MESSENGERS: A Distributed Computing Environment for Autonomous Objects

Munehiro Fukuda, Łubomir 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 96-20

August 5, 1996

Abstract

MESSENGERS is a distributed system based on the principles of autonomous objects. It facilitates distributed parallel computing as cooperative work among autonomous objects called Messengers, which carry their own behavior in the form of a program. Messengers exhibit two abilities: navigational autonomy and dynamic composition. They create and navigate through a modifiable computational network, and coordinate dynamic function invocations as they visit new nodes. Hence, problems including nondeterminism can be solved in parallel by Messengers. This flexibility is realized by having an interpreter daemon running at each physical node. The MESSENGERS interpreter daemon is distinct from that of other autonomous-object-based systems in terms of its support for efficient parallel processing. It functions as a micro-kernel for autonomous objects rather than a simple interpreter. It supports inter-Messengers communication, synchronization, and virtual-time-based function scheduling. In this paper, we describe the architecture of MESSENGERS, focusing specifically on performance-oriented features, present its performance evaluation, and discuss optimal granularity of each object.

Contents

1 Introduction

For many years, most distributed systems have been based on the computing paradigm where each machine runs a program embedding all computing "intelligence" and exchanging passive computed results as messages. Since a program running at each machine is precompiled and calls library functions optimally coded for inter-process communications, computation-intensive distributed applications can perform with ultimate efficiency obtained from underlying hardware and the operating system. On the other hand, such a computing paradigm does not provide the support to deal with non-deterministic problems which, for example, require network reconfigurations, non-deterministic communication, and dynamic process coordination. Therefore, each application needs to include its own anticipated dynamic features *a priori* or inevitably limit its flexibility.

Recently, autonomous objects have begun to be recognized as a new computing paradigm for distributed systems. They carry the "intelligence" or control of applications over a network. In other words, autonomous objects decide their behaviors and network navigation dynamically as they propagate through the network. This is achieved by having each machine running an interpreter daemon which exchanges autonomous objects with other daemons, interprets each object as a program, and carries out the tasks requested by each object. The more high-level dynamic features the interpreter daemon provides, the more flexible decisions can be made by autonomous objects at run time. For example. if the interpreter keeps track of all its neighboring nodes, an autonomous object can propagate itself to all neighbors of the current network node without having to enumerate all the nodes. Instead it uses high-level operations, supported by the interpreter, which replicates the object as necessary and sends a copy to all neighbors. At the same time, the interpreter should be efficient enough to realize fast context switching and task execution, since multiple autonomous objects must be typically exchanged over the system in order to maintain a certain level of the parallelism. Hence, a good trade-off between flexibility and efficiency to obtain high performance for various applications is of a great interest.

Existing systems based on autonomous objects include Telescript [Whi94], WAVE [SB94], and HTTP-base mobile agents [LDD95]. Since these systems are intended primarily for information retrieval, electric commerce, or interactive transactions, their interpreters put their greatest emphasis on network resource utilization and security rather than distributed parallel computing. None of them has reported any study of trade-offs between flexibility and efficiency as discussed above, nor have they published any performance evaluations.

In this paper we describe MESSENGERS, an autonomous-object-based system developed at the University of California, Irvine, which aims at distributed parallel

computing. An autonomous object called a Messenger in our system, carries a complete program (referred to as its script), together with its current status information, including a program counter and local variables. This defines the Messenger's behavior. The program is written in a subset of C language, which we will refer to as MESSENGERS-C. This includes two kinds of predefined functions, which allow network navigation and task coordination. At each machine, a Messenger interpreter daemon exchanges Messengers with other daemons and interprets the scripts of visiting Messengers. The primary emphasis of MESSENGERS implementation is on performance-oriented features provided by the interpreter daemon while keeping the paradigm's flexibility. This paper first presents its most important such features, then describes the system architecture realizing these features, and finally examines the performance and the appropriate computation granularity of various distributed applications.

2 MESSENGERS Paradigm

2.1 Distinctive Features

MESSENGERS consists of interpreter daemons, each of which runs on a different machine and communicates with the other daemons through an underlying common network protocol (Unix sockets in our present implementation), similar to other autonomous-object-based systems. These daemons realize a network-transparent environment where any logical computational network can be constructed regardless of the actual physical network. A Messenger has its own identity as an autonomous object and can decide at run time where it wishes to navigate next and what tasks it is to perform there. The behavior of each Messenger is described by its script, which it carries along as it moves through the computational network. According to a classification defined in [BFD96], MESSENGERS is superior to others in terms of its combined navigation and coordination capabilities:

- *Navigational autonomy*
  This is the ability of each object to navigate through the network according to its own behavioral script. A Messenger's navigational autonomy is provided by built-in navigational statements that allow the Messenger to go to specific nodes in the network, follow specific links emanating from the node in which it currently resides, or cloning itself to pursue independent paths. The navigation may take advantage of the current network topology or state of the computation. For example, a Messenger may be propagated to all neighboring nodes, broadcast to all existing nodes, or follow all links which have the same link weight. Logical computational networks are dynamically constructed by Messengers using

navigational statements that create and delete logical nodes at run time and do not necessarily correspond to the underlying physical network. As a result, Messenger scripts for network navigation and construction are independent from and reusable for any physical network structure.

- *Dynamic composition*
  This is the ability of each object to coordinate the invocation of functions precompiled into the machine's native mode. A Messenger's dynamic composition is realized by built-in task-coordinating statements that permit the Messenger to invoke and control the execution of unrestricted precompiled C functions at each visited node. A Messenger can choose three styles of function coordinations: (1) invoking a node-resident precompiled program as an independent child process of the current interpreter daemon, (2) linking and running a node resident precompiled program as part of the daemon process dynamically, or (3) carrying as part of its script a program image in some form, (i.e. as source code, intermediate code, or native code) and invoking it at a visited node. Hence, a Messenger orchestrates the execution of a group of native-mode functions dynamically. The Messenger's script itself is interpretive, and thus slow, however it offers the flexibility of a dynamically changeable environment for open-ended applications, where predicting all possible scenarios *a priori* may not be possible.

2.2 Execution Model

The MESSENGERS system is implemented as a daemon running a language interpreter at each physical node (Sun Workstation in our present implementation). The system distinguishes the following three layers of networks:

1. *physical network:* This is an existing electrical hardware link, (i.e. LAN or WAN) connecting workstations.
2. *daemon network:* This network is created onto the physical network. It includes all participating workstations as its nodes. Hence, the nodes are the subset of the underlying physical network nodes. Daemon network links can be made regardless to the underlying physical network. They are used for automatic distribution of logical network nodes which Messengers dynamically create onto this daemon network. Their explanation are given below.
3. *logical network:* This is an application-specific computation network which Messengers create onto the underlying daemon network. Multiple logical network nodes can be created onto the same daemon network nodes, thus assigned onto the same physical node.

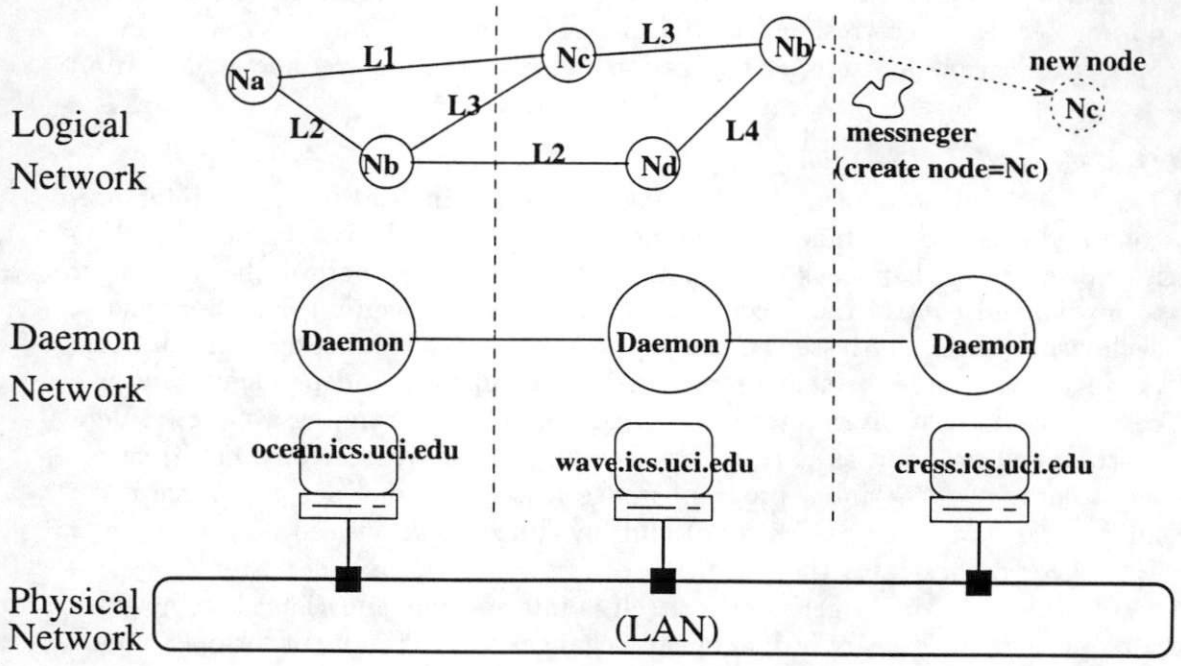

Figure 1: Network Layer

Figure 1 shows how each layer is mapped onto the next lower layer. The interpreter daemon exchanges Messengers with other daemons, and multiplexes logical nodes mapped to its physical node. When a Messenger creates a new logical node, it can not only choose a specific physical node but also let the system map this logical node to a certain physical node automatically. In the latter case, such a new logical node will be assigned onto the current physical node or one of physical nodes reached from the current node along its daemon links.

For each new Messenger, the interpreter continues processing its statements until it encounters one that is navigational or task-coordinating. At a navigational statement, the interpreter passes the Messenger on to the appropriate destinations node(s) in the logical network. If this is within the same physical node, the Messenger is simply moved to the appropriate queue, where it awaits its turn as the interpreter is being multiplexed between the different logical nodes. If the destination is in a different physical node, the Messenger is sent there using Unix sockets. At a task-coordinating statement, the interpreter executes the task as a function. A context switch to another ready Messenger will occur when the interpreter encounters a task-coordinating statement and before it actually processes this statement. Hence, the interpretation of Messenger is serialized between any two navigational or task-coordinating statements.

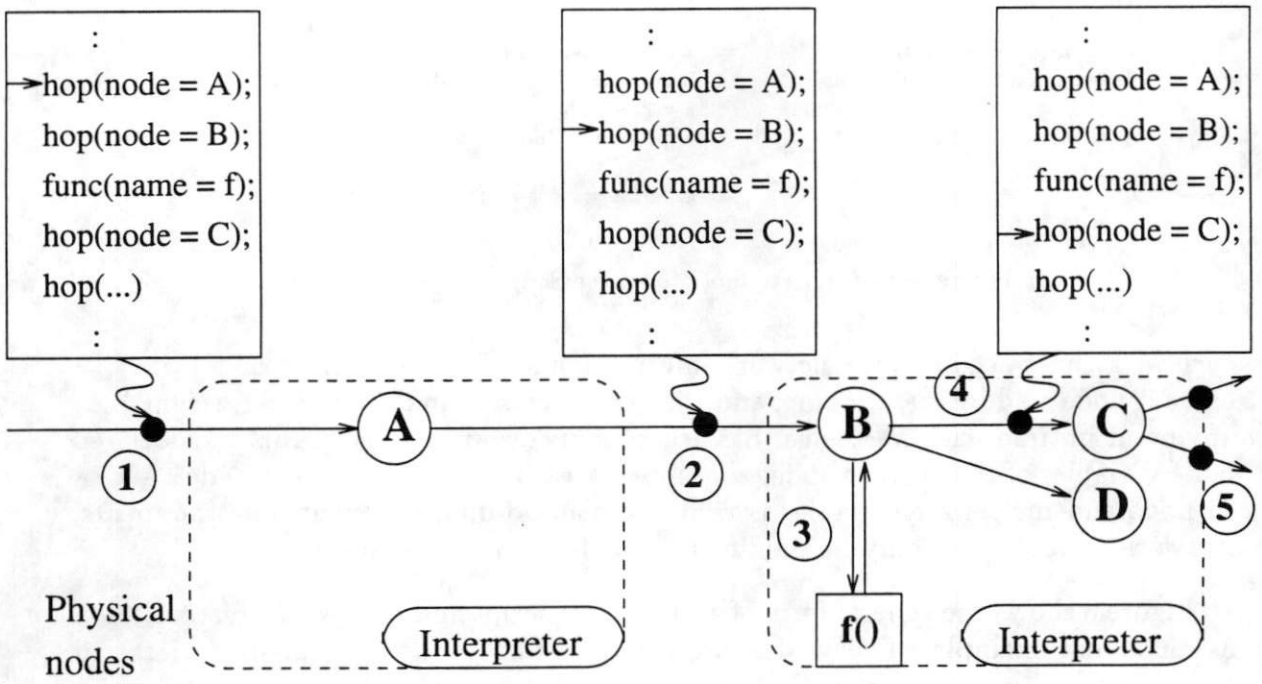

Figure 2: MESSENGERS Execution Model

Figure 2 illustrates these basic principles graphically. It shows two physical network nodes, each running the Messenger interpreter as one of its applications. A logical network of 4 nodes (named A through D) is mapped onto this architecture as shown. A Messenger is received by node A (marked as step 1 in the figure). Its current statement is navigational (*hop(node = B)*), which causes the Messenger to be forwarded to node B (step 2). The next statement, *func(name = f)*, causes the invocation of the node-resident function f() (step 3). Upon returning from the function call, the interpreter carries out the next statement, *hop(node = C)*, which causes a replica of the Messenger to be sent to the neighbors incident to the current node B whose name is C (step 4). Only node C receives this Messengers in this case and thus continues interpreting the next statement, denoted by *hop(...)* in the figure (step 5).

2.3 Language Specification

MESSENGERS-C distinguishes three types of variables, referred to as *messenger, node,* and *network* variables. Messengers variables are private to and carried by each Messenger as it propagates through the logical computational network. Node variables are resident in nodes and shared by all Messengers currently running on the same logical node. Network variables are predefined at each logical node and give

General Form:		*Example:*
	Node variable declaration;	#node "node_variables"
	Messenger variable definition;	int a, b = 1, c = 2; double x;
	S_1;	a = b + c;
	S_2;	hop(node = "node"; link = "link_X");
	...	...
⟶	S_i;	exec(file = "echo"; in = $node);
	...	...

Figure 3: General Form of Messenger Script and an Example

each Messenger access to the network information local to the current node, (i.e. the current node's address and name, and the last traversed link's name and weight.) At any point in time, each Messenger has access to its own messenger variables, declared node variables of the current node, and network variables of the current node. Access to node and messenger variables is achieved using ordinary assignment statements. Network variables can only be modified through navigational statements.

Figure 3 shows the general form of a Messenger script and its possible instantiation as an actual example. Every Messenger script starts with a variables declaration that has the same style as in C. The declaration allows all standard C data types, excluding pointer, union, and unsigned variables. First, node variables are declared in a separate file specified after the keyword *#node*, (i.e. "node_variables" is a file name in the example). Their declaration only provides the necessary mapping information between the Messenger code and the node-resident variables. The actual allocation and content initialization is done by the first Messenger to arrive at a node. Following the node variable declaration, variables local to and carried by the Messenger, (i.e. *a, b, c,* and *x* in this example) are declared as messenger variables. Their initial contents may be also defined at this point. Network variables, (e.g. $node in the example) are referred to without explicit declaration.

The remainder of a Messenger script consists of statements, denoted by S_i. The arrow is used to indicate the current statement, (i.e. the one to be interpreted next). This corresponds to a program counter in a conventional language but must be made part of the Messenger, rather than the processor state, since the Messenger migrates between different nodes. Each S_i is one of the following three types of statements:

Computational statements:

1. *assignment:* A Messenger residing in a particular node may: (1) read and update this node's variables; (2) read and update its own messenger variables; and (3) read the current node's network variables. Arbitrary expressions are permissible in the assignment and may include all of the common arithmetic and logic operators provided in C.

2. *control statement:* A Messenger may perform all the common control statements supported in C, such as if-then-else, while, do-while and break. These may access the same types of variables as assignment statement.

Navigational statements:

A Messenger may create new logical links and nodes, change or delete existing ones, and move arbitrarily through the network by following links or jumping to specific nodes.

1. *create(node, link, weight, physical)*: Create a new logical *link* leading to the specified *physical* node, along which the Messenger moves during this operation, associate *weight* with this *link*, and create a new logical *node* at the end of the new link, (i.e. on the *physical* node which the Messenger reaches). All parameters are optional. If omitted, the new node/link has no explicit name or weight and is created on a physical node chosen automatically by the system. Hence the last parameter allows the user to explicitly control the logical-to-physical node mapping. For instance, *create(node=a)* makes the system choose a certain physical node automatically, create a new physical node named 'a' onto this physical node, and move the requesting Messenger onto it.

2. *hop(node, link, weight, physical)*: Cause the Messenger to be forwarded to *node* on *physical* node along *link* if the link has the associated *weight*. As with *create*, all parameters are optional, which offers great flexibility in specifying the Messenger's next node destinations. If there are multiple destinations, a separate copy of the Messenger is propagated to each destination. For instance, *hop(node=a)* propagates the Messenger along all logical links emanating from the current node to all neighboring nodes named "a". If a node address is given as a node parameter, the Messenger jumps directly to the specified node. In addition, the parameters support "wild card" matching, which results in a powerful multicast mechanism for each Messenger. For instance, *hop(node=*; physical='laguna')* sends a copy of the Messenger to all nodes on the physical node named 'laguna', regardless of any logical links.

3. *delete(node, link, weight, physical)*: Forward the Messenger as with *hop*, but also delete the links traversed by each copy of the Messenger. In addition, if this action erases all links from the departing node and there are no other Messengers currently residing in it, the node is also deleted. No direct jump to the specific node is allowed in this statement.

Task-coordinating statements:

This allows the Messenger to load and execute ordinary C functions, precompiled in native mode of and residing on the current physical node, or being carried by the Messenger as it propagates.

1. *exec(filename, arguments, [nowait])*: Invoke a precompiled function specified by *filename* as a separate concurrent process and pass *arguments* to it. If *nowait* is specified, the function may continue operating even after the Messenger has left the node (or has terminated).

2. *func(filename, function_name, in_arg, out_arg)*: Invoke a precompiled function specified by *filename* and *function_name* as part of the MESSENGERS interpreter, wait for its completion, and return the results back to the invoking Messenger program. *filename* is given as a Unix path name and is resolved relative to the file system accessible by the current node. The *in_arg* and *out_arg* are arguments passed to and returned from the function respectively. The invoked function also receives pointers to the node variables area and the memory space holding the messenger variables and thus can manipulate them directly. The loading of the function is triggered dynamically when it is invoked for the first time.

   A function image can be carried by encapsulating it into a character array of Messenger variable and invoked by jumping to this data area.

2.4 The System Libraries

The MESSENGERS environment provides a library of functions that extend the basic capabilities of the language and its interpreter as described so far. The library realizes the creation, destruction, synchronization, and communication of Messengers. Among those features, the most basic functions implemented thus far are those of creation and destruction of Messengers.

Messengers Injection:

The injection of new Messengers is accomplished using the function *m_inject*. This injects a Messenger from a file containing the Messenger's behavior. It may also supply arbitrary initial parameters to the newly created Messenger. The function has two forms, depending on its intended use: one may be invoked from the Unix shell, thus allowing the use to create new Messengers on the fly; the other is used inside Messengers programs, thus allowing Messengers to create progenies at run time. The duplication of new Messengers is achieved by *m_duplicate* or *m_fork* which duplicates the calling Messenger inside the current node. The only difference is that *m_duplicate* has the duplicated Messenger start from the beginning while *m_fork* has the Messenger start right after its calling point (like Unix fork).

The injection of Messengers using these library functions should not be confused with the automatic replication of Messengers during navigation. The latter is implicit and is the result of the Messenger's following multiple logical links during a *hop* statement. The functions *m_inject*, *m_duplicate*, and *m_fork*, on the other hand, cause an explicit creation of a new Messenger inside a node.

Messengers Destruction:

A Messenger may terminate itself by executing the *exit* statement or it may be killed by another Messenger using a function *kill*.

2.5 Performance-Oriented Features

The introduction of an interpretation layer to the Unix environment brings several concerns regarding performance. First, function coordination should be realized with an efficient interface for passing arguments and returning values between these two layers. Otherwise, a function invocation itself incurs a heavy overhead, thus requiring coarse granularity. Second, the ensemble of Messengers at each node should be supported by fast synchronization mechanisms. Busy waiting in the interpreter layer would completely degrade performance. Third, some form of conventional message-passing facilities should be supported by the daemons, rather than forcing all communication to be programmed using Messengers. For instance, two messengers running on different nodes should be able to communicate via a daemon-supported communication channel rather than employing another Messenger working just as a data carrier. Finally, to support certain applications, such as distributed parallel simulations, the virtual time [Jef85] should be maintained by the daemons rather than broadcast by Messengers at the user level. These concerns are addressed by the performance-oriented features of MESSENGERS discussed below. Almost all of these features are accessible through library functions listed in Table 1 at the end of this section.

Function Call Interface

Function calls from the interpretation layer to the Unix environment are made using the *func* statement. Figure 4 shows the general form of a *func* statement and its possible instantiation as an actual example. Arguments in the *func* statement, (i.e. those listed after *in=* keyword) are copied into a new memory area, and its pointer, (i.e. *local in Figure 4) is passed to the C function. Any data type of constant values or variables such as messenger, node, and network variables can be passed as arguments (by value). In addition to arguments, a C function is given two pointers, (i.e. *node and *msgr in the example) to the node variables area of the current node and to the messenger variables area of the calling Messenger. Multiple values can be returned to the calling Messenger by invoking a library function, *back2messenger*.

In Messenger's script

General Form:	*Example:*
func(file=filename; name=funcname; in=args; out=ret_vals);	func(file="a.out"; name="func1"; in=x,y; out=a,b);

In C program

General Form:	*Example:*
	typedef struct { .... } NODE;
	typedef struct { .... } MSGR;
	typedef struct {double x, y;} LOCAL;
func_name(node_pointer, local_pointer, msgr_pointer)	func1(node, local, msgr)
char *node_pointer, *local_pointer, *msgr_pointer;	NODE *node, LOCAL *local, MSGR *msgr;
{	{
....	....
statement;	m = local→x * local→y
....	....
back2messenger(pointers_to_return_values);	back2messenger(&m, &n);
}	}

Figure 4: Function Interface

They are copied into variables listed after *out=* keyword of *func* statement in the enumerating order. Hence, a C function can get direct access to variables in the MESSENGERS interpretation layer and reflect its multiple computation results to this layer without any data conversion or replication. However, using pointers to node and messenger variable areas has the disadvantage that a priori knowledge of all node and messenger variables is coded into the function body.

Inter-Messengers Communication

There are two communication channels:

1 shared node variables
2 mailboxes (one per logical node)

Any Messenger can read/write any node variables on the node where it currently resides. To access node variables on remote nodes, the Messenger would dispatch a special Messenger to carry data to/from the remote node. In most cases, this is unnecessarily complex to programs and inefficient in terms of overhead. Hence mailboxes have been provided. They are available for both local and remote communication and provided through library functions: *m_send* and *m_recv*. (See Table 1.) Their restriction is that a logical node's mailbox is shared by all Messengers on this node.

Synchronization

Since node variables are shared among Messengers residing on the same logical node, it is natural to use these variables for inter-Messengers synchronization. This is accomplished by telling the interpreter daemon which node variable a Messenger

wants to synchronize on and what condition the variable should satisfy. At each interpretation cycle the daemon checks in native mode whether the variable satisfies the specified condition, and if so, resumes a Messenger waiting on this condition immediately. A Messenger describes the desired synchronization condition to the daemon using the library function, *m_sched_node* shown in Table 1. This scheme is much faster than busy waiting, where a Messenger repeatedly checks the condition in interpretive mode.

Global Virtual Time Maintenance

The concept of virtual time plays an important role in distirbuted computing. For instance, an application may consist of several stages, each of which must start after the complete termination of its previous stage. This is known as a ditributed termination detection. Another example is the distributed parallel simulation, where Messengers behave as simulation entities on a computational network representing a simulation space as incrementing their simulation time. With the concept of virtual time, Messengers can be scheduled along their virtual time line that is associated with each different stage of a ditributed termination problem or a simulation time. In order to make virtual time consistent over the system, its management should be supported by the system with efficient system-wide synchronizations rather than reimplemented at user level for each new application.

In our implementation, each interpreter daemon maintains its virtual time, so that Messengers can synchronize their execution along the virtual time line. This is realized by library functions, *m_sched_time_dlt* and *m_sched_time_abs*, which suspend the calling messenger for a certain virtual time interval or until a specified absolute virtual time, respectively. If there is no more ready Messenger at the current virtual time, the interpreter daemon increments its virtual time and resumes Messengers scheduled at the new virtual time. Since interpreter daemons work at their own speeds independently, a time lag may occur among their virtual times. When the daemon receives a Messenger whose time stamp is older than its current virtual time, it rolls back its computation to this time stamp. Such a rollback may require sending anti-Messengers to annihilate Messengers which have been mistakenly sent out. This is known as optimistic simulation. All interpreter daemons also communicate with one another to find the system-wide minimum virtual time which is known as the global virtual time (GVT). This is the lower bound beyond which the computation will never be rolled back [Fuj90]. Since GVT maintenance imposes significantly increased inter-daemons communications, virtual-time-based computation may be turned on and off by calling library functions, *m_gvtstart* and *m_gvtstop*, respectively. However, GVT concept always remains even in the case when virtual-time-based computation is turned off. (In other words, *m_gvtstart* resumes GVT at the virtual time which has been stopped by the last *m_gvtstop*.)

Adaptive Function Scheduling

This feature automatically varies the rate of invocation of a function called from a Messenger, based on the values being computed. In an event-driven simulation, where continuous change, typically described by differential equations, is approximated by discretized function, one of the main difficulties is determining an appropriate time step Δt for the next event. This is because a small Δt increases cost while too large a Δt may loose the fidelity of the computed results, and consequently require the computation to be rolled back. Such a user-driven rollback is not only expensive due to its interpretive mode but also difficult to program. To solve this problem, MESSENGERS offers a facility called adaptive function scheduling, where choosing appropriate Δt and the possible rollbacks of the computation are handled automatically. The basic idea is to increase, (e.g. double), the Δt at which the function is reinvoked as long as the predicate is satisfied, and to decrease it when the predicate is false. This permits the rate of function invocation to be continuously and automatically adjusted based on an arbitrary predicate involving any messenger variables. See the *m_sched_derivative0, 1, and 2* library functions listed below.

Inter-Messengers Communication
1. *m_send(address, data)* sends the content of *data* to the node specified by a system-unique node *address*.
2. *m_recv(address, data)* receives the oldest message sent to the node specified by *address* and copies the content into *data*.
Synchronization
1. *m_sched_node(node_variable, predicate_function, messenger_varible)* schedules the calling Messenger to resume its execution when *node_variable* satisfies *predicate_function*. The latter takes *messenger_variable* as its arguments and returns true or false.
GVT Scheduling
1. *m_sched_time_dlt(delta_time)* suspends the calling Messenger for a virtual time interval specified in *delta_time*.
2. *m_sched_time_abs(abs_time)* suspends the Messenger until an absolute virtual time *abs_time*.
Adaptive Function Scheduling
1. *m_sched_derivative0(function, predicate_function, messenger_variable)* involves only the currently computed value of *function*, which is passed a pointer to a messenger variables area. It keeps increasing the invocation interval of *function*, as long as the *predicate_function*, taking *messenger_variable* as its argument, returns true.
2. *m_sched_derivative1(function, predicate_function, messenger_variable)* involves the current and the last value of *function*, that is, it considers the estimated first derivative of *function* (its slope) based on its current and previous values. It keeps increasing the invocation interval, as long as the the rate of change (*function'*) satisfies *predicate_function*.
3. *m_sched_derivative2(function, predicate_function, messenger_variables)* involves the current and the last two values of *function*, that is, it estimates the rate of change of *function*'s slope, (i.e. the second derivative, *function''*) over the last two intervals, and keep increasing the interval, as long as *function''* satisfies *predicate_function*.

Table 1: Library Functions to Interface to Performance-Oriented Features

3 System Architecture

This section discusses the implementation techniques of MESSENGERS. First, we describe data structures used to schedule Messengers, manage logical computational networks, coordinate functions, and to communicate with external user processes. Then, we explain how the MESSENGERS interpreter use and manipulates these structures, considering issues of higher performance.

3.1 Network Structure

Daemon to Physical Network Mapping

Each daemon maintains network information necessary to communicate with other daemons. It maintains a *Physical Routing Block (ROUTE)* for each physical node containing a daemon to which it is connected. This includes its IP address, socket address structure and socket descriptor.(See Figure 5.) The *ROUTE* is the only data structure affected by the underlying network protocol, (i.e. the socket in our implementation). This implements the mapping of the daemon network onto the physical network.

Logical to Daemon Network Mapping

Each logical node is maintained by a *Node Control Block (NCB)*, which contains the node address, logical name and a pointer to its node variables area and a list of logical links emanating from this node. The node variables area includes a list of Messengers sharing this area in addition to the actual node variables.

Each logical link within the same daemon node is implemented as a pair of *LINK* structures connected via pointers to the corresponding *NCB*. Each logical link spanning two daemon nodes is split into two pairs of *LINK*s, one pair residing on each daemon node. They are connected on one side to their corresponding *NCB*s and on the other side to a *ROUTE*, which allow them to communicate. To identify the source and destination *NCBs* residing on a remote daemon node, a *LINK* contains the remote *NCB*'s system unique node address in addition to its link name and weight. Figure 5 describes the implementation of these logical-to-daemon and daemon-to-logical network mappings.

3.2 Messenger Management

Each Messenger has its own identity, including an independent program and private data, like a Unix process. However, it is regarded as only an intelligent message

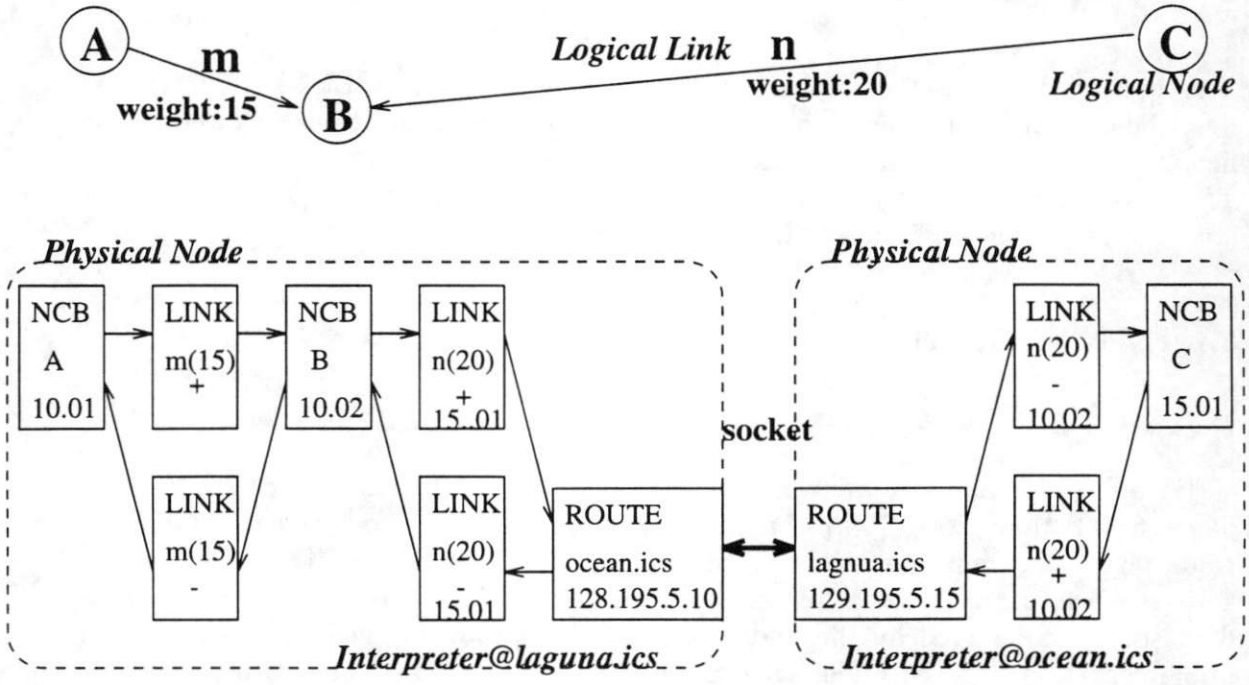

Figure 5: Implementation of Logical Network

interpreted by daemon processes, therefore it does not need its own virtual address space, file descriptors, and a full processor status. Of importance is that a Messenger must have a weight light enough to migrate to different physical nodes quickly and to be switched from one to another frequently. The run-time interpretation of a high-level language not only requires much processing time but also carries a large amount of status information, including parsing trees and symbol tables. To resolve this problem, each Messenger script is transformed into a machine-independent byte code by the MESSENGERS compiler before being injected into the system. Hence, the amount of information necessary to manage a Messenger is comparable to a thread, rather than a Unix process. The information is contained in a *Messenger Control Block (MCB)*, which is allocated to each Messenger upon its injection into the system. As shown in Figure 6, an *MCB* mainly consists of the Messenger's ID, its byte code's file name, and several pointers indicating the byte code, the current code position to be interpreted, its messenger variables area, and the node variables area of the current node.

The scheduling mechanism is realized using three types of queues: a *ready mcb queue* to include *MCB*s that are ready for interpretation, a *suspended mcb queue* to include *MCB*s that are waiting for the return from a precompiled function, and *outgoing mcb queues*, each of which keeps *MCB*s leaving for each physical destination.

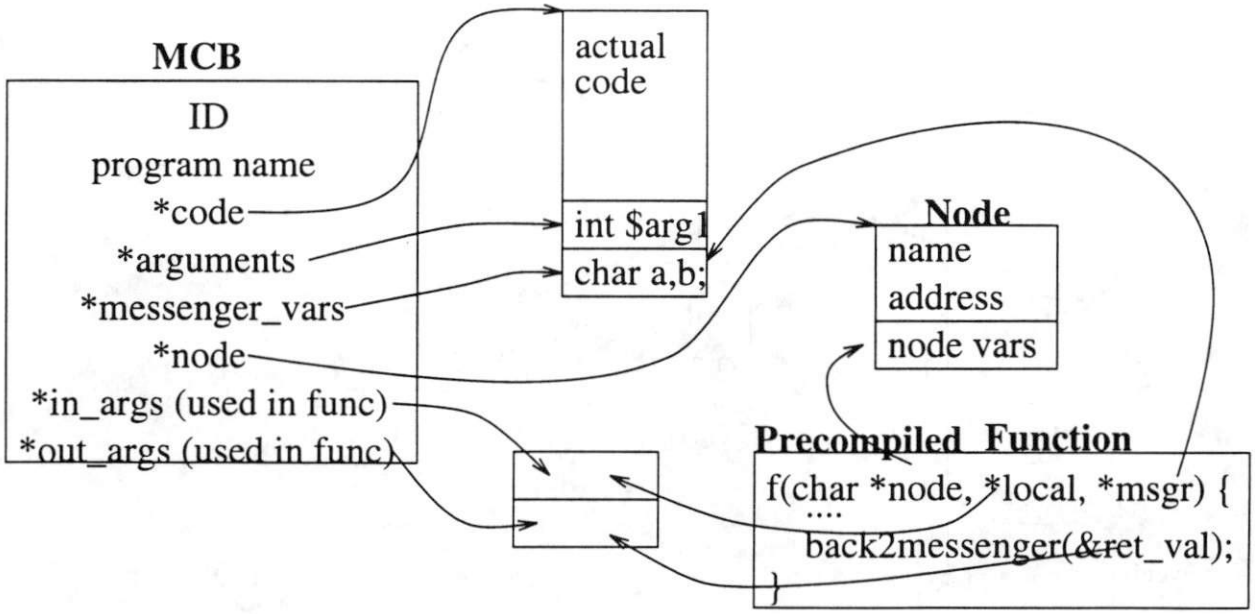

Figure 6: Messenger Control Block (MCB)

(There will be further improvements to realize a prioritized scheduling scheme in the future.)

3.3 Function Scheduling

As explained in Section 2.5, a Messenger has the ability to invoke specialized functions that suspend it for a certain period of time or to schedule the invocation of arbitrary functions as future events based on virtual time. This is implemented by maintaining an *EVENT* data structure, which records a function invocation scheduled at the current logical node. Each entry contains the calling Messenger's MCB, the entry/continuation point of this function, and the virtual time to resume its execution. Whenever a Messenger calls a precompiled function, an *EVENT* is allocated to schedule its invocation at the same virtual time. (As described in Section 2.5, virtual time concept is always maintained.) When the scheduling library function, *m_sched_time(time)*, is called, an *EVENT* is allocated to schedule the calling Messenger or function at *time*.

All allocated *EVENT*s are maintained by a splay tree [ST85] for each logical node so that the leftmost leaf always contains the earliest *EVENT* in virtual time for this logical node. The individual splay trees for all logical nodes on the same daemon are also maintained as a splay tree such that the leftmost leaf always contains the

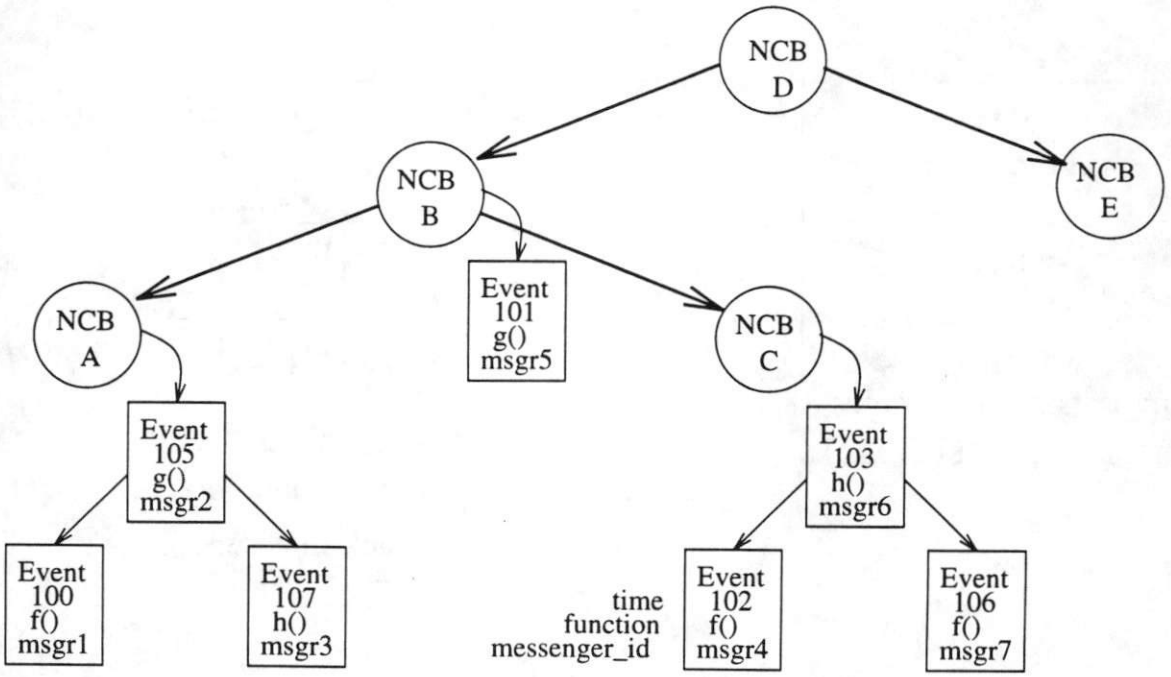

Figure 7: Hierarchical Structure of Event Queues

earliest *EVENT* over all logical nodes of the given daemon. Figure 7 shows the resulting hierarchical splay tree structure of *EVENT*s. The reason why we employ this scheduling queue is that MESSENGERS will be able to migrate logical nodes between daemons for dynamic load balancing in the future. This hierarchical structure permutes all future *EVENT*s associated with a migrated logical node to be found and packed efficiently.

3.4 External Interface

Messengers and external Unix processes, for example a TCL manager, need to have a specific communication channel rather than use the Unix file system, for reasons of performance. Upon initialization, each interpreter daemon acquires a Unix-supported shared region and a message queue associated with a user-defined key, and allows Messengers to use them to interface with external processes running on the same physical node. The key is specified in a *.messengers_profile* file that is located in a user's home directory and used for a daemon initialization. The external processes can access these Unix-supported shared region and message queue by reading the key and calling the *shmget* and *msgget* Unix system calls with this key.

1. *Interface Buffer (IFBUF):* This is a Unix-supported shared memory region which consists of two sub-spaces. One shows the interpreter daemon's current status and the other is open for communications with external processes.

2. *Interface Port (IFPORT):* This is a Unix-supported prioritized 2k-byte message queue. Several priorities are reserved for the MESSENGERS daemon but the others may be used for communications with external processes.

Since both IFBUF and IFPORT are only static interface media, a synchronization protocol between Messengers and external processes must be agreed upon *a priori.*

3.5 Behavior of the Interpreter

Figure 8 describes the interpreter daemon's behavior. The MESSENGERS interpreter daemon alternates the following two phases: (1) interpreting Messengers until there is no more ready Messengers, (line 3-23) and (2) executing scheduled functions unless there is any ready Messengers (line 24-27). During each phase it also exchanges Messengers with other daemons whenever a certain combination of conditions are satisfied (line 21-22 and 28-29).

(1) Interpretation

Once the interpreter daemon picks up a ready Messenger, it continues processing its statements while they are computational (line 5-6 in Figure 8). Upon encountering a *hop* statement (line 9), the daemon traces all *LINK*s from the current node, (i.e. *NCB*) to locate the destinations. If the destination is a local *NCB*, the current Messenger, (i.e. *MCB*) is simply attached to this *NCB* (line 10) and goes to the end of *ready mcb queue* for the next interpretation. If the destination is a remote node (line 11), (i.e. *ROUTE*), the *MCB* is enqueued into the end of the *ROUTE*'s *outgoing mcb queue* to wait for being sent (line 12). The other navigational statements, namely *create* and *delete* achieve the creation and deletion of a node respectively (line 7-8) in addition to the operations shown above.

Upon encountering a *func* or *exec* statement (line 14-15), a new *EVENT* to schedule a function invocation is inserted into the current node's *future event queue* (line 16). This insertion entails tree rotations of the hierarchical future event queue. The current *MCB* is linked into *suspended mcb queue.*

Figure 9 illustrates how a Messenger is handled by the interpreter daemon from its creation to migration. A new Messenger is injected from an external process onto *IFBUF*, enqueued into a *ready mcb queue*, and waits for its interpretation. Once it is picked up from *ready mcb queue*, the Messenger is processed as described above,

```
1   daemon() {
2       for(;;) {
3           while ((mcb = ready_mcb_dequeue()) != NULL) {
4               for (;;) {
5                   switch(mcb→instruction[mcb→counter++]) {
6                   case COMPUTATIONAL: compute(); continue;
7                   case CREATE:        create_new_ncb(); goto HOP;
8                   case DELETE:        delete_current_ncb();
9                   case HOP:           if(next_ncb == LOCAL) {
10                                              relink(mcb, next_ncb); ready_mcb_enqueue(mcb);
11                                          } else
12                                              outgo_mcb_enqueue(mcb, next_route);
13                                          break;
14                  case FUNC:
15                  case EXEC:          event→messenger = mcb;
16                                      local_future_event_enqueue(event); susp_mcb_enqueue(mcb);
17                                      break;
18                  }
19                  break; /* context switch */
20              }
21              if (communication_conditions == TRUE)
22                  exchange_messengers();
23          }
24          if (ncb = global_future_event_dequeue() != NULL)
25              if (event = local_future_event_dequeue(ncb) != NULL) {
26                  execute(event→function); ready_mcb_enqueue(event→messenger);
27              }
28          if (communication_conditions == TRUE)
29              exchange_messengers();
30      }
31  }
```

Figure 8: Pseudo Code of Interpreter

enqueued into an appropriate *outgoing mcb queue* when it wants to migrate to a remote node, and sent out onto a socket in the next communication phase.

As described before, the interpretation of each Messenger is serialized between any two navigational or task-coordinating statements. This means that a Messenger must explicitly relinquish control to another Messenger using one of *create, delete, hop, func* and *exec* statements. Although such a non-preemptive scheduling carries the risk of hanging an interpreter daemon easily by a Messenger in an infinite loop, it has two advantages from the performance point of view. One is reducing the number of context switches between Messengers. Another is that non-interruptibility (e.g. for critical region enforcement) is guaranteed without any explicit user-defined synchronization mechanisms. If such non-interruptibility is programmed in each Messenger script. inter-Messenger coordination must be enforced at interpretation level, which will degrade the system performance. Instead, when synchronization is needed for cooperation, Messengers may use the *m_sched_node* function (see section 2.5), supported as a library function. Hence, the CPU time will not be wasted unnecessarily.

(2) Function Execution

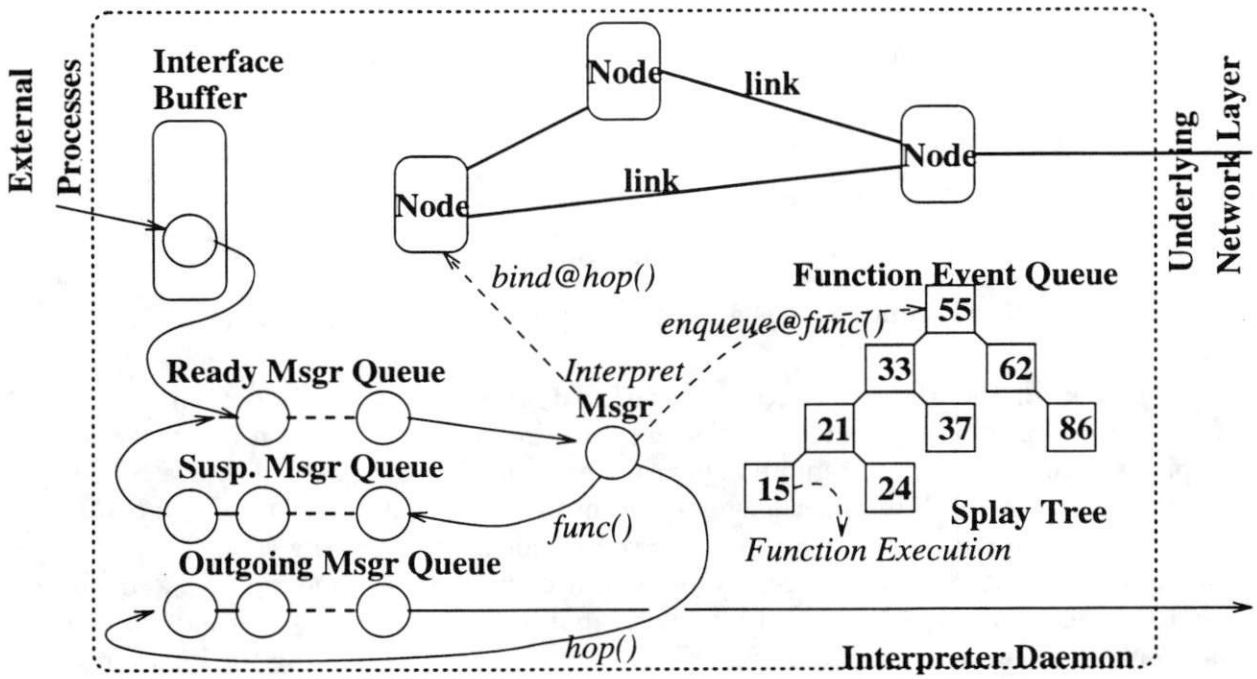

Figure 9: Interpreter's Behavior

Function executions are initiated only when there is no more ready Messengers to interpret. This design is based on the following two observations. First, Messengers which do not call precompiled functions should be interpreted and sent to their next destinations as quickly as possible without the overhead incurred by dynamic function linking. Second, under a virtual-time-based computation, Messengers do not have their own virtual time recorded with their *MCB*s. Only when they are exchanged among daemons, are they sent with the source daemon's latest virtual time. An interpreter daemon updates its local virtual time when it executes a precompiled function scheduled at a certain virtual time. Hence, all Messengers inside this daemon are regarded to have the same new virtual time. If the daemon executes a scheduled function although there are still ready Messengers to be interpreted, even Messengers which do not call a scheduling function at all, are inevitably shifted into a new virtual time.

The interpreter daemon picks up of the leftmost *NCB* leaf of the hierarchical event queue, the leftmost *EVENT* leaf, and then executes a function specified in this *EVENT*. If such a function is not linked yet, the daemon performs a dynamic link system call to load this function before its execution. This deletion of the *EVENT* reorganizes both levels of the future event queue by tree rotations so that the next *EVENT* to be processed is ready to be picked up.

(3) Network Communication

The interpreter daemon initiates an inter-daemons communication when:

- there are no more ready Messengers, or
- the number of outgoing Messengers exceeds a given threshold, or
- an interval timer generates an interrupt, or
- the local virtual time exceeds a given interval

Until any of the above conditions is satisfied, the daemon pools outgoing Messengers into an *outgoing mcb queue* for each destination *ROUTE*. If a Messenger is duplicated to two or more remote logical nodes located on the same physical node, only one copy of the Messenger is enqueued into the destination *ROUTE*'s *outgoing mcb queue*. The parameters for the above communication triggering conditions, (i.e. the number of outgoing Messengers to be sent at once, the real time interval, and the virtual time interval) are specified by a user upon the daemon initialization and can be changed at run time through *IFBUF*. These are used for adjusting the amount of data transferred between physical nodes in one burst for the better performance.

4 Performance Evaluation

In this section discussion, we first summarize the current implementation status, next we show the overhead incurred by the interpretation on a single machine and the network latency, and finally discuss the performance results for parallel computing using various applications. We will also discuss the best computation granularity for each application.

4.1 Current Implementation Status

We have implemented the MESSENGERS interpreter and language compiler. The system library currently includes primary functions to create, duplicate, identify and destroy Messengers; other functions are still under the construction. The system does not yet support inter-Messengers synchronization and virtual-time-based scheduling. Thus, these two features must still be handled by each application independently. Since any function invocation causes a context switch between Messengers, synchronization is currently realized by testing a given node variable and calling a dummy function if the node variable does not satisfy a given condition. The virtual time is maintained by broadcasting a new virtual time to all nodes using *hop(node = *)*.

Item	Descriptions
Workstations	Sun SPARC Station ELC, 16M memory
Communication Media	10Mbps Ethernet
Network Nodes	18 Workstations (One of them is an NFS server.)
	+ 43 PCs without bridge/router

Table 2: Physical Configuration for Performance Evaluation

The interpreter daemon is currently running as a single Unix process. It does not spawn any threads. Only blocking read and write operations are used for data transfers over the network. Hence, there is no concurrency inside a daemon. Table 2 summarizes the current physical configuration used for our performance evaluation.

4.2 Performance of Interpreter

First, we concentrate on the performance of the interpreter itself when it interprets Messengers without any precompiled code. Our performance goal is to keep the interpreter's overhead within one order of magnitude slower than the execution speed in native mode.

The performance evaluation consists of six tests. Test 1 compares MESSENGERS' arithmetic interpretation with native arithmetic computation. The arithmetic operation is a 500,000-time repetition of a floating-point division, (i.e.$a = b/c$). Test 2 shows the performance of a *func* statement interpretation vis-a-vis an ordinary function call in native mode. The test repeatedly calls a dummy function 100,000 times. Test 3 compares the creation of a Messenger with that of a thread and a Unix process. The creation is repeated 30,000 times. A Messenger's size is 640 bytes, whereas a thread allocates only 177-byte data which correspond to messengers variables and an MCB. Test 4 examines the context switch of a Messenger and of a thread. 24,000 context switches of Messengers are generated by having 1,200 Messengers call a dummy function 20 times. 24,000 context switches of threads are similarly generated. Test 5 evaluates a Messenger's local hop within the same physical node. The elapsed time of 24,000 local hops is compared with that of 24,000 context switches of threads. Test 6 compares a Messenger's hop over a physical network with an inter-threads communication through a socket. The byte size of the hopping Messenger is 800 bytes and is the same as that of the data transferred between threads. This test repeats 3,000 data transfers over a physical network.

As shown in Table 3, for all the tests except Test 2, the time required by the interpreter was less than 13 times the native execution time. The result of Test 2 is caused by the fact that a function invocation always involves a context switch

Test #	programs	time (seconds)	magnitude (times as slow)
Test 1	*Arithmetic interpretation vs execution*		
	(a) Sequential C program	2.117	1.0
	(b) Messengers	27.193	12.8
Test 2	*Func statement vs ordinary function call*		
	(a) Sequential C program	0.190	1.0
	(b) Messengers	32.506	171.1
Test 3	*Creation: Messenger vs thread/process*		
	(a) Thread	26.569	1.0
	(b) Messengers	77.996	2.9
	(c) Unix process	3879.680	146.0
Test 4	*Context switch: Messenger vs thread*		
	(a) Thread	9.759	1.0
	(b) Messengers	11.185	1.1
Test 5	*Local hop vs thread's context switch*		
	(a) Thread	9.759	1.0
	(b) Messengers	101.52	10.4
Test 6	*Remote hop vs thread communication with a socket*		
	(a) Thread/socket	10.266	1.0
	(b) Messengers	70.913	6.9

Table 3: Performance Result on a Single Machine

operation. This indicates that a further improvement of the interpreter daemon is needed to avoid invoking a meaningless series of context switch operations when there is only one Messenger on a physical node. Test 4 shows that there is little difference between Messengers and threads in terms of their context switch. The overhead seen in Test 3 is for a complete duplication of a Messenger as compared to thread. The overhead observed in Test 5 and 6 includes the destination node search and the flow control over the physical network.

As a result, it is hard to limit the interpreter's overhead within several times of the native speed, however with our goal, if a Messenger invokes a precompiled function which includes more than 1,000 floating-point operations, the performance degradation will be held within only several percent even on a single machine.

4.3 Network Latency

The physical network communication is the major factor that diminishes the effect of parallelism. In order to hide the network latency, the application program needs to overlap communication with computation or, when the application does not permit that, granularity of computation will be coarse enough to outweigh the communication overhead. We have investigated the latter by following test: from a single node, a Messenger is duplicated to a given number i of other physical nodes, where each computes a given number j of dummy floating-point divisions. All Messengers then

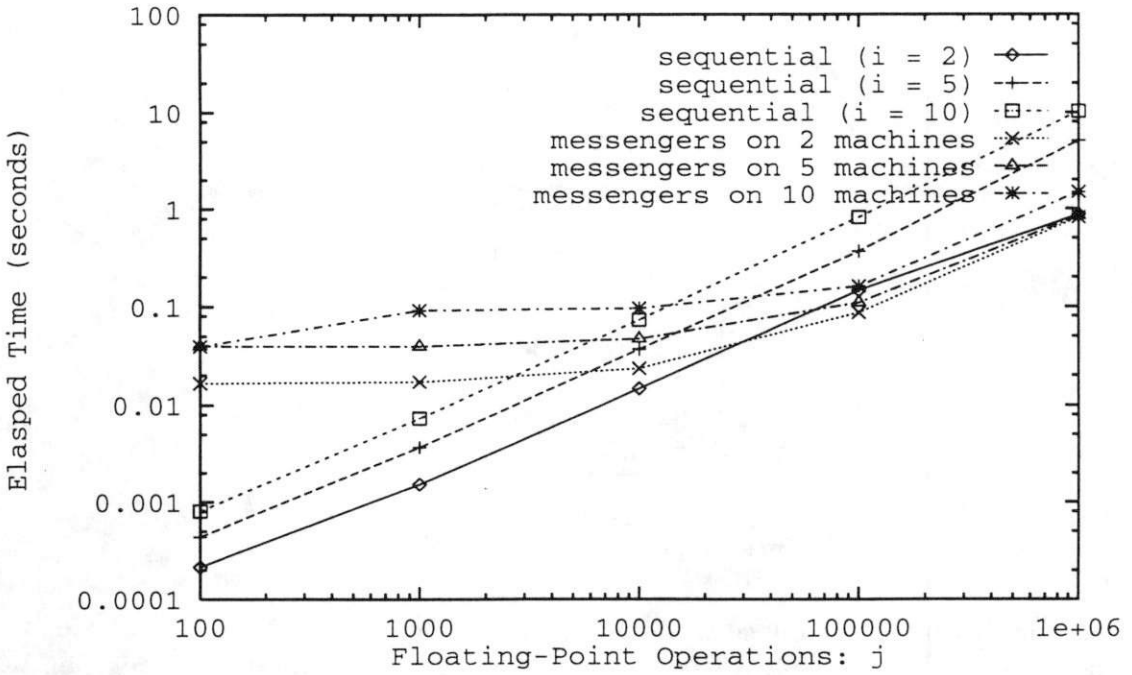

Figure 10: Network Latency Test

return to the original station node. For comparison, a sequential program executes the same amount of computation that includes $i \times j$ divisions. Figure 10 shows the result. This shows that the order of 15,000-30,000 floating-point operations are necessary at each physical node to compensate for the network latency. In the following subsection, we will show the performance of four kinds of applications, some of which are very simple and include only several or even no floating-point operations. Therefore we cannot expect their performance to be very good compared to sequential execution. However, assuming that programs from the similar application arenas but with much large grain size exist, we have conducted the same communication-versus-computation test shown above for each application, and will show the grain threshold necessary to overcome the communication overhead.

4.4 Performance of Applications

We coded the following four application programs:

1. *Pharmaco-kinetic Simulation:* This program simulates the distribution over time and metabolism of various toxins by different organs of a living organism. In col-

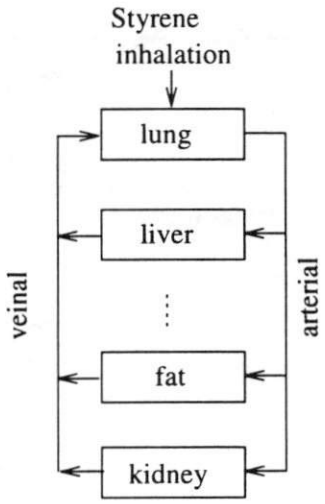

Figure 11: Pharmaco-Kinetic Simulation of Inhaled Styrene

laboration with UCI's College of Medicine, we implemented a Styrene-inhalation program using MESSENGERS as a control language to coordinate the operation and interaction of compiled node-resident functions, which carry out the actual computations of the model [FMB+96]. The basic approach is to map each of five organs onto a separate logical node as shown in Figure 11. This node contains the necessary sets of differential equations and constants describing the organ's behavior. The Styrene-carrying fluids, such as blood, are implemented as waves of consecutive Messengers, which pass from the lung node to the other organs and back along the predefined paths. As they pass through the organs, they trigger the execution of appropriate functions to compute the new Styrene concentrations for the current simulated time increment. The lung node becomes the generator of the virtual time increments, which it sends to all other organs with each wave of Messengers. This program is an example of dynamic model computation (also referred to as intra-Messengers coordination in [FBD96]).

2. *Shortest Path:* This program uses Dijkstra's algorithm [CLR90] to find single-source shortest paths on a weighted $N \times N$ mesh for the case where all edge weights are nonnegative. We construct such a mesh of logical nodes over the physical nodes. The logical nodes are named $n00$ through $n(N-1)(N-1)$. As shown in Figure 12, shortest paths from $n00$ to all the other nodes are found by a Messenger, injected initially into node $n00$, and replicated along all vertical and horizontal links to compute the distances from $n00$. Upon arriving at each node, each of Messengers compares its own distance traveled thus far with that computed by a previously visiting Messenger and stored at that node. If the Messenger's distance is shorter, it updates the node's distance and continues

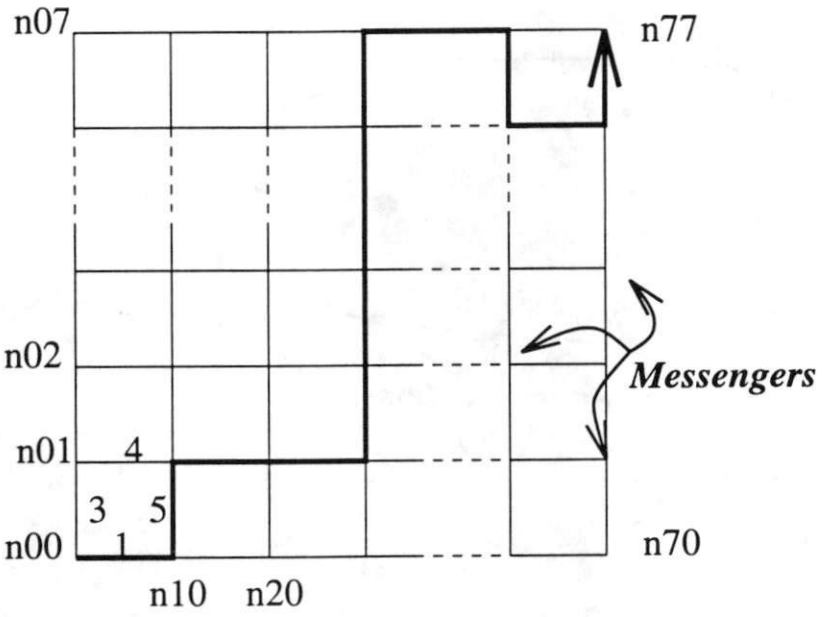

Figure 12: Shortest Path

its propagation. Otherwise, it goes back to $n00$ to terminate. Since $n00$ has a node variable that keeps the number of active messengers, a Messenger that reads 0 from this variable is the last active Messenger and thus recognizes the termination of this distributed application. This application in an example of inter-Messengers coordination and is expected to benefit from parallelism if the logical mesh network is mapped over more physical nodes.

3. *Convex Hull:* This is a divide-and-conquer program to solve a convex hull of n points given in the plane. The program first divides these 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 Shamos's algorithm [PS85] to merge the hulls at each level as it back-tracks. As shown in Figure 13, this program dynamically constructs a binary tree, each node of which combines two convex hulls from its left and right child nodes. This binary tree is mapped onto physical nodes at run time. Initially, a Messenger is injected to create the root of this tree. It then replicates itself at each level to create two child nodes. When reaching a leaf node, each Messenger works as a carrier of a sub convex hull from the leaf back to the root. Since there is no data dependency among nodes at the same level of the tree, the divide-and-conquer or bisection is a typical algorithm applied to loosely coupled memory machines. Given a large number of points, the more physical nodes are used, the more parallelism is this

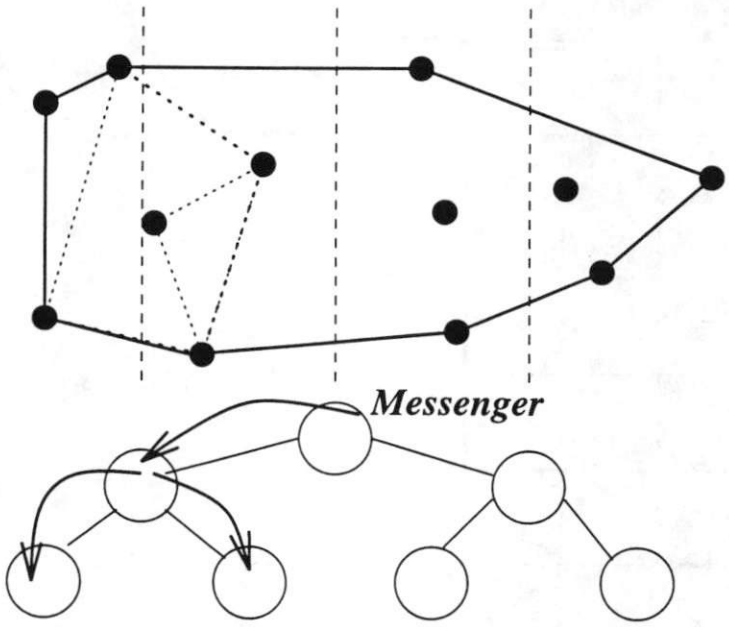

Figure 13: Convex Hull

program expected to achieve.

4. *Wa-Tor:* This is a simple Monte Carlo ecological simulation program implementing a model by Dewdney [Dew84], in which idealized fish and sharks live, move randomly, breed, and eat one another in a two-dimensional ocean grid. Figure 14 shows this simulated space. It is subdivided into several subspaces (striped along the y-axis in the present implementation), which are represented by logical nodes and distributed over physical nodes. Each individual fish selects one unoccupied neighboring place, as it breeds its progeny in the departing place every certain period of time. Each shark selects a neighboring place where a fish occupies and eats it. Otherwise, the shark moves just as a fish does. If it cannot eat any fish in a given period of time, it starves. Each individual fish and shark is implemented as a Messenger, which carries its own behavior as described above. This is a good example of an application requiring efficient inter-Messengers coordination and hence is a good candidate for studying scalability issues on efficient number of physical nodes.

The performance of each application has been compared with that of its corresponding sequential program.

Figure 15(a) shows the result of the pharmaco-kinetic simulation (inhalation of styrene). Since the organism is modeled as five distinct tissue nodes, we used one

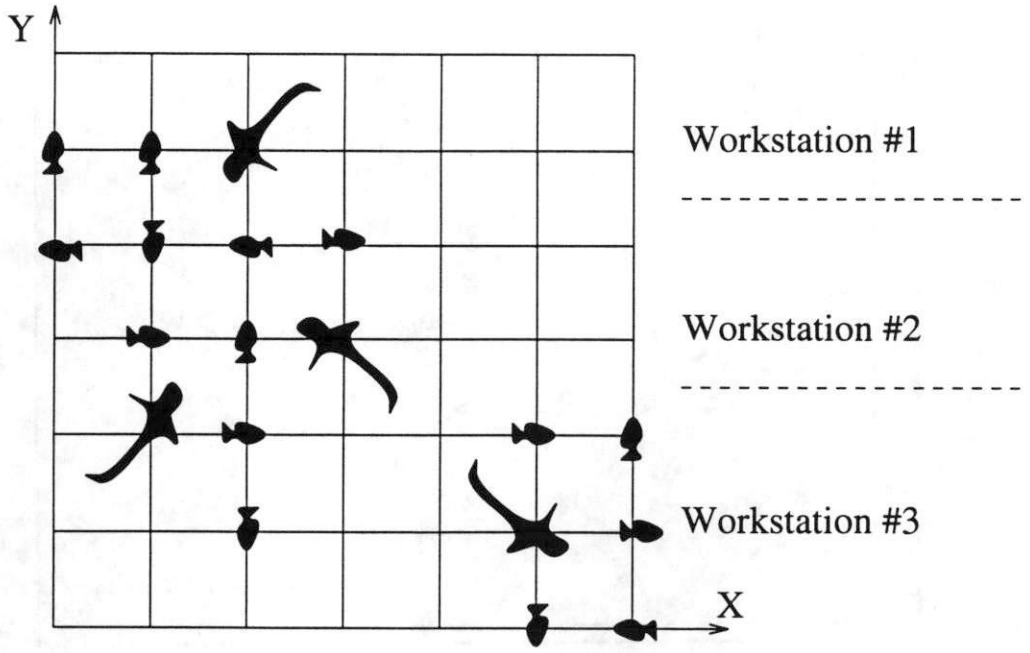

Figure 14: Wa-Tor

to five workstations. The result is disappointing, showing a 2-3 orders of magnitude slower for Messengers. There are two main reasons: (1) the computation size at each tissue node is only several floating-point operations, and hence the resulting parallelism cannot compensate for the high communication penalty, (2) all Messengers behaving as a styrene carrier, a virtual time watcher, and a file recorder must synchronize with each other using busy-waiting due to the lack of an inter-Messengers synchronization library function. Next, we have conducted a communication-vs.-computation test to see how much computation must be preferred at each node to overcome the communication overhead. Figure 15(b) shows the result when each computation node performs a given number of (dummy) floating-point divisions. The result shows more than 10,000 floating operations are required for this type of application.

Figure 17(a) shows the result of the shortest path program. As shown in Figure 16, we divided the 8×8 mesh of logical nodes into 2, 3, 4, 6, and 9 pieces, each of which is mapped onto a different workstation. The result of MESSENGERS shows one magnitude slower than the corresponding sequential program. The main reasons are: (1) the amount of computation at each node is minimal (no floating-point operation), (2) the Messenger program always runs in interpretive mode, never calling any precompiled function, (3) all Messengers need to go back to $n00$ node to implement a distributed termination. To investigate granularity, we had a Messenger invoke a dummy function at each logical node, which repeats a given number of floating-point divisions.

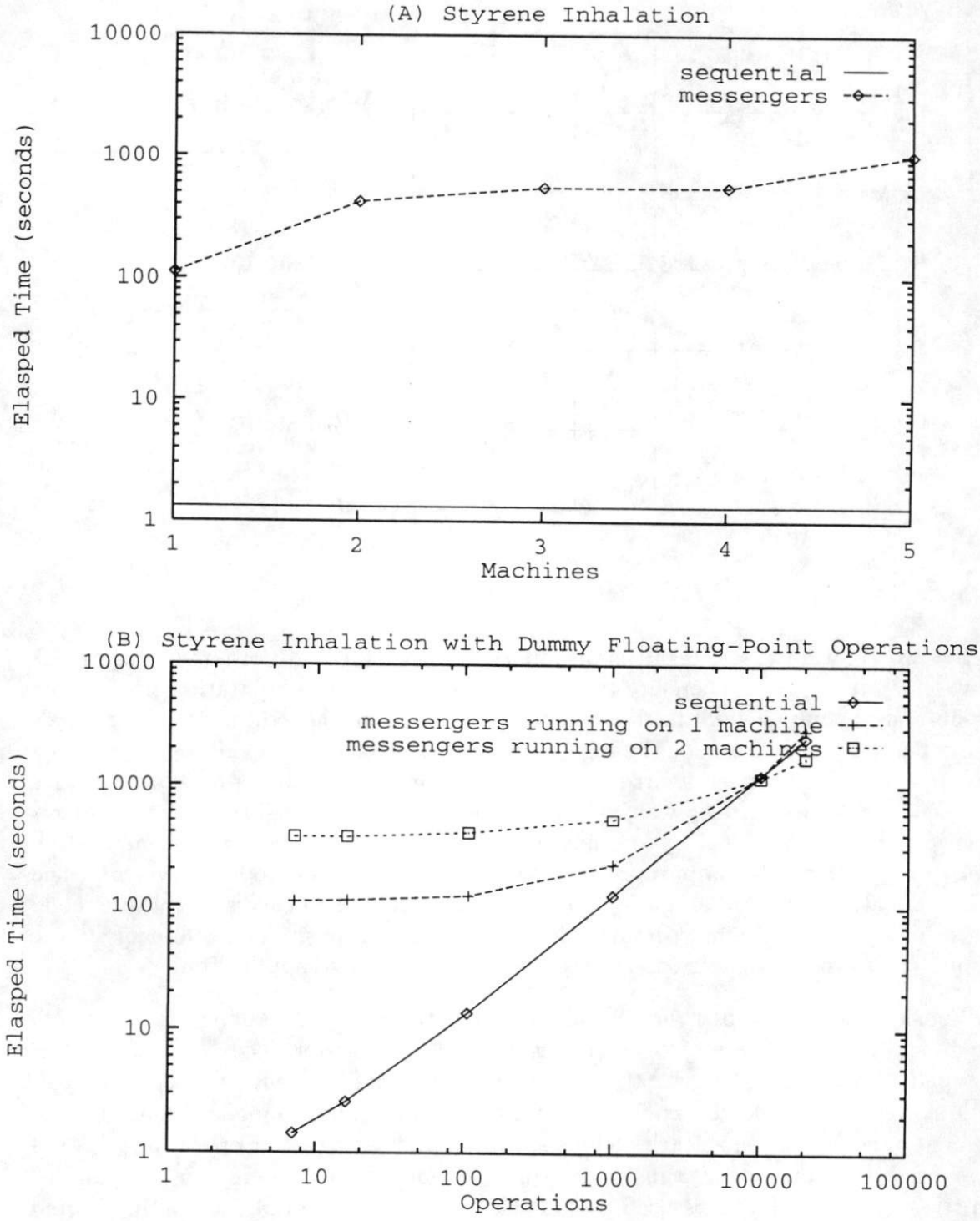

Figure 15: Performance Result of Styrene-Inhalation Program and Its Coarse Grain Analogue

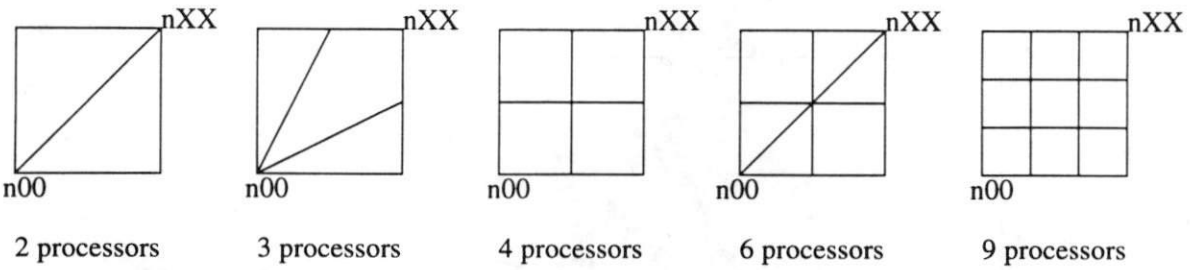

Figure 16: Logical to Physical Network Mapping for Shortest Path Program

We compared MESSENGERS with two different sequential programs: the one that performs a breadth-first search and the other that performs a depth-first search. The latter algorithm can be simply coded by using recursive function calls, however it may search more possible paths than the former one. In MESSENGERS, the interpreter always interprets Messengers in a breadth-first fashion, hence the breadth-first search, (i.e. the better algorithm) is automatically chosen although a user does not carefully intend it. Figure 17(b) shows this comparison. This type of application requires 2000 to 3000 floating-point operations to compete with the sequential version. However, MESSENGERS becomes competitive at only 70 floating-point operations if it is compared with the naive sequential program based on the depth-first search.

Figure 18(a) shows the performance results of the convex hull program which solves 1024, 4096, and 32768 points respectively. For more than 4096 points, the effect of parallel processing beats the communication and interpretation overheads. Since a Messenger has been coded to always carry all points, the more workstations are given, the more communication overhead is incurred and hence performance does not improve beyond four machines. In some bisection programs, however, such as a quadrature problem for numerical integration [GM85], only a single computation result at each node must be passed back to its parent node, and therefore the amount of data transferred is insignificant. We have evaluated the performance of such a case by having a Messenger repeat a certain number of dummy floating-point divisions at each node of the tree but not carry any data between nodes. Figure 18(b) shows this result. Similar to the other examples, the threshold to obtain an improvement from parallel computing is approximately 10,000 floating-point operations at each node.

Figure 19(a) shows the performance results of the Wa-Tor problem. The simulation space is represented as a 32×32 matrix and striped into 2, 4, and 8 pieces, each of which is mapped onto a different workstation. We injected 500 Messengers. These Messengers do not include any floating-point operations but just several simple integer assignments. Hence, similar to the shortest path program, the result is not competitive with the corresponding sequential program. Next, we had each Messenger execute dummy floating-point divisions. As shown in Figure 19(b), MESSENGERS

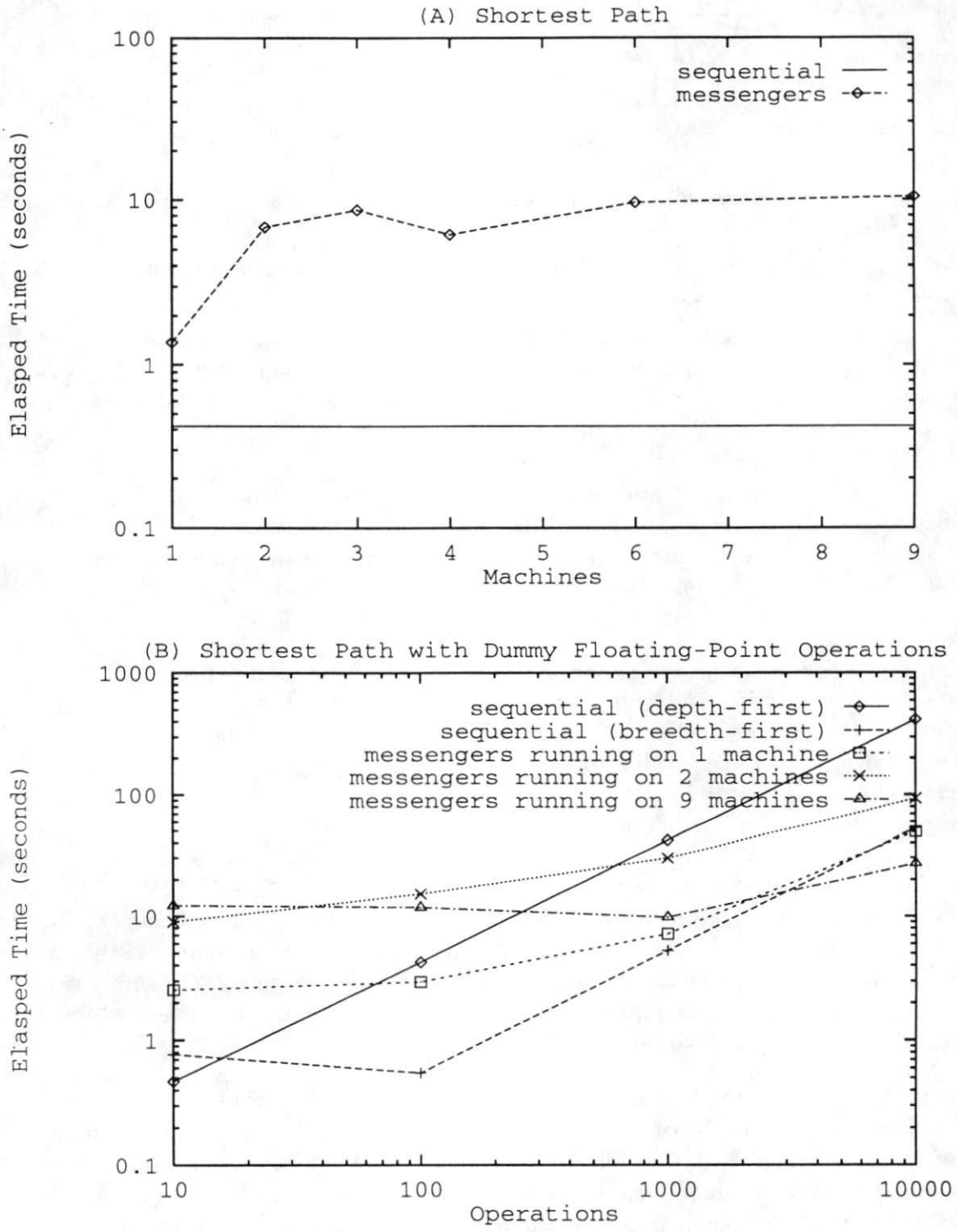

Figure 17: Performance Result of Shortest-Path Program and Its Coarse Grain Analogue

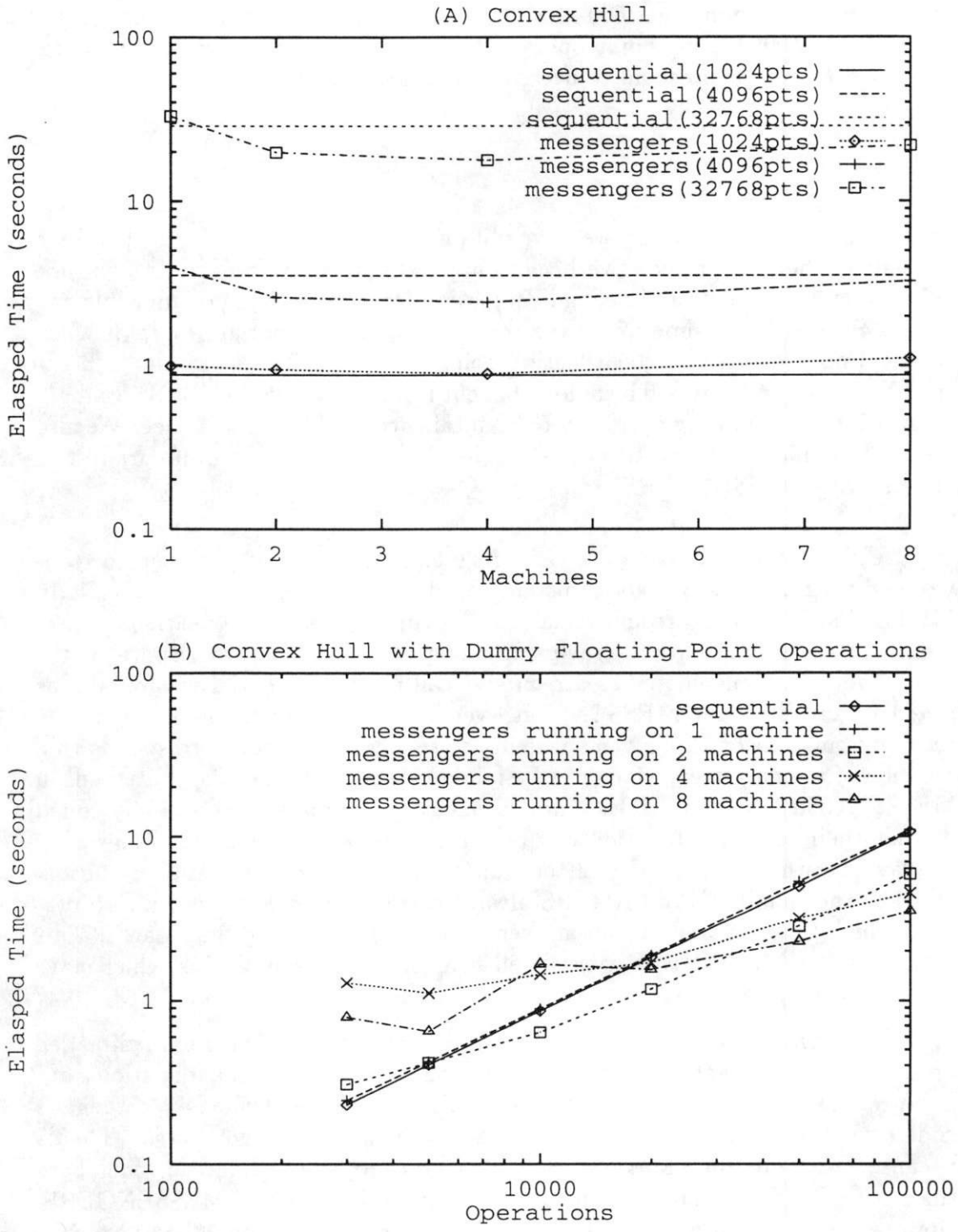

Figure 18: Performance Result of Convex Hull Program and Its Coarse Grain Analogue

is superior to the sequential program if each Messenger executes 400 or more floating-point operations. Assuming that 500 Messengers are uniformly distributed over eight physical nodes, 2,500 floating-point operations at each physical node, which comes from $400 \times 500/8$, are a threshold to overcome the network overhead.

4.5 Discussion

From our performance evaluation we have obtained two numerical results: (1) for a single machine, the interpreter's overhead is approximately one order of magnitude above the sequential execution time, and (2) for applications runining on multiple machines, MESSENGERS is superior to the correnspondingsequential programs when each physical node executes 10,000 floating-point operations. These results indicate the type of applications that will be able to benefit from the MESSENGERS paradigm and the style of programming necessary to maintain acceptable performance. We also made several significant observations that emphasize the advantages of programming applications with MESSENGERS.

1. *MESSENGERS versus Recursive Call:* As seen in the shortest path and the convex hull program, many algorithms are based on the recursive function call. In MESSENGERS, such a recursive call can be emulated as its navigational statement. In the shortest path, traversing along all vertical and horizontal paths from a node is implemented as a recursive call in the sequential version and as a *hop* statement in the MESSENGERS version. In the convex hull program, a recursive subdivision of a given space in the sequential version corresponds to a creation of a binary tree, which can be made by a repetition of *hop* statement in MESSENGERS version. Unless these sequential programs are optimally coded so that their recursive function call performs a breadth-first search, they may display a depth-first searching pattern due to the nature of sequential execution. On the other hand, MESSENGERS always performs a breadth-first interpretation, which tends to find out an answer faster than the depth-first search. This means that MESSENGERS works well against naive applications which have been intuitively coded but not yet optimized.

2. *MESSENGERS versus Thread Programming:* Programming with multithreaded processes is an alternative approach to code distributed parallel applications and is expected to be more efficient in the following two cases: (1) when a Messenger's code size is large, and (2) when Messengers are explosively generated. This is because each individual Messenger has its own code, whereas threads can share the same code. Therefore, copying a code is always overhead in MESSENGERS. However, we have observed that there is no difference between MESSENGERS and multithreading in terms of context switch. Also, *MCB*s which have been used for old Messengers are not completely freed but pooled for the future reuse.

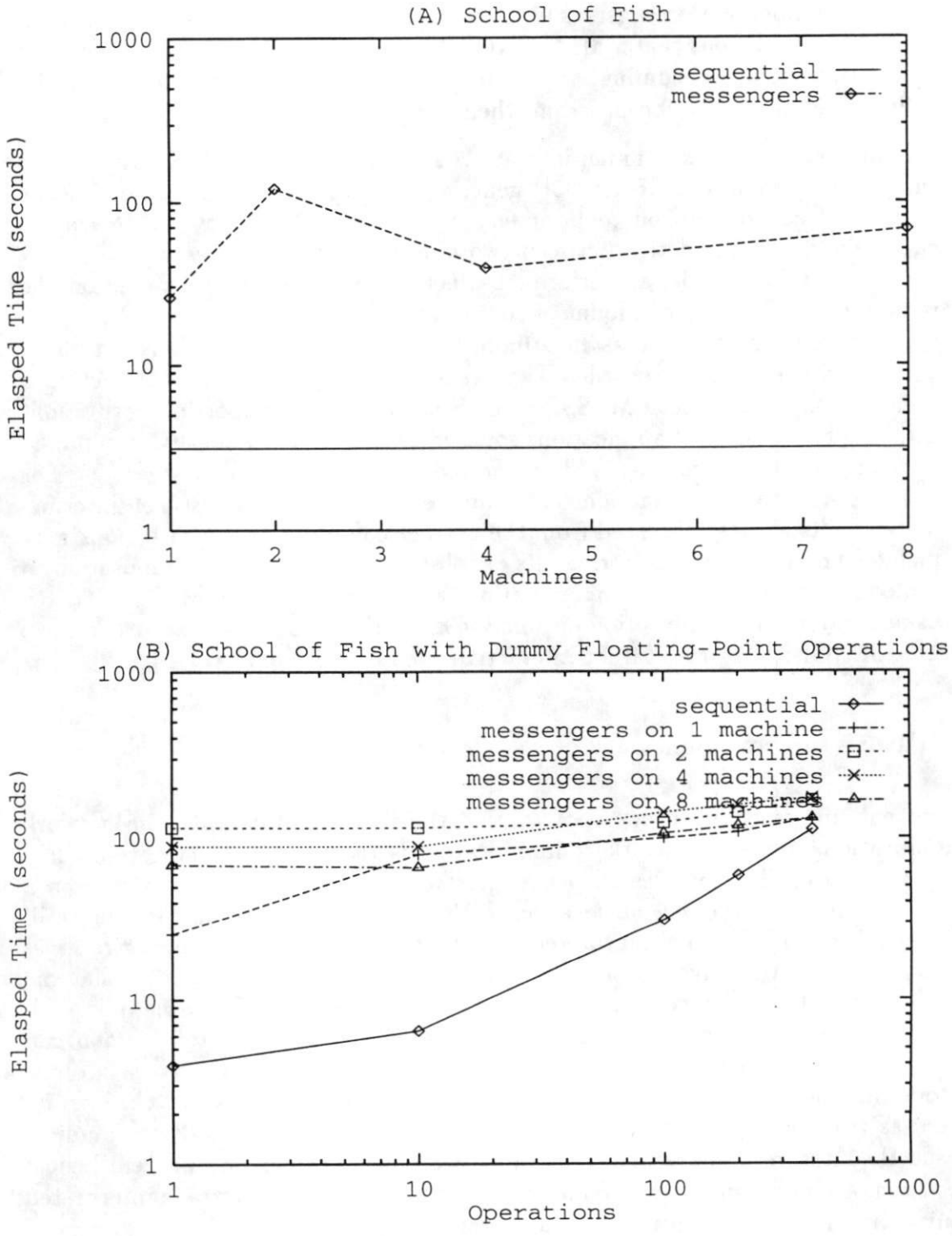

Figure 19: Performance Result of Wa-Tor and Its Coarse-Grain Analogue

Hence, MESSENGERS is competitive with multithreading for applications such that the number of Messengers is kept constantly or its growth is constrained. In addition, it is obvious that a MESSENGERS user does not have to be concerned with flow control, scheduling, and memory management, all of which must be coded by each application program when using multithreaded processes.

3. *Interactive Operations:* Human-interactive applications such as DIS (Distributed Interactive Simulations [BBB+93]) generally demand that a response time be within $\frac{1}{20}$ seconds. In our experiment on the network latency, a Messengers can make a one-round trip between two different physical nodes within an order of ten milliseconds, as performing 10,000 floating-point operations on the remote node. The same amount of computation is included by general-purpose benchmark programs such as Livermore Loops. Therefore, the elapsed time in our experiment can be regarded as a response time just enough to react to a user's demand and hence MESSENGERS is capable of supporting such applications. These types of applications sometimes need to dynamically manipulate the computation in progress. This is referred to as "computational steering". For instance, the next set of actions or parameters is determined based on intermediate results already observed from the current computation. Such actions may include handling intermediate results or rolling back the current computation to some previous status. Such manipulations are easily realized in MESSENGERS using its dynamic composition of function execution and GVT features. Hence, MESSENGERS is practical for interactive open-ended applications

4.6 Future Improvements

The current implementation does not overlap the interpretation with the network communication. Once a network communication is triggered, the interpreter daemon is blocked until it receives all Messengers which have been flushed out from a sender daemon's *outgoing mcb queue* at once. However, the receiver daemon may still have ready Messengers to be interpreted. Furthermore, once the interpreter receives the first incoming Messenger, it should immediately start interpreting it so that the communication delay in receiving the following Messengers can be overlapped with the interpretation. Using asynchronous I/O handling, the interpreter daemon can hide the network latency, and therefore the computational granularity at each physical node may be reduced to less than 10,000 floating-point operations. This can be realized as two concurrent threads which are in charge of the network communication and the interpretation respectively. Since Messengers running at different logical nodes can not modify any data outside of their current node, they can be interpreted concurrently by spawning an interpretation thread.

Another improvement will be needed for the migration and creation of Messengers.

Although Messengers and threads do not differ significantly in their context switch issues, Messengers are much slower than threads in their creation. This is due to the fact that the complete image of an original Messenger must be copied into a new Messenger, whereas a new thread needs only a new copy of local variables as its own stack. However, if the original Messenger and its progenies are implemented to share the same byte code, the main overhead incurred by a Messenger creation will be the copying of messenger variables only. This may also be exploited to improve the migration of Messengers. If a Messenger has once visited a node, its byte code is saved and reused when the same Messenger returns there in the future. Hence, Messengers' weight will further be reduced and their performance will be much closer to that of threads.

MESSENGERS offers the important capability to develop an application incrementally since new Messengers can be dynamically injected and the underlying computational network is changeable at run time. However, once an application is completely developed, it may no longer requir this capability, in which case an approach using multithreaded processes would be more efficient. At that point, the flexibility provided by MESSENGERS becomes an unnecessary burden, which the user would like to trade for better performance. This may be accomplished by automatically converting the Messenger-based application into a fully compiled efficient distributed program, where multithreaded processes communicate with one another via ordinary send and receive primitives.

5 Conclusion

In this paper we have described a distributed computing environment for autonomous objects from two perspectives: flexibility and efficiency. The MESSENGERS paradigm offers the great flexibility to autonomous objects in terms of their navigational autonomy and dynamic composition. They may navigate through dynamically changing computational networks and schedule function invocations at each node according to their internal individual behaviors. As we have described in [FBD96], MESSENGERS may be viewed as a coordination paradigm for distributed computations using a logical network as the underlying computational medium. Such applications fall into two categories. The first involves intra-Messenger coordination, which regards a network as a control/data-flow graph where Messengers carry intermediate results and coordinate functions at each node. The second involves inter-Messengers coordination, which regards a network as a simulated environment or a database where each Messenger works as an independent computation entity. For both models, MESSENGERS system provides efficient support for coordination, so that Messengers can invoke precompiled functions and interacts with each other with little overhead.

The MESSENGERS system is now in operation at University of California, Irvine and has been used by students not only in our group but also in an undergraduate ICS class, and by a research group of the Community and Environmental Medicine Department. We have three future plans for MESSENGERS system: (1) implementing GVT features, (2) incorporating the ability of multi-threading and asynchronous I/O handling, and (3) developing a translator to generate static parallel threaded programs from Messengers so that an application could execute in native mode under multithreading. With these features, we will develop several distributed simulation programs which will be a new basis of our research on dynamic load balancing, proximity management and scalability [MB96].

Acknowledgment

We are grateful to Fehmina Merchant for sharing her MESSENGERS application programs with us, to Bozhena P. Bidyuk for her design and implementing of the MESSENGERS compiler, and to Leonard Megliola III for his assistance in the performance evaluations.

References

[BBB+93] C. Bouwen, J. Brann, B. Butler, S. Knight, et al. The dis vision - a map to the future of distributed simulation. Technical report, DIS Steering Committee, October 1993.

[BFD96] Lubomir F. Bic, Munehiro Fukuda, and Michael B. Dillencourt. Distributed computing usiing autonomous objects. *IEEE COMPUTER*, August 1996.

[CLR90] Thomas H. Cormen, Charles E. Leisorson, and Ronald L. Riverst. *Introduction to Algorithms*, chapter 25, Single-Source Shortest Paths, pages 515 – 549. The MIT Press, 1990.

[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.

[FBD96] Munehiro Fukuda, Lubomir F. Bic, and Michael B. Dillencourt. Intra- and inter-object coordination with messengers. *In Proc. of First International Conference on Coordination Models and Languages (COORDINATION'96)*, pages 179 – 196, April 1996.

[FMB+96] Munehiro Fukuda, Katherine Morse, Lubomir Bic, Michael Dillencourt, Edward Lee, and Daniel Menzel. A novel approach to toxicology simulation based on autonomous objects. *SCS Western Multiconference, Simulation in Medical Science*, pages 127 – 132, January 14 - 17 1996.

[Fuj90] Richard M. Fujimoto. Optimistic approaches to parallel discrete event simulation. *Transactions of the Society for Computer Simulation*, Vol.7(No.2):153 – 191, 1990.

[GM85] 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.

[Jef85] David R. Jefferson. Virtual time. *ACM TOPLAS*, Vol.7(No.3):404 – 425, July 1985.

[LDD95] Anselm Lingnau, Oswald Drobnik, and Peter Domel. An http-based infrastructure for mobile agents. *World Wide Web Journal - Fourth International World Wide Web Conference Proceedings*, December 14 - 17 1995.

[MB96] Susan L. Mabry and Lubomir F. Bic. Distributed biomedical simulation in speedes. *In Proc. of SCS Simulation Multiconference, High Performance Computing Symposium*, pages 36 – 41, April 1996.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry An Introduction*, chapter 3, Convex Hulls: Basic Algorithms, pages 95 – 149. Springer-Verlag, 1985.

[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, June 1994.

[ST85] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, Vol.32(No.3):652 – 686, 1985.

[Whi94] J. E. White. Telescript technology. Technical report, General Magic, Inc., Mountain View, CA 94040, 1994.