Distributed Individual-Based Simulation Using Autonomous Objects

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

Technical Report 97-46

January 21, 1998

Abstract

Individual-based simulation modeling structures an entire complex and open-ended application as a collection of individual autonomous entities. The behavior of the entire system is simulated as interactions among such entities and hence the application development focuses primarily on describing the entities behaviors. Therefore, this modeling scheme attracts many scientists dealing with complex models from the battlefield to particle-level simulations. A natural implementation of individual-based modeling is to use the philosophy of autonomous objects, i.e., mobile entities navigating autonomously through their underlying computational network. We have developed MESSENGERS, an autonomous-objects-based system aimed at a general-purpose distributed computing, and, especially, distributed simulations. It is the first system to provide 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. The environment supports both conservative and optimistic simulations. In this paper, we discuss MESSENGERS' advantages for application development from the software engineering point of view, and show conditions, such as problem size, entity granularity and scalability, necessary for MESSENGERS to exploit its parallelism and achieve competitiveness with conventional message-passing executions.

Contents

1 Introduction

Most traditional simulation programs are based on a top-down mathematical description of the entire simulation model. However, all models cannot be efficiently described using this approach, especially when they include large, complex, and/or open-ended properties. Another approach is "individual-based" modeling, which structures the entire model as a collection of individual simulation entities. With this approach, the behavior of a complex system results from interactions among those autonomous entities. These focus on their own local behavior, while the entire model achieves its open-ended growth by incrementally incorporating new types of entities. Furthermore, it is relatively easy to apply this approach to multiprocessors or multicomputers by distributing simulation entities to each computing node [LS95, MSC94].

Typical examples of such "individual-based" simulations are interactive battle simulations [Com94, Rog92], traffic modeling [Res94], particle-level simulations in physics [HE88], and various individual-based simulation models in biology or ecology [Mic96, Lan95]. Related to the latter is also the study of collective behavior in AI, which investigates the mechanisms that result in complex and highly coordinated behaviors of groups of individuals, such as a school of fish [Dew84] or an ant colony [CJ90], while each individual has only a very limited local knowledge of the "problem" and a set of simple rules to follow.

The main concern is how individual-based simulations should be implemented in actual computing systems. Most systems represented by Timewarp OS [JBW+87] and SPEEDES [Ste92] have assumed the top-down mathematical modeling where large processes describing partial differential equations or stochastic models are bound to processors in a static computational flow and exchange their messages along the 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 be introduced, (3) entity interactions are fixed *a priori* or all entity information must be broadcast periodically, which limits scalability.

The best-known system for individual-based simulations is Swarm [MBLA96] in which a simulation entity is an object described in an object oriented language and is called an *agent*. Thus, a simulation is organized as a collection of agents, named 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 efficient simulation environments: scheduling tools for entity interactions, the support for probes, graphic interface, and the control for simulation scenarios. However, Swarm does not directly aim at distributed simulation, and thus users need to be concerned with distributed coordination of entity interactions by themselves.

Recently, the philosophy of autonomous objects has received great popularity for network management and distributed processing. Autonomous objects are mobile entities, capable of navigating autonomously through their underlying computational network and launching tasks at the nodes they visit. Using autonomous objects as individuals is a natural extension to distributed individual-based simulations. In this approach, a collection of autonomous objects are interacting with one another as entities and moving over a logical network constructed as a virtual space. WAVE is one of the first autonomous-object-based systems, which aims at distributed simulations, especially DIS as one of its application domains [BBCS94]. It is a complete environment consisting of a specialized language to express arbitrary autonomous objects behaviors, and a run-time system of interpreters [SB94]. In spite of its goal, WAVE however has several disadvantages for simulations: (1) objects named *waves* with different roots cannot communicate with one another directly, (2) *waves* are interpretive objects and become heavy when they call precompiled functions, which must be invoked as independent processes, and (3) its distributed termination detection works only for *waves* with the same root and thus cannot be used for virtual time management.

We have developed MESSENGERS, an autonomous-object-based system aiming at general-purpose distributed computation [BFD96]. The system provides objects with navigational autonomy, various inter-objects communication schemes, and dynamic function linking. MESSENGERS is the first system to facilitate a virtual-time computing environment for autonomous objects. This paper presents the first experiment to apply the paradigm of autonomous objects to distributed individual-based simulations. Furthermore, the MESSENGERS virtual-time environment supports not only conservative but also optimistic simulations. In this paper, we demonstrate that MESSENGERS allows users to simply and intuitively program individual models as autonomous objects within a virtual-time environment, and shows several performance results: entity granularity and scalability suitable to distributed computing, comparison with a static message-passing model, and conditions under which optimistic synchronization performs better than a conservative scheme.

2 MESSENGERS Paradigm

2.1 Principles of Operations

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

[1] case (Messengers), while the system as a whole is denoted by small capitals (MESSENGERS).

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. The lowest level is the *physical network* (a LAN or WAN), which constitutes the underlying computational nodes. Superimposed on the physical layer is the *daemon network*, where each daemon is a UNIX process running a MESSENGERS language interpreter. The *logical network* is an application-specific computation 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 [Bid96]. 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. This gives a programming style much easier than the commonly used approaches to individual-based simulation modeling. In particular, the user does not have to include all models of simulation entities and space in the simulation program at a time. Rather, these models are described in a separate *Messenger script* and incrementally injected as an independent Messenger object into the system. While some Messengers are dynamically constructing or modifying the logical network corresponding to a simulation space, other Messengers representing individual entities may roam over the logical network.

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 enable the Messenger to perform arbitrary computations. They include all standard C assignment and control statements, involving arbitrary variables and constants; (2) Navigational statements distinguished as *create()*, *delete()*, and *hop()* endow the Messenger with mobility, permitting it to create and destroy logical nodes and/or links, and to move within the logical network; (3) Function invocation statements permit the dynamic loading and invocation of precompiled C functions to be executed in native mode. The *exec()* statement

spawns a separate concurrent process for the invoked function, while the *func()* statement 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 [FBDM98].

2.2 Virtual Time Support

All distributed features that enhance the basic language capabilities are made available through a system library of functions which Messengers invoke using the *func* statement. Through this system library, the MESSENGERS daemon provides a virtual-time environment in which Messengers can schedule their activities along a virtual time line.

Each scheduled event suspends the involving Messenger until the conditions associated with this event are satisfied. The triggering conditions may be simply future points in time to wake up the Messenger but may also 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, its further interpretation is suspended until virtual time reaches the time specified in the event or the system satisfies the specified predicate.

Virtual time is automatically maintained over 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 of the system periodically decide the globally minimal future time of events, called GVT, (i.e., Global Virtual Time), by exchanging their local minimal future time of scheduled events, called LVT, (i.e. Local Virtual Time). This guarantees consistent virtual-time-oriented interpretations of Messengers with respect to the advancing GVT.

The following lists the functions interfacing to the virtual-time environment supported in MESSENGERS.

M_gvtstart()

The MESSENGERS' virtual time environment is not activated until a Messenger invokes the *M_gvtstart()* function. The invocation may occur at any daemon. GVT initially starts from 0, and may be suspended using M_gvtstop() (see below). A subsequent *M_gvtstart()* reactivates the virtual time environment such that GVT continues with the last value saved at the time of deactivation. In other words, all daemons periodically exchange their LVTs to update GVT during the virtual time computation mode, while they simply maintain GVT in the inactive mode.

M_gvtstop()

The virtual time environment is deactivated once a Messenger invokes the *M_gvtstop()* function. The invocation may occur at any daemons. However, the system recognizes only one invocation whose virtual time is the earliest among possible multiple invocations, it waits until GVT reaches this deactivating time, and then disables the virtual time environment.

M_getlvt()

This function returns the current LVT to the calling Messenger.

M_sched_time_dlt(delta_time)

The daemon suspends the interpretation of a Messenger calling *M_sched_time_dlt()* for the number of virtual time units given in *delta_time.*

M_sched_time_abs(absolute_time)

The daemon suspends the interpretation of a Messenger calling *M_sched_time_abs()* until its LVT reaches an absolute value given in *absolute_time.* If the content of *absolute_time* is already smaller than the LVT, the daemon simply ignores the suspension of this Messenger and continues to interpret it.

M_sched_time_node(node_variable, predicate, messenger_variable, delta_time)

This event-scheduling function involves two conditions: One is a period of virtual time given in *delta_time* to suspend the calling Messenger. The other is a predicate associated with node and Messenger variables to suspend the Messenger until it is satisfied. The predicate may be only a simple logical operator. The daemon resumes the Messenger when either of these two conditions is satisfied.

Examples: Figure 1 shows two Messenger scripts, *creator()* and *traverser()*, which create a linear logical network and traverse it, respectively. The *traverser()* Messenger starts from an *INIT* node one virtual time unit later than the *creator()* Messenger (line 9), and both suspend themselves at each node for one unit (line 5 and 12). Therefore, the *traverser()* will follow but never pass the *creator()*, as shown in Figure 2.

2.3 Daemon's Behavior

In this section, we will concentrate on how the virtual-time environment is realized inside the MESSENGERS daemon. The overall explanation of the daemon's behavior is given in [Fuk97].

```
(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 1: An example of Messenger scripts using the virtual-time environment

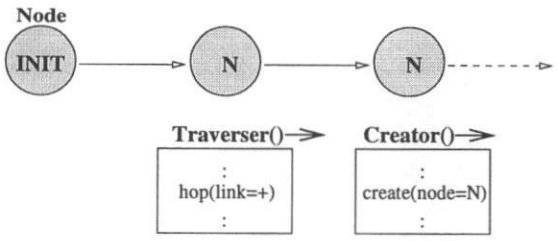

Figure 2: One Messenger follows another Messenger creating a network

There are two reasons why virtual time is realized at system level rather than user level:

1. **Better programmability:** The problem in distributed simulation is reduced into distributed termination detections, whose goal is guaranteeing that there are no more events to be processed at a given virtual time point. The added complexity in mobile autonomous objects is that the objects are roaming over the network while propagating and terminating themselves as well as scheduling future events. It is much easier for the system, rather than the ever, to detect active and in-transit autonomous objects and events, which in turn permits users to focus on simulation modeling.
2. **Better performance:** Implementing virtual time means repeating distributed termination detections, which requires periodical system-wide synchronizations. Needless to say, efficiency cannot be expected with an implementation of such synchronizations at the application layer. Especially for an optimistic simulation, the system-level implementation is much better suited, since it can easily save the history of old object states, recover these states, and annihilate erroneously propagated objects, since the system manages all objects.

To achieve better performance, the MESSENGERS daemon adopts several new implementation techniques. The rest of this section will concentrate on explaining them.

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 *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 *Function Event Queue*, which is implemented as a splay tree, and then pushes the Messenger into *Suspended Messenger Queue*. When the LVT reaches the time scheduled in that event, the daemon picks it up from *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.

Conservative synchronization has 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

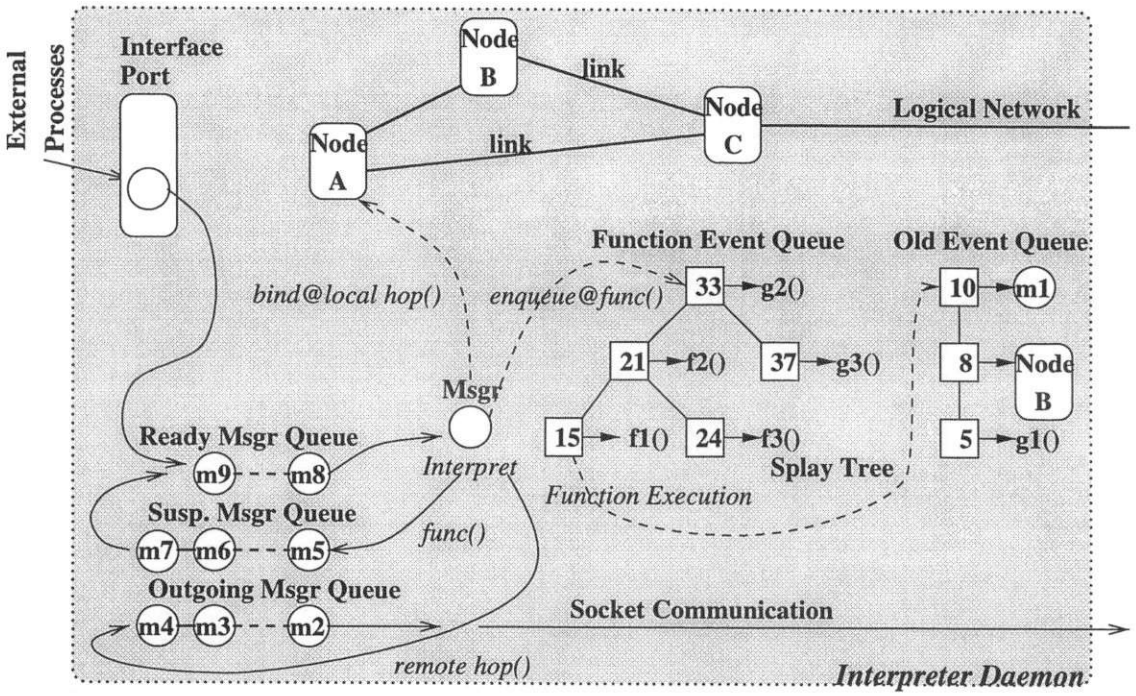

Figure 3: Implementation of event-driven computation in MESSENGERS

by incrementing their own LVT regardless of the current GVT value. While daemons are working more independently than with conservative synchronization, this may cause inconsistent cases of delivering a Messenger to a daemon whose LVT is newer than the Messenger's time stamp. In such cases, the daemon must turn back its LVT to the old time stamp and recover its old state. To be prepared for such a rollback operation, the daemon incrementally saves results of its interpretation and event processing. In 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 *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. Every periodical GVT decision, each daemon performs a garbage collection by deallocating events in *Old Event Queue* which are older than a new GVT.

The underlying inter-daemons communication uses either TCP/IP or UDP, which is specified in a user's system configuration file. Figure 4 describes the layered implementation of inter-daemons communication. The highest level is a user's script including navigational statements and event-scheduling functions. Under the script

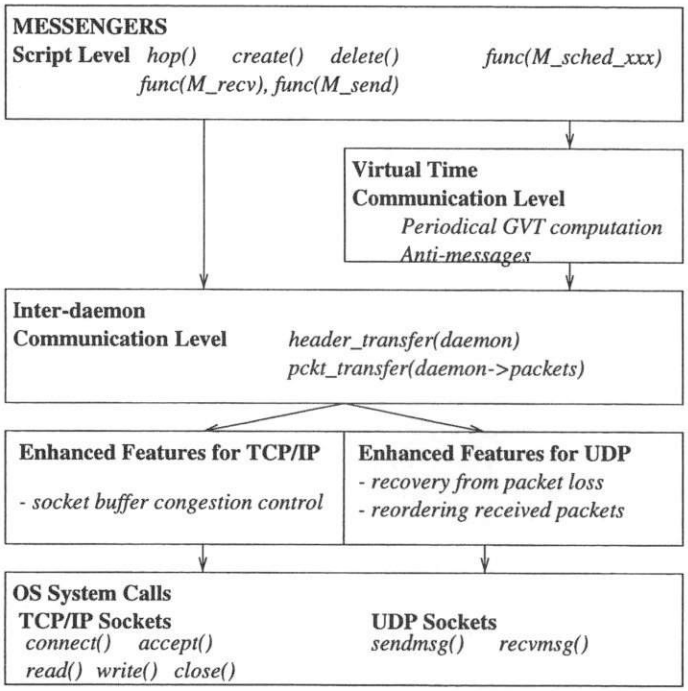

Figure 4: Implementation of inter-daemons communication

level is the virtual time communication that automatically creates messages for GVT decisions and Messenger cancellations. All Messengers and virtual-time-related messages are disassembled into headers and packets whose transmission is handled by two internal functions: *header_transfer()* and *packet_transfer()*. Depending on TCP/IP or UDP, these two functions support reliable communications. Under TCP/IP, they prevent socket buffers from congestion by having two threads write and read data through sockets in turn, while under UDP they work on recovery from packet loss and reordering received packet.

Section 3 and 4 will verify better programmability and performance resulting from the virtual time implementation at the system level discussed here.

3 Simulation Modeling

As described in Section 1, distributed individual-based simulations are realized by two design principles: (1) the modeling of entity behavior, including their interactions in the simulated space, and (2) the coordination and timing of their interactions in the physical network. The main interest of application programmers lies in the former. The latter is the result of distributed computation and 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. This section demonstrates MESSENGERS' superiority over a conventional message-passing system such as PVM, from this programmability point of view.

3.1 Individual-Based Simulations

To understand the typical entity behavior in this class of simulations, we first consider the ant evolution program, whose goal is finding the best genetic code for ants to bring back food to the nest efficiently. The ant genetic code is represented as a bit string called a "genotype". The best genetic code 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. Figure 5 illustrates this simulation. An ant senses its surrounding information: the presence of food, the presence of a nest and the amount of pheromone. It may thereafter take one of the following actions: moving to any of the four neighboring cells, picking up a unit of food if the ant has found it, dropping a unit of food if the ant has arrived at its nest, and dropping some units of the pheromone.

The decision to move to a neighboring cell depends on each ant's genotype. It may always select the one whose units of the pheromone is larger than any other neighboring cells, always move randomly, use the pheromone only when its amount exceeds a given threshold, or even stay at the current position. The decision scheme may also prohibit an ant to share the same cell with others. In any case, the ant's behavior can be modeled as a simple repetition of taking action and moving to a new position, which is being paced using virtual time:

```
for (;;) {
        M_sched_time_dlt(delta);
        action();
        new_position();
}
```

We expect that this basic structure can be applied to many individual-based simulations, while 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 DIS deals with a much smaller number of entities, such

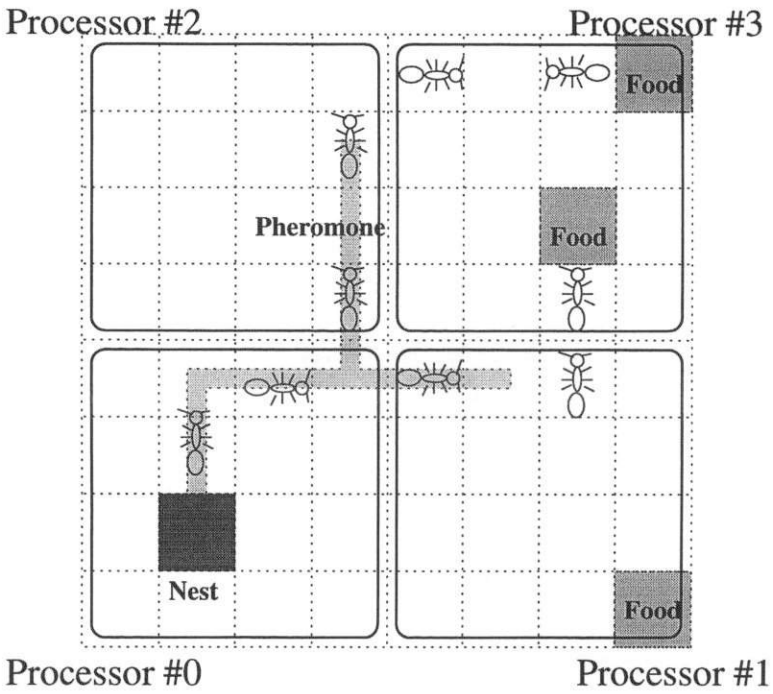

Figure 5: Ant evolution Program: an example of individual-based simulations

as tanks and fighters, but need a large amount of computation to decide their next actions. Thus, we implemented this basic repetition of *action()* and *new_position()* in both MESSENGERS and PVM for not only discussing the programmability but also finding the number and granularity of entities necessary for MESSENGERS to achieve performance competitive with PVM.

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, avoids choosing a cell occupied by another entity at the current virtual time, and recomputes a new neighboring cell when it collides with another entity with a higher entity ID at the same cell. Figure 6 illustrates this deterministic behavior.

Three entities with IDs 84, 97, and 103 residing at cells $[0,1]$, $[1,1]$, and $[2,0]$ are now deciding their new positions. They are given a direction-decision algorithm that makes all entities choose their left-hand cell first and, if occupied, keep selecting other cells in a counter-clock wise direction until an empty one is found. The entity #84 finds no more cell on its left side and thus moves downward to cell $[0,0]$. The entity #103 locates the entity #84 to its left and thus moves downward to cell $[1,0]$. The entity #97 first moves to its left-hand cell $[1,0]$. However, it detects a collision with the entity #103, and goes back to its original position due to its smaller ID. It then

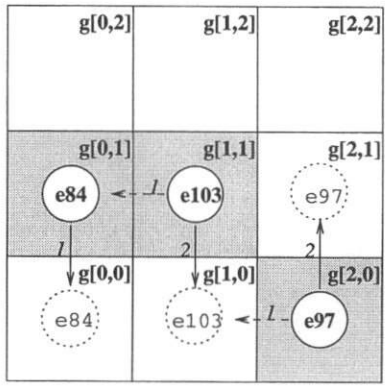

Figure 6: 2D walk simulation: a simplified form of individual-based simulation

chooses the cell $[2, 1]$ as its new position.

3.2 Programming with MESSENGERS

Figure 7 and 9 shows two Messenger scripts, *grid()* and *entity()*. The former constructs a two-dimensional grid, the latter describes a walking entity.

The *grid()* Messenger starts with the lower-left corner of the grid (lines 2 and 3), consecutively creates vertical links of one column (line 4 and 5), and comes back to the origin (line 6). The rest of the script then repeatedly constructs new columns by piling up rectangles on top of each other adjacent to the already constructed portion of the grid. Figure 8 illustrates this process. It starts by drawing the lowest vertical edge (line 9). The Messenger then creates each box by: hopping along the left vertical edge of the rectangle from the south-west up to the north-west corner (line 14), creating an upper horizontal edge to the north-east corner (line 16), creating a right vertical edge down to the south-east corner (line 18), and jumping back to the north-west corner (line 13). Logical-to-physical node mapping is performed implicitly by selecting the desired number of daemon nodes and their creation threshold to control the clustering of logical nodes (See Section 2.4.3 Node Mapping in [Fuk97].)

The *entity()* Messenger schedules its behavior every 10 virtual time units (line 3). It first checks its termination condition (lines 4-8). For instance, the termination may be satisfied when the *entity* reaches its final destination cell or when it is outside of the simulation space. Then, the *entity* calls the function *compute()* (line 9), followed by deciding on one of the four neighbors as its next destination through its direction-decision function named *walk()* (line 11), and hopping to the chosen direction (lines 13-16). If the cell is not occupied or the *entity* collides with another *entity* whose ID is smaller (line 17), the *entity* records its own ID as the new resident ID (line 19) and waits until virtual time is incremented (line 20). Otherwise, the *entity* must move back

```
(1) grid() {
(2)   create();
(3)   addr_0 = $address;
(4)   for (j = 0; j < size; j++)
(5)     create(link = +"vertical");
(6)   hop(node = addr_0);
(7)   for (i = 1; < size; i++) {
(8)     addr_nw = addr_0;
(9)     create(link = +"horizontal");
(10)    addr_se = $address;
(11)    addr_0 = $address;
(12)    while (j = 1; j < size; j++) {
(13)      hop(node = addr_nw);
(14)      hop(link = +"vertical");
(15)      addr_nw = $address;
(16)      create(link; link = +"horizontal");
(17)      addr_ne = $address;
(18)      create(node = addr_se; link = -"vertical");
(19)      addr_se = addr_ne;
(20)      j++;
(21)    }
(22)    hop(node = addr_0);
(23) } }
```

Figure 7: A Messenger script for 2D grid creation

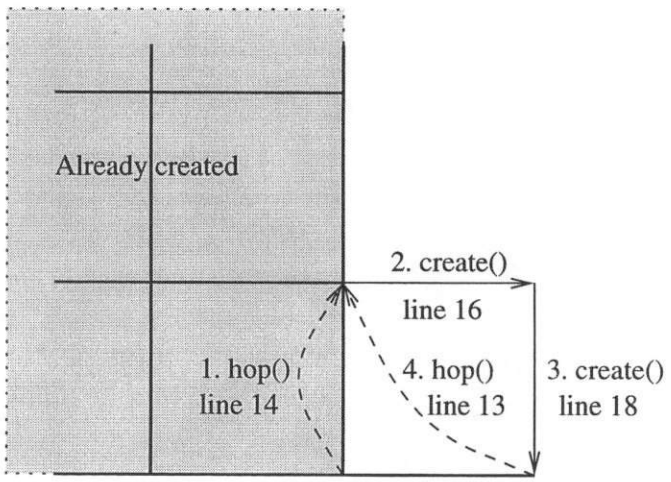

Figure 8: 2D grid creation using MESSENGERS

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

Figure 9: 2D walk simulation using MESSENGERS

to the original cell (line 24) 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 20) but was awaken by someone with a higher ID (line 21), 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 28) and marks the new cell (line 30).

We note that the entity walk specification described in Figure 6 was translated directly and intuitively into the concrete script level of Figure 9, which was possible due to the small semantic gap between the abstract and the programming levels of the simulation models. Once a *grid()* Messenger creates a 2D grid over a physical network, a set of *entity()* Messengers can start walking immediately on a distributed system without any user-level descriptions of low-level coordinations, (i.e. entity migrations, interactions, virtual time delivery, and network maintenance.) Simulation users can therefore focus on their model design, embodied in the functions "*compute*", "*walk*", and "*terminate_cond*".

3.3 Programming using Message-Passing

Figure 10 and 11 show the corresponding programs using message-passing. First, the master process spawns a given number of slave processes, each of which is in charge of simulating entities in its assigned simulation sub-space (lines 3-5). The master process also works as a virtual time manager which confirms that all slaves are finished with the current GVT (lines 10-11), and broadcasts a new GVT (lines 8-9).

The slave process first initializes its assigned space (lines 14-15), and then goes into a loop of entity simulations (lines 16-46). It investigates the termination condition for all entities residing in its assigned space (line 18-19), kills the ones which satisfy the condition (lines 20-21), and executes actions for all the alive entities through the *compute()* function (line 23). The slave then repeats a nested loop for entity exchanges with its neighbors which consists of two phases (lines 25-44). The first exchange phase (line 26-31) performs the following tasks: the computation of each entity's new logical position (line 26-27), the migration of entities whose new positions are outside of the assigned space (line 30), and the merger of exchanged and resident entities into a new list (line 31). The second exchange phase (line 32-44) performs the followings: the detection of entity collisions (line 34), the migration of entities that have been swept out due to collisions with those with a higher ID (line 42), and the merger of entities into a new list (line 43).

Each slave process knows who are its neighbors in advance, and thus maintains a list for entities sent to each neighbor. When an entity is out of the assigned space as a result of its movement to a new position or backing-off to the original cell, it is enqueued into the appropriate list by the *enqueue_migr()* function (line 47-52). Using this list, the *migration()* function performs the entity migration to each neighboring process (line 54-64).

Obviously, the message-passing code is much longer than the corresponding Messenger scripts, in spite of the fact that the *init_space()*, *enqueue_migr()* and *migration()* functions are still at an abstract level. The length of an actual PVM implementation is more than 900 lines of C code, while an MESSENGERS version consists of 200 lines of Messenger scripts and 230 lines of C programs. The message-passing version must take a master-slave form, coded as two different programs. Since the intelligence lies in each slave process rather than each entity, the slave must divide the entity walking algorithm into three *while* loops: (1) entity termination and computation, (2) entity walk and migration, and (3) entity back-off and migration, each of which must be performed in batch for all entities. Therefore, the conventional message-passing program supports no simple intuitive implementation of the simulation model. Furthermore, the low-level coordination of simulation entities is left completely up to the simulation model designer.

```
(1) master() {
(2)   for (i = 0; i < slaves; i++) {
(3)     spawn("slave", &(slave_id[i]));
(4)     send(slave_id[i], sub_region, entity_list, sim_time);
(5)     recv(&ack);
(6)   }
(7)   for (gvt = 0; gvt <= sim_time; t += 10) {
(8)     for (i = 0; i < slaves; i++)
(9)       send(tids[i], gvt);
(10)    for (i = 0; i < slaves; i++)
(11)      recv(&ack);
(12) } }

(13) slave() {
(14)   recv(&sub_region, &entity_list, &sim_time);
(15)   init_space(sub_region, entity_list, sim_time);
(16)   for(;;) {
(17)     recv(&gvt);
(18)     while (e = dequeue(entity_list, gvt)) != NULL) {
(19)       if (terminate_cond(e->pos_x, e->pos_y)) {
(20)         grid[e->pos_x][e->pos_y].old_resident = NONE;
(21)         free(e);
(22)       }
(23)       compute();
(24)     }
(25)     for (trial = 0; trial < 4; trial++) {
(26)       while (e = dequeue(entity_list, gvt)) != NULL) {
(27)         walk(e->pos_x, e->pos_y, e->param, &e->new_x, &e->new_y);
(28)         enqueue_migr(e, e->new_x, e->new_y);
(29)       }
(30)       migration();
(31)       merge(local_list, tmp_list);
(32)       while (entity = dequeue(entity_list, gvt)) != NULL) {
(33)         g = &grid[e->new_x][e->new_y];
(34)         if (g->old_resident== NULL && g->new_resident->my->id < e->my_id) {
(35)           enqueue_migr((nr = g->new_resident), nr->pos_x, nr->pos_y);
(36)           g->new_resident = e->my_id;
(37)           enqueue(tmp_list, e);
(38)           continue;
(39)         }
(40)         enqueue_migr_list(e, e->pos_x, e->pos_y);
(41)       }
(42)       migration();
(43)       merge(local_list, tmp_list);
(44)     }
(45)     if (gvt == sim_time) break;
(46) } }
```

Figure 10: 2D walk simulation using message-passing (line: 1 - 46)

```
(47) enqueue_migr(e, dist_x, dist_y) {
(48)   if (dist_y > max_y) enqueue(north_migr_list, e);
(49)   else if (dist_x > max_x) enqueue(east_migr_list, e);
(50)   else if (dist_y < min_y) enqueue(south_migr_list, e);
(51)   else if (dist_x < min_x) enqueue(west_migr_list, e);
(52)   else enqueue(tmp_list, e);
(53) }

(54) migration() {
(55)   send(north_neighbor, north_migr_list);
(57)   send(east_neighbor, east_migr_list);
(58)   send(south_neighbor, south_migr_list);
(59)   send(west_neighbor, west_migr_list);
(60)   recv(north_neighbor, north_migr_list);
(61)   recv(east_neighbor, east_migr_list);
(62)   recv(south_neighbor, south_migr_list);
(63)   recv(west_neighbor, west_migr_list);
(64) }
```

Figure 11: 2D walk simulation using message-passing (line: 47 - 61)

3.4 MESSENGERS versus Message-Passing

The preceding section demonstrated that MESSENGERS allows simulation users to focus on their model design much more than the message-passing paradigm from two points of view: (1) the understandability of simulation programs, and (2) the low-load coordination of entity interactions. In addition, there are several other fundamental differences between these two paradigms:

- Individual-based applications involve not only entity modeling but also their simulation space management. These two issues are separated clearly in the MESSENGERS implementation. The simulation space is created by an independent Messenger, such as *grid*, and may persist until the MESSENGERS system is shut down. Hence, simulation entity Messengers can be injected at any time during the simulation. In contrast, the message-passing version where a slave process runs at each physical node interwines the simulation space management and the individual entity behavior.

- The message-passing implementation is fixed in terms of its functionality. Any modification or extension would require the program to be modified, recompiled, and redistributed to the physical nodes. The MESSENGERS implementation, on the other hand, is open-ended and thus arbitrarily extensible. Specifically, 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. Hence, the MESSENGERS paradigm facilitates interactive and incremental simulation modeling.

- With the message-passing implementation, each slave node program must be told who its neighbors are, (e.g. IP names in PVM). This hard-wired programming requires recompilation and explicit supply of the neighboring information whenever the physical topology changes. In contrast, Messenger scripts are not affected by such a node mapping problem. Logical nodes may be distributed implicitly as described with the *grid* code.

4 Performance Comparisons

The Messenger script itself is interpretive and thus slower than a native code execution when executed on a single machine. However, 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 granularity of the computation. Similarly, it displays slower performance than the message-passing version where static processes execute native program and communicate with one another using an optimized communication library. Such performance slow-down will however be 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.

Figure 12 shows the topology of a daemon network and the mapping of a 2D grid over the network when we use five and ten workstations respectively. One of them is always dedicated to a new GVT computation, and thus four or nine CPUs are actually working for the entity simulation. All workstations are Sun SPARC Station SS5's (32MB memory each) connected by a 10Mbps Ethernet in the same sub network, where no other irrelevant user programs are running.

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 Messenger scripts, which do not use the MESSENGERS virtual time environment but implement it at the script (i.e. application) level. Initially, a set of simulation entities are placed together in the center portion of a 2D grid, each occupying its own cell. The *walk()* function used for each entity to decide the next position returns a random direction, and thus we simulate a random propagation of entities walking from the center.

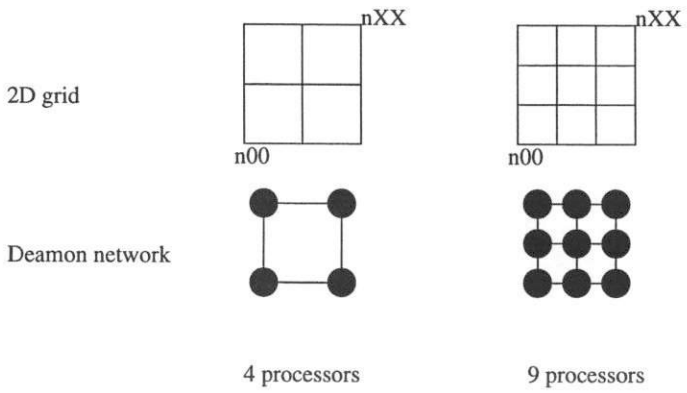

Figure 12: Logical-to-daemon network mapping for 2D grid

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 *compute()* 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 13 shows the performance of these two different grid sizes.

10×10 gird simulation:
With five CPUs, MESSENGERS' computation granularity in the non-virtual time mode needs more than 160,000 floating-point operations to overcome the single CPU version. This is reduced to 64,000 operations in the virtual time mode. On the other hand, the PVM version achieves speed-up over the single CPU version around 50,000 operations. Therefore, 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 CPUs. This is because the problem size is small as compared to communication latency.

20×20 grid simulation:
With five CPUs, MESSENGERS' computation granularity in both non-virtual time and virtual time modes becomes smaller, (i.e. 50,000 and 30,000 floating-point operations to become competitive with the single CPU version). However, PVM needs only 2,500 floating-point operations. MESSENGERS is approximately 2.5 times slower than PVM at 30,000 floating-point operations and thus looses its competitiveness. Both MESSENGERS and PVM show their scalability with ten CPUs. In particular, MESSENGERS in virtual time mode reduces the breaking point from 30,000 to 20,000

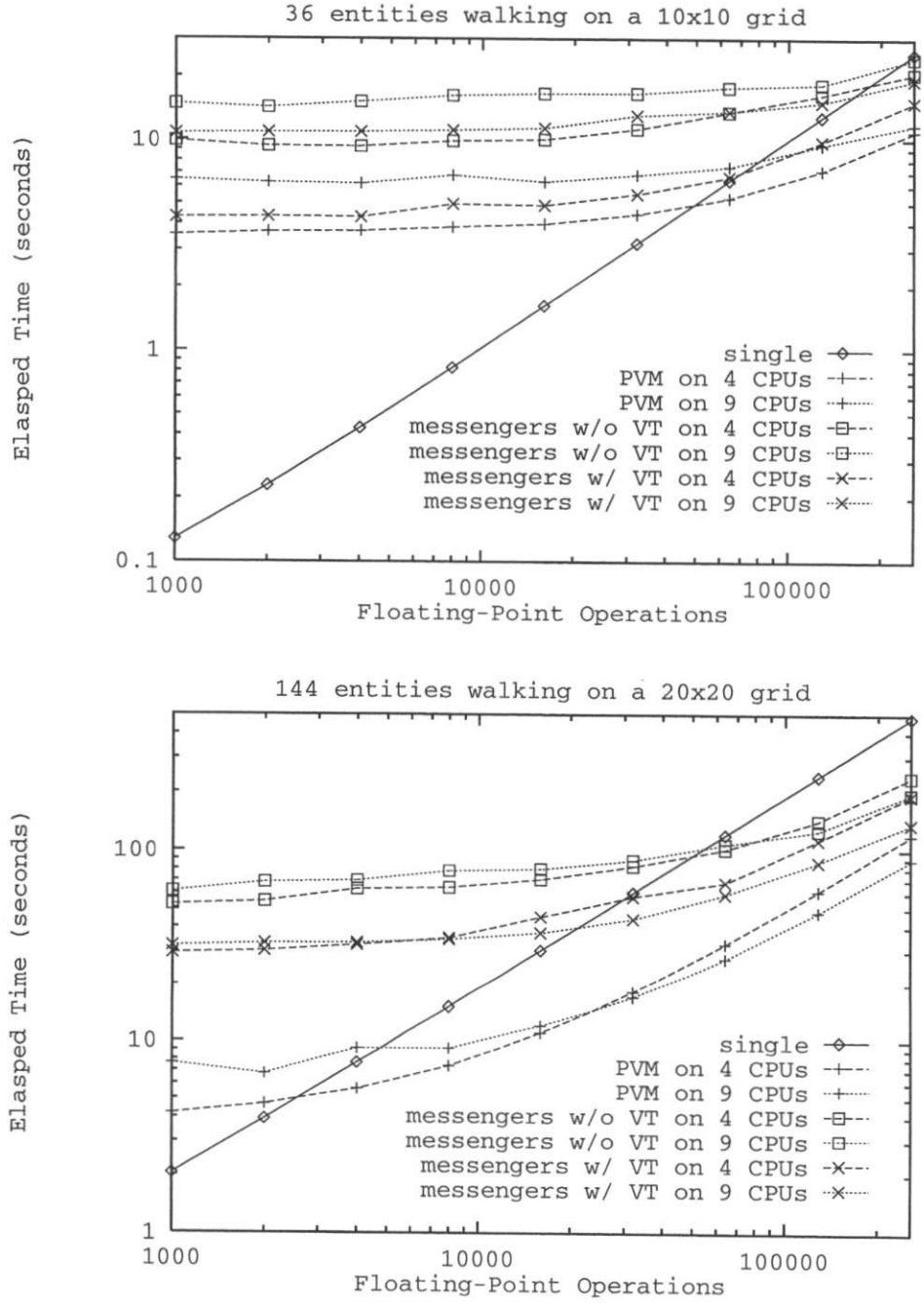

Figure 13: Comparison between PVM and MESSENGERS for 2D random walk

floating-point operations and slightly shrinks the large performance distance between PVM and MESSENGERS.

In summary, MESSENGERS is competitive with PVM for a smaller problem size when it uses the virtual time mode. To minimize an entity granularity and take advantage of CPU scalability, the problem size must be enlarged, which however lowers PVM's break-even point to surpass the single CPU version much more than MESSENGERS and thus weakens its competitiveness. This drawback is mainly caused by the context switches and function calls occurring in MESSENGERS, which increase with a larger number of simulation entities.

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. For instance, with two dummy events per hop, a Messenger schedules itself to hop at virtual time T and at time, $T+10$ and $T+20$ only calls the *compute()* function. It then repeats these three operations.

Figure 14 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 *compute()* 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 in 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

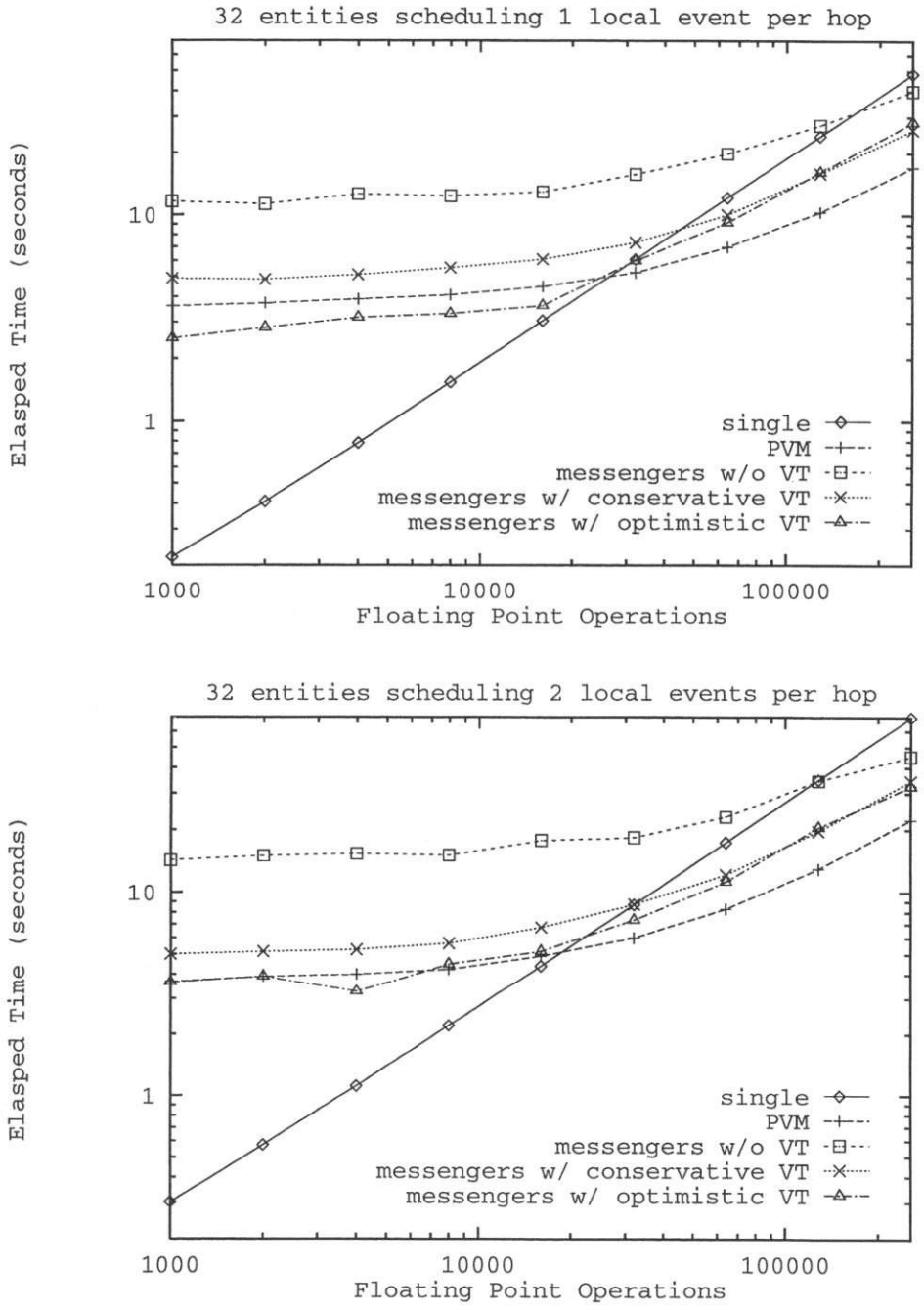

Figure 14: Comparison between conservative and optimistic synchronizations

degraded and finally matches that of the conservative scheme at 100,000 floating-point operations. The insertion of two dummy events between *hop*s does not change the efficiency of optimistic synchronization. The smaller the computation, the better the performance.

Optimistic synchronization didn't achieve better performance with ten CPUs than five CPUs. One of the reasons is that the master CPU needs to communicate with more slave CPUs for each GVT decision. However, the main reason lies in logical-to-physical node mapping. As shown in Figure 12, one CPU is surrounded by four neighbors, so that it receives more cancellation messages than other CPUs, causes more rollbacks, and thus becomes a bottleneck.

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.

Our performance evaluation showed the following three results: (1) the system-level implementation of virtual time performs two-times better than the user-level implementation, (2) the performance of conservative synchronization is competitive with PVM for small scale simulation domains where tens of entities are moving over four CPUs, and (3) the performance can be further improved by optimistic synchronization but its scalability is as small as that of the conservative scheme.

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

[BBCS94] L. Bic, P.M. Borst, M. Corbin, and P.S. Sapaty. The WAVE control protocol for distributed interactive simulation. In *Proc. 11th DIS Workshop on Standars for the Interoperability of Distributed Simulations*, pages 519–533, Orland, FL, September 1994. Institute for Simulation and Training.

[BFD96] Lubomir F. Bic, Munehiro Fukuda, and Michael B. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, pages 55–61, August 1996.

[Bid96] B. Bidyuk. MESSENGERS-C compiler manual. Report MSGR-06, University of California, Irvine, CA, 1996.
`http://www.ics.uci.edu/~bic/messengers`.

[CJ90] Robert J. Collins and David 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.

[Com94] DIS Steering Committee. The DIS vision: A map to the future of distributed simulation. *Institute for Simulation and Training*, 1994.

[Dew84] 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.

[FBDM98] Munehiro Fukuda, Lubomir F. Bic, Michael B. Dillencourt, and Fehmina Merchant. Distributed coordination with MESSENGERS. *Science of Computer Programming*, page to appear, in 1998.

[Fuk97] Munehiro Fukuda. *MESSENGERS: A Distributed Computing System Based on Autonomous Objects.* PhD thesis, University of California, Irvine, CA 92697, June 1997.

[HE88] R.W. Hockney and J.W. Eastwood. *Computer Simulations using Particles.* IOP Publishing Ltd, Bristol, Great Britain, 1988.

[JBW+87] David Jefferson, Brian Beckman, Fred Wieland, et al. Distributed simulation and the Time Warp Operating System. Technical Report No.870042, UCLA, Computer Science Dept., August 1987.

[Lan95] Christopher G. Langton. *Artificial Life : An Overview (Complex Adaptive Systems)*. MIT Press, Cambridge, MA, August 1995.

[LS95] Helmut Lorek and Michael Sonnenschein. Using parallel computers to simulate individual-orieted models in ecology: A case study. In *Proc. ESM'95 European Simulation Multiconference: Modelling and Simulation*, pages 526–531, Prag, 5-7 June 1995. SCS International.
`http://offis.offis.uni-oldenburg.de/projekte/ecotools/`.

[MBLA96] Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. Technical report, Santa Fe Institute, Santa Fe, NM, June 1996.
`http://www.santafe.edu/projects/swarm/`.

[Mic96] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs, 3rd Edition.* Springer-Verlag, 1996.

[MSC94] William Maniatty, Boleslaw Szymansk, and Tom Caraco. Implementation and performance of parallel ecological simulations. In *Proc. IFIP WG10.3 Working Conference on Applications in Parallel and Distributed Computing*, volume vol.A-44, pages 93–102, Caracas, Venezuela, April 1994. Elseveir Science Publishers.
`http://www.cs.rpi.edu/research/tempest/`.

[Res94] M. Resnick. Changing the centralized mind. *Technology Review*, pages 30–40, July 1994.

[Rog92] Ralph V. Rogers. Implementing system simulation of C3 systems using autonomous objects. In *Proc. IEEE/AIAA 11th Digital Avionics Systems Conference*, pages 275–280, Seatle, WA, 5-8 October 1992. IEEE.

[SB94] 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.
`http://ww.ira.uka.de/132/wave/wave.html`.

[Ste92] Jeff S. Steinman. SPEEDES: A multiple-synchronization environment for parallel discrete-event simulation. *International Journal in Computer Simulation*, pages 251–286, 1992.