B. Thread A first sends a 16-byte data to thread B. Both threads then perform j floating point operations. Finally, thread B sends a 16-byte data to thread A. The third program is similar to the second but uses two Messengers. Messenger B hops to another workstation; both Messengers perform j floating point operations; Messenger B then hops back to A's site.

The three programs are designed to measure the effects of granularity on speedup. The curves show that both `lwp` threads and Messengers achieve speedup over sequential execution once a certain granularity (between 1000 and 10000 operations) is reached. The results show further that, past a certain granularity, the additional overhead of MESSENGERS becomes negligible as compared to conventional multi-threading execution.

programs	times of slow down
Remote hop vs thread communication with a socket	
(a) Thread/socket	1.0
(b) Messengers	6.9

Table 2. Network Performance

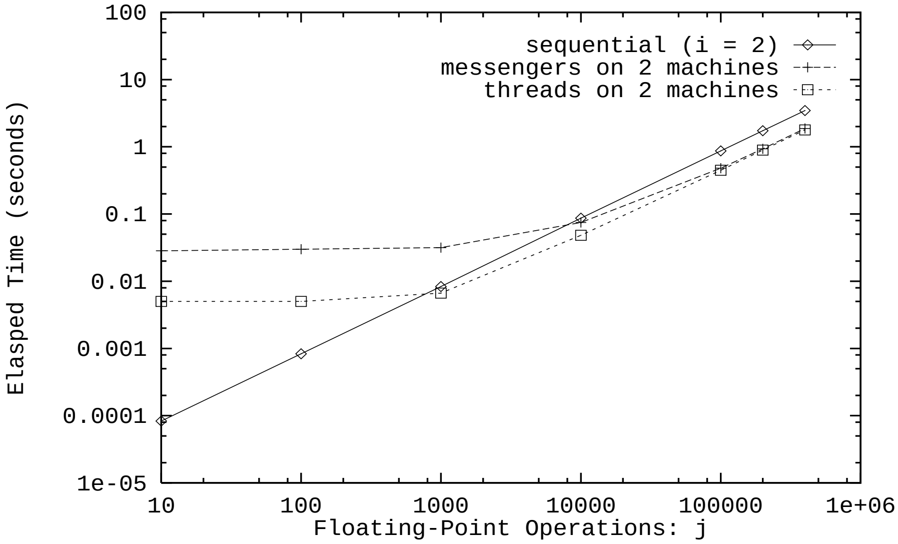

Fig. 2. Network Latency Test

3.3 Specific Applications

The performance measurements described in the two preceding subsections are generic. We have evaluated MESSENGERS' performance further using three concrete programs: a toxicology simulation, convex hull computation, and Wa-Tor, which take advantage of MESSENGERS' capabilities: *dynamic composition*, *topology invariance*, and *proximity correspondence* respectively.

A Toxicology Simulation This is a time-driven simulation, developed in collaboration with UCI's College of Medicine, which simulates the distribution over time and metabolism of inhaled Styrene by different organs of a living organism [11].

The basic approach is to map each organ onto a separate logical node as shown in Figure 3. Each node contains the necessary sets of differential equations and constants describing the organ's behavior. The Styrene-carrying fluids (blood) are implemented as waves of consecutive Messengers. Initially, a Messenger is injected into the Lung node, where it computes the starting Styrene concentration. The Messenger then replicates itself along all arterial links, carrying the current Styrene value to all the other organs. At each node, the Messengers trigger the execution of the appropriate function to compute the new Styrene concentrations for the current simulated time increment. Each Messenger then

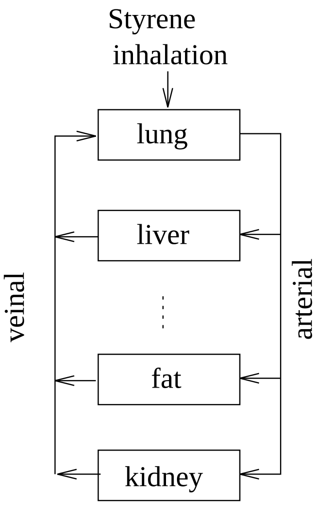

Fig. 3. Styrene Inhalation: an Example of Dynamic Decomposition Application

hops back to the Lung node along the veinal link and deposits its Styrene value. All but the last arriving Messenger terminate. The only surviving Messenger then starts the next cycle with the new values.

The main advantage of the MESSENGERS implementation is that it supports the dynamic composition of the application. In particular, the circulating Messengers could be modified at runtime to invoke different functions, thus changing the model's functionality on the fly. This capability also supports interactive and incremental software development and use.

To assess the performance of the MESSENGERS implementation, we compared three versions of the program: a MESSENGERS version running on one machine, a MESSENGERS version running on two machines, and a sequential version implemented in C, not using MESSENGERS. The basic cycle alternates between computation and communication and hence its performance depends largely on the granularity of each node computation relative to the communication overhead. To study the trade-offs, we inserted dummy computations, which arbitrarily increase the number of operations performed at each node. The dummy computations consisted of a "func" call to a C program in the MESSENGERS version and a dummy loop in the C version, so the additional overheads imposed were roughly the same. Figure 4 shows that the price for the dynamic composability is high: 10,000 operations at each node are necessary to compensate for the interpretation and the communication overhead[2]. The figure shows

[2] Note that the MESSENGERS implementation on one machine never actually "catches up" to the C implementation. Rather, with 10,000 or more operations at each node, the communication overhead becomes very small compared to the computation cost.

further that the break-even point between running on a single machine versus two machines is at about the same level of granularity.

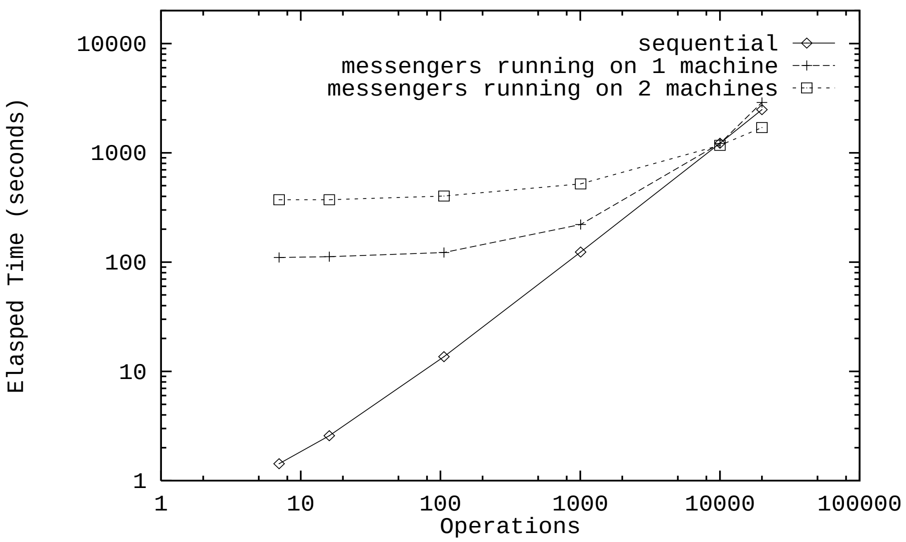

Fig. 4. Performance Result of Dynamic Decomposition Application

Convex Hull Computation This is a divide-and-conquer program for finding a convex hull of n points given in the plane [12]. It illustrates MESSENGERS' ability to compute in networks whose topology and size are unknown *a priori* and even modifiable at run time.

As shown in Figure 5, the program divides the points into two subsets, containing $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ points respectively, such that all points in the first subset lie to the left of all points in the second subset along the x-axis. It then computes the convex hulls of the subsets recursively, using the same divide-and-conquer approach. The partial solutions are merged at each level as the program back-tracks from the recursion.

This application requires a binary tree structure to be built dynamically by a frontal wave of Messengers. Initially, a Messenger creates the root of the tree. From the root, the Messenger repeatedly creates a pair of left and right child nodes by duplicating itself and dividing the given set of points into two subspaces. This is repeated for a given number of times, based on the number of available processors to work on the problem. Once the desired tree depth has been reached, each Messenger invokes a function that locally computes a convex hull of points in the corresponding subspace. During the next phase,

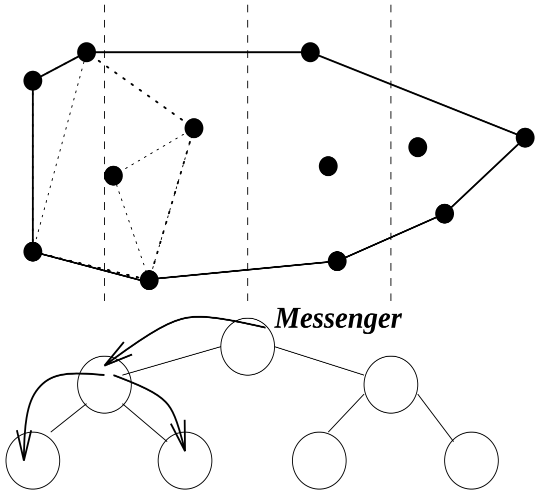

Fig. 5. Convex Hull: an Example of Network-Invariant Computation

the Messengers then work as carriers of the partial convex hulls from the leaves toward the root, merging these at each level.

Figure 6(a) shows the performance results of the convex hull program for solving the problem with 1024 through to 32768 points. This shows that for slightly above 1000 points, the effect of parallel processing on two or four machines outweighs the communication and interpretation overheads, resulting in actual speed-up. The performance however does not improve beyond four machines. There are two possible explanations for this. One is that the problem size is too small to gain from further parallelism. The second is that the current implementation of MESSENGERS supports only a fixed-size variable area, so the Messenger has been coded to always carry all points, resulting in increased communication overhead which is not strictly necessary to solve the actual problem.

To compensate for the effect of this additional communication overhead, we ran another experiment. We modified the above divide-and-conquer program to only perform a certain number of dummy floating-point operations at each node of the tree as it is being constructed but not to carry any data between nodes in either direction. This abstracts the behavior of certain other bisection programs, such as a quadrature problem for numerical integration [13], where only a single computation result must be passed back from a child to its parent node at each level, and therefore the communication overhead is constant. Figure 6(b) shows the result. The threshold to obtain an improvement from parallel computing on two nodes is approximately 6,000 floating-point operations at each node. Performance can be improved further using four or eight machines if the granularity is sufficiently large.

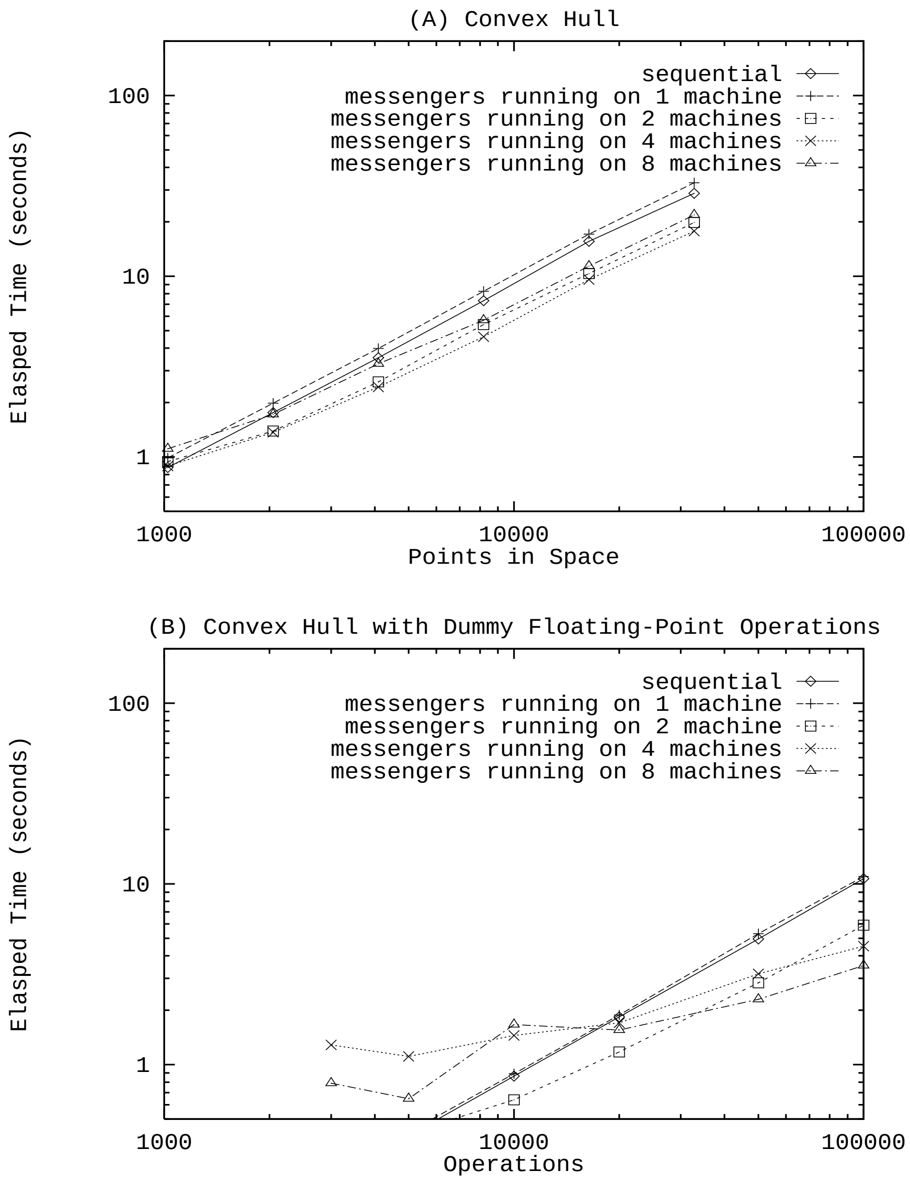

Fig. 6. Performance Result of Network-Invariant Application and Its Coarse Grain Analogue

Wa-Tor This is a simple Monte Carlo ecological simulation program implementing a model by Dewdney [14], in which idealized fish and sharks live, move randomly, breed, and eat one another in a two-dimensional ocean grid. The program illustrates MESSENGERS' capability to support individual-based simulations, where complex global behaviors emerge from simple local interactions of individuals, each implemented as a separate Messenger.

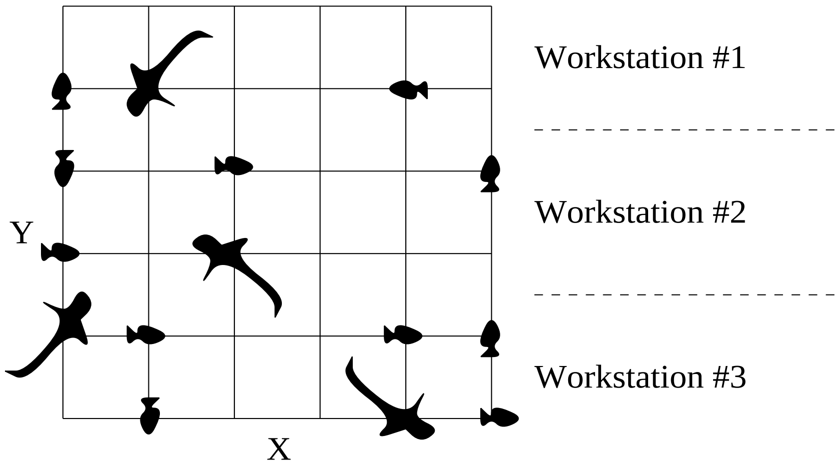

Fig. 7. Wa-Tor: an Example of Individual-Based Simulations

Figure 7 shows this simulated space, which is partitioned and distributed over physical nodes. Each individual fish and shark is implemented as a Messenger. At each step, a fish computes its new position, and possibly creates a progeny by spawning a copy of itself. If the new position is outside of the ocean strip assigned to the current node, the Messenger hops north or south to the appropriate neighboring node. The behavior of a shark is similar but includes the ability to kill other fish it encounters.

Figure 8 shows the performance results of the Wa-Tor program under increased computational granularity of each individual. The simulation space is represented as a 32×32 matrix and striped into 2, 4, and 8 segments. There are 500 Messengers representing fish and sharks. The results show that a relatively low number of operations (approximately 400) is sufficient to compensate for the interpretation and communication overhead. Furthermore, there is some speedup between different numbers of machines despite the relatively small number of individuals being simulated.

4 Conclusion

We have designed MESSENGERS as an autonomous-objects-based system for distributed computing and investigated its performance. The following consider-

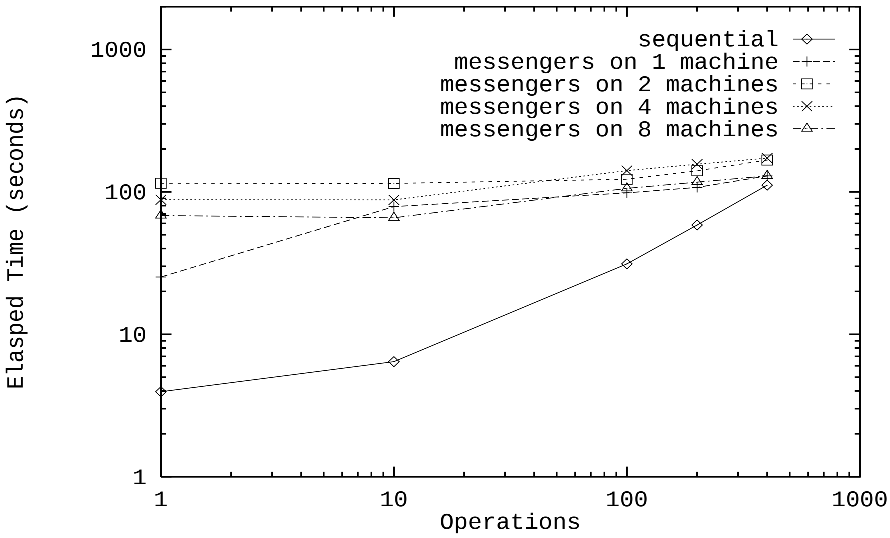

Fig. 8. Performance Result of Individual-Based Simulation

ations play an important role in determining the performance of autonomous object-based systems:

1. The cost of starting up a new autonomous object, and the cost of switching between objects. For example, in some autonomous-object systems, each object is a separate process.
2. The cost of passing parameters to native code. For example, in some systems only string parameters are allowed, so each invocation of native code requires encoding and decoding of parameters.
3. The ability of native code to efficiently access the state of the calling autonomous object.
4. The ability of objects to carry native code with them.
5. The extent to which the system provides built-in support for synchronization and coordination.

MESSENGERS has been developed as a system that supports general-purpose computation. As discussed in Section 2, the MESSENGERS language is a subset of C, and MESSENGERS is designed to coordinate native code written in C. MESSENGERS supports each of the specific performance issues listed above as follows: (1) The costs of an object's context switch and migration are closer to those of threads than processes. In particular, this applies to the context switch between interpreted Messenger code and native code; (2) MESSENGERS supports the C calling sequence with all possible C types, thus arbitrary parameters can be passed directly from MESSENGERS to C code; (3) Native code can access variables that contain the state of the Messenger; (4) Native code can be carried by Messengers; (5) each Messenger can schedule its actions using virtual time [15] maintained automatically by MESSENGERS interpreter daemons.

While the MESSENGERS interpreter overhead is on the order of one magnitude, the actual performance of applications depends on the ratio of interpreted versus directly executed code in each Messenger. With distributed applications, the most important source of overhead is communication. Since MESSENGERS' *hop* statement is more costly than a simple message send, applications need to be written at a larger level of granularity.

As shown in Figure 2, the additional cost for the *hop* statement becomes negligible in comparison with conventional socket communication when the computation granularity grows beyond 10,000 floating-point operations. Furthermore, at the same level of granularity, the effect of scalability is observed in the divide-and-conquer algorithm as shown in Figure 6-B. We are also conducting performance comparisons between MESSENGERS and PVM (Parallel Virtual Machine) [16] using various numerical computations, such as Mandelbrot set generation and matrix multiplication. The preliminary results we have obtained indicate that MESSENGERS' scalability is highly competitive with that of PVM.

All measurements presented in this paper were carried out on a slow broadcast-based network (10Mbps Ethernet) shared by many workstations. Hence these results represent a baseline for future improvements. One such improvment is a port to a new dedicated environment centered around a 100Mbps collision-free switch, which is currently being installed in our laboratory.

We are presently developing a new version of MESSENGERS which will provide support for both conservative and optimistic global virtual time maintenance. We expect these features to improve the running time of MESSENGERS applications such as the Wa-Tor program of Section 3. These enhancements will reduce the interpreter overhead by moving the burden of maintaining a global clock and achieving synchronization out of the user-written interpreted code and into the actual daemon code.

Fault tolerance is another problem that we will address as our future work. The interpreter daemon will be enhanced to detect whether any of its neighboring daemons on the daemon network enter a faulty condition. Once such faulty neighbors are detected, new Messengers can be dispatched or detoured to correctly functioning neighbors so that their computations can continue.

In view of the many benefits derived from using an autonomous-objects-based paradigm and the fact that MESSENGERS has not yet been optimized for performance in any systematic way, we feel that this paradigm is a promising tool for distributed computing. Additional information on MESSENGERS may be obtained via our WWW page: `http://www.ics.uci.edu/~bic/messengers`.

References

1. M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant. MESSENGERS: Distributed computing using mobile autonomous objects. *Information Sciences*, 1997. To appear.

2. A. Lingnau, O. Drobnik, and P. Dömel. An HTTP-based infrastructure for mobile agents. In *Fourth International World Wide Web Conference Proceedings*, pages 461–471, Sebastopol, CA, December 1995. O'Reilly and Associates.
3. P.S. Sapaty and P.M. Borst. An overview of the WAVE language and system for distributed processing of open networks. Technical report, University of Surrey, UK, 1994.
4. J.E. White. Telescript technology. Technical report, General Magic, Inc., Mountain View, CA 94040, 1994.
5. D.B. Lange and D.T. Chang. IBM Aglets workbench, programming mobile agents in Java. White paper, IBM Tokyo Research, Japan, September 1996.
6. R.S. Gray. Agent tcl: A transportable agent system. In *Proc. Fourth Int'l Conf. on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.
7. Ciovanna Di Marzo Serugendo, Murhimanya Muhugusa, Christian Tschudin, and Jurgen Harms. Survery of theories for mobile agents. Technical report No.106, University of Geneva, Switzerland, November 1996.
8. M. Fukuda, L.F. Bic, M. Dillencourt, and F. Merchant. Intra- and inter-object coordination with MESSENGERS. In *First Int'l Conf. on Coordination Models and Languages (COORDINATION'96)*, Apr. 1996.
9. L.F. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8), Aug. 1996.
10. A. Huth and C. Wissel. The simulation of the movement of fish schools. *Journal of Theoretical Biology*, 156:365–385, 1992.
11. M. Fukuda, K.L. Morse, L.F. Bic, M. Dillencourt, E. Lee, and D. Menzel. A novel approach to toxicology simulation based on autonomous objects. In *SCS Western MultiConference, Simulation in the Medical Sciences*, Jan 1996.
12. Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction*, chapter 3, Convex Hulls: Basic Algorithms, pages 95 – 149. Springer-Verlag, 1985.
13. D. H. Grit and J. R. McGraw. Programming divide and conquer on a MIMD machine. *Software Practical Experiment*, Vol.15(No.1):41 – 53, January 1985.
14. A. K. Dewdney. Computer recreations sharks and fish wage an ecological war on the toroidal planet wa-tor. *Scientific American*, pages 14 – 22, December 1984.
15. D.R. Jefferson. Virtual time. *ACM TOPLAS*, Vol.7(No.3):404 – 425, July 1985.
16. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Macine A User's Guide and Tutorial for Networked Parallel Computing.* The MIT Press, Cambridge, Massachusetts, 1994.

This article was processed using the LaTeX macro package with LLNCS style