Global Virtual Time Support for Individual-Based Simulations

Munehiro Fukuda, Lubomir F. Bic, and Michael B. Dillencourt

Department of Information and Computer Science
University of California, Irvine
e-mail: {mfukuda, bic, dillenco}@ics.uci.edu

Abstract

An individual-based simulation organizes the application as a collection of autonomous entities. The behavior of the system is simulated as interactions among such entities and hence the application development focuses primarily on describing the entities' behaviors. A natural implementation of individual-based modeling is to use autonomous objects, i.e., mobile entities navigating autonomously through their underlying computational network. This is further simplified if the programming and execution environment supports global virtual time management. MESSENGERS is a virtual-time computing environment for autonomous objects, and therefore the present work is the first experiment in applying the paradigm of autonomous objects to distributed individual-based simulations. We discuss MESSENGERS' advantages for application development from the software engineering point of view and compare its performance with conventional message-passing executions.

1 Introduction

Many traditional simulation programs are based on a top-down mathematical description of the entire simulation model. An alternative approach is *individual-based modeling*, in which the entire model is structured as a collection of individual simulation entities. With this approach, the behavior of the model is determined by the behavior of the individual entities and their interactions. These models are open-ended, because new types of entities can be incorporated incrementally into the model. It is relatively easy to apply this approach to multiprocessors or multicomputers by distributing simulation entities to each computing node. Typical examples of such individual-based simulations are interactive battle simulations, traffic modeling, particle-level simulations in physics, and various individual-based models in biology, ecology, and Artificial Life. These all investigate mechanisms that result in complex and highly coordinated behaviors of groups of individuals, such as schools of fish or ant colonies, with each individual following a simple set of rules.

Several approaches to implementing individual-based simulations have been proposed. Systems such as Timewarp OS [8] and SPEEDES [10] use a top-down approach: processes describing partial differential equations or stochastic models are statically bound to processors, and exchange their messages along available communication paths. With these conventional systems, users may face the following difficulties: (1) entities are allocated statically and not allowed to migrate, (2) new entities cannot easily be introduced, (3) entity interactions are fixed *a priori* or all entity information must be broadcast periodically, which limits scalability.

In the Swarm system [9], a simulation entity is an object described in an object-oriented language and called an *agent*. A simulation is organized as a collection of agents, called a *swarm*, with a schedule of events over those agents. Like many object-oriented systems, Swarm supports hierarchical structuring of objects, which in turn allows a swarm to be includeded as an agent in another swarm. Swarm libraries provide users with an efficient simulation environment including scheduling tools for entity interactions, a graphic interface, and control for simulation scenarios. However, Swarm does not directly aim at distributed simulation, so users need to directly address the distributed coordination of entity interactions.

In this paper we consider an alternative approach to individual-based simulation using *autonomous objects*. Autonomous objects are mobile entities, capable of navigating autonomously through their underlying computational network and launching tasks at the nodes they visit. They represent a natural means of implementing individual-based simulations: each individual entity can be represented as a distinct member of a collection of autonomous objects, moving through a logical network represented as a virtual space and interacting with one another.

We discuss distributed individual-based simulation in MESSENGERS, an autonomous-object-based system aimed at general-purpose distributed computation. MESSENGERS provides objects with navigational autonomy, various inter-objects communication schemes, and dynamic function linking, as described in [1]. We have implemented system-level support for virtual-time, which is the primary subject of this paper. MESSENGERS is the first system to incorporate a virtual-time computing environment for autonomous objects. The MESSENGERS virtual-time environment supports both conservative and optimistic simulations. We demonstrate that MESSENGERS allows users to simply and intuitively program individual models as autonomous objects within a virtual-time environment, and we give some performance results.

2 MESSENGERS Paradigm

MESSENGERS is a system that supports the development and use of distributed applications structured as collections of autonomous objects, called Messengers[1]. To allow Messengers to navigate autonomously through the network and carry out their tasks, the MESSENGERS system is implemented as a collection of interpreter daemons instantiated on all physical nodes participating in the distributed computation. A daemon's task is to continuously receive Messengers arriving from other daemons, interpret their behaviors, described as programs carried as part of each Messenger, and send them on to their next destinations as dictated by their behaviors.

The MESSENGERS system involves three levels of networks: (1) The *physical network*, a LAN or WAN, which constitutes the underlying computational nodes; (2) The *daemon network*, where each daemon is a UNIX process running a MESSENGERS language interpreter; and (3) The *logical network*, an application-specific computational network created on top of the daemon network. At system startup, a single logical node, named *INIT*, is created on every daemon node. Any Messenger may be injected (from the shell or by another Messenger) into any of the INIT nodes and it may start creating new logical nodes and links on the current or any other daemon.

Messenger programs, referred to as *Messenger scripts*, are written in a subset of C and are compiled into a form of byte code for more efficient transport and parsing [2]. 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. The user does not have to specify programs for all simulation entities at startup time. Instead, each entity can be described in a separate Messenger script and incrementally injected as an independent Messenger object into the system.

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

A Messenger script is a sequence of statements, which can be of one of the following types: (1) Computational statements, including all standard C assignment and control statements, involving arbitrary variables and constants; (2) Navigational statements (*create()*, *delete()*, and *hop()*) that endow the Messenger with mobility, permitting it to create and destroy logical nodes and/or links, and to move within the logical network; and (3) Function invocation statements that permit the dynamic loading and invocation of precompiled C functions to be executed in native mode; *exec()* spawns a separate concurrent process for the invoked function, while *func()* invokes the function as part of the current Messenger's behavior and returns its results to it. The details of the MESSENGERS language specification are not the focus of this paper; they are described in [6].

2.1 Virtual Time Support

The virtual time support in MESSENGERS consists of a system library of functions which Messengers can use to schedule their activities along a virtual time line. Messengers can suspend their execution until a triggering condition occurs. The triggering condition may be as simple as a time to wake up, or it may include a predicate applied to Messenger and node variables. Once a Messenger schedules an event by calling one of the event-scheduling functions explained below, it is suspended until the triggering condition occurs.

Virtual time is automatically maintained

[1]The individual autonomous objects are denoted by mixed case (Messengers), while the system as a whole is denoted by small capitals (MESSENGERS).

M_gvtstart(): Begin activation of GVT.

M_gvtstop(): End activation of GVT.

M_getlvt(): Return current LVT to the calling Messenger.

M_sched_time_dlt(t): Suspend interpretation of the calling Messenger for t virtual time units.

M_sched_time_abs(t): Suspend interpretation of the calling Messenger until daemon's LVT reaches the value t.

M_sched_time_node(n, predicate, m, t): Suspend interpretation of the calling messenger until either (1) the predicate associated with the node variable n and the messenger variable m is true, or (2) t virtual time units have elapsed.

Figure 1: The MESSENGERS virtual time functions

```
 (1) creator() {
 (2)   func(name = "M_gvtstart");
 (3)   while (1) {
 (4)     create(node = "N"; link = +);
 (5)     func(name = "M_sched_time_dlt"; in = 1);
 (6)   }
 (7) }

 (8) traverser() {
 (9)   func(name = "M_sched_time_dlt"; in = 1);
(10)   while (1) {
(11)     hop(link = +);
(12)     func(name = "M_sched_time_dlt"; in = 1);
(13)   }
(14) }
```

Figure 2: An example of Messenger scripts using the virtual-time environment

throughout the system in either a conservative or an optimistic manner, which can be selected in a user's system configuration file. In both cases, all daemons in the system determine the globally minimal future time of events, called Global Virtual Time (GVT), by periodically exchanging their local minimal future times of scheduled events, called Local Virtual Time (LVT). This guarantees consistent virtual-time-oriented interpretations of Messengers with respect to the advancing GVT. The virtual-time functions environment supported in MESSENGERS are listed in Figure 1. More details about their specification can be found in [5].

Figure 2 gives a simple illustration of the virtual time features. It contains two Messenger scripts, *creator()* and *traverser()*, which create a linear logical network (line 4) and traverse it (line 11), respectively. The *traverser()* Messenger starts from an *INIT* node one virtual time unit later than the *creator()* Messenger (line 9). Both suspend themselves at each node for one unit (line 5 and 12). Hence the *traverser()* will follow but never pass the *creator()*.

2.2 Daemon's Behavior

In this section, we will describe how the virtual-time environment is realized inside the MESSENGERS daemon. A more detailed explanation of the daemon's behavior is given in [4]. The underlying inter-daemon communication is based on a layered implementation, described in [5], that can use either TCP/IP or UDP as specified by the user in a configuration file.

Figure 3 illustrates the implementation of event-driven simulations in the MESSENGERS system. The MESSENGERS daemon receives a new Messenger injected from an external process through a Unix-supported message queue, which we call an *interface port*, and enqueues it into the *Ready Messenger Queue*. Once the daemon picks up the Messenger from this queue, it continues processing the Messenger script as long as its statements are computational. If the Messenger calls an event-scheduling function, the daemon enqueues the corresponding event into the *Function Event Queue*, which is implemented as a splay tree, and then pushes the Messenger into the *Suspended Messenger Queue*. When the LVT reaches the time scheduled in that event, the daemon picks it up from the *Function Event Queue*, and moves the corresponding Messenger into the *Ready Messenger Queue*. This may be repeated any number of times. Eventually, the Messenger will terminate or perform a navigational command. In the latter case, the daemon enqueues it into *Outgoing Messenger Queue* and thereafter sends it out through the socket interface during the next inter-daemon communication phase.

Both conservative and optimistic synchronization are supported. Under *conservative* synchronization, all daemons process only Messengers and events scheduled at GVT. After this processing, the daemons find out the minimal future time among events unprocessed in their own *Function Event Queue*, exchange it as their next possible LVT, and update GVT to the minimal value.

Optimistic synchronization permits all daemons to process Messengers and events by incrementing their own LVT regardless of the current GVT value. This allows daemons to work more independently than with conservative synchronization, but inconsistencies may arise when a Messenger is delivered to a daemon whose LVT is newer than the Messenger's time stamp. If this occurs, the daemon must turn back its LVT to the old time stamp and recover

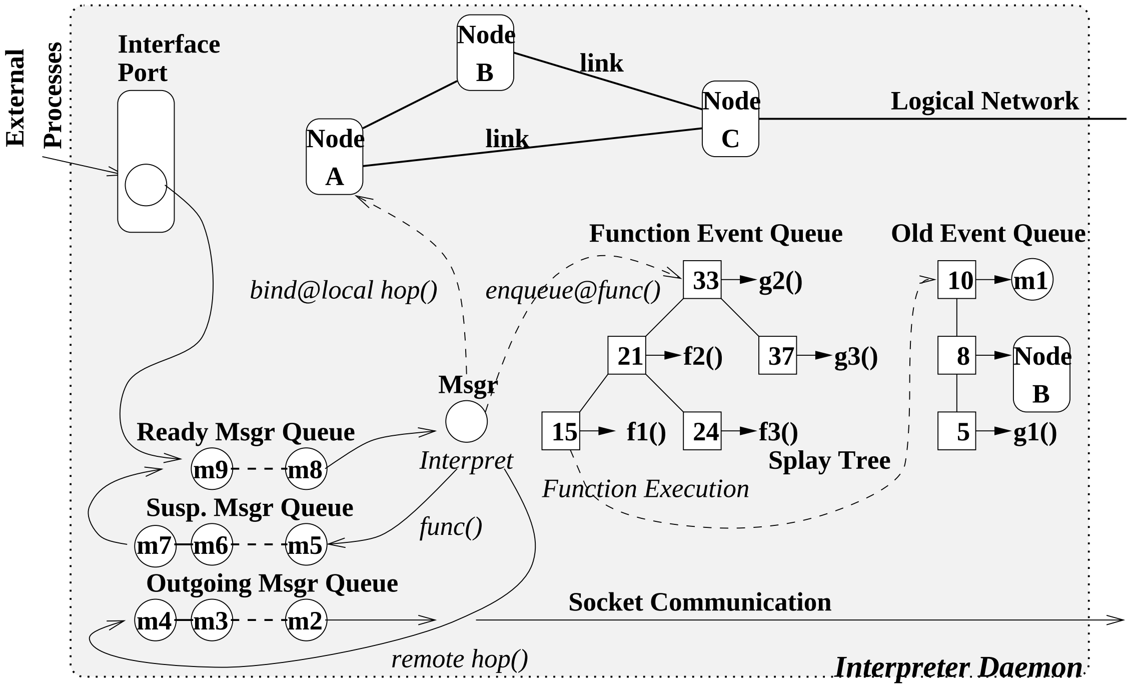


Figure 3: Implementation of event-driven computation in MESSENGERS

its old state. To be prepared for such a rollback operation, the daemon incrementally saves results of its interpretation and event processing. Prior to interpreting a new Messenger, invoking a precompiled function, processing an event, or modifying a logical network, the daemon saves the current state of the corresponding data structures, associates them with old events, and enqueues these events into the *Old Event Queue*. Upon a rollback, the daemon pops events out of this queue in a counter-chronological order, and incrementally returns to its appropriate earlier state. When the daemon detects that it has erroneously sent out Messengers, it sends the corresponding cancellation messages to the appropriate destinations. When the GVT is updated, each daemon performs a garbage collection by deallocating events in *Old Event Queue* which are older than the new GVT.

3 Simulation Modeling

Implementations of distributed individual-based simulations integrate two distinct activities: (1) the modeling of entity behavior, including their interactions in the simulated space, and (2) the coordination and timing of the interactions in the physical network. The main interest of application programmers lies in the first activity but the second activity is required for the results of the distributed computation to be correct. The necessary effort depends on the programming style required and the simulation features provided by each system. With MESSENGERS, application programmers are largely relieved from this coordination problem and thus may focus on the entity behavioral modeling.

3.1 Individual-Based Simulations

To illustrate the typical entity behavior in this class of simulations, we consider an ant evolution program [3], whose goal is finding an encoding of the behavior necessary for ants to bring back food to the nest efficiently. The encoding is represented as a bit string called a "genotype". The encoding is derived over multiple life generations, each of which consists of the following two stages: (1) simulating the food-searching behavior of ants with different genotypes, and (2) generating offsprings by the selection, crossover and mutation of the best group of parent genotypes. The first stage is an example of individual-based simulations. Over a 2-dimensional space, ants with different genotypes start from their nest, roam in search for food, and bring it back to the nest. While roaming over the space, they may leave pheromone trails which guide the other ants to the food source. An ant senses its surrounding information: the presence of food, the presence of a nest and the amount of pheromone. It may then take one of several actions—moving to one of the four neighboring cells, picking up a unit of food if found, dropping a unit of food if the ant has arrived at its nest, dropping some units of the pheromone—based on its encoding and the characteristics of the current cell. The decision scheme may also prohibit an ant to share the same cell with others. The ant's behavior can be modeled as a simple repetition of taking action and moving to a new position, with the actions being paced using virtual time:

```
while(1) {
   M_sched_time_dlt(delta);
   action();
   new_position();
}
```

This same basic structure can be applied to many individual-based simulations, although the number and granularity of entities vary for each type of application. For instance, particle simulations deal with millions of particles whose actions may be simple collisions, while battlefield simulations involve a much smaller number of entities, such as tanks and fighters, that need a large amount of computation to decide their next actions. We implemented this basic repetition of *action()* and *new_position()* in both MESSENGERS and PVM [7] to assess programmability and to find the number and granularity of entities necessary for MESSENGERS to achieve performance competitive with PVM. The PVM implementation, omitted here due to space limitations, is presented in detail in [5].

We use a two-dimensional grid for a simulation space and have entities walk with a direction-decision algorithm, which may choose a neighboring cell to move to randomly or according to a given goal. To obtain a deterministic simulation result, each entity is given a different ID. It avoids choosing a cell occupied by another entity at the current virtual time by choosing a different neighboring cell when it collides with another entity with a higher entity ID at the same cell.

3.2 Programming with MESSENGERS

Figure 4 shows the Messenger script *entity()*, which describes the behavior of the walking entity. This schedules its behavior every 10 virtual time units (line 3). It first checks its termination condition (lines 4-8). Then, it calls the function *action()* (line 10), followed by deciding on one of the four neighbors as its next destination through its direction-decision function named *walk()* (line 12), and hopping to the chosen direction (lines 15-18). If the cell is not occupied or the *entity* collides with another *entity* whose ID is smaller (line 19), the *entity* records its own ID as the new resident ID (line 22) and waits until virtual time is incremented (line 23). Otherwise, the *entity* must move back to the original cell (line 28) and repeat the same sequence with another destination. This repetition must also be performed by an *entity* that has been waiting for a new virtual time (line 23) but was awaken by someone with a higher ID (line 24), who has the right to occupy the cell. Once virtual time is incremented, it is guaranteed that all movements have been decided, and thus each *entity* erases its residency in the previous cell (line 32) and marks the new cell (line 34).

```
 (1) entity() {
 (2)   while(1) {
 (3)     func(name = "M_sched_time_dlt"; in = 10);
 (4)     func(name = "terminate_cond"; in = pos_x, pos_y;
 (5)          out = condition);
 (6)     if (condition) {
 (7)       old_resident = NONE;
 (8)       exit;
 (9)     }
(10)     func(name = "action");
(11)     for (trial = 0; trial < 4; trial++) {
(12)       func(name = "walk"; in = pos_x, pos_y, param;
(13)            out = dir, param);
(14)       old_node = $address;
(15)       if (dir == N)      hop(link = +"vertical");
(16)       else if (dir == E) hop(link = +"holizontal);
(17)       else if (dir == S) hop(link = -"vertical");
(18)       else               hop(link = -"holizontal");
(19)       if (old_resident == NONE &&
(20)            new_resident < my_id) {
(21)         alarm = new_resident;
(22)         new_resident = my_id;
(23)         func(name = "M_sched_node";
(24)              in = alarm, "==", my_id, 1);
(25)         if (alarm != my_id)
(26)           break;
(27)       }
(28)       hop(node = old_node);
(29)     }
(30)     new_node = $address;
(31)     hop(node = old_node);
(32)     old_resident = NONE;
(33)     hop(node = new_node);
(34)     old_resident = my_id;
(34)     new_resident = NONE;
(35)     alarm = NONE;
(36) } }
```

Figure 4: 2D walk simulation using MESSENGERS

The MESSENGERS implementation has several noteworthy features:

- The entity walk specification was translated directly and intuitively into the concrete script level of Figure 4, which was possible due to the small semantic gap between the abstract and the programming levels of the simulation models. The virtual time support provided by MESSENGERS was essential to program the coordination among the different entities effectively.

- The management of the simulation space was handled by a different Messenger (not shown in this paper) and thus clearly separated from the code describing the simulation entity.
- Under MESSENGERS it is possible to introduce another set of instances or arbitrary new entities of a different type in the simulation space at runtime, without recompiling or even halting the ongoing simulation. Thus the implementation is open-ended and arbitrarily extensible.

4 Performance Comparisons

Although the Messenger script itself is interpretive and thus slower than a native code execution when executed on a single machine, the MESSENGERS daemon allows the script to call precompiled C functions, and uses multiple computing nodes. Hence, the MESSENGERS may achieve better performance than a single processor execution depending on the granularity of the computation. In addition, the MESSENGERS program exhibits slower performance than a message-passing version in which static processes execute native program and communicate with one another using an optimized communication library. However, the performance slow-down is mitigated by MESSENGERS's flow control mechanism for socket communication and the system level implementation of virtual time synchronization. Therefore, we focus on the following two evaluations: (1) the entity granularity necessary to achieve speed-up over the corresponding single processor version and (2) the effect of optimistic synchronization. Our experiments were run on two different daemon networks, with five and ten workstations respectively. In either configuration, one of the workstations was dedicated to the GVT computation, and thus four or nine CPUs were actually working on the entity simulation. All workstations were Sun SPARC Station SS5's (32MB memory each) connected by a 10Mbps Ethernet in the same subnetwork. We briefly summarize the results here, they are described more extensively in [5].

The Messenger scripts and message-passing programs discussed in Section 3 are used for the evaluations. The message-passing version uses PVM. In addition, we also implement another version of

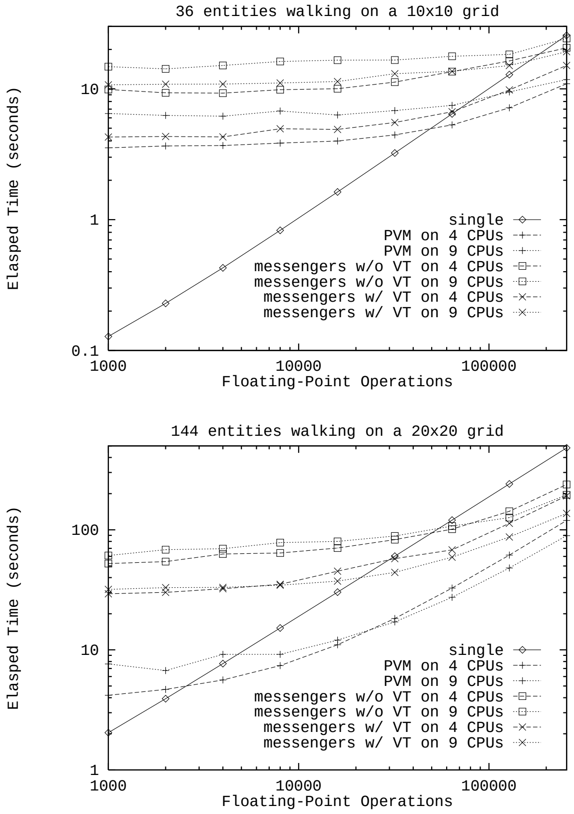


Figure 5: Comparison between PVM and MESSENGERS for 2D walk

Messenger scripts, which do not use the MESSENGERS virtual time environment but implement it at the script (i.e. application) level.

4.1 PVM versus MESSENGERS

We investigate two different sizes of the problem. One is the simulation of 36 entities walking over a 10×10 grid, and the other of 144 entities over a 20×20 grid (a four-time increase). The *action()* function defined as an entity's action has no contents other than dummy floating-point operations. For each simulation of the 10×10 and 20×20 grids, we measured its performance when changing the number of the floating-point operations in order to find a break-even point of computation granularity to perform better than a single CPU version. Figure 5 shows the performance of these two different grid sizes.

On the 10×10 grid simulation with five CPUs, MESSENGERS in the virtual time mode runs within only a 1.3 times slow-down as compared to PVM. With ten CPUs, the performance of both MESSENGERS and PVM is degraded below that of four

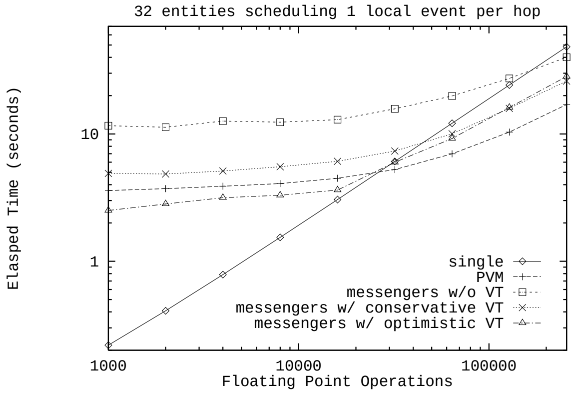
CPUs, because the problem size is small as compared to communication latency. On the 20×20 grid, with five CPUs MESSENGERS is approximately 2.5 times slower than PVM floating-point operations. Both MESSENGERS and PVM show their scalability with ten CPUs. In particular, MESSENGERS in virtual time mode reduces the breaking point slightly, shrinking the large performance difference between PVM and MESSENGERS.

4.2 Conservative versus Optimistic Synchronizations

We investigate under what conditions optimistic synchronization performs better than the conservative scheme for MESSENGERS. We focus on the simulation of 32 entities walking on a 10×10 grid only, since MESSENGERS shows its competitiveness with PVM for a smaller problem size. Indeed, better performance is not expected for a larger problem due to more context switches and function calls, which in turn incur more state-saving overhead in optimistic synchronization.

In general, the more events each physical node schedules locally, the better performance optimistic synchronization achieves over the conservative scheme. If Messengers schedule no local events and hop every 10 simulation cycles, there is no difference between these two schemes in terms of the number of messages required for GVT decision, while optimistic synchronization incurs state-saving and rollback overhead. Therefore, we evaluated the performance with one or two dummy events inserted between any two *hop* statements in each Messenger script.

Figure 6 shows the performance using five CPUs when one and two dummy events are inserted respectively. We again measured elapsed time while changing the number of floating-point operations included in the *action()* function. MESSENGERS takes advantage of optimistic synchronization the most efficiently with only one dummy event in the range of small computations. In particular, MESSENGERS using optimistic synchronization is even faster than PVM. However, such superiority is limited to the range where computation granularity is too small for both MESSENGERS and PVM to gain from parallelism due to communication overhead, (i.e., below 20,000 floating-point operations). Beyond this threshold, the performance of optimistic synchronization is gradually degraded and finally matches that of the conservative scheme at 100,000 floating-point operations. Optimistic synchronization didn't achieve better performance with ten CPUs than five CPUs.



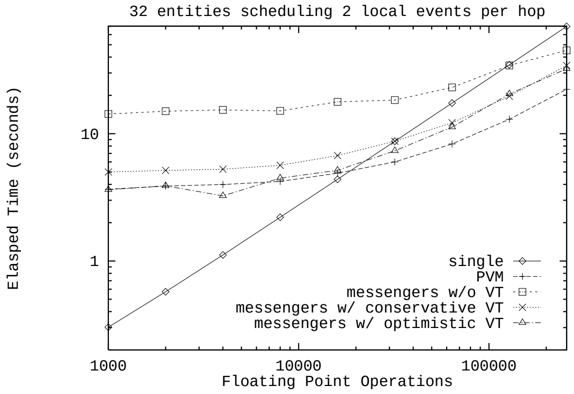


Figure 6: Comparison between conservative and optimistic synchronizations

In summary, MESSENGERS can achieve its performance close to PVM using optimistic synchronization but its truly effective range where it also surpasses the single CPU version is limited to between 20,000 and 100,000 floating-point operations. Such efficiency is obtained under the following conditions: (1) a relatively small problem size, (2) one or more events scheduled between *hop* statements, (3) a small multiprocessor configuration, and (4) an efficient logical-to-physical node mapping.

5 Conclusion

MESSENGERS is the first autonomous-object-based system incorporating a virtual time environment. We have demonstrated that the paradigm of autonomous objects in conjugation with virtual time eases individual-based simulation modeling. The daemons coordinate all interactions among Messenger objects along a virtual time line, and therefore users can focus on the design of their entity models. With MESSENGERS' navigational autonomy, models are programmed in Messenger scripts from the entities' view point.

We are presently addressing the performance so that MESSENGERS will be competitive with PVM in any range of computation granularity and network size. One such improvement is full compilation and native-mode execution of Messenger scripts. Rather than interpreting objects, the MESSENGERS daemons handle threads, which execute compiled code of Messenger scripts. Thus we can completely eliminate the overhead incurred by interpretations and function calls from the script layer. Another improvement is load balancing where overloaded daemons will be off-loaded by having logical nodes and Messengers residing there migrate to other lightly-loaded daemons. In view of the software engineering benefits to simulation modeling derived from the paradigm of autonomous objects and the fact that we are still improving MESSENGERS' performance, we feel that MESSENGERS is a promising tool for distributed individual-based simulations.

References

[1] L. F. Bic, M. Fukuda, and M. B. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, pages 55–61, August 1996.

[2] B. Bidyuk. MESSENGERS-C compiler manual. Report MSGR-06, University of California, Irvine, CA, 1996.

[3] R. J. Collins and D. R. Jefferson. AntFarm: Towards simulated evolution. In *Artificial Life II: Proceedings of the Workshop on Artificial Life*, pages 579–601, Santa Fe, NM, February 1990. Addison-Wesley.

[4] M. Fukuda. *MESSENGERS: A Distributed Computing System Based on Autonomous Objects.* PhD thesis, University of California, Irvine, CA 92697, June 1997.

[5] M. Fukuda, L. F. Bic, and M. B. Dillencourt. Distributed individual based simulation using autonomous objects. ICS Technical Report UCI-ICS 97-46, University of California, Irvine, CA, January 1998.

[6] M. Fukuda, L. F. Bic, M. B. Dillencourt, and F. Merchant. Distributed coordination with MESSENGERS. *Science of Computer Programming*, page to appear, in 1998.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jieng, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.

[8] D. Jefferson, B. Beckman, F. Wieland, et al. Distributed simulation and the Time Warp Operating System. Technical Report No.870042, UCLA, Computer Science Dept., August 1987.

[9] N. Minar, R. Burkhart, C. Langton, and M. Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. Technical report, Santa Fe Institute, Santa Fe, NM, June 1996.

[10] J. S Steinman. SPEEDES: A multiple-synchronization environment for parallel discrete-event simulation. *International Journal in Computer Simulation*, pages 251–286, 1992.