UNIVERSITY OF CALIFORNIA,

IRVINE

MESSENGERS: A Distributed Computing System Based on Autonomous Objects

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Munehiro Fukuda

Thesis Committee:

Professor Lubomir F. Bic, Chair

Professor Michael B. Dillencourt

Professor Daniel B. Menzel

1997

The dissertation of Munehiro Fukuda is approved
and is acceptable in quality and form
for publication on microfilm:

Committee Chair

University of California, Irvine

1997

DEDICATION

To

my parents

for all their support

Contents

List of Figures

List of Tables

Acknowledgments

I owe an especially deep debt of gratitude and appreciation to Professor Lubomir Bic, my thesis advisor and mentor for all his guidance and support. He has given me an invaluable education in the process of research. I would like to express my appreciation to my other committee members, Professor Michael Dillencourt and Professor Daniel Menzel for their intellectual support and continuous encouragement from the inception of my Ph.D. research. My gratitude is due also to my candidacy committee members, Professor Tatsuya Suda and Professor Alexandru Nicolau for early insights helpful to this study.

I also owe a great deal to my ICS colleagues for their contributions to the MESSENGERS project: Fehmina Merchant and Jason Cahill for sharing their application programs, Wa-Tor and Mandelbrot set generation, Bozhena Bidyuk for her design and implementation of the compiler, Alexander Thornton for his implementation of the assembler, Leonard Megliola for his assistance in the performance evaluations, Ken Tang for developing tools, Katherine Morse for her verification of the system and its library through her application development, Dominik Jergus for his porting work, and Susan Mabry for her many discussions with me. My appreciation is also due to colleagues from the UCI College of Medicine: Edward Lee for his valuable assistance in coding a toxicology simulation program on MESSENGERS and Cindy Smithies for using MESSENGERS for her medical simulation. Among them, Fehmina has been the most deeply involved in using MESSENGERS from its initial stage. Her patient verification made the system more robust for general release.

I would like to acknowledge those who have used MESSENGERS. I would like to thank ICS 198/199 undergraduate and ICS 247 graduate students for their developing application programs on MESSENGERS. Special thanks to Weiyu Tang for sharing his matrix multiplication program with me. I am also grateful to off-campus users from several universities. With deep thanks to Matt Simmons from Purdue for his information on porting MESSENGERS to other systems.

My life in graduate school was intellectually stimulating and a pleasure thanks to my friends: Tim McMahon, Oscar Aurelio, Pei Suen, Hisashi Norikane, Celio Albuquerque, Ana Azevedo, Miguel Campos, Veronica Leis, and Antenor Carvalho.

A special thanks to the Schaefers, Suzanne and John, who have been my friends, mentors, and guardians. Without them I might not have completed my graduate work.

I am very grateful to the ICS staff and support. Especially Susan Moore, Theresa Kolonecky, Candy Mamer, and Carol Rapp, who displayed admirable patience in guiding me through many administrative details. John Romine and Lynette Bellini gave me the SUN architectural information necessary for my dissertation and revived a backup copy of MESSENGERS at a crucial point.

I would like to acknowledge those who provided me with technical information from Japan: Hiroki Murata, Yasushi Negishi, Shiho Ogino, and Takashi Machida. I am also grateful to the ICS visiting faculty for giving me their research suggestions: Dr. Takashi Watanabe from Shizuoka University, Dr. Shinya Kobayashi from Kanazawa University, Dr. Tetsuo Hironaka from Hiroshima City University, Dr. Shinji Shimojo from Osaka University, Dr. Yuji Oie from Nara Graduate Institute of Technology, and Dr. Hiroshi Nakamura from Tokyo University.

This research was supported by: IBM Japan Overseas Graduate Study Scholarship, University of California Graduate Student Fellowship, and University of California Regents Dissertation Fellowship. I would like to especially thank Dr. Shigenori Shimizu and Ms. Kaoru Ueno for their assistance and encouragement through the IBM Japan Overseas Graduate Study Scholarship Program.

Finally, I would like to thank my family. My aunt, Kyoko Yoshida has often sent me pictures and cards in her letters which kept me in synch with the changing seasons back home. I thank my parents, Eiichi and Fumiyo and my sister, Yoshiko for their constant and gentle love and encouragement. Despite the geographical distance, I felt a strong family bond, which helped me through the difficulties I have faced.

Curriculum Vitae

Munehiro Fukuda

March 1986	**B.Eng.** in Information Engineering University of Tsukuba, Japan
March 1988	**M.Eng.** University of Tsukuba, Japan Masters Thesis: *Research on Hardware for Distributed Systems* Professor Takashi Masuda, Chair
1988 - 1991	**Associate Researcher** in Workstation Hardware Group, Tokyo Research Laboratory, IBM Japan
1991 - 1993	**Researcher** in Parallel Systems Hardware Group, Tokyo Research Laboratory, IBM Japan
Spring 1994	**Teaching Assistant**, Class: Principles of Operating Systems, Information & Computer Science, University of California, Irvine
June 1995	**M.S.** in Information & Computer Science University of California, Irvine
Spring 1996	**Teaching Assistant**, Class: Project in Operating Systems, Information & Computer Science, University of California, Irvine
June 1997	**Doctor of Philosophy** in Information & Computer Science University of California, Irvine Dissertation: *MESSENGERS: A Distributed Computing System Based on Autonomous Objects* Professor Lubomir F. Bic, Chair.

Field of Study

Distributed computing systems — Autonomous objects, virtual time computations, mobile agents, distributed programming coordination

Awards and Scholarships

1989	IBM Corporation First Patent Application Award
1992	IBM Japan Invention Achievement Award
1993-1996	IBM Japan Scholarship for Overseas Graduate Study
1995-1996	UCI Graduate Student Fellowship
1996	UC Regents Dissertation Fellowship

Publications

Fehmina Merchant, Lubomir F. Bic, Peter M. Borst, Malcom Corbin, Michael B. Dillencourt, Munehiro Fukuda, and Peter S. Sapaty.
Simulating Autonomous Objects in a Spatial Database.
9th European Simulation Multiconference, Prague, Czechk Republic, June 1995.

Munehiro Fukuda Katherine L. Morse, Lubomir F. Bic, Michael B. Dillencourt, Edward Lee, and Daniel B. Menzel.
A Novel Approach to Toxicology Simulation Based on Autonomous Objects.
SCS Western Multi-Conference (Simulation in the Medical Sciences,) pages 127-132, San Diego, January 14-17, 1996

Munehiro Fukuda, Lubomir F. Bic, Michael B. Dillencourt, and Fehmina Merchant.
Intra- and Inter-Object Coordination with MESSENGERS.
1st International Conference on Coordination Models and Languages - COORDINATION'96, in Springer LNCS 1061, pages 179-196, Cesena, Italy, April 15-17, 1996

Lubomir F. Bic, Munehiro Fukuda, and Michael B. Dillencourt.
Distributed Computing Using Autonomous Objects.
IEEE Computer, No. 8 Vol. 29 August 1996.

Munehiro Fukuda, Lubomir F. Bic, and Michael B. Dillencourt.
Performance of the MESSENGERS Autonomous-Objects-Based System.
1st International Conference on Worldwide Computing and Its Applications - WWCA'97, Tsukuba, Japan, pages A/1/3/1-A1/3/15, March 10-11, 1997 (also to appear in Springer LNCS)

Munehiro Fukuda, Lubomir F. Bic, Michael B. Dillencourt, and Fehmina Merchant.
Messages versus Messengers in Distributed Programming.
International Conference on Distributed Computing Systems - ICDCS'97, Baltimore, USA, May 27-30, 1997

Munehiro Fukuda, Lubomir F. Bic, Michael Dillencourt, and Fehmina Merchant.
MESSENGERS: Distributed Programming Using Mobile Autonomous Objects.
Journal of Information Sciences, North Holland, to appear in 1997

Munehiro Fukuda, Lubomir F. Bic, Michael B. Dillencourt, and Fehmina Merchant.
Distributed Coordination with MESSENGERS.
Special Issue on Coordination, Science of Computer Programming Journal, North Holland, to appear in 1998

Abstract of the Dissertation

MESSENGERS: A Distributed Computing System Based on Autonomous Objects

by

Munehiro Fukuda

Doctor of Philosophy in Information and Computer Science

University of California, Irvine, 1997

Professor Lubomir F. Bic, Chair

This research addresses the software architecture and techniques necessary to apply the philosophy of autonomous objects to general distributed computing. For this purpose, I implemented an autonomous-objects-based system called MESSENGERS, and evaluated its functionalities and performance.

Most distributed systems are based on the process-oriented approach where concurrent processes communicate with each other via messages. Autonomous objects are mobile entities, capable of navigating autonomously through the underlying computational network and performing independent and/or collaborative tasks at the nodes they visit. The autonomous-objects approach to distributed computing has several functional advantages over the conventional process-oriented approach. Autonomous objects can be programmed irrespective of the underlying network topology and hardware architecture. Computations based on autonomous

objects are fault-tolerant since objects can migrate or be dispatched to other correctly functioning nodes. Autonomous objects can also better encapsulate algorithms, thus narrowing the semantic gap between distributed algorithms and their implementation.

However, the autonomous-objects-based approach has only been applied to network service tasks where the communication latencies and the speed of human interaction overshadow the cost of object interpretation and the overhead incurred from coarse-grained and string-oriented computation. In order to extend the application domain to general-purpose distributed computing on LANs, it is essential to address the performance issues and thus support finer-grained parallelism.

MESSENGERS supports general-purpose computing while maximizing flexibility such as navigational autonomy and dynamic task decomposition. It is the first powerful coordination paradigm that incorporates three fundamental concepts; (1) navigational autonomy, (2) dynamic composition, and (3) temporal and spatial interaction. The system is realized as a Unix daemon process running a language interpreter at each machine. Each daemon process exchanges autonomous objects with other daemons and interprets them until they depart to another site.

This dissertation demonstrates both MESSENGERS' programming superiority to and performance competitiveness with conventional message-passing programming. It also proposes a new scheme for virtual-time synchronization among autonomous objects and shows that the proposed scheme achieves speed up over other synchronization schemes under certain conditions.

Chapter 1

Introduction

This research addresses the software architecture and techniques necessary to apply the philosophy of autonomous objects to general distributed computing. For this purpose, I implemented an autonomous-objects-based system called MESSENGERS, and evaluated its functionalities and performance.

Most distributed systems are based on a process-oriented approach: intelligence of computations is local to the concurrent processes communicating via passive messages. In contrast, the autonomous-objects-based approach embeds the intelligence of computations in mobile entities (autonomous objects) capable of navigating autonomously through the underlying computational network and performing independent and/or collaborative tasks at the nodes they visit.

An autonomous-objects-based approach to distributed computing provides several functional advantages such as machine-independent object description, bypassing faulty machines, and tight encapsulation of algorithms in objects. On the other hand, due to the nature of object interpretation, its recent applications have been limited to network service tasks which may not necessarily be processed faster than the speed of human interaction. However, in order to extend the application domain to general-purpose distributed computing on LANs, it is essential to address the performance issues and thus support finer-grained parallelism.

MESSENGERS supports general-purpose computing while maximizing flexibilities such as navigational autonomy and dynamic task coordination. The system is realized as a collection of Unix daemon processes running a language interpreter at each machine. Each daemon process exchanges autonomous objects with other daemons and interprets them until they terminate or depart to another site. The objects carry their programs written in a subset of C.

This dissertation discusses the features and the architecture of MESSENGERS to make the autonomous-objects-based approach to distributed computing a reality. It also analyzes MESSENGERS' qualitative and quantitative competitiveness with ordinary process-oriented approaches. The dissertation is further organized as follows: The remainder of Chapter 1 reviews existing approaches to distributed computing, highlights their limitations, and clarifies the objectives of this dissertation work. Chapter 2 details the MESSENGERS system in terms of architecture, language, and distributed features. Chapter 3 compares MESSENGERS with conventional message-passing systems from a qualitative point of view. Chapter 4 demonstrates that MESSENGERS' performance is competitive with other message-passing systems discussed in Chapter 3. Chapter 5 discusses how temporal synchronization among autonomous objects can be efficiently realized by our proposed scheme of virtual-time optimistic synchronization. Chapter 6 reviews related works and differentiates this Ph.D. research from them. Chapter 7 concludes the discussion and points out possible future extensions derived from this research.

1.1 Approaches to Distributed Computing

The popularization of LANs and the emergence of high cost/performance workstations have made distributed computing over clustered workstations possible. Naturally, the fastest implementation of distributed systems is using multi-threaded processes communicating through a low-level network interface, (e.g., a socket.) However, such systems depend on hard-wired machine-dependent programming, which makes it difficult to respond to the typical concerns of distributed computing. These concerns include network-wide control of computation instances, network-transparent resource access, and reusability and/or programmability of application code. With the growing prominence of distributed systems, it is critical to construct a transparent environment to coordinate computation instances over the system while maintaining high performance.

Several approaches have been studied and implemented by other projects including distributed shared memory, higher-level message-passing environments, and coordination languages/paradigms [BST89, And91]. Distributed shared memory (DSM) [Li88] gives each computation activity the illusion of a global shared memory. DSM attempts to replace all underlying network communication with memory accesses in the same way as on a physically shared memory architecture, so that programmers can use the shared memory paradigm for distributed coordination. For instance, a message broadcast to all computation instances is simply implemented by a memory write operation, and synchronization among them is realized with P/V operations. The paradigm is derived from a tightly-coupled shared-memory multiprocessor model where the user is in control of parallelism but does not need to deal with data distribution. The main drawback of DSM is performance, which is a high price to pay for the convenience of programming.

Using message-passing at the system level is extremely tedious and hence a number of higher-level environments have been proposed recently to facilitate this task. One of the best known representatives of such an abstract message-passing environment is the Parallel Virtual Machine (PVM) [GBD+94]. This provides a machine-independent dynamic collection of virtual processes, where all inter-process communication is uniformly realized by the same primitives *pvm_send* and *pvm_recv* regardless of location. PVM also provides various process control primitives to users. However, PVM's primary limitation is that all primitives are inlined in each application as function calls. This not only results in software engineering problems such as low understandability and maintainability of the application code but also requires a recompilation of the code upon a system reconfiguration.

Coordination paradigms, as represented by Linda [CG89], emphasize the separation of the computation and communication (or coordination). The interactions among execution instances are described in a coordination language separate from intra-instance computations that may be coded in ordinary programming languages such as C or C++. For that purpose, Linda provides the abstraction of a global logical space, called a *tuple space*, which consists of arbitrary data and possibly function tuples. Through operations on this *tuple space*, computation instances not only interact with each other but can also spawn new computations.

All these approaches to distributed computing are process-oriented. The intelligence of computations is contained in execution instances which exchange passive data with each other. Although the process-oriented approach can implement every conceivable coordination aspect (from a simple interaction to a complete process migration) by combining existing primitives, it has several functional disadvantages: (1) computations may be scattered over multiple execution instances, which increases the semantic gap between distributed algorithms and their implementations; (2) a faulty process may jeopardize an entire computation; and (3) a process cannot migrate to another machine in a heterogeneous network.

Contrasting with the process-oriented approach is one that incorporates intelligence into messages, which autonomously migrate over the network and access passive data local to the nodes they visit. This approach has been pioneered in Intelligent Email by having email navigate itself autonomously to whom it may be of interest [RD89], and has been used by various mobile agent systems. We refer to such messages as autonomous objects. This autonomous-objects-based approach is the one that we use to coordinate distributed computing. The next section discusses its superiority over the process-oriented approach and the hurdles to clear for applying the autonomous-objects-based approach to distributed computing.

1.2 Autonomous-objects-based Approach

Autonomous objects are mobile entities capable of navigating autonomously through the underlying computational network and performing independent and/or collaborative tasks at the nodes they visit. The autonomous-objects-based approach can address the inherent limitations of the conventional process-oriented approach [Wal82]. Autonomous objects are generally programmed in an interpretive language, and thus their code is independent of the underlying network topology and hardware architecture. Computations based on autonomous objects generally have greater fault tolerance since objects can migrate or be dispatched to other functional nodes. Autonomous objects can also encapsulate complete algorithms and thus narrow the semantic gap between distributed algorithms and their implementation. The autonomous-objects-based approach has two potential performance advantages: reduce the amount of network communication in the client-server model by having an autonomous object work at the server site on behalf of its client, and obtain inherent parallelism by propagating autonomous objects over a network.

However, the autonomous-objects-based approach has been applied primarily to mobile agents, which focus on network service tasks such as information retrieval and electronic commerce over WANs, (e.g., the Internet) where the communication latencies and the speed of human interaction overshadow the cost of object interpretation. Two systems, BPEM and WAVE, were among the first to apply the autonomous-objects-based approach to general-purpose distributed computing by seeking to harness the computational capacity of the underlying network to speed up some computational task. BPEM [BL87] is a computational model designed to facilitate the parallel processing of knowledge, represented by semantic nets. WAVE [Bor92] is a complete environment consisting of a specialized language to express arbitrary autonomous objects behaviors, and a run-time system of interpreters. Obviously, systems consisting of simple interpreters cannot be competitive with other approaches to distributed computing. Each of the following naive implementation features results in a large performance loss: using existing general-purpose interpreters to interpret all the code, spawning a new process to perform each new task, transferring entire object images during migration, and having a process other than the interpreter handle blocking socket operations. To date, there are no systems applying the autonomous-objects-based approach to general-purpose distributed computing while maintaining performance comparable with other approaches.

1.3 Research Objective

Applying the autonomous-objects-based approach to distributed computing includes not only the development of specific implementation techniques but a complete design of the software architecture and a precise formulation of features of the autonomous-objects-based system. Therefore, the research objectives are three-fold:

1. Designing the software architecture of an autonomous-objects-based system for general distributed computing and bringing an actual system into practical use;
2. Analyzing inter-object synchronization problems and designing a new virtual time synchronization scheme suitable to autonomous objects; and
3. Demonstrating that our scheme is competitive with other public domain distributed computing systems.

Chapter 2

MESSENGERS System

MESSENGERS is a coordination paradigm for distributed systems based on the concept of autonomous messages [BFD96], which we refer to as Messengers to differentiate them from ordinary passive messages. A Messenger is a self-contained object, consisting of a program and current state, including a program counter and local variables. The program, which is a combination of interpreted MESSENGERS code and native C code, describes the Messenger's behavior, which allows it to navigate autonomously through the underlying communication network and carry out computation in the nodes it visits.

The following sections present MESSENGERS' distinct features in terms of the fundamental concepts, the principles of operations, and the internal software architecture.

2.1 Fundamental Concepts

MESSENGERS is based on the following three fundamental concepts:

- *Navigational Autonomy* – A computation under MESSENGERS involves the dynamic construction of a logical network that is mapped onto the underlying physical network. The construction (and possible subsequent destruction) of this logical network is performed by one or more Messengers. Any Messenger is also capable of navigating through the logical network, replicating itself by following multiple links, and deciding what actions to take at each node. The navigational statements described in Section 2.3 permit the construction/destruction of and the movement through the local network.

- *Dynamic Composition* – In addition to constructing and moving around the logical network, each Messenger is capable of invoking and controlling the execution of ordinary C functions accessible on various nodes visited by the Messenger. Hence, each application may be viewed as a collection of ordinary C functions, whose execution, (i.e., invocation, termination, synchronization, and communication) is coordinated in both time and space by Messengers propagating autonomously through the computational network.
- *Temporal and Spatial Interaction* – Concurrent Messengers can coordinate their activities through a collection of specialized synchronization and communication mechanisms in both time and space to accomplish their mission.

We note that none of the above three concepts is unique to MESSENGERS. There is, however, no system that uses all three together for the sole purpose of application coordination. Notably, autonomous navigation is used by a number of systems to create "mobile agents" (see Section 6.1). While their primary purpose is to roam the Internet, MESSENGERS uses navigation as a primary structuring tool, which, in conjunction with dynamic functional composition, yields a powerful coordination paradigm for general-purpose distributed computation.

2.2 Execution Model

The MESSENGERS system involves three levels of networks as illustrated in Figure 2.1:

1. *Physical Network:* This is the lowest level network (a LAN or WAN), which constitutes the underlying computational resource.

2. *Daemon Network:* This network is superimposed over the physical layer. Each daemon node is a UNIX process running a MESSENGERS language interpreter. This involves selecting an arbitrary subset of the physical nodes to run the daemon process on and specifying arbitrary inter-daemon links. Each such link is mapped onto a path of one or more links in the physical networks and is used to guide the mapping of logical links and nodes at run time. The main purpose of the daemon network is to provide a virtual computational resource, thus isolating the applications from the details of the physical network topology.

   The daemon network topology is defined in a configuration file, referred to as *messengers_profile* that each daemon reads at system initialization. Specifically, this file declares all physical nodes comprising the system, associates them with daemon names, and defines for each daemon node which nodes are its logical neighbors, and selects one of them as a system master node that handles system-wide synchronization.

3. *Logical Network:* This is an application-specific computation network created on top of the daemon network. Multiple logical network nodes may be created on the same daemon network node, thus running on the same physical node, and they may be interconnected by logical links into an arbitrary topology.

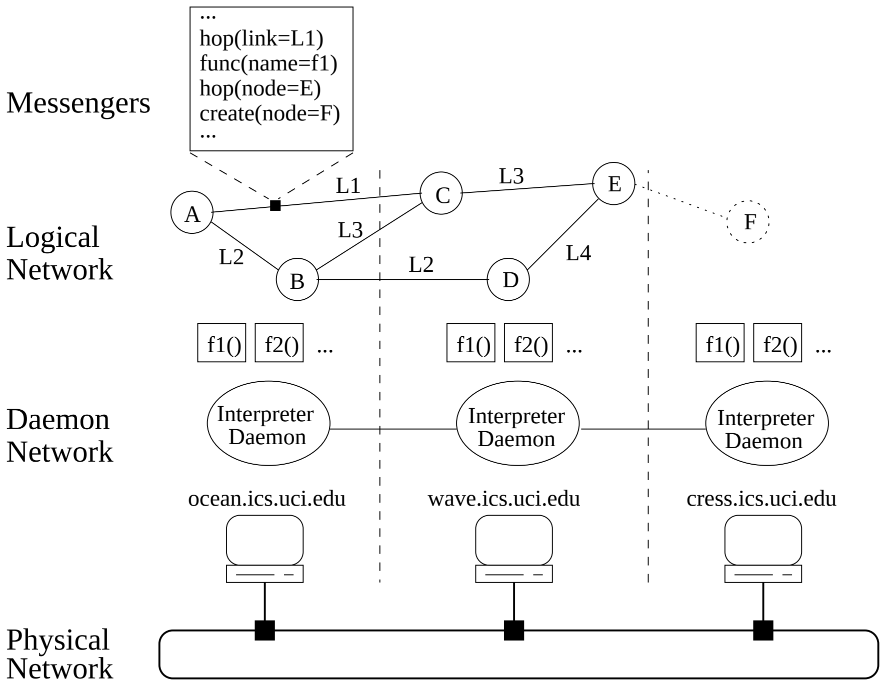


Figure 2.1. Execution model

It is this network that is used by Messengers when navigating through the system.

Messenger programs, referred to as *Messenger scripts*, are written in a subset of C (which we refer to as MESSENGERS-C) and are compiled into a low-level interpretive language 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.

Before giving the details of MESSENGERS-C language specification, we illustrate the behavior of Messengers graphically using Figure 2.1. This shows a Messenger in transit between two logical nodes, A and C. The Messenger's script fragment shows "hop(link=L1)" as the current statement that effected the transfer of the Messenger from the current node A along link L1. At the destination node, C, the Messenger's next statement, "func(name=f1)", will invoke the node-resident function f1. Upon returning from the function call, the next statement, "hop(node=E)", causes the Messenger to transfer to the new destination node E. From node E, the Messenger's next step is a *create statement*, which creates a new node, F, connected to the current node E. The link is unnamed and the physical node is unspecified, which causes the creation of the new node on a physical node chosen by the system. This allows the logical network to spread throughout the daemon network using system defaults or mapping information supplied in the create statement.

2.3 Language Specification

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

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");
...	...
$\longrightarrow$ S_i;	exec(file = "echo"; in = $node);
...	...

Figure 2.2. General form of Messenger script and an example

the current node. Access to *node* and *messenger variables* is achieved using ordinary assignment statements. *Network variables* are modified automatically and implicitly through navigational statements.

Figure 2.2 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 a computational, a navigational, or a task-coordinating statement:

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 an assignment statement.

Navigational statements:

A Messenger may create new logical links and nodes, change or delete existing nodes, and move arbitrarily through the network by either following links or jumping to specific nodes. The following is a high-level description of the navigation statements. Exact semantics is specified in [Fuk96b].

1. *create(node, link, weight, daemon)*: Create a new logical *link* leading to the specified *daemon* 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 *daemon* 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 daemon node chosen automatically by the system. Hence the last parameter allows the user to explicitly control the logical-to-daemon node mapping. For example, *create(node=a)* has the system choose a certain daemon node automatically, create a new logical node named 'a' on this daemon node, and move the requesting Messenger onto this node 'a'.

2. *hop(node, link, weight, daemon, NODE, LINK, WEIGHT)*: Cause the Messenger to be forwarded to *node* on *daemon* 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=*; daemon="pacific")* sends a copy of the Messenger to all nodes on the daemon node named "pacific" regardless of any logical links. Parameters in uppercase are used to change network variables while the Messenger is propagating to the destination nodes. For example, *hop(link=x, LINK=z)* changes in "z" all the link named "x" along which the Messenger navigates.

3. *delete(node, link, weight, daemon)*: 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 itself 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 daemon node, or being carried by the Messenger as it propagates.

1. *exec(file=filename [; in=argument] [; out=argument;] [nowait])*: The parameter *filename* is a Unix path name that specifies a file containing an executable C program. This must contain a *main*() function, which is spawned as a separate Unix process, i.e., operating concurrently with the invoking MESSENGERS. The arguments specified by *in* are passed to the new process at the time of creation and results are returned to the invoking Messenger via the *out* argument. If *nowait* is specified, the process may continue operating even after the parent Messenger has left the node (or has terminated). With this option, no results are returned to the invoking Messenger.

2. *func(file=filename; name=func_name [; in=argument] [; out=argument])*: Similar to *exec*, the parameter *filename* is a Unix path name specifying a file that must contain one or more executable C functions (but not necessarily a complete program with a *main*() function). The function specified by the parameter *func_name* is invoked and the arguments specified by *in* are passed to it. Unlike *exec*, the function executes as part of the current Messenger, that is, as part of the MESSENGERS interpreter process. The invoking Messenger is blocked until the function terminates and returns its results to the Messenger script via the *out* argument.

2.4 Distributed Environment

A distributed environment that enhances the basic capabilities of MESSENGERS-C is made available through a library of functions invoked by a *func* statement [Fuk96a]. Such an environment includes user-level object control, and inter-Messengers communication and synchronization. Unlike these library-supported features, logical-to-daemon node mapping is controlled by the combination of the *messengers_profile* file and the order in which the nodes are created. This section details the library call mechanism, the library-supported features, and the node mapping strategy.

2.4.1 Library Call

Library calls from the interpretation layer to the Unix environment are made using the *func* statement. Figure 2.3 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 a pointer to it, (i.e., *local in Figure 2.3) is passed to the C function. Any data type of constant value or variable such as *messenger*, *node*, and *network variable*s can be passed as arguments (by value). In addition to arguments, the C function is given two pointers (*node and *msgr in the example) to the *node variable*s area of the current node and to the *messenger variable*s area of the calling Messenger. Multiple values can be returned to the calling Messenger by invoking a library function, *back2messenger*. They are copied into variables listed after the *out=* keyword of *func* statement in the enumerated order. Hence, a C function can get

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);

C program
(invoked through a func statement)

General Form:	*Example:*
	typedef struct { .... } NODE; typedef struct { .... } MSGR; typedef struct {double x, y;} LOCAL;

func_name(node_ptr, local_ptr, msgr_ptr)
char *node_ptr, *local_ptr, *msgr_ptr;
{
....
statement;
....
back2messenger(pointers_to_ret_vals);
}
	func1(node, local, msgr)
	NODE *node, LOCAL *local, MSGR *msgr;
	{
	....
	m = local→x * local→y
	....
	back2messenger(&m, &n);
	}

Figure 2.3. Function interface

direct access to variables in the MESSENGERS interpretation layer and reflect its multiple computation results back to this layer without any data conversion or replication.

2.4.2 Object Control

Object creation (injection), termination, and context-switch in MESSENGERS are accomplished as follows:

Messengers Injection:

The injection of new Messengers is accomplished using the function *M_inject*. This loads a Messenger from a file containing the Messenger's behavior and injects it into a logical network. 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 user 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 Messengers is achieved by *M_duplicate* or *M_fork* which duplicate the calling Messenger inside the current node. The only difference is that *M_duplicate* has the duplicated Messenger start from the beginning of its script 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 navigation along 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 *M_kill(id)*, where *id* is the ID of the Messenger to be killed. The Messenger can obtain its own ID by *M_getmid* and a child Messenger ID upon spawning it by *M_duplicate* or *M_fork*.

Messengers Context Switch:

Messengers are non-preemptive. Their context switch does not occur until the current Messenger explicitly relinquishes control to another Messenger using either a navigational or a task-coordinating statment. This guarantees the non-interruptibility of a Messenger's execution and thus avoids spin lock (busy wait) at the application level. Therefore, when the current Messenger needs to resume another one, it must temporally leave the current logical node or call some function (which may be a dummy.)

We evaluate the qualitative and quantitative advantages of these three object handling primitives in Sections 3.1.1 and 4.1.

2.4.3 Node Mapping

MESSENGERS supports both explicit and implicit mapping of the logical network structure onto the daemon network and thus on the physical network. Explicit mapping is accomplished by specifying in each *create* statement the daemon node on which the new logical node is to be mapped. This gives the programmer complete control over the logical-to-physical mapping. Implicit mapping is performed by the system, but the user can control the mapping through appropriate use of the following parameters and constructs:

- The *size* of the daemon network. This determines the number of physical nodes potentially involved in the computation and hence the degree of parallelism.
- The *topology* of the daemon network. This controls the fan-out during logical node creation, i.e., the number of separate physical directions into which the logical network extends.
- The *creation threshold value* for each daemon. This determines the number of logical nodes created by the same Messenger (and its replicas) in any

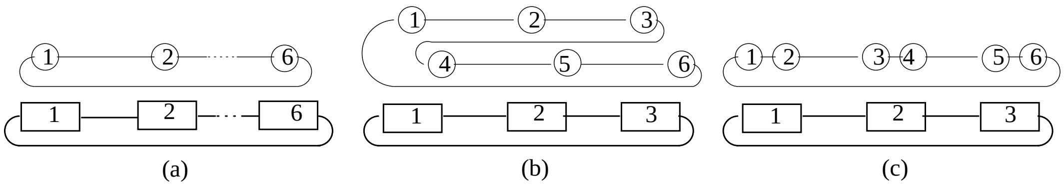


Figure 2.4. Mapping of a ring structure

given daemon before it must move onto a neighboring daemon. Hence the threshold controls the clustering of the computation.

The exact rule is as follows. After N logical nodes have been created on the same daemon node (where N is the threshold value), the interpreter forwards the next new logical node to one of its neighboring daemon nodes. The threshold counter is then reset to zero. The neighboring daemon nodes are selected in round-robin fashion.

- The *order* in which the nodes are created by the application. This, in conjunction with the threshold and daemon topology and size, determines the actual mapping of logical to daemon nodes.

To illustrate the above concepts graphically, consider first the mapping of a simple static one-dimensional structure, such as a ring, onto the physical network. Assuming the n logical nodes are created in ascending order starting from 1, we can produce a 1-to-1 mapping by selecting n daemons, interconnecting them into a ring, and specifying the creation threshold as 1. This forces each new logical node to be created on a new daemon, resulting in the mapping shown in Figure 2.4a. Figures 2.4b and c show how granularity can be increased by reducing the size of the daemon network. Figure 2.4b is the result of reducing the number of daemons by half while keeping the creation threshold still at one. The logical ring wraps around the daemon network twice. In Figure 2.4c, the threshold value has been doubled, thus resulting in a clustering of two logical nodes per daemon. Note that each of the mappings were accomplished by modifying the daemon topology/size or the creation threshold values, all of which are defined outside of the actual application program. Hence it is possible to adapt the application to different physical environments without reprogramming or recompilation.

Similar mapping control techniques can be applied to multi-dimensional structures, permitting the user to control the clustering of individual elements, columns, rows, or planes of such structures. For example, consider an $n \times n$ toroidal grid as illustrated in Figure 2.5. A Messenger can create this grid one row at a time. The first row is created by creating the node $(1,1)$, then $(1,2)$ (along with the link from $(1,1)$ to $(1,2)$), then $(1,3)$, (along with the link from $(1,2)$ to $(1,3)$), and so on. After $(1,n)$ is created, the messenger creates the link from $(1,n)$ to $(1,1)$. It then creates the node $(2,1)$, along with the link from $(1,1)$ to $(2,1)$. The remaining rows are created in a similar manner. The final step is creating the remaining vertical links. The mapping control parameters interact with this creation scheme as follows. If the threshold value is set to n, all logical nodes in a given row will be mapped onto the same daemon node. If the threshold value is set to 1 and there are n daemon nodes, each logical node in each row will be mapped onto a different daemon node, and all logical nodes in a given column will be on the same daemon node. It is also possible to (1) cluster multiple columns or rows, or (2) cluster any portions of rows or columns.

It is also possible to use the mapping parameters and constructs described above to produce a suitable mapping of a dynamic logical network structure. A common approach is to choose a granularity based on the amount of computation in each node and the cost of communication among nodes and then to attempt to

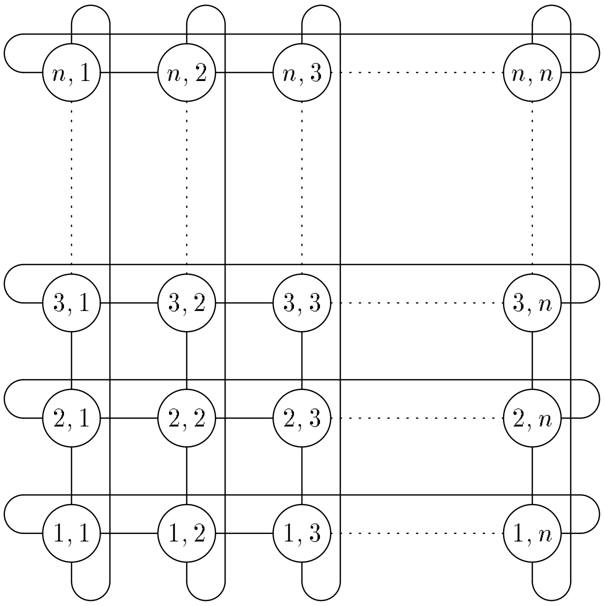


Figure 2.5. Mapping of a toroidal grid structure

spread the granules evenly over the physical resources as they are being generated by the computation.

2.4.4 Communication and Synchronization

Permitting Messengers to open direct communications would result in more efficient point-to-point burst data transfers. In that case, the only limiting factor would be the underlying communication hardware. However, keeping track of the communication/synchronization partners, which would continuously move over a network, would be prohibitive. Therefore, inter-Messengers communication and synchronization take place via indirect media shared among Messengers rather than through channels directly binding particular Messengers. Such media are *node variable*s for Messengers residing on the same logical node, mail boxes for remote communication, and global virtual time for temporal synchronization (both

Features	Local	Local and Remote
Communication	Assignment to node variables *node_var=msgr_var1; msgr_var2=node_var*	Mailbox messaging *M_send()/M_recv()*
Synchronization	Node varialbe synchronization *M_sched_node()*	Virtual time synchronization *M_sched_time_dlt/abs()*

Table 2.1. Local/remote communication and synchronization

local and remote). Table 2.1 summarizes how inter-Messengers communication and synchronization can be achieved locally and remotely. In the following, we discuss each implementation:

Shared Variables

When communicating Messengers are residing on the same logical node, data are exchanged by read/write operations on node variables. This is the same as with the shared memory programming model. Each Messenger chooses which variables it will have access to by specifying the appropriate file of node variables declarations (see Section 2.3.)

Mailbox Messaging

To access node variables on remote nodes, the Messenger could dispatch a special Messenger to simply carry data to/from the remote node. In most cases, this is unnecessarily complex to program and inefficient in terms of overhead. Therefore, mailbox messaging is provided to alleviate this problem. Each logical node has a mailbox. Through *M_send(node, data)* and *M_recv(node)* library functions, where *data* may be any type, communicating Messengers may send and receive any size of *data* to/from a mailbox associated with *node*. Since Messengers do not have to distinguish which logical nodes are remote, mailbox messaging is available for both local and remote communications.

Node Variable Synchronization

To eliminate the overhead of busy-waiting while testing for a shared *node variable* to satisfy a given condition, MESSENGERS provides a special synchronization function, *M_sched_node(node_variable, condition, messenger_variable, delta_ time)*, which suspends the calling Messenger until *node_variable* and *msgr_variable* satisfy a given *condition* or until the system-maintained virtual time has reached the current virtual time plus *delta_time*. The difference between *node variable* synchronization and ordinary spin wait is that the MESSENGERS daemon checks which *node variable* satisfies a synchronization condition and resumes the Messenger waiting on that variable. This test and resumption is performed only when the daemon switches between Messengers. More precisely, the MESSENGERS daemon checks the wait conditions of the Messengers suspended on a given logical node only when the last Messenger at this node has accessed any *node variables* and thus may have enabled some of the waiting Messengers.

Virtual Time Synchronization

The simplest way to synchronize remote Messengers is by a rendezvous in that these Messengers meet at the same logical node and synchronize on a *node variable*. Indeed, the rendezvous is very common among existing mobile-agents system and addresses simple synchronization problems. However, as discussed in Section 5.1.1, it is insufficient to implement a global synchronization among Messengers since they are spawned, terminated, and moved over the network dynamically. To mitigate this complexity, MESSENGERS offers a globally maintained virtual time to applications. The *M_sched_time_dlt(delta_time)* and *M_sched_time_abs(abs_time)* library functions allow a Messenger to schedule itself along the virtual time line.

The former wakes it up at the current virtual time plus *delta_time* and the latter resumes it at the absolute virtual time specified by *abs_time*. We detail how the virtual time synchronizations take place in several applications using these two library functions.

2.5 System Architecture

This section discusses the various implementation details 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 manipulates these structures, considering issues of performance.

2.5.1 Network Structure

As described in Section 2.2, MESSENGERS maintains two layers of network structures over the physical network: the daemon network and the logical network. This separation offers the following advantages:

1. The daemon links may be used to guide the mapping of new logical nodes to physical nodes (see Section 2.4.3.)
2. When logical nodes are created using the MESSENGERS' implicit node mapping strategy, the daemon links coincide with logical links. Except for direct hops, Messengers use logical links as their navigating paths, and therefore the daemon network reflects which daemons must communicate with each

other more frequently. MESSENGERS virtual time synchronization uses this knowledge to prevent rollback explosions (see Section 5.2).

3. A daemon node name may be different from the corresponding IP name. For instance, the physical node "pacific.ics.uci.edu" may be simply given "A" as a daemon node name. This makes Messenger scripts completely independent of the physical address space. The actual physical-to-daemon node mapping is achieved by associating the daemon name with the corresponding IP address in a *messengers_profile* configuration file (see Section 2.2.)

The following explains the implementation of the daemon and logical networks.

Daemon-to-Physical Network Mapping

A daemon manages each connection to another daemon with a *Physical Routing Block* (ROUTE). A ROUTE includes the assigned daemon's IP address, socket address structure and socket descriptor (see Figure 2.6.) The ROUTE is the only data structure affected by the underlying network protocol, (i.e., the socket in our implementation.) In addition to the physical information, each ROUTE also indicates if the daemon is a neighbor or non neighbor and if it is a system master or not (see Section 2.2.) Hence, the daemon network is defined by the collection of the neighboring daemons' mutual ROUTE information.

Logical-to-Daemon Network Mapping

A logical node is maintained by a *Node Control Block* (NCB), which contains the system-wide node address (consisting of the daemon node's IP address and the daemon-local node ID), logical name, 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 currently residing in this logical node and sharing this area.

A logical link within the same daemon node is implemented as a pair of LINK structures; one for each direction (even in the case when the link is undirected.) Each LINK contains the logical link name, its weight, and orientation. For example, in Figure 2.6, the physical node *laguna.ics* contains two logical nodes A and B with the corresponding system-wide node addresses 128.195.5.10.01 and 128.195.5.10.02. For simplicity, Figure 2.6 truncates the first three numbers, (i.e., 128.195.5) from their actual address and indicates them as 10.01 and 10.02 respectively. The nodes are interconnected by a logical link directed from A to B. This is recorded by the "+" sign in the LINK leading from A to B and a "–" sign in the other LINK. Both LINKs also contain the weight, 15. This implementation makes it uniform and easy to search an appropriate logical link from both the source and the destination NCB when a daemon interprets navigational statements.

Each logical link spanning two daemon nodes is split into two pairs of LINKs, one pair residing on each daemon node. They are connected on one side to their corresponding NCBs 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 also contains the remote NCB's system-unique node address as shown in Figure 2.6 in addition to its link name and weight.

2.5.2 Messengers Management

Each Messenger is managed in a *Messenger Control Block* (MCB) shown in Figure 2.7. The entry *ID* is the Messenger's ID local to the current daemon and

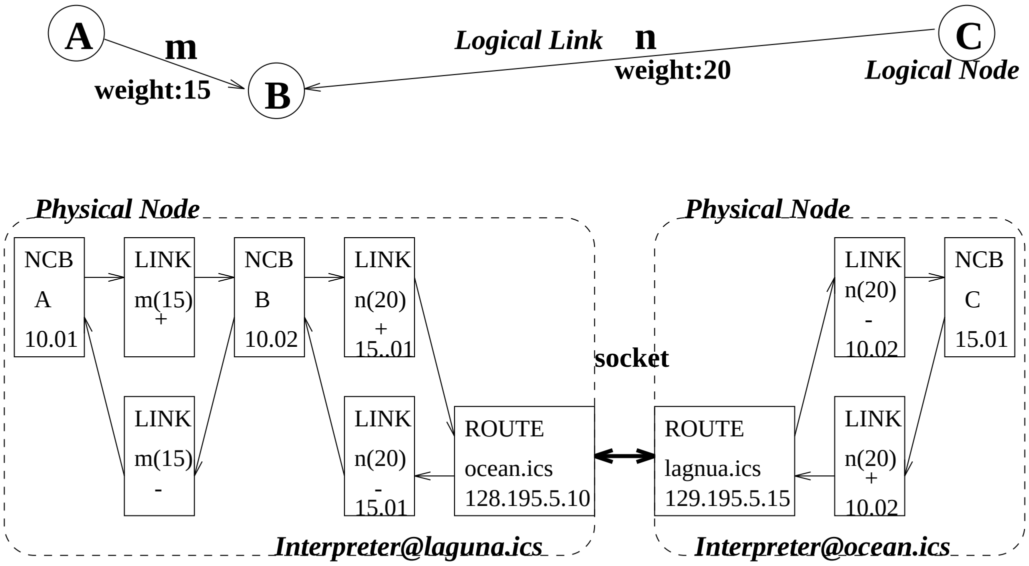


Figure 2.6. Implementation of logical network

may change when the Messenger moves to another daemon. The entry *program name* is the path name of a file containing the Messenger's byte code. The entry *code* points to the byte code area in memory. The entry *arguments* is a pointer to the area storing arguments passed when the Messenger is injected with *M_inject*. This area can include any number of arguments which may be of any data type. Figure 2.7 shows a Messenger which has only one argument and its data type is integer. The entry *messenger_vars* points to the area storing *messenger variables*. The example in Figure 2.7 shows that the area has two character-type *messenger variables* *a* and *b*. The entry *node* points to the area storing the name, address, and *node variables* of the current logical node. The last two entries *in_arg* and *out_arg* point to the input and output arguments for func statement.

The scheduling mechanism in each daemon node is realized using three types of queues: a *ready mcb queue* to include MCBs that are ready for interpretation, a *suspended mcb queue* to include MCBs that are waiting for the return from a

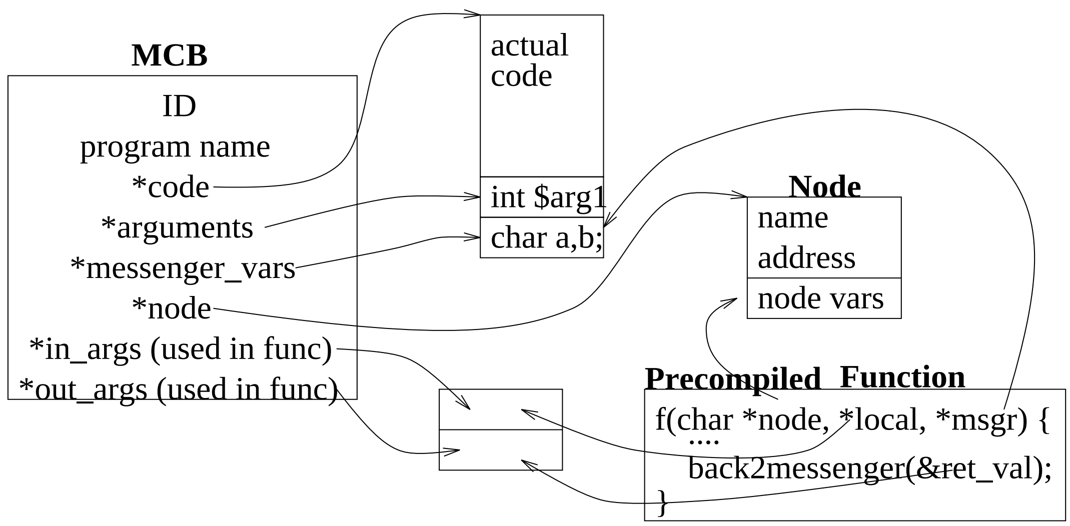


Figure 2.7. Messenger Control Block (MCB)

precompiled function, and *outgoing mcb queues*, each of which keeps MCBs of Messengers leaving for the corresponding destination daemon node.

Similar to threads, Messengers share their byte code if they are identical. When a new Messenger is injected with its file path name, the daemon checks if it has already loaded the byte code from the same file with the same last modification date. If so, it bypasses the file load operation and simply makes the injected Messenger point to the existing code area. Otherwise, the daemon loads the new byte code and keeps it for future reuse if it has enough memory space even after all Messengers sharing it are terminated or transferred.

2.5.3 Function Event Management

Precompiled functions are connected to the MCB as shown in Figure 2.7. They are retrieved from the file system and their connections are established when they are called for the first time by the Messenger on the current logical node.

Each function gets a pointer to the *node variables* area, the *messenger variables* area, and the two areas for input and output parameters.

The daemon handles a function call from a Messenger as an event to be scheduled. Such an event is recorded in an EVENT data structure consisting of the virtual time to resume the calling Messenger and several pointers: one to the calling MCB, one to the precompiled function, and one to the area storing function arguments. When a *M_sched_time_dlt(dlt_time)* or *M_sched_time_abs(abs_time)* library function is called, the virtual time slot of the EVENT it set to the current virtual time plus *dlt_time* or to *abs_time*. If a Messenger calls a function immediately, the EVENT's virtual time slot records the current virtual time. The daemon processes events when the virtual time reaches their scheduled time. The EVENT structure also stores the wait condition when a Messenger schedules its invocation on some node variable using *M_sched_node*. When an EVENT satisfies its wait condition, the daemon processes it regardless of its virtual time slot.

All scheduled EVENTs 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 earliest EVENT over all logical nodes of the given daemon. Figure 2.8 shows an example of the hierarchical splay tree structure of EVENT for 5 logical nodes. The merits of this scheduling queue are the following:

1. The entire set of EVENTs associated with the same logical node can be easily located and thus manipulated at once. For instance, they can be immediately

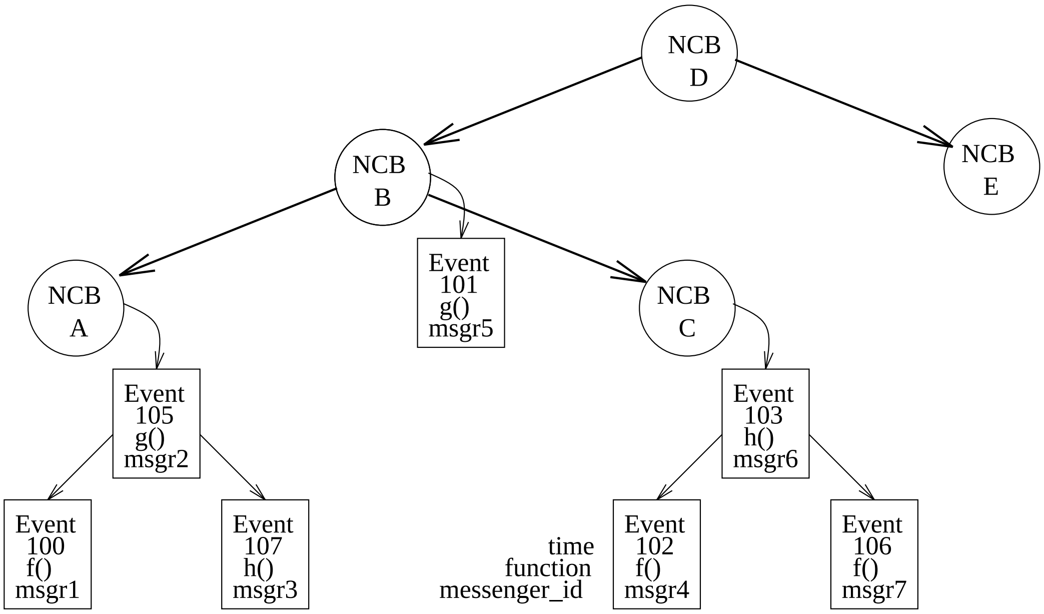


Figure 2.8. Hierarchical structure of event queues

canceled upon deleting their logical node or moved upon migration of this node.

2. When the MESSENGERS daemon detects during interpretation that a certain node variable has been accessed by the current active Messenger, it screens only the EVENTs associated with this node to check if there are any Messengers waiting on this node variable.

2.5.4 Communication Management

The MESSENGERS daemon communicates with others using the TCP/IP socket interface. A new socket connection is opened between any two daemon during each communication phase and is closed again when all Messengers have been exchanged. This is because Unix does not allow more than 64 socket descriptors to be opened simultaneously. The main problem of the socket interface is the upper limit on the send/receive queue buffer. When two processes send a lot of data to each other at the same time, they may easily saturate each other's buffers and fall into a deadlock. The flow control to avoid this congestion is implemented using two threads in MESSENGERS. Generally, a MESSENGERS daemon exchanges Messengers using only one thread referred to as a *master thread*. The socket send and receive operations are achieved by asynchronous write and read system calls respectively. Upon an incomplete write indicating that the buffer is full, the *master thread* checks if there are socket connection requests to receive Messengers from other daemons, and if so it spawns another thread, referred to as a *slave thread*, which is in charge of only socket receive operations. Upon an incomplete read, the *slave thread* switches to the *master thread* in order to resume suspended socket send operations. From then on, the *master* and *slave threads* alternate between writing and reading depending on the current contents of the buffer.

The daemon keeps track of which type of Messengers have been sent to and received from any remote daemon. Upon sending a Messenger, the daemon bypasses an actual transfer of its byte code if it has already sent the same type of Messenger to the same destination in the past. To implement this bypass operation, MCBs previously received from or sent to a remote daemon are recorded in the corresponding ROUTE data structure.

In addition, when a Messenger is sent to two or more logical nodes located on the same remote daemon, the daemon enqueues only one copy of the Messenger (with multiple addresses) into the destination ROUTE's *outgoing mcb queue*, and can therefore avoid redundant Messenger's byte code transfers.

In Section 4, we measure the performance of communication techniques discussed above: multi-threaded flow control, byte code sharing and caching, and omission of redundant Messengers to be sent.

2.5.5 External Interface

To permit communication between Messengers and external Unix processes, (e.g., a TCL manager), each MESSENGERS interpreter daemon prepares a Unix-supported shared memory area and a message queue. We refer to the former as an *interface buffer* and to the latter as an *interface port*. They have the following form:

1. *Interface Buffer:* This is a Unix-supported shared memory area consisting of two sub-spaces. One is used as the daemon's status indicator as well as its control panel. In other words, external process can not only check the interpretation status but also change the daemon's behavior. The other is available to users.
2. *Interface Port:* This is a set of Unix-supported prioritized 2k-byte message queues accessed through the Unix *msgsnd* and *msgrcv* functions. The top three queues are reserved for injecting new Messengers from the Unix shell and control processes spawned from Messengers through *exec*, while the other queues are available to users.

The usage of the external interface, including the procedure to plug them into external processes, the memory map of the *interface buffer*, and the reserved and available port numbers of the *interface port*, are detailed in [FBD96].

Since both the *interface buffer* and *port* are only interface media without any typing or structuring information, a synchronization protocol between Messengers and external processes must be agreed upon *a priori* at the application level.

2.5.6 Behavior of the Interpreter

Figure 2.9 describes the interpreter daemon's behavior. This alternates between the following two phases: (1) interpreting Messengers until there are no more ready Messengers, (line 3-25) and (2) executing scheduled functions (line 26-29). During each phase it also exchanges Messengers with other daemons whenever a certain combination of conditions are satisfied (line 23-24 and 30-31).

(1) Interpretation

Once the interpreter daemon picks up a ready Messenger, it continues processing its statements as long as they are computational (line 5-6 in Figure 2.9). Upon encountering a *hop* statement (line 9), the daemon traces all LINKs from the current node, (i.e., NCB) to locate the destinations. If the destination is a local NCB, the current Messenger (i.e., its MCB) is simply attached to the end of the *ready mcb queue* for this NCB (line 10). If the destination is a remote node (line 12) (i.e., a ROUTE), the MCB is enqueued to the end of the ROUTE's *outgoing mcb queue* to await its transfer (line 13). The other navigational statements, namely *create* and *delete*, perform the creation and deletion of the corresponding NCB respectively (line 7-8) followed by the same operations as for *hop*.

Upon encountering a *func* or *exec* statement (line 15-16), a new EVENT to schedule a function invocation is inserted into the current node's *future event queue*

```
daemon() {
    for(;;) {
        while ((mcb = ready_mcb_dequeue()) != NULL) {
            for (;;) {
                switch(mcb→instruction[mcb→counter++]) {
                case COMPUTATIONAL: compute(); continue;
                case CREATE:             create_new_ncb(); goto HOP;
                case DELETE:             delete_current_ncb();
                case HOP:                if(next_ncb == LOCAL) {
                                             relink(mcb, next_ncb);
                                             ready_mcb_enqueue(mcb);
                                         } else
                                             outgo_mcb_enqueue(mcb, next_route);
                                         break;
                case FUNC:
                case EXEC:               event→messenger = mcb;
                                         local_future_event_enqueue(event);
                                         susp_mcb_enqueue(mcb);
                                         break;
                }
                break; /* context switch */
            }
            if (communication_conditions == TRUE)
                exchange_messengers();
        }
        if (ncb = global_future_event_dequeue() != NULL)
            if (event = local_future_event_dequeue(ncb) != NULL) {
                execute(event→function); ready_mcb_enqueue(event→messenger);
            }
        if (communication_conditions == TRUE)
            exchange_messengers();
    }
}
```

Figure 2.9. Pseudo code describing interpreter's behavior

(line 17). This insertion may entail tree rotations of the hierarchical future event queue. The current MCB is then linked into the *suspended mcb queue.*

As described before, the interpretation of each Messenger is serialized between any two navigational or function-invocation statements. This means that a Messenger must explicitly relinquish control to another Messenger using a *create, delete, hop, func* or *exec* statement. 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 were programmed in each Messenger script, inter-Messenger synchronization had to be enforced at interpretation level, which would degrade the system performance. Instead, when synchronization is needed for cooperation, Messengers may use the *m_sched_node* function as described in Section 2.4.4. Hence, the CPU time will not be wasted unnecessarily.

(2) Function Execution

Function executions are initiated only when there are no more ready Messengers to interpret. This design is based on the following two observations. First, Messengers that 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 MCBs. Only when they are exchanged among daemons, are they sent with the source daemon's latest virtual time. At that moment, all Messengers inside this daemon are regarded to have the same new virtual time. Messengers that do not call a scheduling function at all must complete their execution at the current time and leave the daemon before its local virtual time is incremented. If the daemon were to execute a scheduled function before all ready Messengers terminated, it would increment its local virtual time and thus automatically shift all ready Messengers into this new virtual time.

For each function invocation, the interpreter daemon picks up the leftmost EVENT of the leftmost NCB leaf of the hierarchical event queue, and executes the function specified in this EVENT. If this function is not linked yet, the daemon first performs a dynamic link system call to load this function. The deletion of the EVENT reorganizes both levels of the future event queue by tree rotations so that the EVENT to be processed next is ready to be picked up as the leftmost node.

Figure 2.10 illustrates graphically how a Messenger is handled by the interpreter daemon from its injection to migration. A new Messenger is injected from an external process through an *interface port*, enqueued into the *ready mcb queue*, and waits for its interpretation. Once it is picked up from the *ready mcb queue*, the Messenger is processed as described above. If it schedules a function, the corresponding event is enqueued into the *future event queue* and the Messenger is entered into the *suspended mcb queue*. When the event is processed, the Messenger goes back into the ready mcb queue. This may be repeated any number of times. Eventually, the Messenger will terminate or perform a navigational command. In the latter case, it is enqueued into an appropriate *outgoing mcb queue*, and sent out through the socket interface in the next communication phase.

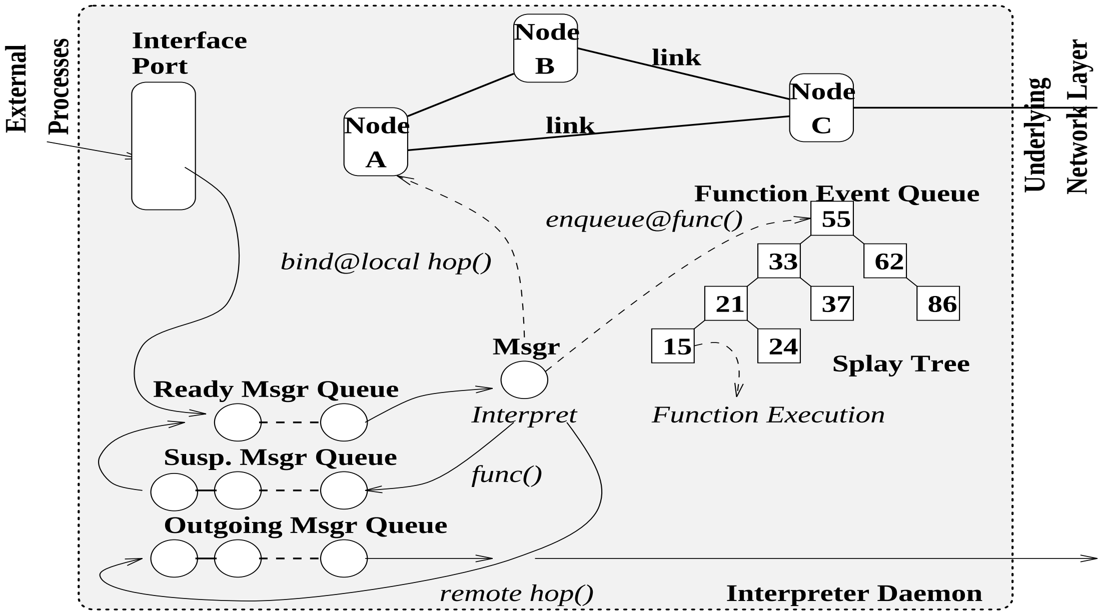


Figure 2.10. Illustration of interpreter's behavior

(3) Network Communication

The MCBs of outgoing Messengers are pooled in the *outgoing mcb queue* for each destination ROUTE until the interpreter daemon initiates the next inter-daemons communication phase, which is triggered when:

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

The parameters for the above triggering conditions, (i.e., items #2, #3, and #4) are specified by the user upon the daemon initialization and can be changed at run time through the *interface buffer*. These parameters control the amount of data transferred between daemons in one burst and may be adjusted for better performance. As described in Section 2.5.4, the daemon optimizes the transfers of Messengers, avoiding transfers of redundant byte code and congestions.

In Figure 2.9, the inter-daemon communication is handled by the *exchange_ messengers* function (lines 24 and 31). It opens a socket connection to each destination daemon and sends MCBs spooled in the corresponding ROUTE's *outgoing mcb queue* while receiving MCBs sent from other daemons through another socket connection. The received MCBs are immediately enqueued to the *ready mcb queue*. Therefore, as long as *mcb*s are received, they are interpreted with higher priority than the event processing. As explained before, this strategy is employed due to the two reasons: (1) processing and forwarding ready Messengers as soon as possible, and (2) avoiding shifting ready Messengers that do not call scheduling functions into a new virtual time.

Chapter 3

Qualitative Evaluation of MESSENGERS

This chapter focuses on the qualitative analysis of MESSENGERS in terms of its programmability, understandability, openness, and network-transparency. To show MESSENGERS' excellence in these aspects, we compare MESSENGERS with two ordinary message-passing programming schemes: multi-threaded communicating processes (simply referred to as *threads* in subsequent discussion) and PVM (Parallel Virtual Machine).

The chapter consists of two major parts: general observations and application-specific discussions. The former discusses how these three programming paradigms facilitate the control of parallel execution instances for the users and hide the complicated management of system resources from them. The latter examines through various applications how MESSENGERS can address the most important features necessary to code each of these applications more efficiently than the message-passing model.

3.1 General Observations

This section investigates the basic features that control execution instances and their communication. Such features are classified into two categories: the user interface and the underlying environment. The user interface provides the procedures for object control and the protocol for inter-objects communication. The underlying environment is the actual implementation of objects and their communication, which users may have to take into consideration when coding application programs. The following discussion is based on these two categories.

3.1.1 The Programming Perspective

We first discuss the application programming perspective of *threads*, PVM, and MESSENGERS, which includes the user control of object instantiation and communication over a "network". To provide network-transparency and scalability, object control and communication should not be described with physical parameters, such as IP addresses, but rather with high-level abstractions. We also focus on how the primitives are abstracted in each system.

Object Control Primitives

The following comparison examines distinctive features of each system to create, terminate, and switch between execution instances.

1. **threads:** Thread primitives are available only inside the same local processor, since the concept of thread is based on the shared memory programming paradigm. All requests to interact with remote threads must be sent to and

processed by the designated processors where the threads are working. There are no thread primitives to realize thread migrations or synchronizations over a network, and thus the users need to take care of them by combinations of local thread primitives and socket functions.

2. **PVM:** As its name suggests, PVM abstracts processors over a network and allocates new execution instances to processors chosen automatically by the system or as explicitly specified in each user program. However, once a new process is allocated to a particular site, it is hard to migrate it to an other site. Unlike threads, PVM primitives work for remote execution instances. Various synchronization primitives are available over a network including synchronous and asynchronous send/receive operations, group communication (multicast) and barrier synchronization.

3. MESSENGERS: Messengers are created and destroyed both implicitly and explicitly. The former case is achieved by MESSENGERS' navigational statements applied to a logical network. Copies of a Messenger are propagated to multiple remote nodes, which may be explicitly specified in the navigational statements or selected automatically by the system. The latter case is achieved by MESSENGERS' library functions, which are available only within the calling physical node and thus functionally similar to thread primitives. Synchronizations among Messengers are achieved by busy-waiting on node variables if they are on the same logical node or by using MESSENGERS virtual-time synchronizations on the same or on different nodes.

Table 3.1 summarizes object control primitives supported by *threads*, PVM, and MESSENGERS. As shown, navigational statements plus the system library of

Features	*threads*	PVM	MESSENGERS
remote create	not available	pvm_spawn()	create(), hop()
remote kill	not available	pvm_kill()	hop() + kill()
distributed synchronization	not available	pvm_barrier()	virtual time
migration	not available	not available	hop()
implicit create	not available	not available	create(), hop()

Table 3.1. Object handling primitives

MESSENGERS give the users various object control features that are superior to those of *threads* and PVM.

Communication Primitives

The following comparisons show the communication primitives featured by each system when remote objects communicate with each other.

1. **threads:** All communication with remote threads must be achieved via a low-level socket interface. TCP/IP abstracts network communication as a simple stream-oriented connection using read/write operations, however, it is available only for end-to-end connection-oriented communication. UDP, on the other hand, is connectionless and facilitates message multicasting, but it does not always guarantee a successful completion of messaging, which must be assured by the thread programs themselves. In either case, the destination of the socket interface must be specified using an IP address, rather than a receiver process ID, thread ID, or some other logical address.

2. **PVM:** The underlying idea of PVM communication is sending or multicasting data between processes in a uniform fashion regardless of whether the destination processes are on local or remote sites. Thus PVM provides general *pvm_send/mcast* and *pvm_receive* primitives, which relieve PVM programmers from using low-level communication setup procedures. Data sent

through these primitives is not necessarily a simple byte stream, like with sockets, but rather a packet consisting of various types of data. To allow this, PVM provides powerful packing/unpacking functions which automatically store and extract necessary bytes, integers, and double floating-point data in messages. This enables PVM to operate in heterogeneous network environments. The disadvantage of PVM communication primitives is that they need an explicit destination process or process group ID. This has it difficult to implement mailbox messaging, where messages could be sent to common mailboxes and later picked up by any processes.

3. MESSENGERS: Messengers may directly visit their destinations by navigational statements to communicate with remote Messengers. The navigational statements propagate the calling Messengers to various multiple destinations such as neighboring logical nodes, neighboring daemon nodes, or other sets of connected or disconnected nodes. Messengers automatically pack, carry, and unpack its *messenger variables*. However, these operations are supported for homogeneous networks only. Also, the current implementation only supports fixed-size *messenger variables* areas and thus all data must be carried at all times. To alleviate this problem, MESSENGERS also supports mailbox messaging in that Messengers can send only the necessary data to logical destination nodes where any Messengers can pick up these messages at any time.

Table 3.2 summarizes the communication primitives supported by each of the *threads*, PVM, and MESSENGERS systems. MESSENGERS covers all items considered in the table.

Features	*threads*	PVM	MESSENGERS
desitnation	IP address	process ID	logical node
multicast	UDP	pvm_mcast()	create(), hop(), delete()
packing/unpacking	bytes only	any data	automatic for homogeneous networks
mailbox	not available	not available	M_send(), M_recv()

Table 3.2. Communication primitives

3.1.2 The Implementation Perspective

This section focuses on how each of the *thread*, PVM and MESSENGERS systems actually implements new activities and their communications when invoked from application programs. From the application programming point of view, object control and communication primitives discussed in the previous section should work for an unlimited number of objects and infinite lengths of data transferred. In practice, application programmers must often be aware of the system resource limitations and implementation problems. Another concern is where to start new execution instances. Ideally, new objects should be mapped automatically to the least loaded machines, however, most systems will not provide such an optimized mapping. In the following, we discuss these system implementation problems in more detail.

Object Implementation

The following compares *thread*, PVM and MESSENGERS in terms of their execution instance sizes. It also shows the threshold number of objects created, which rapidly degrades the system performance and the absolute upper limit of creations.

1. **threads:** Since a thread includes only its own CPU status and thread information, such as ID, priority, and calling stack, its size is much smaller than

that of a process. Therefore, threads are generated, destroyed, and switched from one to another with much less overhead than processes. Although the thread stack areas are handled automatically by the default stack manager, programmers may still need to minimize stacks when creating a large number of threads. This is because a page called a "red zone" is inserted between neighboring thread stacks in order to protect these from corrupting each other [KSS96], and thus the number of threads spawned is still restricted by the available number of inserted pages. For instance, the maximum number of threads is limited to approximately 1000 in Sparc SLC and 5000 in Sparc SS5 running SunOS.

2. **PVM:** The execution instance in PVM is a process, whose control mechanism is quite heavy. This overhead may neutralize parallelism when new processes are dynamically created during the execution of an application or when the number of processes created exceeds the total number of processors. On the Unix system, the number of processes spawned from the same parent process is limited to approximately 20 to 30.

3. MESSENGERS: The size of a Messenger is highly dependent on the amount of Messenger variables, however, its control block (MCB) is quite small (350 bytes) and its script, whose size is generally only a few kilo-bytes, is shared by all Messengers originated from the same script file. Hence, Messengers are as light as threads. Since they are realized in the heap area of the daemon process, they do not waste memory space like threads' red zone, which permits more Messengers to be created than threads. The efficiency of Messenger control primitives is demonstrated in Section 4.

Flow Control

Network flow control becomes a major concern especially when the application involves a large number of workstations and cannot predict the communication flow *a priori*. The following discusses each system's control mechanism and some open problems.

1. **threads:** Application programmers must completely take care of any flow control problems using socket functions and I/O control libraries [Ste90]. Without careful flow control, applications may saturate socket send/receive buffers, packetize too small or large size of data, or be blocked on a connection establishment. In any of these case, user processes may fall into an I/O waiting state. For instance, assume that two remote threads simultaneously exchange data with each other. If the data size is beyond the socket receive buffer, both threads cause a deadlock. In this case, the thread on each side must be switched to another (new) thread which takes care of receiving the data transferred from the other side instead of the blocked thread. Such flow control is realized by (1) adjusting socket options, such as socket send/receive buffer size and the maximum data size sent at once, (2) making read/write system calls asynchronous, (3) spawning a new thread reading data from a saturated buffer or switching to another ready thread, and (4) handling interrupts from the ready network device so that the blocked thread can be woken up. It is burdensome to implement these schemes in each application.

2. **PVM:** Since the PVM daemon takes care of the network flow control, user processes can send and receive any size of data at any time without any concerns about congestions and deadlocks. However, all data transfers between remote processes are handled by daemon processes. PVM does not use Unix

shared regions or message queues, because their sizes may be too restrictive for some applications. This means that data transferred over the network must be copied into the sending daemon and from the receiving daemon. These copying operations may incur large overhead when applications send a large amount of data.

3. MESSENGERS: As described in Section 2.5.4, MESSENGERS spawns two concurrent threads, one for sending data and the other for receiving data whenever it detects that data previously sent or expected to be received has not actually been sent or received. Since Messengers are active inside each interpreter daemon's private address space, extra copying operations as seen in PVM are avoided. However, MESSENGERS, unlike PVM, has not yet been tuned up systematically for socket communications, and may therefore be slower than PVM when sending a small size of data. This performance is examined in Section 4.1.2.

Execution Instance Mapping

The following comparison introduces the execution instance mapping provided by each system and clarifies underlying problems to be addressed.

1. **threads:** Users are required to manually start new multithreaded Unix processes on explicitly selected machines. Threads are thereafter spawned inside each process through thread library functions.

2. **PVM:** When a PVM program spawns a new task, it can have the PVM daemon choose the most appropriate machine to balance the system load. The user can control this mapping scheme by giving a specific type of machine architecture or a particular host name. However, these restrictions are

associated with a given physical configuration and must be changed upon a system reconfiguration.

3. MESSENGERS: As described in Section 2.4.3, the user is in control of distribution of new logical nodes over the system using MESSENGERS' threshold parameter and the daemon topology. Since this parameter is given in a configuration file, Messenger scripts are independent of the mapping problem. MESSENGERS' navigational statements also permit users to specify particular machine names. This, however, introduces machine dependency and the Messenger scripts may have to be modified upon a system reconfiguration, similar to PVM.

3.2 Specific Applications

This section discusses MESSENGERS' superiority over the message-passing paradigm in software engineering, using several distinct application models: master-slave, information diffusion, network circulatory simulation, divide-and-conquer, multi-staged numerical computation, and individual-based simulation.

3.2.1 Master-Slave

We start our application-based qualitative analysis with the program in Figure 3.1, which shows a message-passing implementation of the common manager-worker (or master-slave) paradigm. This program is an adaptation of a PVM program [GBD+94], where unnecessary details, such as data packing/unpacking, buffering, etc, have been abstracted away for clarity. The program is divided into

```
 (1) manager() {
 (2)   for (i = 0; i < ntask; i++)
 (3)     child[i] = spawn(worker);
 (4)   for (i = 0; i < ntask; i++)
 (5)     send(child[i], next_task());
 (6)   while ((task = next_task()) != NULL) {
 (7)     res = recv(any_woker);
 (8)     i = who_sent(res);
 (9)     send(child[i], task);
(10)     process(res); }
(11)   for (i = 0; i < ntask; = i++) {
(12)     res = recv(any_worker);
(13)     i = who_sent(res);
(14)     kill(child[i]);
(15)     deposit(res); }
(16) }

(17) worker() {
(18)   while (TRUE) {
(19)     task = recv(manager);
(20)     res = compute(task);
(21)     send(manager, res); }
(22) }
```

Figure 3.1. Manager-worker using message-passing

two distinct functions. The behavior of the worker is very simple: it repeatedly receives a task (line 19), computes the result (line 20), and sends it back to the manager (line 21). The manager is more complicated. It first spawns a number of workers (lines 2-3) and sends each one a task, obtained using the function *next_task()*, to work on (lines 4-5). It then enters a while loop (line 6) during which it repeatedly receives a result from any worker that finished its work, determines the worker's ID, sends it the next task to work on, and deposits the received result for subsequent processing and output (line 7-10). When there is no more work to do, the manager enters the final loop (line 11) during which it collects the remaining results and kills the workers (lines 12-15).

The corresponding Messengers version of this program is shown in Figure 3.2. This shows the script of a single Messenger, which is injected into the INIT node

```
 (1) manager_worker() {
 (2)   create(daemon = *);                    /* create a link to each slave */
 (3)   hop(link = $link);                     /* hop back to the master      */
 (4)   for (;;) {
 (5)     func(name = "next_task"; out = task);
 (6)     if (task == NULL) break;
 (7)     hop(link = $link);                   /* hop to the slave            */
 (8)     func(name = "compute"; in = task; out =res);
 (9)     hop(link = $link);                   /* hop to the master           */
(10)     func(name = "deposit"; out = res);
(11) } }
```

Figure 3.2. Manager-worker using MESSENGERS

of some daemon. Its first course of action is to create logical nodes connected to the current node on every neighboring daemon (line 2). This also causes a replica of the Messenger to be created, each of which now executes in the new node while the original Messenger disappears; that is, it has cloned itself into a number of independent "workers". Each of the Messengers hops back to the original node by specifying the last traversed link, *$link* in a *hop()* statement (line 3). It then attempts to get a new task to work on (line 5). If successful, it hops back to its logical node—again by specifying the last traversed link (line 7)—where it computes the result (line 8), carries it back to the master node (line 9), and deposits it there (line 10). It continues hopping back and forth, each time solving a new task, until there is no more work to do. At that point it ceases to exist.

Even though both programs implement the same basic paradigm, the MESSENGERS program offers two important advantages:

- The Messengers code is conceptually much simpler as it only deals with one program. The message-passing version consists of two separate programs resulting in two types of concurrent activities—a manager who embodies most of the "intelligence" and a "mindless" worker. That is, the manager is in charge of the entire operation, including spawning the workers, supplying them with work, and killing them when all work is completed, while each worker only processes the tasks supplied to it. The Messengers version makes the workers "intelligent," thus eliminating the need for a manager. Each worker is capable of shuttling autonomously between the central node containing the tasks and its own node where it carries out its work. This is possible for two reasons. First, Messengers are mobile and hence can get their own work. Second, the logical network created initially (line 3) provides the "exogenous skeleton" for the application. The logical nodes can be thought of as work areas connected to the central node by corridors. Each worker can literally travel to the central place to take a new task and then carry it to its own work place to solve it. This distributes all the complexity of the central manager over the individual workers. The main reason why this is not possible with a non-navigational language is the inability to maintain a data structure on any node without some activity (process) associated with it. Thus, even if we endowed the worker processes with the necessary intelligence to autonomously request new tasks and deposit the results, (e.g., using RPCs), an active manager process would still be necessary to "guard" the central repository of tasks and partial results.

- The Messengers program is considerably shorter. This despite the fact that the message-passing version is only written in pseudo code—a lot of detail would have to be added to make this program run under PVM or any other system. The Messengers code, on the other hand, is fully executable. The reduction in code length is not the result of using a high-level language.

Messengers programming is comparable to message-passing in that both deal with parallelism explicitly. Specifically, our *create* is comparable to a *spawn* operation in PVM while a *hop* is similar to a *send/receive* operation. The reduction is due to our reformulation of the manager-worker paradigm that distributed the intelligence of the central manager over the workers, which is not possible using message-passing.

3.2.2 Information Diffusion

The control of large heterogeneous networks requires the ability to deal with a dynamically changing topology as physical nodes are dis- and re-connected, to query their status and to disseminate information to them. To illustrate the ability of MESSENGERS in this context, we consider the relatively simple problem of sending (diffusing) some piece of information (a message) to all currently operational nodes in the network, assuming there is no general broadcast facility. A standard solution to this problem using conventional send and receive primitives, taken from [And91], is shown in Figures 3.3. It uses neighbor set communication to send the message from one node to all its active neighbors.

Figure 3.3 shows the node program that must be running on each node participating in the diffusion. The diffusion is initiated by sending a message (msg) to any of the nodes. Upon receiving a message (line 2), the node sends that message to all its neighbors (lines 3-4). To avoid leaving extra messages buffered on the network and to correctly detect termination of the distributed computation, each node then waits to receive ($nneighbors - 1$) redundant messages sent by its

```
diffuse() {
  msg = receive();
  for (i = 0; i < nneighbors; i++)
    send(neighbor[i], msg);
  for (i = 0; i < nneighbors - 1; i++)
    msg = receive();
  deposit(msg);
}
```

Figure 3.3. Information diffusion using conventional message-passing

neighbors. These redundant messages replace the explicit acknowledgments that would otherwise have to be sent before the node programs could safely terminate.

Figure 3.4 shows a diffusion program using MESSENGERS. Given a logical network whose topology is identical to the underlying daemon network, the program can be injected into any of the daemon nodes, and spreads over the logical network. The diffusion is accomplished using the *hop* statement (line 3), which replicates the Messenger along all links incident on the departing logical node and thus moves each copy to each neighboring logical node, (i.e., each neighboring daemon node). Each receiving daemon node continues interpreting the next command (line 4), which determines if the node has already been visited before. If not, the node variable *visited* is set to TRUE to indicate that it now has been visited (line 6). The Messenger then continues executing its repeat loop (line 2), which again replicates it to all neighboring daemon nodes. If the condition on line 4 fails, indicating that the node has already been visited, the Messenger dies (line 7).

The MESSENGERS program offers the following advantages over the message-passing version:

- The message-passing program is not tolerant of node failures/disconnections. Node failures can result in expected messages not being received, which can

```
diffuse(msg) {
  for(;;) {
    hop();
    if (visited == FALSE) {
      func(name = "deposit"; in = msg);
      visited = TRUE; }
    else
      exit;}
}
```

Figure 3.4. Information diffusion using MESSENGERS

prevent the program from terminating. If nodes are reconnected during the running of the program, it can result in premature termination of the program at a node, leaving extra buffered messages on the network. These anomalies occur because each node program must decide when to terminate based on its interaction with its neighbors. The node program makes this decision by counting the number of messages it receives, and node failures or reconnections can disrupt the count.

The MESSENGERS program, on the other hand, always explores the current topology. If a node is disabled (before or after it has been visited), it does not prevent other nodes from being reached, provided the network remains connected. Similarly, a new node can simply start the interpreter and thus automatically become reachable by Messengers. The main reason for the increased tolerance of node failures is that a Messenger decides whether to terminate solely on the basis of whether the current node has been visited and does not have to count messages.

- The message-passing algorithm requires more data communication operations than the MESSENGERS algorithm, since each link in the spanning tree is traversed twice (by a message) in the message-passing algorithm and once (by a Messenger) in the MESSENGERS algorithm. Links that are not part of the spanning tree are traversed twice in both cases. The additional data communication overhead incurred by the message-passing algorithm is minor if the network is a dense graph, (e.g., a fully connected topology), but it may be significant if the network is a sparse graph, (e.g., a planar grid.)

- The message-passing program is useful only when *multiple diffusions* are to be performed, using the same algorithm. This is because initiating the message-passing computation requires sending the node program of Figure 3.3 to each node by some mechanism and then initiating the node program. If only one message is to be diffused, the mechanism used to send the program could be used to send the message itself to each node rather than the program. Changing the dissemination algorithm or running a different network application also requires the corresponding application programs to be sent to each node. With MESSENGERS, all application-specific information is carried by the Messengers while all nodes run the same (application-independent) interpreter, which runs continuously in every node.

3.2.3 Circulatory Simulation

This application illustrates how an application may be structured as a collection of functions, residing on different nodes of a network, where the invocation of these functions and the exchange of data among them is orchestrated by Messengers. The purpose of such an organization is to permit interactive manipulation of the model at run time. This may be accomplished by terminating a

Messenger as it passes through a node and injecting a newer version, thus altering the sequence of functions it will invoke. We consider a Toxicology simulation, which we have developed in collaboration with UCI's College of Medicine [FBDM96, FMB+96]. This program simulates the distribution over time and metabolism of inhaled Styrene by different organs of a living organism [RA84]. The basic approach is to map each organ onto a separate logical node as shown in Figure 3.5. 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, whose script is described in Figure 3.6. Initially, a Messenger containing this script is injected into the Lung node, where it computes the Styrene concentration using the amount of inhaled Styrene and the compounded amounts carried along the veinal links, `c_veinal` (initially zero). The Messenger then replicates itself along all arterial links (line 4), carrying the newly computed Styrene value to all the other organs. At each node, the Messengers trigger the execution of the appropriate function to compute the new Styrene concentrations for the current simulated time increment (line 5). Each Messenger then hops back to the Lung node along the veinal link (line 6) and adds its Styrene value to `c_veinal` (line 7). All but the last arriving Messenger terminate (lines 8-9). The only surviving Messenger continues its for-loop, thus repeating the same cycle with the new values. This continues for a number of cycles specified by the `break_point` parameter supplied at the original Messenger's creation.

Figure 3.7 shows the corresponding message-passing version. The program consists of two functions, *lung()* and *organ()*, that work as a master and a worker

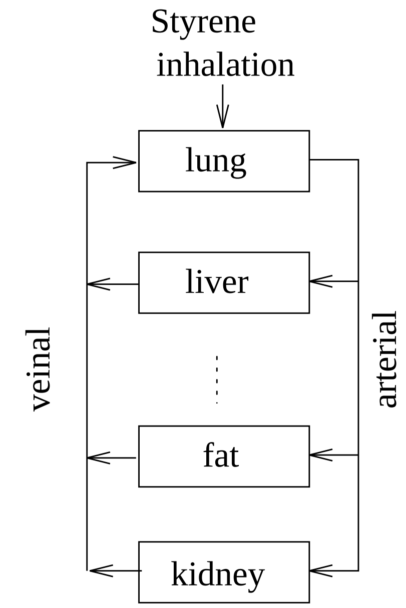


Figure 3.5. Styrene inhalation: as an example of circulatory simulation

```
styrene(break_point) {
  for(gvt < break_point) {
    func(name="new_lung_conc"; in=c_venial, inhaled_sty; out = styrene);
    hop(link = "arterial");
    func(name = "new_organ_conc"; in = styrene; out = styrene);
    hop(link = "venial");
    c_venial += styrene;
    if (i_am_not_the_last_messenger)
      exit;
    gvt++;}
}
```

Figure 3.6. Messenger script for styrene inhalation

```
 (1) lung(break_point) {
 (2)   for (i = 0; i < norgan; i++)
 (3)     organ[i] = spawn(organ);
 (4)   for(gvt = 0; gvt < break_point; gvt++) {
 (5)     new_lung_conc(c_venial, inhaled_sty, &styrene);
 (6)     for (i = 0; i < norgan; i++)
 (7)       send(organ[i], styrene);
 (8)     for (i = 0; i < norgan; i++)
 (9)       c_venial += recv(any_organ);
(10) } }

(11) organ() {
(12)   while (TRUE) {
(13)     styrene = recv(lung);
(14)     new_organ_conc(&styrene);
(15)     send(lung, styrene); }
(16) }
```

Figure 3.7. Styrene inhalation program using message-passing

similar to the master-worker problem described in Section 3.2.1. The *lung()* function first spawns a given number of child processes that execute the *organ()* function. Then, at the beginning of each simulation cycle, the *lung()* function computes its Styrene concentration from the amount of inhaled Styrene and the compounded amounts obtained from organ nodes in the previous cycle (line 5). The *lung()* function then multicasts the newly computed Styrene value to all organ nodes (line 7). Each organ node has invoked the *organ()* function that receives the Styrene value for each cycle (line 13), computes the new Styrene concentrations (line 14), and sends it back to the lung node. The *lung()* function sums up the new Styrene concentration from all the organ nodes (line 9) and then starts the next simulation cycle (line 4).

We can observe that MESSENGERS has two important capabilities:

- MESSENGERS provides for a clear separation between computations, as expressed by the individual node functions, and coordination, which is the order of function invocations and the transport of information among them; the latter is accomplished by Messengers. The message-passing version, on the other hand, mixes computations and coordination in the same code, and therefore does not permit such separation. This separation allows each layer to be designed and implemented separately. The Messengers do not need to know anything about the functions they invoke and, consequently, these can be supplied or changed at runtime. Similarly, the computational functions need not know when and by whom they will be invoked to perform their tasks. Such separation enables to *reuse* the coordination layer in the design of other similar models. For example, the above styrene inhalation model can be turned into a cardio-vascular simulation model [MB96] by replacing the functions computing styrene concentrations by corresponding functions computing blood pressures and/or volumes, and carrying those values between the organ nodes in the same way as the current styrene concentrations.

- MESSENGERS does not require the entire experiment to be set up *a priori*. Rather, each simulation run is composed dynamically as Messengers propagate through the network and invoke the various compute functions. Hence the simulation model is *open-ended* in terms of its functionality and can arbitrarily by modified or extended incrementally and interactively at runtime. It allows both the definition and the state of a simulation experiment to be modified at runtime by either substituting different functions to be invoked or by replacing the circulating Messengers by different ones as necessary. Hence intermediate results observed from the current simulation run may be used as immediate feedback to steer the ongoing experiment. For instance, when the MESSENGERS circulation pauses (for example as a result of a breakpoint), the user may terminate the current version of the Messenger, modify its composition, and reinject it. The Messenger could be modified to invoke a different function already stored at a node, (e.g., an alternative to `new_organ_conc`), or it could change the definition of `new_organ_conc`. On other hand, the message-passing version must be stopped and restarted from the beginning of its sequence whenever the user needs any modifications.

3.2.4 Divide-and-Conquer

In this section we consider a class of problems that require the dynamic construction of a data structure, such as a search tree, that is independent of the underlying physical network topology. The mapping of the data structure onto the physical network thus becomes an important issue.

We consider a divide-and-conquer algorithms for finding a convex hull of n points given in the plane [PS85]. As shown in Figure 3.8, the program divides the points into two subsets, containing $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ points respectively, such that all points in the first subset lie to the left of all points in the second subset along the x-axis. It then computes the convex hulls of the subsets recursively, using the same divide-and-conquer approach. The partial solutions are merged at each level as the program back-tracks from the recursion.

The easiest way to implement the above algorithm as a sequential program is to use recursion to construct and subsequently collapse the binary tree. In a distributed environment, recursion must be emulated by explicitly creating new processes and specifying the exchange of information among them. Figure 3.9 shows a possible distributed implementation using message-passing. Lines 8–10

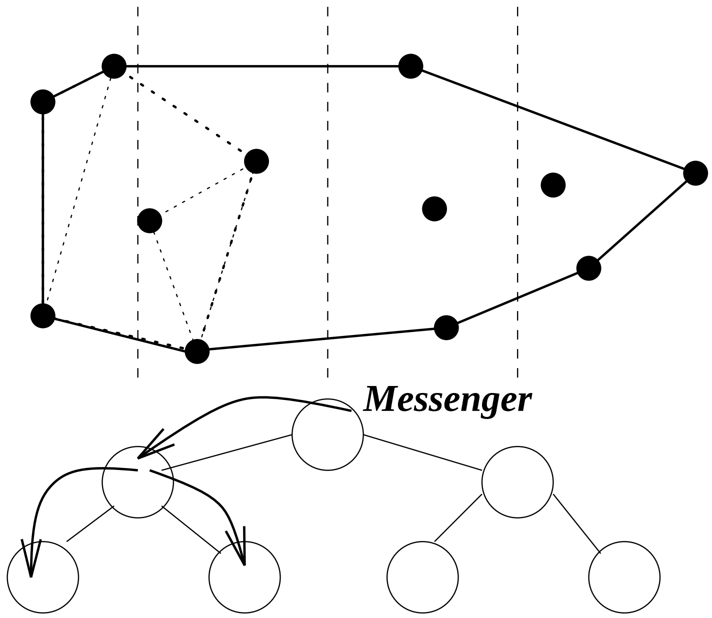


Figure 3.8. Convex hull: an example of divide-and-conquer algorithms

implement the subdivision of n points into two parts, each of which is sent to a newly created child process, *l_child* and *r_child* (line 6 and 7). The sending program then awaits the arrival of the partial results (lines 11 and 12), which it combines into a single result (line 13) and passes to its parent (line 16). If the tree branch reaches the bottom, the result is computed locally (line 15) and passed to the parent (line 16).

The corresponding MESSENGERS program is shown in Figure 3.10. Initially, a Messenger called *convex_hull* is created and passed n points as its parameter. The first *for*-loop (lines 2–7) constructs the binary tree. At each iteration, it creates two new links labeled "left" and "right", respectively, and replicates itself along these links (line 3). At the destination node, which is created at the same time as the new link, it determines whether it is the left or the right child and takes the appropriate half of the points for itself (lines 4–7). This is repeated until the number of points

```
convex_hull(points) {
  parent = parend_id();
  if (parent != VOID)
    recv(top_pt, tail_pt, level);
  if (level > 1) {
    left_process = spawn(convex_hull);
    right_process = spawn(convex_hull);
    mid_pt = top_pt + (tail_pt - top_pt)/2;
    send(left_process, points, top_pt, mid_pt-1, level/2);
    send(right_process, points, mid_pt, tail_pt, level/2);
    left_pt = recv(left_process);
    right_pt = recv(right_process);
    my_pt = merge(left_pt, right_pt);
  } else
    my_pt = convex(top_pt, tail_pt);
  send(parent, my_pt);
}
```

Figure 3.9. Convex hull program Using message-passing

is small enough to be handled locally in the current node (line 8), which produces the lowest level result *my_pt*. The Messenger then starts backtracking toward the root (line 10). The minus sign in the *hop* statement indicates that it follows the link ("left" or "right") in the opposite direction of its creation (indicated by the plus sign in the *create* statement). In each node the Messenger checks if it is the first to arrive there by examining the variable *other_pt* (line 11). If this is empty, the Messenger deposits the partial result it carries in that variable (line 12) and terminates itself (line 13). The second Messenger arriving at the same node from the other child node will detect that its sibling has already been there and thus continues its journey toward the root after having combined its own partial result (*my_pt*) with that of its sibling (*other_pt*) deposited at the node earlier (line 15). Eventually, a single Messenger will arrive at the root carrying the complete result.

The user can control the mapping of logical nodes onto daemon nodes using the constructs as described in Section 2. In particular, if the threshold value is set to 1, then each left child will be on the same daemon node as the parent, but each

```
(1) convex_hull(points) {
(2)   for (i = 0; i < max_level; i++) {
(3)     create(link = +"left", +"right");
(4)     mid_pt = top_pt + (tail_pt - top_pt)/2;
(5)     if ($link == "left") tail_pt = mid_pt-1;
(6)     if ($link == "right") top_pt = mid_pd;
(7)   }
(8)   func(name = "convex", in = top_pt, tail_pt; out = my_pt);
(9)   for (i = max_level; i > 0; i--) {
(10)    hop(link = -);
(11)    if (other_pt == NULL) {
(12)      other_pt = my_pt;
(13)      exit;
(14)    } else
(15)      func(name = "merge"; in = my_pt, other_pt; out = my_pt);
(16)  }
(17) }
```

Figure 3.10. Messenger script for convex hull

right child will be on a different node. This is because the number of nodes that has been created on the current daemon node is 1 before the *create* statement in line 3 is executed, so the right child is sent to a different daemon node. The counter is then reset, as described in Section 2, so the left child is then created on the same daemon node. This particular mapping scheme reduces computational contention, since the left and the right children computations run concurrently on different nodes. It also reduces communication overhead, since only one of the two links created on line 3 (and only one of the two corresponding hops on line 10) causes a Messenger to migrate between daemon nodes.

The two programs, while similar in their length and complexity, are very different in their programming style. The first is written from the point of view of stationary processes cooperating with each other in the construction of the search tree. The second program is written from the point of view of the Messenger, which constructs the tree by replicating itself at each node and then retraces its paths back toward the root. These two distinct point of view have a number of important implications:

- The abstract algorithm distinguishes three separate phases: (1) construct the binary tree by subdividing the work at each level; (2) compute the results at the leaf nodes; (3) merge all partial results by retracing the tree toward the root. The MESSENGERS program follows this intuitive description of the algorithm much more closely than the message-passing program. It clearly preserves these three phases in its structure: the construction and retracing of the tree corresponds to the two for-loops that surround the local computations at the leaf nodes. In contrast, the message-passing program does not preserve this structure. It recursively forks the function *convex_hull* to construct the tree but it must also handle the local computations and the merging of the partial results during each cycle. This is because there is no easy way to describe the backtracking from the leaves toward the root as a separate loop. Instead, each newly created process must wait for the partial results to be returned from its two children before it can terminate.

  Another way to look at this is to consider the *locus* of computation. With the MESSENGERS program, it is possible to step through the program sequentially. For any statement, there may be multiple instantiations of that activity but we never have to suspend an activity to consider another. In particular, when the execution on any particular branch reaches the "compute" function, there are no activities earlier in the program suspended and waiting for the current computation to return to them. Hence there is always a single locus of the computation. In contrast, the message-passing program can only be understood by considering the interactions among multiple concurrent activities. In particular, after spawning the two child processes, each parent remains active waiting for their replies. Hence, when tracing a child's executions, we also need to keep track of each of the suspended ancestors higher in the tree. It is this distributed locus of the computation that makes the writing and understanding of distributed programs so difficult.

- The message-passing program forces the user to deal with two separate concepts at the same time: the creation (and termination) of concurrent activities, and communication, i.e., the exchange of information among these activities. The former is accomplished using the *spawn* statement while the letter uses *send* and *receive* primitives. The algorithmic description, however, does not make any such distinction. It only prescribes the construction and subsequent retracing of a binary tree. The MESSENGERS program follows this abstract description very closely by using only the navigational commands *create* and *hop*. The *create* operation is used during the tree construction and accomplishes both the creation of new nodes and the passing of the necessary parameters (including the locus of the computation) to these nodes. The *hop* operation then achieves the backtracking, including the merging of all the partial results. The smaller semantic gap between the abstract algorithm and the MESSENGERS implementation makes the programs easier to understand.

- To control the mapping of the message-passing program onto the physical network, the *fork* statement needs the location information for the child process. For instance, the corresponding function in PVM is *pvm_spawn* that must be given a string including a machine name or an architecture group name, otherwise PVM automatically selects an available machine to run a spawned process. With the MESSENGERS implementation, the actual mapping is controlled by the creation threshold value and the daemon network topology as explained above.

3.2.5 Numerical Computation

Distributed numerical computations involving matrices include repetitions of an identical stage or (an identical sequence of stages) where each processor shuffles or rotates its assigned sub-block of the matrices at the end of the each computation. Such data-centric computations are frequently used in the message-passing model. We demonstrate MESSENGERS' competitiveness even for this type of computations.

The block matrix multiplication is one such example, which has been used to demonstrate PVM's programming. Given square matrices A, B, and C, $C = AB$ is computed by a grid of $m \times m$ tasks, denoted as T_{ij} (where $0 <= i, j < m$), each of which initially contains blocks A_{ij}, B_{ij}, and C_{ij}. As shown in Figure 3.11, at each stage k (where $0 <= k < m$), the tasks on the diagonal ($t_{i(i+k)}$) first send their block $A_{i(i+k)}$ to all the other tasks in row i, all tasks then calculate and add $A_{ij} \times B_{ij}$ to C_{ij}, and finally each task t_{ij} sends B_{ij} to its adjacent task $t_{(i-1)j}$, thus rotating the matrix B along its column. The final result is obtained after m repetitions of this cycle.

Figures 3.12 and 3.13 show the programs using message-passing and MESSENGERS respectively.

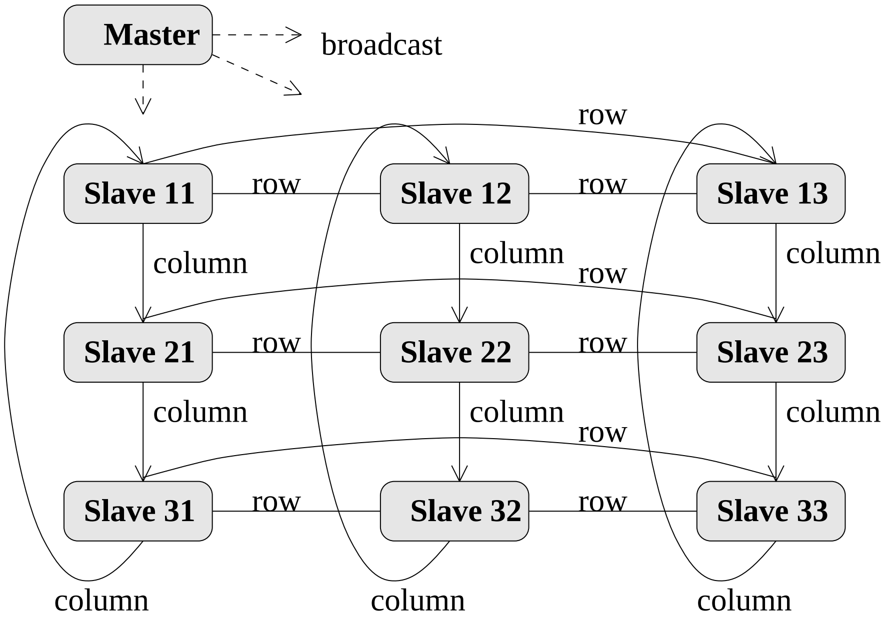


Figure 3.11. Matrix multiplication: an example of numerical computation with barrier synchronizations

```
matrixi_mult(blk, m, i, j) {
  join_group("mmult", get_pid());
  if (parent_id() == VOID) {      /* master process */
    for (i = 0; i < m; i++)
      for (j = 0; j < m; j++)
        child = spawn(matrix_mult, blk, m, i, j);
  } else {                        /* slave processes */
    for (k = 0; k < m; k++)       /* make a multicast group */
      myrow[k] = pid_in_group("mmult", i*m+k);
    for (k = 0; k < m; k++) {
      if (j == (i + k) % m) {
         buf = pack(A, blk/m);
         multicast(myrow, buf);   /* multicast A to all nodes in row */
      } else {
         buf = receive();
         unpack(buf, A, blk/m); }
      multiply(A, B, C);          /* Cij = Aij * Bij */
      buf = pack(B, blk/m);
      send(pid_in_group("mmult", (i-1)*m+j), buf); /* rotate B */
      buf = receive();
      unpack(buf, B, blk/m);
} } }
```

Figure 3.12. Matrix multiplication using message-passing

```
 (1) matrixi_mult(m) {
 (2)   master = $address;
 (3)   procs = m * m;
 (4)   for (k = 0; k < m; k++) {
 (5)     sync = 0;
 (6)     hop(daemon = *);          /* go to all slave nodes */
 (7)     if (node.j == (node.i + k) % m) {
 (8)        func(name="copy"; in=node.A; out=msgr.A); /* multicast A to all */
 (9)        hop(node = ,$address; link = "row",);     /* nodes in the row   */
(10)     } exit;
(11)     func(name = "multiply");   /* Cij = Aij * Bij */
(12)     hop(node = master);
(13)     if (++sync != procs)       /* barrier synchronization */
(14)       break;
(15)     sync = 0;
(16)     hop(physical = *)          /* go to all slave nodes */
(17)     func(name = "copy"; in = node.B; out = msgr.B);
(18)     hop(link = -"column");     /* rotate B to column i - 1 */
(19)     func(name = "copy"; in = msgr.B; out = node.B);
(20)     hop(node = master);
(21)     if (++sync != procs)       /* barrier synhcornization */
(22)       break;
(23) } }
```

Figure 3.13. Matrix multiplication using MESSENGERS

The message-passing version first has the master process spawn slave processes, passing the matrix size, the number of the slave processes, and assigned row/column numbers to each slave(line 3-6). Upon creation, each slave process first registers its ID in the “mmult” process group (line 2), finds other processes’ IDs in the same row from “mmult” and stores them in the *myrow* process ID array (line 9). It then starts an actual matrix multiplication: the slave processes on the diagonal multicast their block A to all the other slaves in the same row (line 11-16); all tasks then calculate $A \times B$ (line 17); and finally each task rotates its B along its column (line 18-21).

The MESSENGERS version assumes that a two dimensional grid has been created. A Messenger injected into the master node broadcasts itself to all slave nodes (line 6). The Messengers on the diagonal hop with A to all the other slave nodes in the same row, including itself ($\$address$) (line 7-9). All Messengers then calculate $A \times B$ (line 11), hop to the master node to synchronize together (line 12-14), hop back to the slave nodes (line 16), rotate their B along their columns (line 17-19), and hop back to the master node to synchronize again (line 20-22).

The main difference between these two codes is how the barrier synchronizations at each stage are achieved. As shown in Figure 3.12, the message-passing model achieves synchronization implicitly by matching message send and receive primitive pairs: multicast and receive (line 6 and 8), and send and receive (line 10 and 11). On the other hand, there are no sends/receives in MESSENGERS. Rather, the matrix subblocks move among the nodes autonomously, each carried by a Messenger. Hence synchronization must be performed explicitly. The simplest solution is to use a centralized manager which serves as a barrier synchronization for each cycle.

This difference in synchronization has two important implications for the design and use of the numerical application.

- The message-passing version must consider two control sequences controlled by an *if-then-else* statement (line 11): one for handling a data send (line 12-13) and the other for a data receive (line 15-16). On the other hand, the MESSENGERS version is completely data-centric and thus such a data transmission is coded in a single sequence of operations (line 8-9).
- Message send/receive pairs in the message-passing model need at least explicit destinations. This means that such destinations must be re-specified when the user changes the number of participating processors. In MESSENGERS, once an $m \times m$ grid is created over a network, a Messenger computing

$C = AB$ does not have to know explicit destinations but simply follow links. Thus, the Messenger script shown in Figure 3.13 is reusable for any size of grid.

Section 4.2.5 reveals the tradeoff between barrier synchronization costs and Messenger script's reusability. We will also demonstrate in Section 5.3.4 how the explicit synchronization using a centralized manager can greatly be simplified by scheduling the application along the virtual time line maintained by the system.

3.2.6 Individual-Based Simulation

In this section we consider a class of applications structured as a collection of entities interacting with one another within a two- or three-dimensional virtual space. Typical examples of such applications are interactive battle simulations [Com94], particle-level simulations in physics [HE88], traffic modeling [Res94], and various individual-based simulation models in biology or ecology [HDP88, Vil92, HW92]. 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 or an ant colony, while each individual has only a very limited local knowledge of the "problem" and a set of simple rules to follow [Res94]. Finally, advanced graphics and animation applications, especially those involving large numbers of individuals, such as the animation of a flock of birds, have also taken advantage of spatially-oriented individual-based modeling [Rey87].

To illustrate the programming style differences in this class of applications, we first consider the simulation of a group of particles performing a random walk on

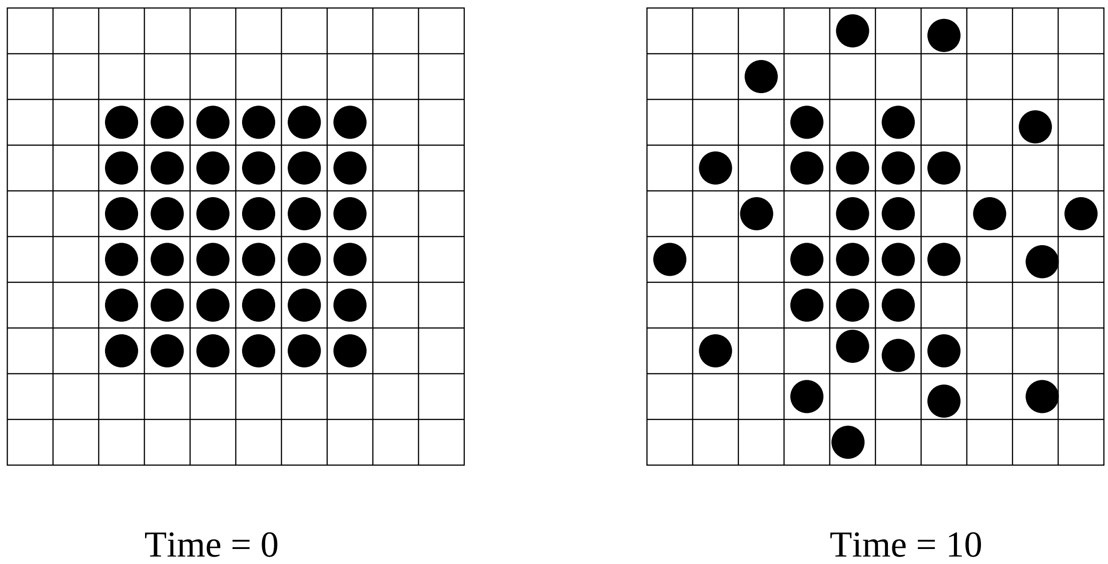


Figure 3.14. 2D random walk: a simple example of individual-based simulation

a 2D grid space as shown in Figure 3.14. During each step, each particle randomly selects and moves to its next position. Since only one particle can occupy any grid cell, collisions with other particles must be resolved. This is based on particle IDs. When two or more particles attempt to occupy the same cell, the one with the higher ID is permitted to stay while the others must return to the original places and choose other available positions.

Figure 3.15 shows a Messenger script describing the behavior of each particle. Assuming that a 2D logical grid has been created over a network, the particle decides on one of four direction: north, east, south, and west (line 3) and hops to the direction (lines 7-11). If a particle with a smaller ID or no particle is currently residing in the cell (line 13), the particle records its own ID as the resident ID (line 15) and waits until the simulation time denoted as *gvt* is incremented (lines 16-19). If, in the mean time, someone with a higher ID arrives before the *gvt* changes (line 18-19), the particle must move back to the original place (lines 20-22) and repeat the same sequence with another destination. This repetition also occurs when a particle detects another particle with a higher ID when it just enters a cell (line 20). The *gvt* must change after all particles find a cell. To guarantee this, each particle records in a certain *node variable* its hopping out of a cell (line 6 and 21), hopping into another cell (line 12 and 23), and its waiting for a new *gvt* value (line 14), so that the *gvt* is incremented when all particles fall into a wait state.

The *gvt* is incremented one by one and maintained by a Messenger separate from the particle Messengers. This *gvt* management Messenger propagates itself to all the slave nodes to assure that there are no in-transit Messengers or active Messengers. It then hops back and increments *gvt*, and again visits all the nodes to updates each node's local virtual time.

There are two possible implementation of this application using message-passing. One is to instantiate execution entities each representing a particle in the same way as the MESSENGERS version. The execution instances may be threads or processes. The difficulty with this implementation is the migration of such execution instances over a physical network. Threads or processes must be explicitly terminated at the source node, transferred over the network, and recreated at the destination node. Obviously the program is much longer and more complicated than the MESSENGERS' version. If the processes do not migrate but only compute their new position within the simulated space, then collision detection becomes a major bottleneck, since every process must exchange information with all other processes during every step.

The other implementation is to run a single process at each physical node, which is assigned a portion of the simulation space. Particles residing on the

```
random_walk() {
  for (i = 0; i < sim_time; i++) {
    func(name = "random"; in = seed; out = dir, seed);
    for (j = 0; j < 4; j++) {
      org_node = $address;
      func(name = "check"; in = m_id, "out");
      switch (dir) {
        case N: hop(link = +"vertical"); break; /* N = 0 */
        case E: hop(link = +"horizontal"); break; /* N = 1 */
        case S: hop(link = -"vertical"); break; /* N = 2 */
        case W: hop(link = -"horizontal"); break;} /* N = 3 */
      func(name = "check"; in = my_id, "in");
      if (resident_id < my_id) {
        func(name = "check"; in = my_id, "sleep");
        resident_id = my_id;
        while (gvt == i + 1) {
          func(name = "M_stall");
          if (resident_id > my_id)
            break; } }
      if (residen_id > my_id) {
         func(name = "check"; in = my_id, "out");
         hop(node = org_node);
         func(name = "check"; in = my_id, "in");
         dir = (dir + 1) % 4;
      } else break;
} } }
```

Figure 3.15. 2D random walk using MESSENGERS

subspace a process takes care of are represented as passive data and exchanged among the neighboring processes when they move beyond the current subspace boundary. Each process computes the next positions of its own particles and sends them to appropriate neighboring processes if their positions are beyond its assigned subspace. Figure 3.16 shows this implementation. Each process maintains its particles in the *local_list*. The function call *test(local_list, gvt)* picks up a particle whose current simulation time denoted by *lvt* is the same as the *gvt* (line 5). The particle decides the next position (line 6) and computes the new xy coordinates (line 12-16). If its xy coordinates are out of the current process's subspace, the particle is enqueued into the appropriate list containing all particles to migrate to the same destination (line 17-21). Otherwise, the particle is temporally enqueued into the *tmp_list*. Each process then exchanges particles with its four neighbors by calling the *migration* function (line 22), which stores incoming particles into the *local_list* (line 40-43). After merging the *tmp_list* to *local_list*, the process checks collisions for each particle (line 24-31). A particle with a higher ID occupies a new cell and forces particles with lower IDs to return to the original cells if they are on the same cell. The function *backoff* enqueues a particle into the list including all particles to return to the same original process (line 26 and 31). Each process repeats the above operations until there are no more particles still moving at the current *gvt*, and thereafter proceeds to the next simulation cycle.

Obviously, the message-passing code is much larger than the MESSENGERS script because of two major reasons. One is that the message-passing version must implement the *migration* function to migrate particles that is handled by the MESSENGERS daemon and thus need not be included in the MESSENGERS script. The other is that the message-passing version needs to decide if each particle's new

```
node_prog() {
  for (i = 0; i < sim_time; i++) {
    init_j(local_list);
    if (test(local_list, gvt) != NULL) {
      while ((cur = dequeue(local_list, gvt)) != NULL) {
        cur->dir = (cur->j++==0) ? random(cur->seed) : (cur->dir + 1) % 4;
        if (cur->j == 4) {
          cur->lvt++;
          enqueue(local_list, cur);
          continue; }
        cur->org_x = cur->new_x; cur->org_y = cur->new_y;
        switch(cur->dir) {
          case N: cur->new_y = cur->org_y + 1; break;
          case E: cur->new_x = cur->org_x + 1; break;
          case S: cur->new_y = cur->org_y - 1; break;
          case W: cur->new_x = cur->org_x - 1; break; }
        if (cur->new_y > max_y) enqueue(n_migr_list, cur)
        else if (cur->new_x > max_x) enqueue(e_migr_list, cur);
        else if (cur->new_y < min_y) enqueue(s_migr_list, cur);
        else if (cur->new_x < min_x) enqueue(w_migr_list, cur);
        else enqueue(tmp_list); }
      migration();
      merge(local_list, tmp_list);
      while ((cur = dequeue(local_list, gvt)) != NULL) {
        if (place[cur->new_x][cur->new_y].resident_id < cur->id) {
          bakcoff(place[cur->new_x][cur->new_y].particle);
          place[cur->new_x][cur->new_y].resident_id = cur->id;
          cur->lvt++;
          enqueue(tmp_list);
        } else
          backoff(cur); } }
      migration();
      merge(local_list, tmp_list);
} } }

migration() {
  send(n_neighbor, n_migr_list);
  send(e_neighbor, e_migr_list);
  send(s_neighbor, s_migr_list);
  send(w_neighbor, w_migr_list);
  recv(n_neighbor, local_list);
  recv(e_neighbor, local_list);
  recv(s_neighbor, local_list);
  recv(w_neighbor, local_list); }
```

Figure 3.16. 2D random walk using conventional message-passing

position is out of the current subspace and spool particles to migrate to the same remote process, which is just replaced with a *hop* statement in the MESSENGERS script.

Before discussing the MESSENGERS' superiority over the message-passing programming, we will introduce another individual-based simulation example whose algorithm is much more sophisticated than the 2D random walk in terms of the following two aspects and thus expected to achieve more speed-up:

- In the 2D random walk, Messengers repeat local hops frequently although they are still in the same physical node. This is because a logical node is assigned to each single cell and thus each daemon node has many logical nodes. Such local hops can be eliminated by having a single logical node manage multiple cells and mapping it to a daemon node.
- Unnecessary remote hops, (i.e., hopping back when collisions occur) can be also reduced by providing the neighboring physical nodes' boundary information to each node. This visibility extension will be handled by a separate space managing Messenger rather than the particle Messengers.

Wa-tor 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 3.17 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 and shark is implemented as a Messenger, which carries its own behavior.

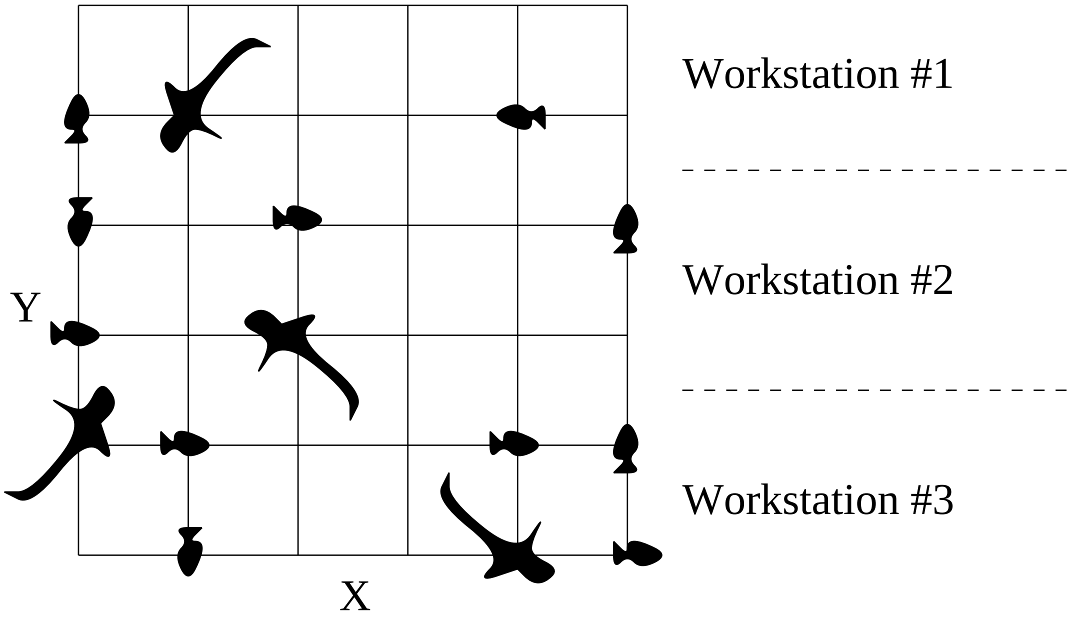


Figure 3.17. Wa-Tor: a more complicated example of individual-based simulation

Figure 3.18 shows the MESSENGERS version of Wa-Tor. The implementation consists of three Messenger scripts. The first describes the behavior of the fish, each of which is programmed to "live" for a number of steps corresponding to the simulation run (line 2). At each step, the fish computes its new cell position (line 3). Each fish coordinates its movement with its nearest neighbors, so the *upd_pos* function must find the neighboring fish and examine their motion vector. Since the radius of visibility may extend into a neighboring cell, the function *upd_pos* function must have access to the boundary strips of the two neighboring nodes (north and south). All the information of cells mapped to the current daemon and the boundary strips are maintained in *node variables*. Lines 6–9 accomplish the migration of the fish; when its y-coordinate falls outside of the current node's area of responsibility, the fish transfers itself to the corresponding neighbor using a *hop* statement, where the plus or minus sign indicates the direction of the link to be traversed.

Another Messenger called, called *ocean_manager* (line 12) is instantiated at each daemon node and exchanges the boundary strips, which must be made visible to the *upd_pos* function. The extraction of the local cells (line 13 and 14) is achieved by checking *node variables* containing boundary cell information.

Initially, the simulation is started by creating the simulated body of water. This is accomplished by a separate Messenger (line 20), which is passed the logical size of the simulated space (*ocean_size*) and the number of cells into which the space is to be subdivided. Each cell is created as a logical node connected to its north and south neighbor using directed links (line 22). For each cell, the Messenger computes the node's area of responsibility (the *min_y* and *max_y* coordinates, line 23) and it injects the *ocean_manager* Messenger into that node (line 24). The desired number of fish Messengers could also be injected into the nodes by the ocean manager or these could be injected separately at the start of the simulation. The simulation run is complete when all fish Messengers have ceased to exist, which may be detected by examining the number of Messengers present in each node.

Figure 3.19 shows the corresponding message-passing version. It consists of a node program that is started at each physical node. Each physical node also has a list, *local_lst*, which contains all the fish instances mapped onto this node. The simulation loop is repeated by all nodes for a given number of times (line 2). Each time the variable *cur* is set to point to the first fish on the list (line 3). The subsequent while loop then steps through the list, updating the position of each fish for the current iteration step (line 5). Similar to the MESSENGERS version, the *upd_pos* function must have access to the boundary strips of the two neighboring nodes (north and south). These two areas, named *n_bndry_strp* and *s_bndry_strp*, respectively, are exchanged periodically among neighboring nodes (lines 17–22) and

```
fish() {
  for (time = 0; time < sim_run; time++) {
    func(name = "upd_pos"; in = n_bndry_strp, s_bndry_strp);
    if (time % bread_period == 0)
      func(name="M_fork");
    /* migration */
    if (my_y < min_y)
       hop(link = +);
    else if (my_y > max_y)
       hop(link = -);
  }
}

ocean_manager() {
  /* boundary strip exchange */
  extract n_bndry_strp from local_cells;
  extract s_bndry_strp from local_cells;
  func(name = M_send, in = n_ngbhr, strp_size, n_bndry_strp);
  func(name = M_send, in = s_nghbr, strp_size, s_bndry_strp);
  func(name = M_recv, out = n_bndry_strp, str_size, n_nghbr);
  func(name = M_recv, out = s_bndry_strp, str_size, s_nghbr);
}

ocean_creator(ocean_size, n_cells) {
  for (i = 0; i < n_cells; i++) {
    create(link = +);
    func(name = "compute_my_cell"; in = ocean_size, i; out = min_y, max_y);
    func(name = "M_inject"; in = "ocean_manager");
  }
}
```

Figure 3.18. Messenger script for Wa-Tor

are used by the *upd_pos* function if necessary. When the new position of the current fish is determined, it is checked against the node's assigned area of responsibility. If its y-coordinate falls below the node's southern border, *min_y* (line 6), the fish is removed from the current list and placed on a special list, *s_migr_lst*, containing all the fish to migrate south (line 7). Lines 8–9 perform the analogous operations for a north-bound migration. The actual migration takes place at the end of the while-loop, once the position of each fish has been updated. Each node sends the lists of migrating fish to its two neighbors (lines 12–13) and, in turn, receives the corresponding lists from these neighbors (lines 14–15), which it merges into its local list (line 16). The final task to be performed at each simulation step is the exchange of the boundary strips among neighboring nodes. Each node determines which of its fish lie in the corresponding north and south boundary strip (lines 17–18) and sends these to the corresponding neighbors (lines 19–20). It then receives the corresponding information from the same neighbors (line 21–22).

The main difference between the message-passing and MESSENGERS versions for individual-based simulation is that with the latter the intelligence is decentralized and embodied in each particle while with the former it is centralized in a single process on every node. This philosophical difference has a number of important practical implications:

- The application deals with two separate issues: the behavior of the simulation entities and the management of the underlying space. These are separated clearly in the MESSENGERS implementation. The simulation space is created by a Messenger such as *ocean_creator* in Wa-Tor and may persist until MESSENGERS system is shutdown. Hence the simulation entity Messenger such as *random_walk* and *fish* can be injected at any simulation time. In

```
node_prog(){
  for (i = 0; i < sim_run; i++) {
    cur = local_lst;
    while (cur != NULL) {
      upd_pos(cur, local_lst, n_bndry_strp, s_bndry_strp);
      if (cur->y < min_y)
         move *cur from local_lst to s_migr_lst;
      elseif (cur->y > max_y)
         move *cur from local_lst to n_migr_lst;
      cur = cur->next;
    }
   /* migration */
    send(n_migr_lst) to n_nghbr;
    send(s_migr_lst) to s_nghbr;
    recv(n_migr_lst) from n_nghbr;
    recv(s_migr_lst) from s_nghbr;
    merge(local_lst, n_migr_ls, s_migr_lst);
   /* boundary strip exchange */
    extract n_bndry_strp from local_lst;
    extract s_bndry_strp from local_lst;
    send(n_bndry_strp) to n_nghbr;
    send(s_bndry_strp) to s_nghbr;
    recv(n_bndry_strp) from n_nghbr;
    recv(s_bndry_strp) from s_nghbr;
  }
}
```

Figure 3.19. Wa-Tor using conventional message-passing

contrast, the message-passing version where a single process runs at each physical node combines the individual entity behavior and the simulation space management. Unlike MESSENGERS the simulation space disappears at the same time when the program completes its execution.

- 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 (1) introduce another set of instances on the space (using the *minject* function); (2) introduce new entities with a modified behavior (by modifying the Messenger script before injecting it); and (3) introduce arbitrary new entities of a completely different type. All of the above can be achieved at runtime, without recompiling or even halting the ongoing simulation, due to the interpretive nature of Messengers. Hence the paradigm facilitates interactive and incremental software development and use.

- The message-passing implementation requires that each node program be told who its neighbors are. Whenever the physical topology changes, the neighboring information needs to be recomputed and explicitly supplied to each node program when it is loaded. This requires the use of a different language, e.g., a shell script, to perform the mapping, or it must be done manually. With Messengers, the same paradigm is used for all tasks. The mapping may be controlled implicitly by selecting the desired number of daemon nodes and their creation threshold to control the clustering of logical nodes. Hence, unlike the message-passing program, the number of logical nodes does not have to match the number of physical nodes.

3.3 Summary

This Chapter has shown that the MESSENGERS programming yields a number of important properties that are much more difficult to achieve with a conventional message-passing model. These include the ability to compute in unknown or dynamically changing network topologies, and the ability to modify or extend the applications' functional capabilities at runtime. Furthermore, as the examples presented in this section demonstrate, MESSENGERS programs generally result in a smaller semantic gap between the abstract algorithms and their actual implementations, which makes them easier to construct, understand, and maintain.

Chapter 4

Quantitative Evaluation of MESSENGERS

This section compares the performance of MESSENGERS, *threads*, and PVM. Each subsection corresponds to that in Section 3.1. The first two subsections show the competitiveness of MESSENGERS' object and network control with *threads* and PVM. The following subsections evaluates MESSENGERS' performance for all the applications discussed in Section 3.1 and reasons about the results from the view points focused on in the first two subsections: (1) the number and size of objects spawned, and (2) the volume, flow, and frequency of network communication.

The performance evaluations have been conducted under four configurations summarized in Table 4.1. The configuration level 1 (or SLC-10E) is the slowest one where Sun SPARC Station SLC's (16MB memory each) are connected by a 10Mbps Ethernet. The configuration level 4 (or SS5-100S) is the fastest one where Sun SPARC Station SS5's (32MB memory each) are connected by a 100Mbps collision-free network switch. Between the levels 1 and 4 are the configuration level 2 (or SS5-10E) where Sun SPARC Station SS5's are connected by a 10Mbps

Level	Name	Hardware Elements	Descriptions
1	SLC-10E	Workstations	Sun SPARC Station SLC, 16M memory
		Communication Media	10Mbps Ethernet
		Physical Nodes	18 Workstations + 43 PCs
2	SS5-10E	Workstations	Sun SPARC Station SS5, 32M memory
		Communication Media	10Mbps Ethernet
		Physical Nodes	43 Workstations + 33 PCs
3	SS5-10S	Workstations	Sun SPARC Station SS5, 32M memory
		Communication Media	10Mbps collision-free switch
		Physical Nodes	6 Workstations
4	SS5-100S	Workstations	Sun SPARC Station SS5, 32M memory
		Communication Media	100Mbps collision-free switch
		Physical Nodes	2 Workstations + 1 server

Table 4.1. Hardware configurations used for performance evaluations

Ethernet, and the level 3 (or SS5-10S) where Sun SPARC Station SS5's are connect by a 10Mbps collision-free network switch.

Since the configuration SS5-10S and SS5-100S are limited to applications which need only up to eight and two machines respectively, almost all the applications have been evaluated using the configuration SS5-10E. However, we used the other configurations for the basic object and communication control tests in the first two subsections in order to validate MESSENGERS' competitiveness through all the configurations.

4.1 General Measurement

4.1.1 Basic Interpreter Performance

The first series of experiments investigates the overhead incurred by the interpreter in terms of four basic operations: (1) arithmetic interpretation, (2) precompiled function invocation, and (3) execution instance creation and (4) its context switch. It compares the performance of these basic operations running MESSENGERS on a single machine and the analogous operations executed by a conventional C program using the Unix **lwp** (lightweight process) multi-threading library. This series of tests are conducted for SUN Sparc Station SLC and SS5. The performance results are summarized in Table 4.2 and Table 4.3.

In terms of the Sparc SLC, (i.e., the slower machine), Test 1 shows that interpreting arithmetic operations in MESSENGERS scripts incurs an order-of-magnitude overhead compared to native code execution. Test 2 shows that MESSENGERS function call is two orders of magnitude more costly than a C function call. Creating a Messenger requires approximately three times as much as spawning a **lwp** thread (but much less than a Unix process). Finally, a Messenger context switch is comparable to a **lwp** context switch (Test 4). A two-order-of-magnitude overhead in MESSENGERS' function invocation comes from (1) the search for the function's entry-point, which requires string comparisons (and occasionally hash table operations), (2) a *setjmp* operation to make the control return right after the function entry point when a *back2messenger* library function is called, (3) a Messenger's context switch, and (4) dynamic linking operations at the first function call.

In terms of the Sparc SS5, we observed that, for Test 2 and 3, MESSENGERS' performance improved relative to the results of Table 4.2. The relative slow-down of Test 1 is caused by the faster floating-point execution of the SS5. In general, a faster floating-point unit increases the relative interpretation overhead. Hence, a large portion of arithmetic operations of any applications should be coded in C functions invoked through a *func* statement, instead of being part of the Messenger script. On the other hand, a larger cache memory reduces memory

Test #	programs	Elasped Time (seconds)	Slow down (magnitude)
Test 1	*Arithmetic interpretation vs execution*		
	(a) Sequential C program	2.117	1.00
	(b) Messengers	27.193	12.85
Test 2	*Func statement vs ordinary function call*		
	(a) Sequential C program	0.187	1.00
	(b) Messengers	31.249	167.11
Test 3	*Creation: Messenger vs thread/process*		
	(a) Thread	26.569	1.00
	(b) Messengers	77.982	2.93
	(c) Unix process	3879.780	146.03
Test 4	*Context switch: Messenger vs thread*		
	(a) Thread	9.759	1.00
	(b) Messengers	11.185	1.15

Table 4.2. Interpreter performance on SUN Sparc SLC

latency. This, in turn, reduces overheads in string operations, hash table operations, and Messenger script duplications, and thus the function call and Messenger creation including these operations are considerably improved on the Sparc SS5. Overall, MESSENGERS interpreter overhead on the SS5 was reduced within one order of magnitude for all tests as shown in Table 4.3.

Since object control, (i.e., creation and context switch) may depend on the size and number of objects, we specifically focus on this issue below. For object creation, MESSENGERS' competitiveness does not change because of two reasons: (1) duplication of a Messenger does not necessarily duplicate its script, because the script is shared by all Messengers originated from the same parent, (2) the overhead incurred by duplicating Messenger variables is proportional to that incurred by thread's automatic variables allocation. For context switch, MESSENGERS's overhead results from queuing operations of their control blocks, which are independent of Messenger script size. As shown in Figure 4.1, the overhead increases linearly with the number of Messengers. In contrast, the threads' context-switch

Test #	programs	Elasped Time (seconds)	Slow down (magnitude)
Test 1	*Arithmetic interpretation vs execution*		
	(a) Sequential C program	0.119	1.00
	(b) Messengers	2.160	18.15
Test 2	*Func statement vs ordinary function call*		
	(a) Sequential C program	0.098	1.00
	(b) Messengers	3.985	40.66
Test 3	*Creation: Messenger vs thread/process*		
	(a) Thread	4.590	1.00
	(b) Messengers	7.759	1.69
	(c) Unix process	259.460	56.53
Test 4	*Context switch: Messenger vs thread*		
	(a) Thread	0.815	1.00
	(b) Messengers	1.234	1.51

Table 4.3. Interpreter performance on SUN Sparc SS5

overhead increases exponentially. This is because a thread stack area needs at least two pages, one for its stack itself and the other for a "red zone" which is a margin between adjacent threads' stacks to protect them from corrupting each other, and thus uses up virtual memory much faster than MESSENGERS.

Finally, it is important to note here that applications will not necessarily run with the overheads shown in Table 4.2 and 4.3. The reason is that every Messenger is a composition of interpreted script code and native-mode C functions, where the latter executes with no additional overhead. Furthermore, communication is typically the most important source of overhead in distributed applications and hence the cost of interpretation is only one small factor of the overall performance. The following section considers the cost of communication.

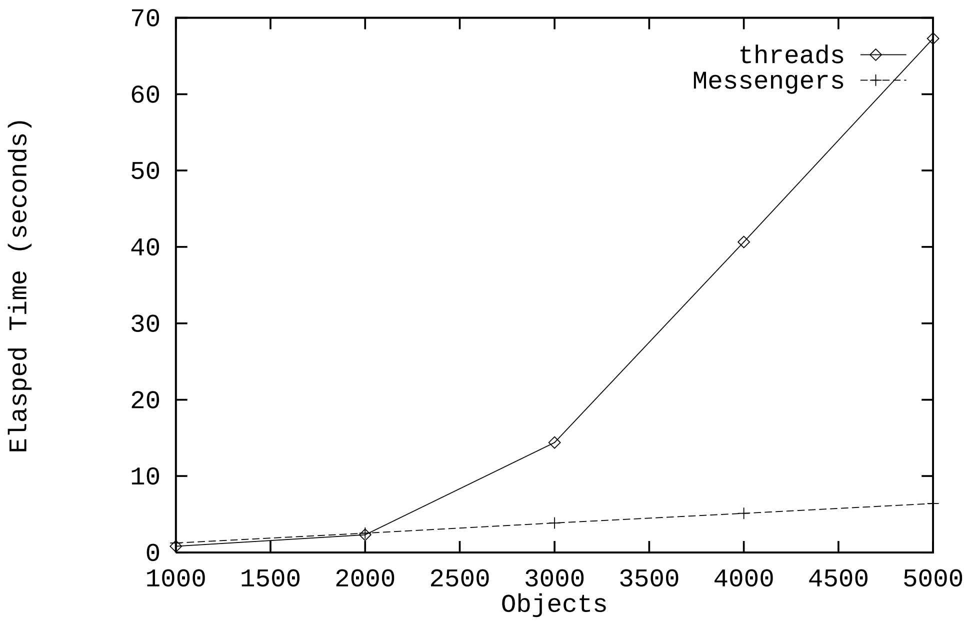


Figure 4.1. Overhead of thread and Messenger context switch on Sparc SS5

4.1.2 Communication Test

Communication is typically the most important feature determining the performance of distributed applications. In this section, we have conducted three communication tests. The first test investigates appropriate computation granularity in distributed computing in order to overcome network latency. The second test explores the data transfer speed of MESSENGERS, *threads*, and PVM when the communication flow is very simple and therefore needs no flow control. The third test evaluates the flow control performance in each of the three systems, given a communication pattern that is likely to induce network congestions.

1. **Network Latency**

This test determines the computation granularity size necessary to overcome communication overhead for parallel computations. It compares four programs. The first is a sequential C program that performs $2 * j$ floating point operations in a loop. The second involves two `lwp` threads, A and B. Thread A first sends a 16-byte data to thread B. Both threads then perform j floating point operations. Finally, thread B sends a 16-byte data to thread A. The third program is similar to the second but uses PVM involving two remote processes. The fourth program uses two Messengers. Messenger A hops to another workstation; both Messengers perform j floating point operations; Messenger B then hops back to A's site.

The four programs are designed to measure the effects of granularity on speedup for three hardware configurations SLC-10E, SSC-10E, and SSC-100S. Figure 4.2(A) shows the results of the slowest configuration SLC-10E. The curves imply that `lwp` threads, Messengers, and PVM all achieve speedup over sequential execution once the granularity increases beyond 1,000, 10,000, and 25,000 operations respectively. The main factor making PVM the slowest among them is that data must be copied from a user process to a PVM daemon at the source node and vise versa at the destination while the amount of data transferred is negligible. The results show further that, past 100,000 operations, the additional overhead of MESSENGERS becomes negligible as compared to conventional multi-threading execution.

Figure 4.2(B) shows the results of the configuration SS5-10E where only the computing nodes are faster than SLC-10E. Neither threads, nor PVM, nor MESSENGERS can achieve speedup over sequential execution until the granularity reaches 400,000 operations. The main reason is, obviously, the slow communication medium. The ratio between the CPU speed and the network latency became considerably worse than with SLC-10E and hence more computation is needed to offest the ratio.

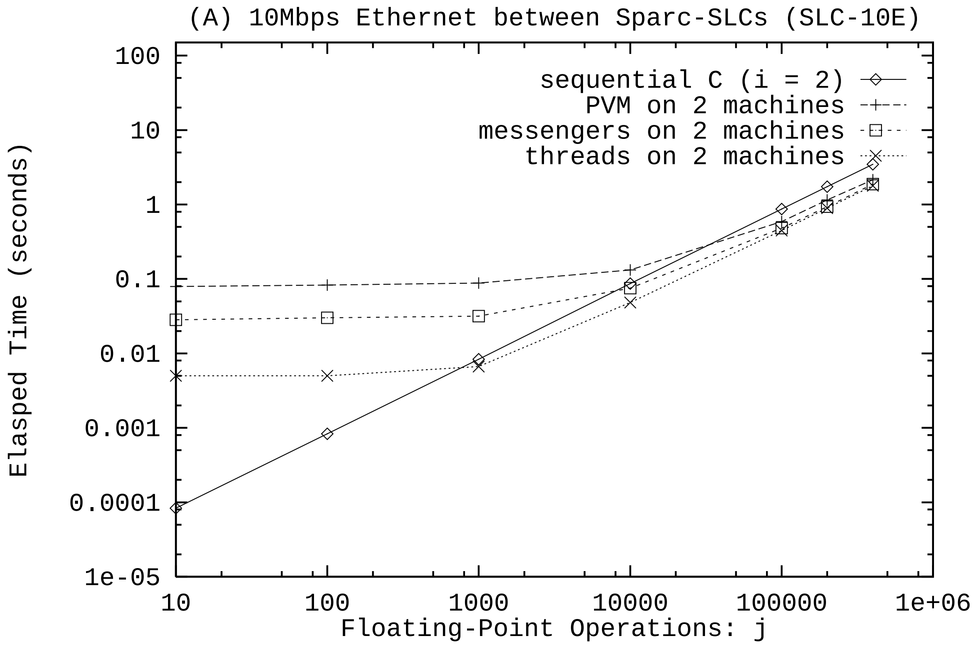
Figure 4.3 shows the result of the configuration SS5-100S where both the computing nodes and the network are faster than SLC-10E. All systems improved with the faster network. Among them, PVM benefited the most.

2. **Ping-Pong Communication**

This communication test investigates how the software overheads incurred by the three programming models change as the amount of data transferred increases. Since the socket setup and the number of read and write operations remain constant, the communication overhead should be bounded by the network speed when the transferred data size becomes large.

The test involves two machines that transfer a certain amount of data between them back and forth like a ping-pong ball (to smooth out fluctuations over time.) As shown in Figure 4.4(A), our prediction is proved correct for MESSENGERS and threads in the configuration SS5-10E. MESSENGERS' overhead becomes negligible as compared to threads beyond 20K-byte-data transfer. However, for PVM the overhad continues to increase with the amount of data transferred. This is because the transferred data must be copied from a user process to a PVM daemon at the source node and vice versa at the destination node. Such data copying operations incur overheads proportional to the amount of data transferred.

In the configuration SS5-100S where only the network is faster than SS5-10E, MESSENGERS' overhead is more visible as compared to threads as shown in



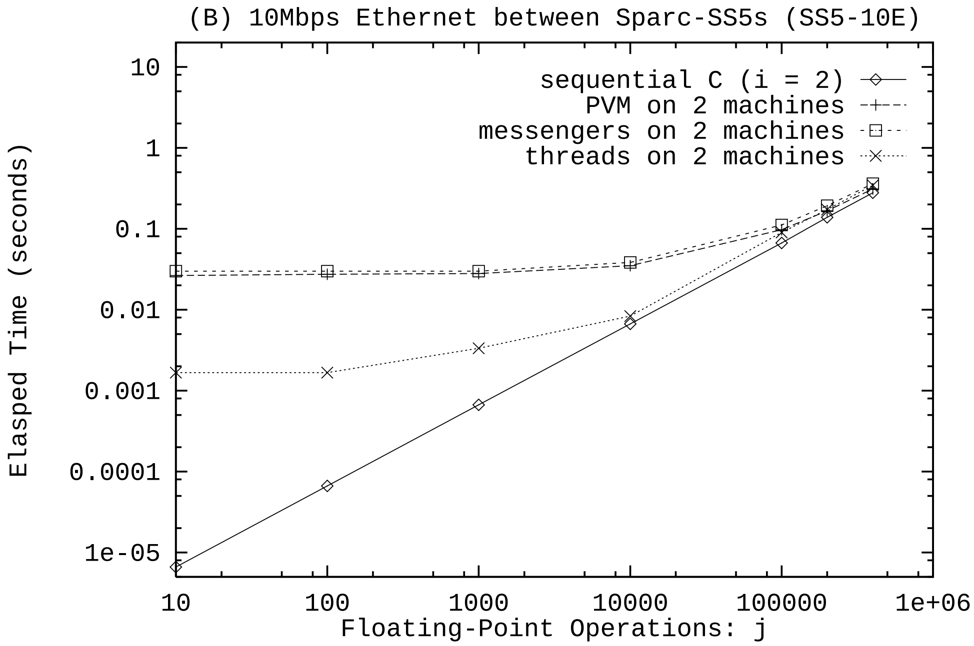


Figure 4.2. 10 Mbps network latency test

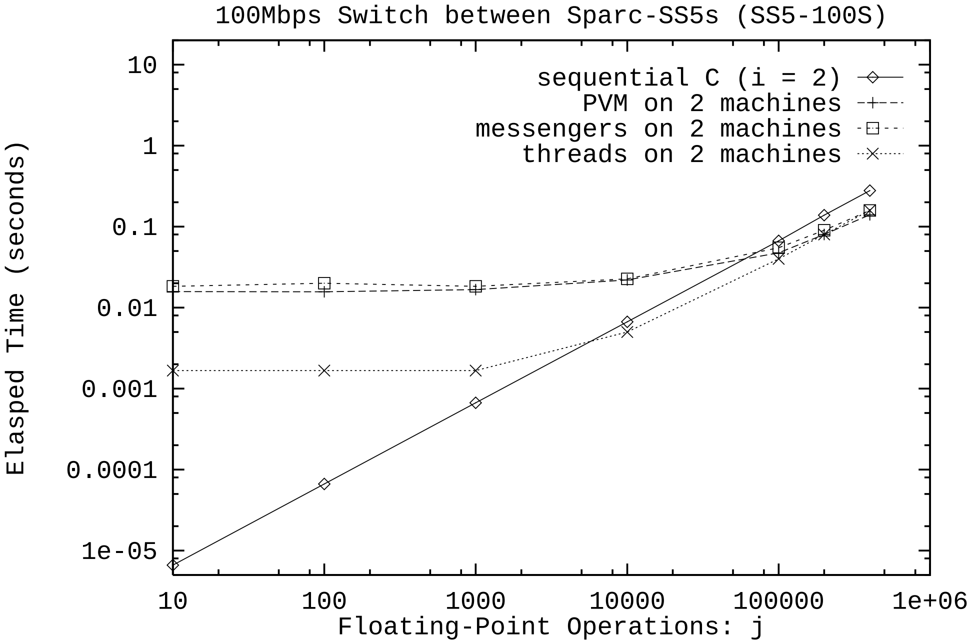


Figure 4.3. 100Mbps network latency test

Figure 4.4(B). PVM is again far behind threads and MESSENGERS. However, as shown in Figure 4.5, PVM benefited more than MESSENGERS from the improved network speed when the amount of data is below 9000 bytes.

3. **Message Exchanging**

The last communication test focuses on flow control performance as the amount of data transferred increases. To induce network congestion, two machines repeatedly transfer a certain amount of data to each other at the same time and then go into a data receiving phase. This distinct two-phase message exchange may easily cause message collisions and saturate the socket receiving buffer.

In SUN OS, 4K bytes is the system default to send data through a socket at once and store it into a socket receiving buffer. Therefore, for the thread

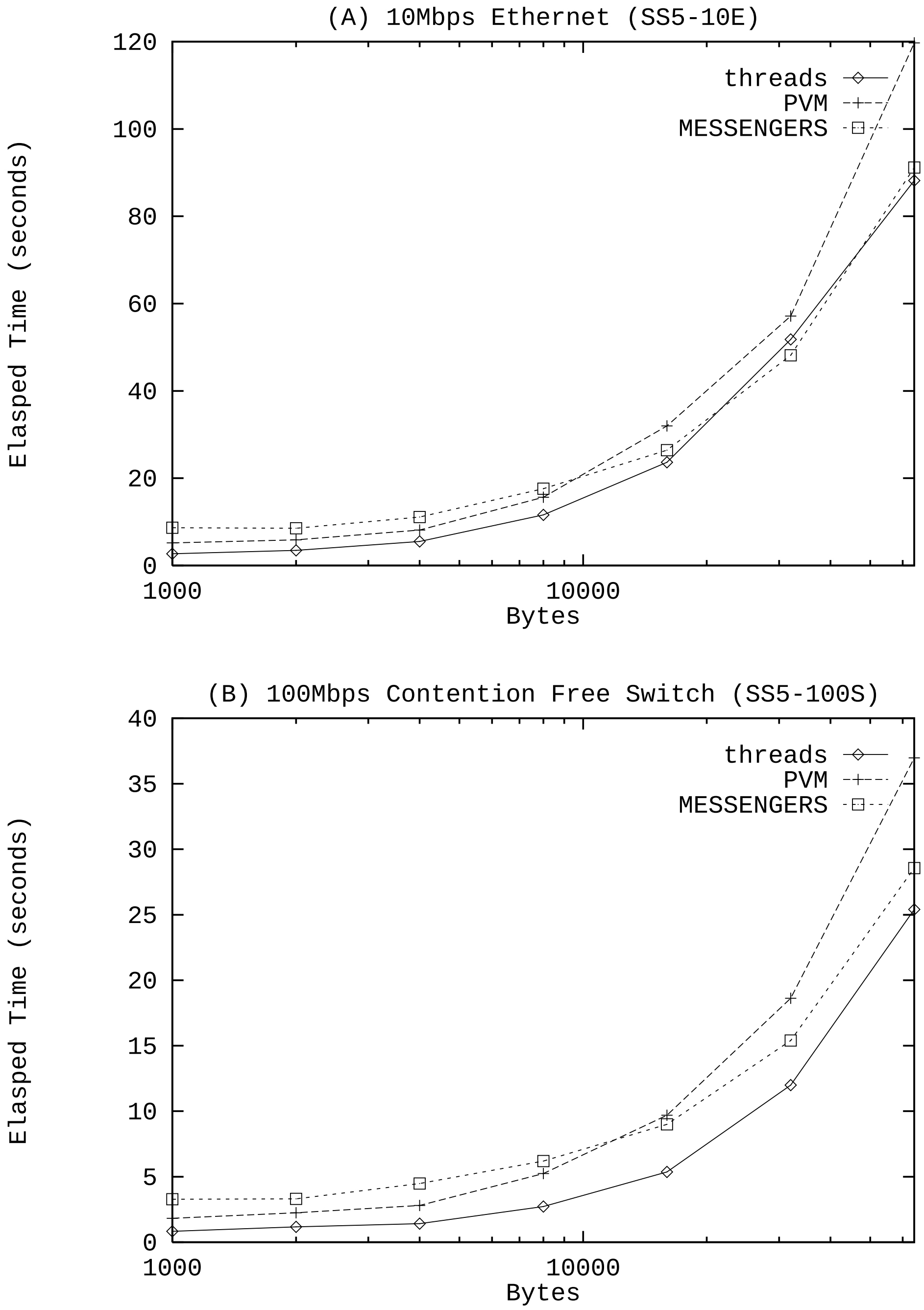


Figure 4.4. Ping-pong communication

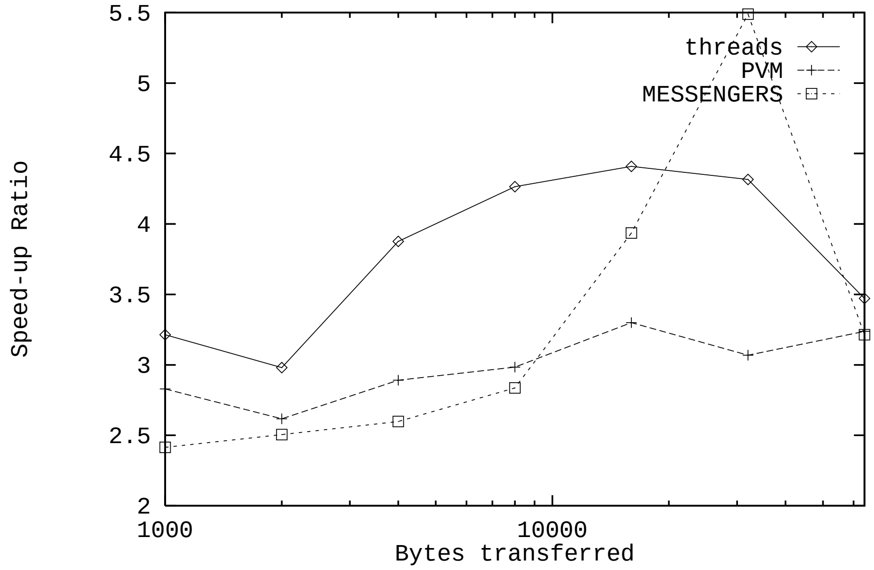


Figure 4.5. Speed-up of ping-pong communication by improving the network speed from 10Mbps to 100Mbps

version, if the size of data is more than 4K bytes, the test program packetizes them into 4K bytes each to avoid a deadlock caused by a network congestion. For the MESSENGERS and PVM versions, the programs transfer the total amount of data to be sent at once.

Figure 4.6(A) shows the result of the configuration SS5-10E. PVM works best when sending small amounts of data, however, its performance falls behind MESSENGERS when data size exceeds 16K bytes because of the data copying operations discussed in the ping-pong test. Furthermore, the threads' performance falls behind that of MESSENGERS beyond 32K bytes.

Figure 4.6(B) shows the result of the configuration SS5-100S. Again, PVM works best for small amounts of data while MESSENGERS becomes the best for large amount of data. Since the network speed in the configuration SS5-100S is much faster than SS5-10E, data is transferred to the destination and saturates the receiving buffer quickly. Figure 4.7 shows the speed-up of the three systems gained by switching the network speed from 10Mbps to 100Mbps. PVM benefited the most from the improved network speed. The thread version displayed the worse performance. Even though, in the thread version, the size of message transferred at once is adjusted not to exceed the receiving buffer, buffer saturation occurs more frequently due to the following reason: the low network latency makes the entire message arrive at the destination in a short time, while the previous message still remains in the receiving buffer. Thus, the operating system must postpone copying this entire message into the buffer. The repetition of such postponement will delay resuming threads from blocking read and write operations.

4.2 Specific Applications

This section discusses MESSENGERS' performance competitiveness with the message-passing paradigm using six applications that have been discussed in Section 3.2.

4.2.1 Master-Slave

This section compares MESSENGERS with PVM when running the master-slave programs. It also investigates computation granularity and message size

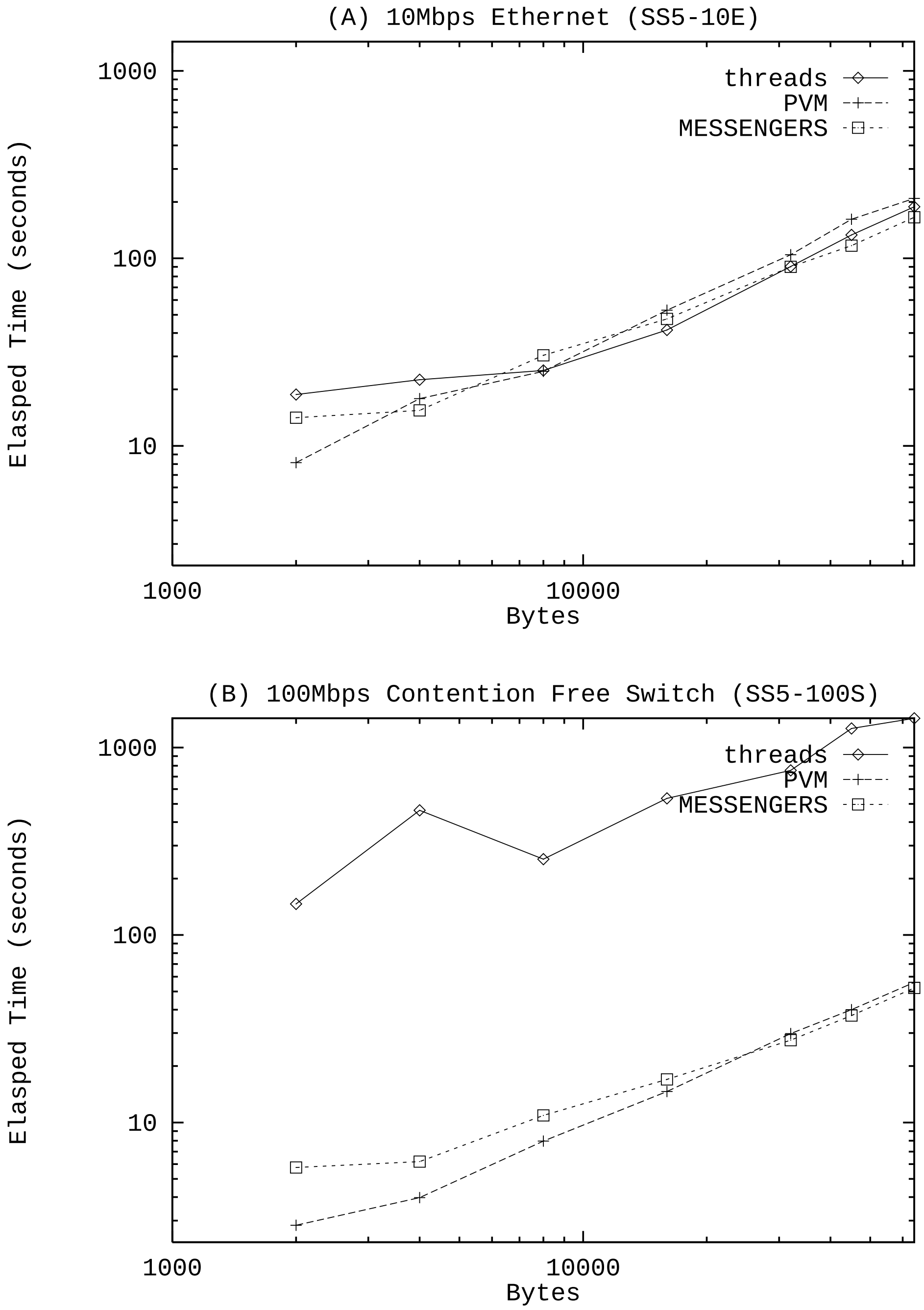


Figure 4.6. Message exchanging

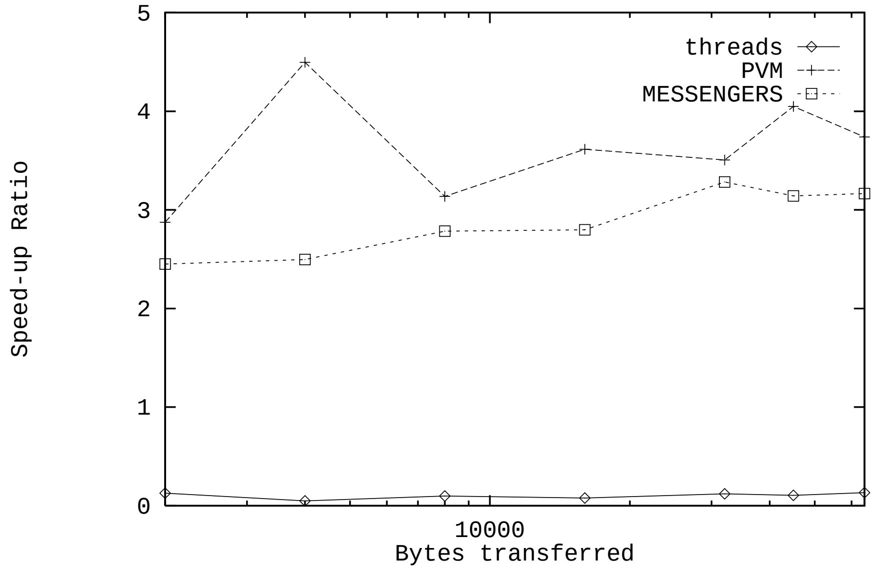


Figure 4.7. Speed-up of message exchanging by improving the network speed from 10Mbps to 100Mbps

appropriate to maintain MESSENGERS' competitiveness for this kind of applications.

We use the codes discussed in Figure 3.1 and Figure 3.2 for PVM and MESSENGERS, respectively. To observe how performance changes by computation granularity, the slave node executes dummy floating-point operations in the compute() function (line 20 in the message-passing version and line 9 in the MESSENGERS version). Also, a certain amount of data is carried by a PVM message and a Messenger every time a new task is processed at a slave node. Figure 4.8, 4.9, and 4.10 show the results of this test when each slave computes 1,000, 10,000, and 100,000 floating-point operations respectively. MESSENGERS are two to four times slower than PVM when the number of slave CPUs is up to nine and the computation granularity is below 10,000 operations. It becomes competitive with

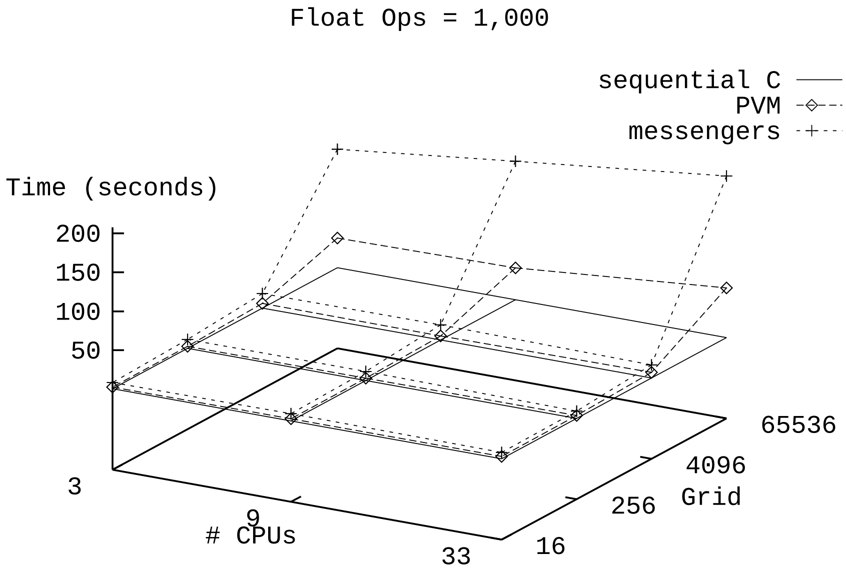


Figure 4.8. Basic master-slave test on SS5-10E (1,000 float-pt ops/job)

PVM when 100,000 operations take place at a slave CPU. This is because the larger computation granularity reduces the frequency of the inter-daemon communication and enlarges the time lag among slave CPUs when they complete each job, so that the master CPU has less chances of congestion.

When the size of data transferred is increased, MESSENGERS' performance is improved as compared to PVM, since PVM's data copying operations incur more overheadn. However, MESSENGERS slows down for transferring too large an amount of data, (i.e., 64K bytes.) This is because the master takes more time to send and receive a larger amount of data, has more chances of communication congestion, and thus induces frequent context switches of message send/receive threads running internally for the MESSENGERS daemon on the master CPU.

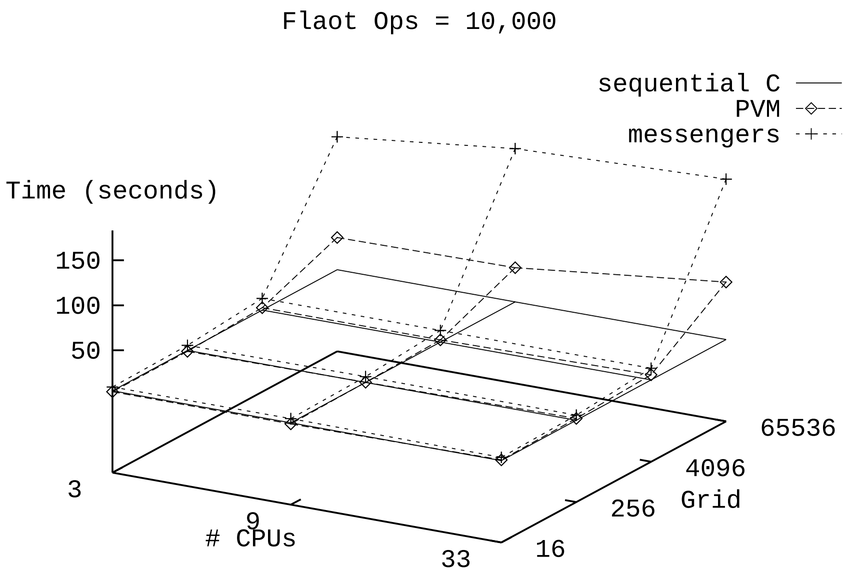


Figure 4.9. Basic master-slave test on SS5-10E (10,000 float-pt ops/job)

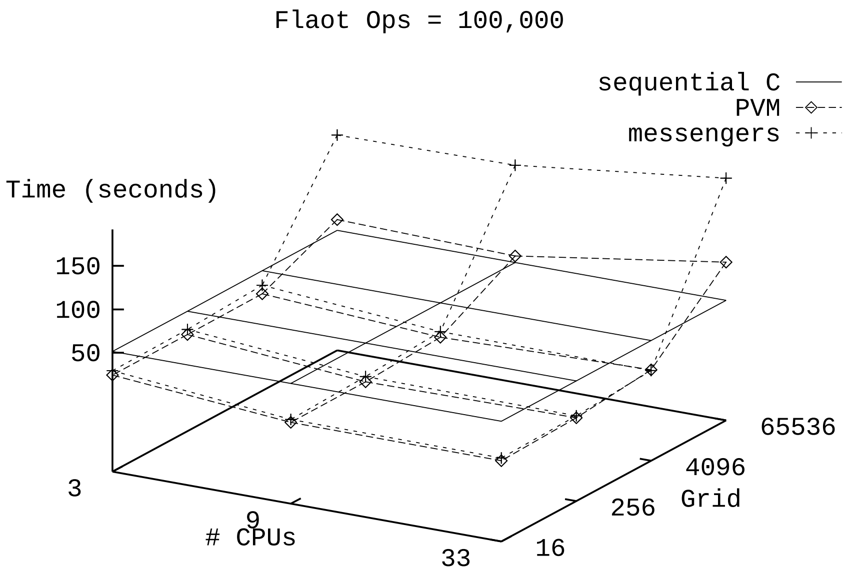


Figure 4.10. Basic master-slave test on SS5-10E (100,000 float-pt ops/job)

We have also considered a practical example categorized as a master-slave type. It is a Mandelbrot set generation, which draws fractal curves on a 2D screen [PS88]. The performance evaluation has been conducted using the same region of Mandelbrot imagery, located at $(-2.0, -1.2, 0.4, 1.2)$ with varying resolutions $(320 \times 320, 640 \times 640,$ and $1280 \times 1280)$. The computation was divided in sub-problems of $8 \times 8, 16 \times 16$, and 32×32 blocks. Each slave program repeats picking up the next unprocessed block and computing the sub-problem at a slave CPU.

Figure 4.11, 4.12, and 4.13 show the performance of MESSENGERS and PVM running this application when image resolutions are 320×320, 640×640, and 1280×1280 respectively. The results obtained are very simple to analyze. The higher the resolution and the smaller the number of blocks, the more competitive becomes MESSENGERS with PVM. The main reason is that only 40 bytes are sent to a slave node as the sub-block information, and therefore the communication becomes the most significant source of overhead when the image is processed with a large number of blocks.

Figure 4.14(A) and (B) show the results of two extreme test cases: the best one with the largest image resolution and the smallest number of blocks and the worse one with the smallest resolution and the largest number of blocks. MESSENGERS achieves even speedup over PVM in the best case.

For all test cases in Figure 4.11, 4.12, and 4.13, MESSENGERS runs 1.5 times slower than PVM for 80% of the test cases. This is because the computation of an image with a 320×320 resolution corresponds to 100,000 floating point computations in the master-slave test program as shown in Figure 4.10.

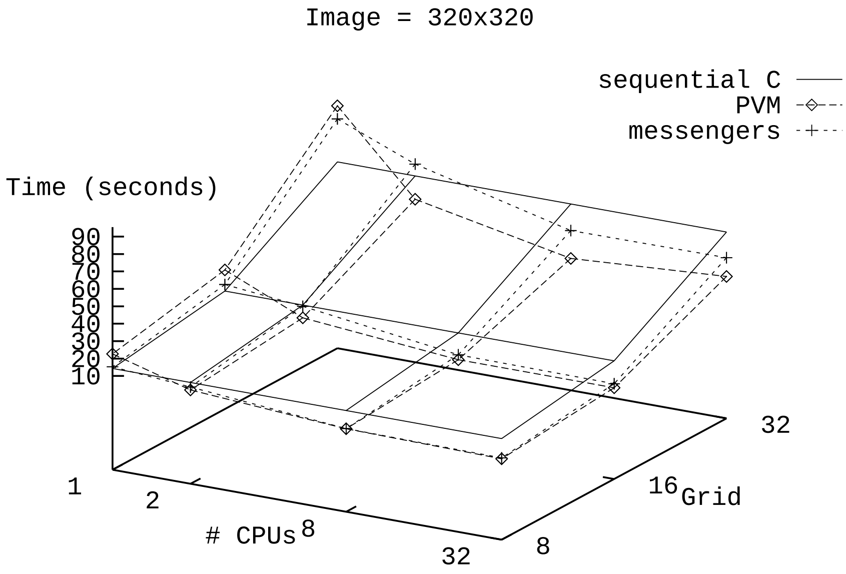


Figure 4.11. Mandelbrot on SS5-10E where image size = 320x320

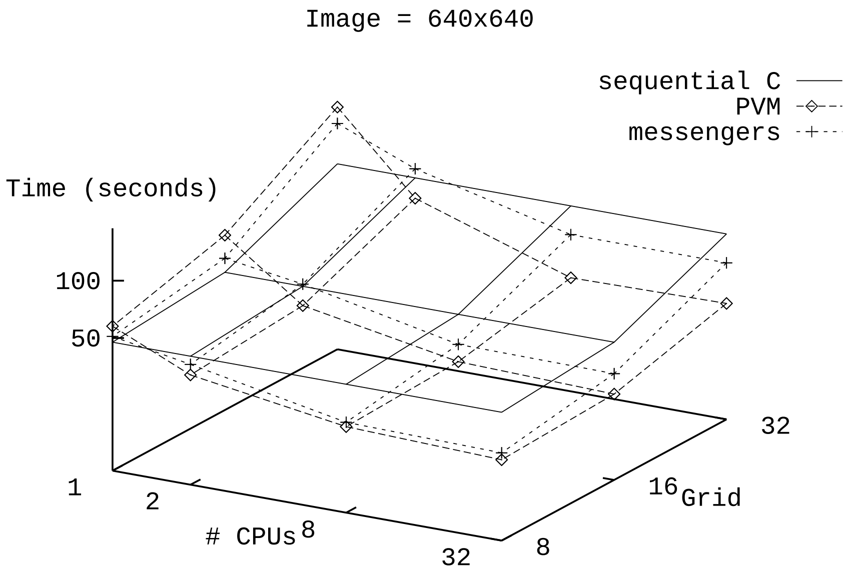


Figure 4.12. Mandelbrot on SS5-10E where image size = 640x640

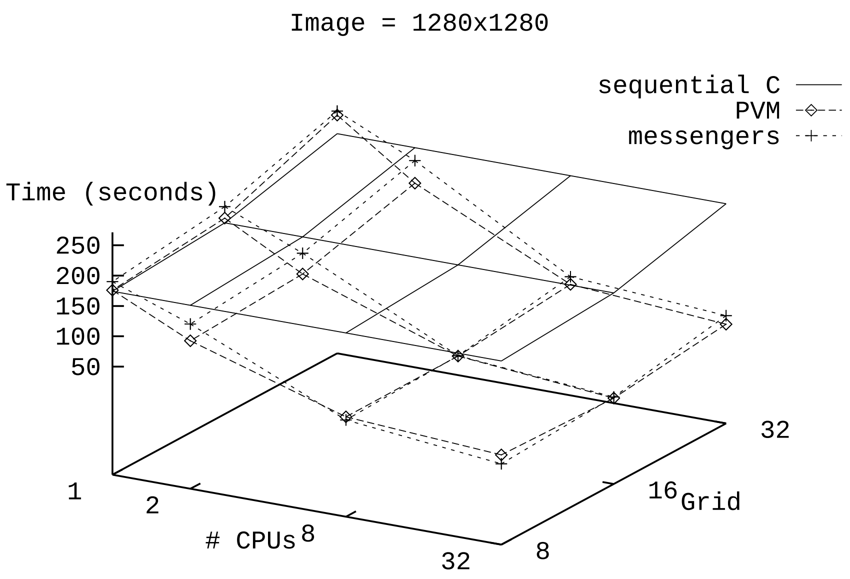


Figure 4.13. Mandelbrot on SS5-10E where image size = 1280x1280

4.2.2 Information Diffusion

This sub-section examines MESSENGERS competitiveness with PVM in terms of the information diffusion algorithm. The same algorithms as described in Figure 3.3 and Figure 3.4 are used to evaluate the performance of MESSENGERS and PVM, respectively. To examine the best computation granularity for this type of application, we have added dummy floating computations performed upon a new message or Messenger received, (i.e., immediately after the first *receive()* on line 2 in the message passing version and after the *create()* on line 3 in the MESSENGERS version.) Also, the message and the Messenger carry a certain amount of data. The evaluation was conducted using a 5×5 grid of CPUs. This is a moderate network topology in terms of its connectivity.

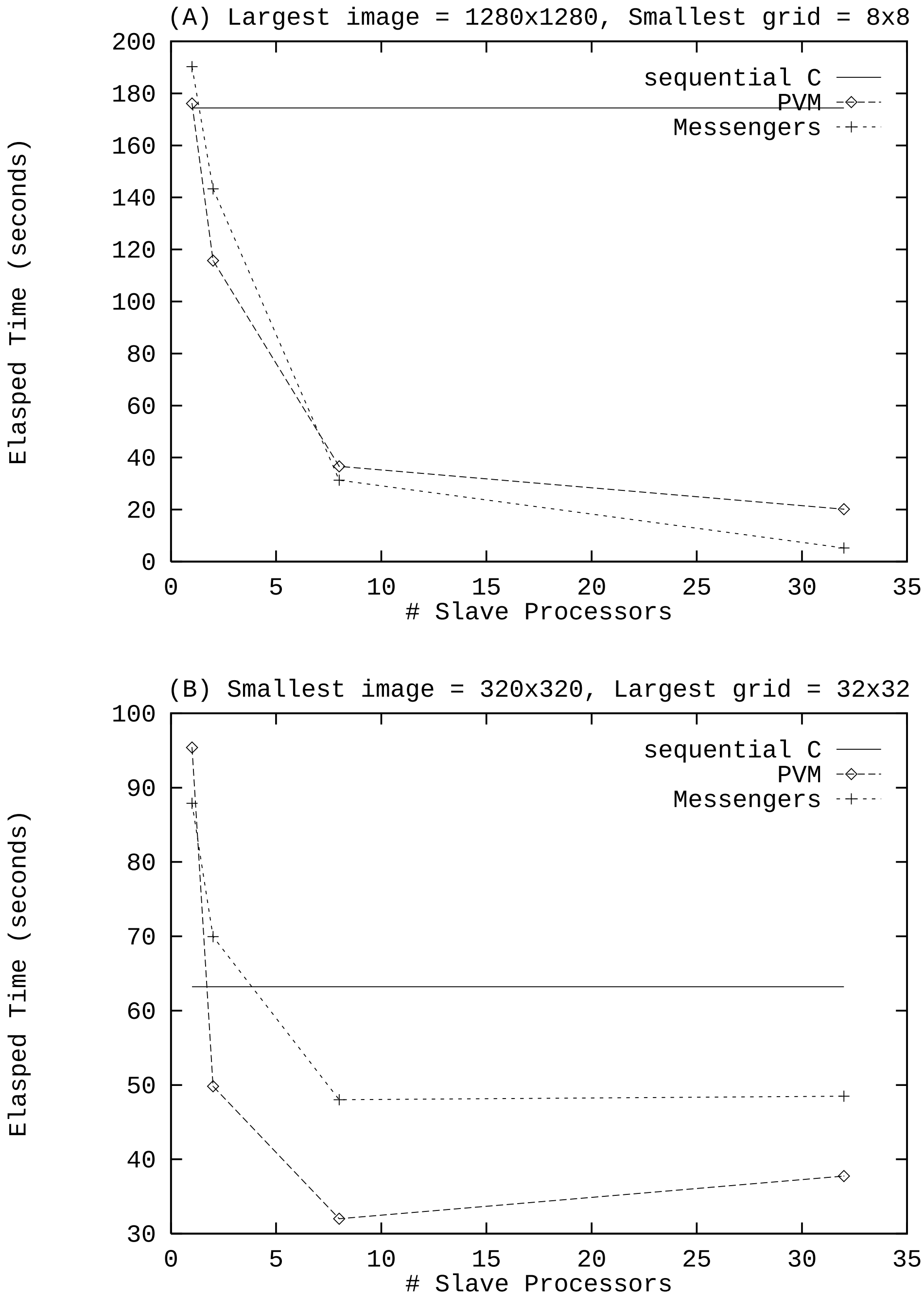


Figure 4.14. The best and worst cases of Mandelbrot set generation

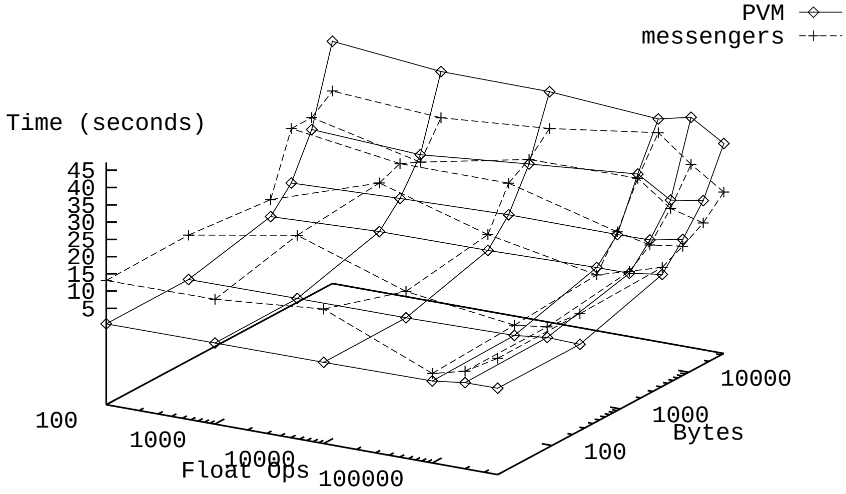


Figure 4.15. Information diffusion on 25 SUN Sparc-SS5s (SS5-10E)

As shown in Figure 4.15, MESSENGERS' performance is competitive with that of PVM beyond 4K-byte-data diffusions. It is even better than PVM beyond 32K bytes with 100 computations at each node. As the computation granularity is increased to 400,000 operations, the threshold of MESSENGERS' performance superiority goes down from 32K bytes to 8K bytes. This result is obtained from three facts: (1) the number of messages exchanged over the network is larger in PVM than MESSENGERS, (2) PVM's data copying operations between daemons and user processes are no longer negligible when large messages are diffused, and (3) the larger computation granularity reduces the number of context-switches between the send and receive threads working inside a daemon to avoid congestion.

4.2.3 Circulatory Simulation

We first use a simple circulatory test program that creates a master and eight slave logical nodes. When using PVM, each logical node is represented by an independent user process. The test involves up to 8 slave CPUs plus 1 master CPU, and uniformly distributes the slave logical nodes to these CPUs. (For instance, when using 2 CPUs, each receives 4 slave logical nodes.) The PVM implementation has a message broadcast from the master to all slaves, which returns from the slaves to the master. The MESSENGERS implementation creates and maps a circulatory logical network over the CPUs as shown in Figure 3.5 and thereafter has Messengers propagated from and returned to the master along all positive links. To assess the computation granularity, dummy computations take place at each slave node immediately after a message or a Messenger arrives there. The data size is fixed to 1K bytes.

Figure 4.16(A) and (B) show the results of the PVM and MESSENGERS implementations. While PVM achieves speedup over the sequential C program when the granularity reaches 10,000 operations, and even demonstrates its scalability, MESSENGERS achieves speedup with only two slave CPUs but does not obtain the effect of scalability at all. This unfavorable result is caused by the congestion of Messengers returning back to the master node. Therefore, the competitiveness of MESSENGERS deteriorates with the number of CPUs, as observed in Table 4.4. Since all returning Messengers must synchronize with each other during every circulation in this type of application, processing even 200,000 dummy operations at each slave is not large enough to make a sufficient time lag among the arrival of

# processors	MESSENGERS (seconds)	PVM (seconds)	Slow down (ratio)
1	614.951	608.880	1.205
2	319.808	510.436	1.207
4	1131.831	264.923	8.004
8	609.810	141.414	7.790

Table 4.4. MESSENGERS vs PVM in basic circulatory test

returning Messengers in each circulation cycle. (They all perform the same 200,000 floating-point computations, so that the distribution does not change.)

The Styrene inhalation program is a more complex example of the circulatory simulations. The main body of this application is the Messenger that implements the Styrene-carrying fluids as described in Figure 3.6. However, in order to control the simulation scenario interactively, we need several additional Messengers that increment simulation time, record the intermediate value of styrene concentration at each node, stop the simulation and kill the Messenger carrying Styrene. Therefore, unlike the simple circulation test program, each MESSENGERS daemon has more Messengers to interpret concurrently and may reduce the frequency of communication congestion.

The results for both the PVM and MESSENGERS versions are shown in Figure 4.17. Again, to asses the computation granularity, dummy floating-point operations are added to the computation of Styrene concentration at each organ node. The result shows that MESSENGERS's scalability is bounded to two CPUs, however, the threshold of computation granularity to achieve speed-up over the sequential version became slightly smaller than with the basic circulation test. This is because the Styrene program application consists of several kinds of Messengers as mentioned above and partially reduced the communication congestion.

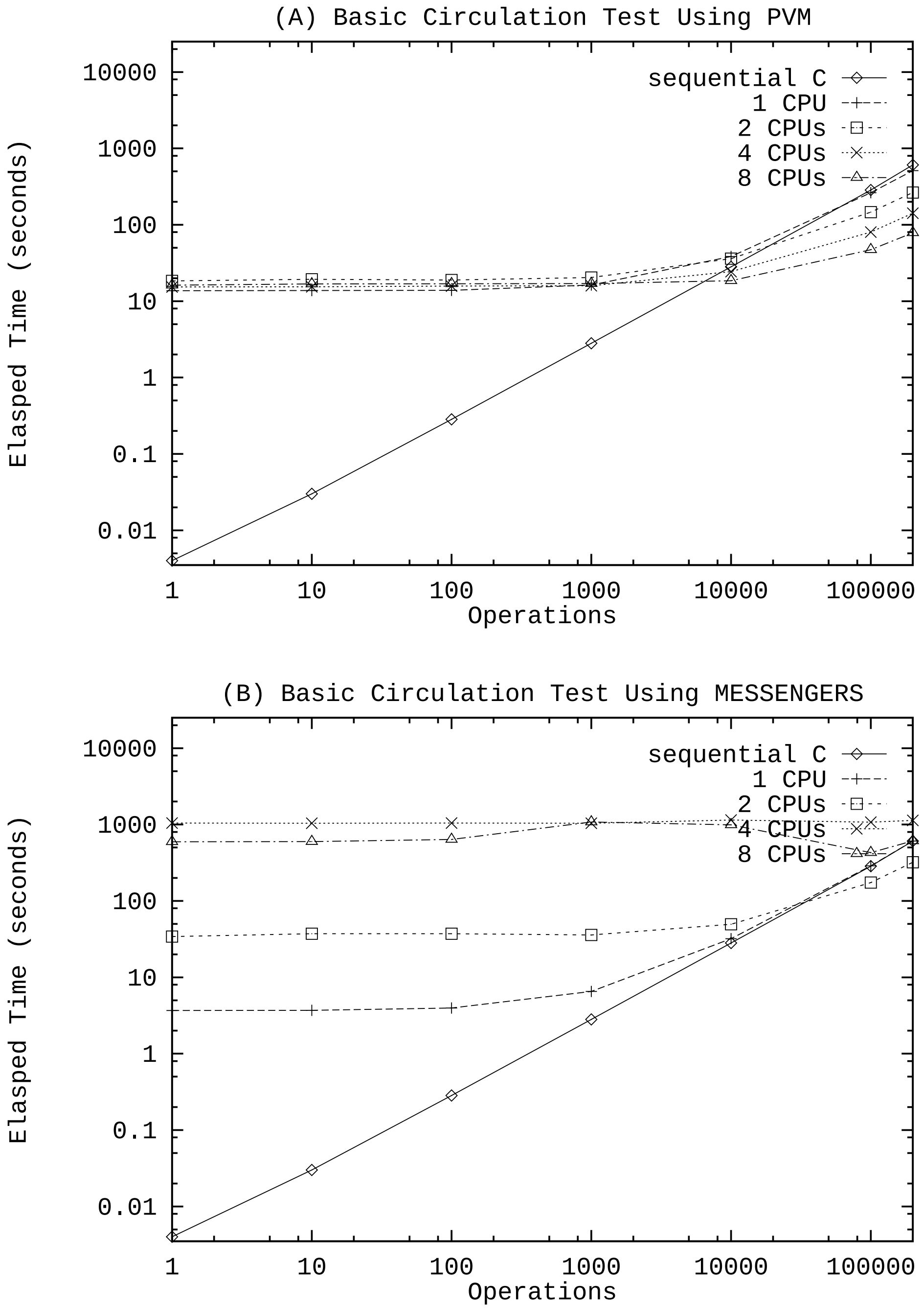


Figure 4.16. Basic circulatory test on SS5-10E

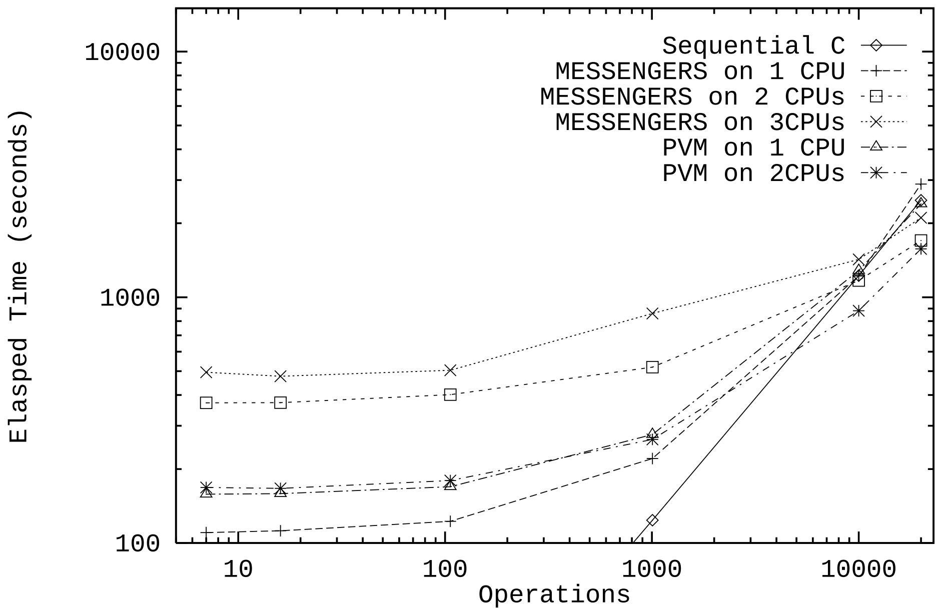


Figure 4.17. Styrene inhalation

Based on this knowledge, we have implemented two new internal functionalities in the MESSENGERS' communication control. One is to only send a single copy of the Messenger binary code when the same types of Messengers are transferred to the same destination. The other is to completely avoid the binary code transfer when the same binary code has already been sent in the past. Table 4.5 shows the effect of these two optimizations. The more CPUs are involved, the greater an effect is obtained.

# processors	Messengers aggregation/omission		Speed up
	Not Supported (seconds)	supported (seconds)	(magnitude)
1	608.122	614.951	0.989
2	588.633	319.808	1.747
4	1352.130	1131.831	1.195
8	6174.400	609.810	10.125

Table 4.5. Effect of Messengers aggregation and redundant Messengers omission

In summary, Messengers needs a considerable amount of computations at each slave node to avoid a network congestion in the circulatory applications.

4.2.4 Divide-and-Conquer

Using two configurations: SLC-10E and SS5-10E, we have compared the performance of Messengers and PVM for the convex hull program introduced in Figure 3.10 and 3.9.

Figure 4.18(A) shows the performance results of the convex hull program for solving the problem with 1024 through to 32768 points in the configuration SLC-10E. This shows that for slightly above 1000 points, the effect of parallel processing on two or four machines outweighs the communication and interpretation overheads, resulting in actual speed-up. The performance however does not improve beyond four machines. There are two possible explanations for this. One is that the problem size is too small to gain from further parallelism. The second is that the current implementation of Messengers supports only a fixed-size variable area, so the Messenger has been coded to always carry all points, resulting in increased communication overhead, which is not strictly necessary to solve the actual problem.

Figure 4.18(B) shows the performance of the same program in the configuration SS5-10E, where the CPU speed is faster than on SLC-10E but the network speed remains unchanged. The transfer operations of the entire set of points are a more significant source of overhead in this configuration, and therefore no speedup has been obtained.

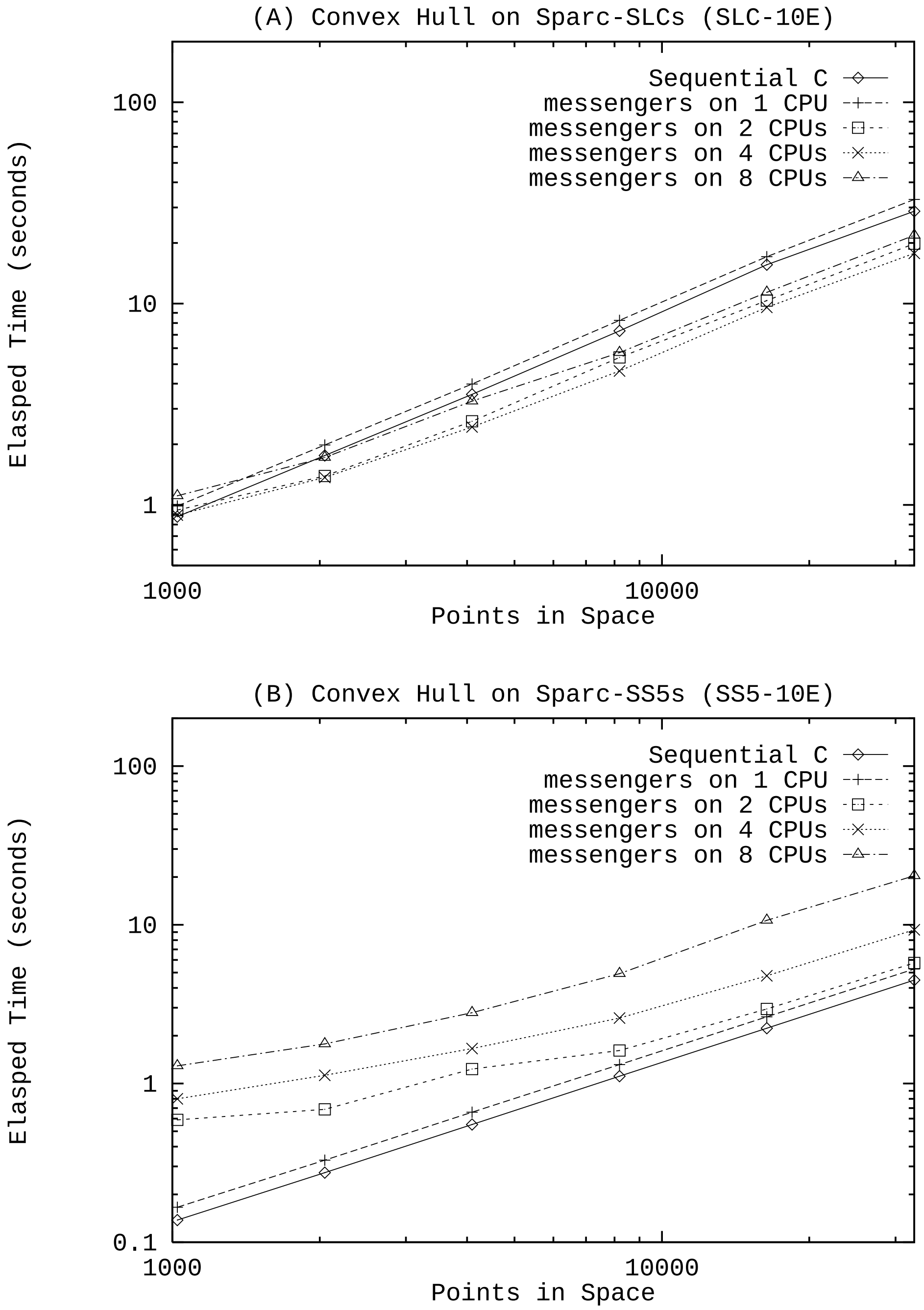
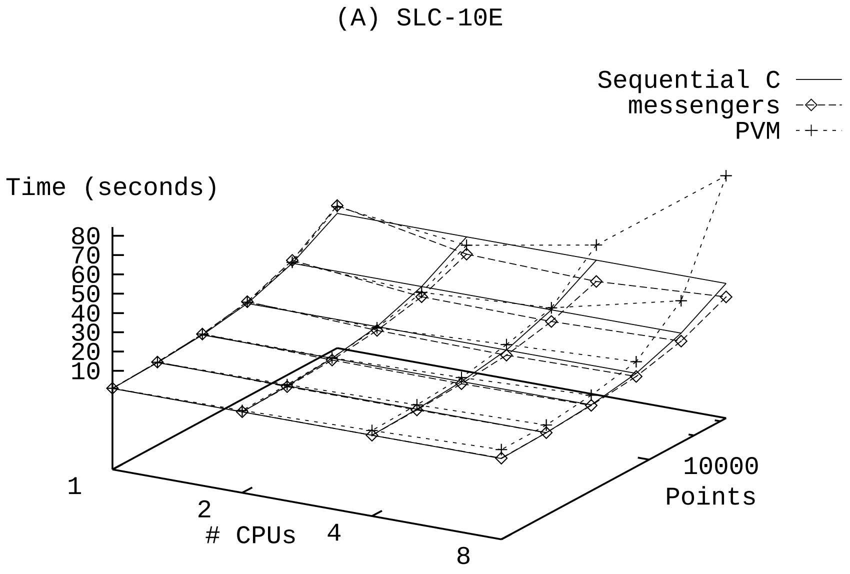


Figure 4.18. Convex hull

The comparison between MESSENGERS and PVM versions is shown in Figure 4.19. In both SLC-10E and SS5-10E configurations, MESSENGERS shows its performance superiority over PVM for almost all test cases except a single CPU version. On a single CPU, the PVM program directly starts as an ordinary Unix process and completes the computation without any intervention by the PVM daemon, whereas the Messenger script must be injected from another Unix process (minject) at the beginning and interpreted by the interpreter daemon until the precompiled C code starts the main computation. In the multiprocessor versions, on the other hand, PVM needs to create two new processes whenever descending along two tree links in addition to sending the complete set of data to them, while MESSENGERS only sends a Messenger that contains all the data and whose navigation cost is negligible as compared to PVM process creation.

To compensate for the effect of the additional communication overhead of carrying the entire data points, we ran another experiment. We modified the above divide-and-conquer program to only perform a certain number of dummy floating-point operations at each node of the tree as it is being constructed but not to carry any data between nodes in either direction. This abstracts the behavior of certain other bisection programs, such as a quadrature problem for numerical integration [GM85], where only a single computation result must be passed back from a child to its parent node at each level, and therefore the communication overhead is constant.

Figure 4.20(A) shows the result of this simple bisection program when running in the configuration SLC-10E. The threshold to obtain an improvement from parallel computing on two nodes is approximately 6,000 floating-point operations at each node. Performance can be improved further using four or eight machines



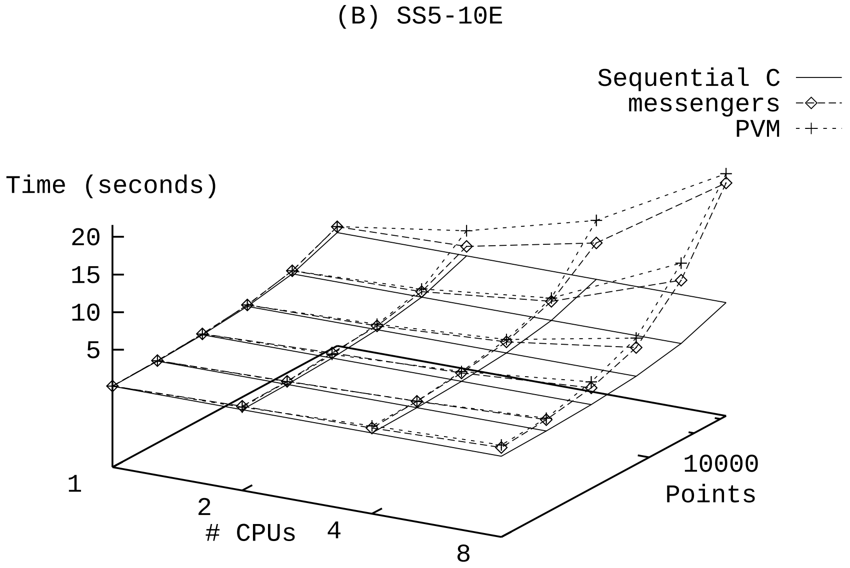


Figure 4.19. MESSENGERS vs PVM when running a convex hull program

if the granularity is sufficiently large. Figure 4.20(B) shows the performance of the same program when running in the configuration SSC-10E. The break-even point is 50,000 floating-point operations at each node, (i.e., almost one order of magnitude larger than SLC-10E) due to the large gap between processor and network speeds.

The comparison among sequential C, MESSENGERS, and PVM versions for the bisection program is shown in Figure 4.21. Again in both SLC-10E and SS5-10E configurations, MESSENGERS's performance is better than that of PVM for all test cases except a single CPU version. The reason for this superiority are the same reasons as for SLC-10E configuration.

4.2.5 Matrix Multiplication

We have conducted the performance comparison among MESSENGERS, PVM, and sequential versions.

To compare the performance with a sequential program, we implemented two versions of matrix multiplication using C. The first is the simple naive algorithm consisting of a triply nested loop, where the outer and middle loops iterate over the rows and columns of the two matrices and the inner-most loop (k) computes the inner product of a row and a column as follows: $C[i,j] = C[i,j] + A[i,k] * B[k,j]$. This naive implementation of matrix multiplication is adequate when the matrices are small. More efficient sequential algorithms have been developed for larger matrices [Pan84, CW90]. Among them, our second implementation is a block-oriented matrix multiplication, where the matrices are partitioned according to the same grid as the parallel algorithms (i.e., 2×2 and 3×3) and the multiplication

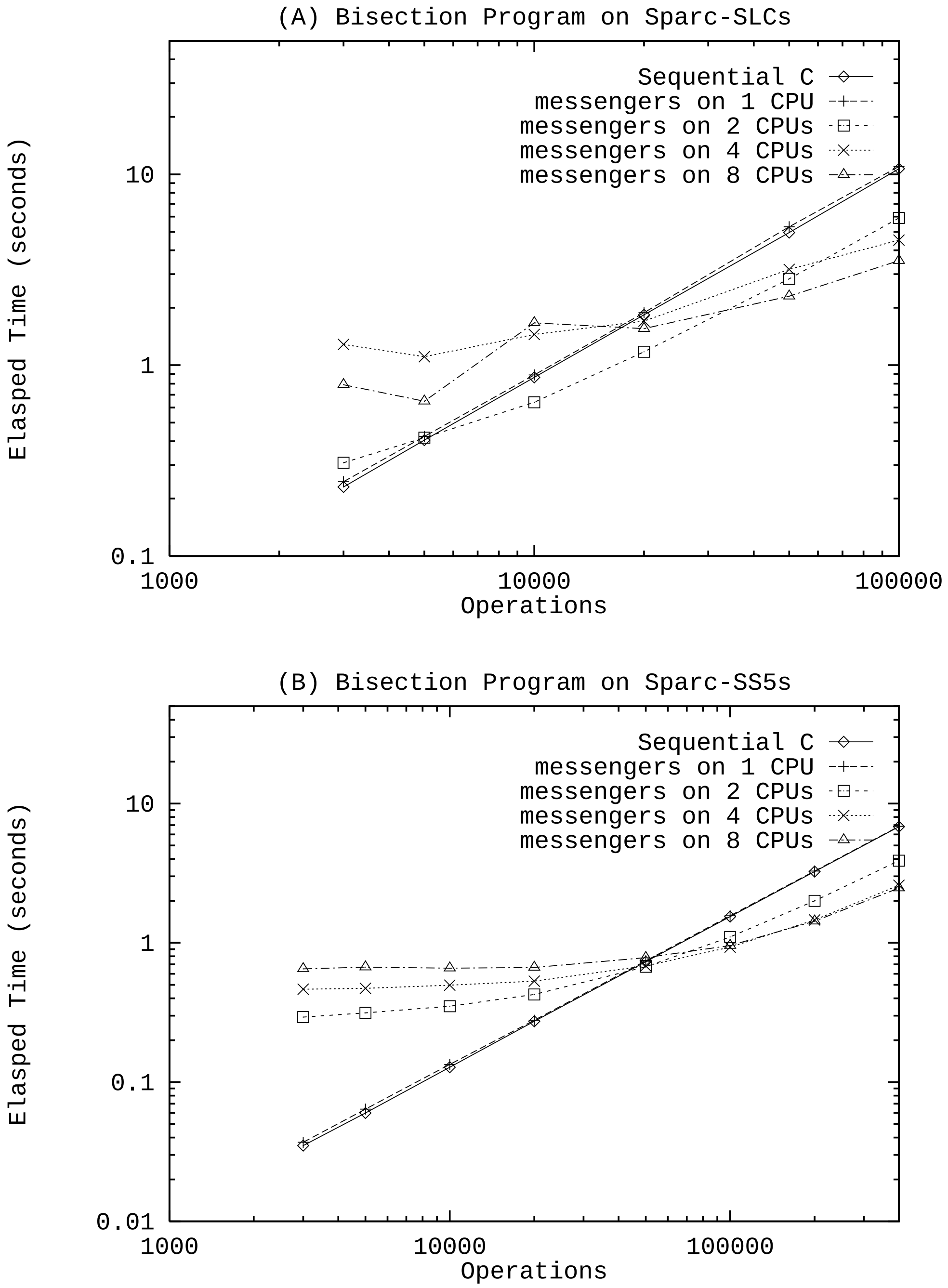
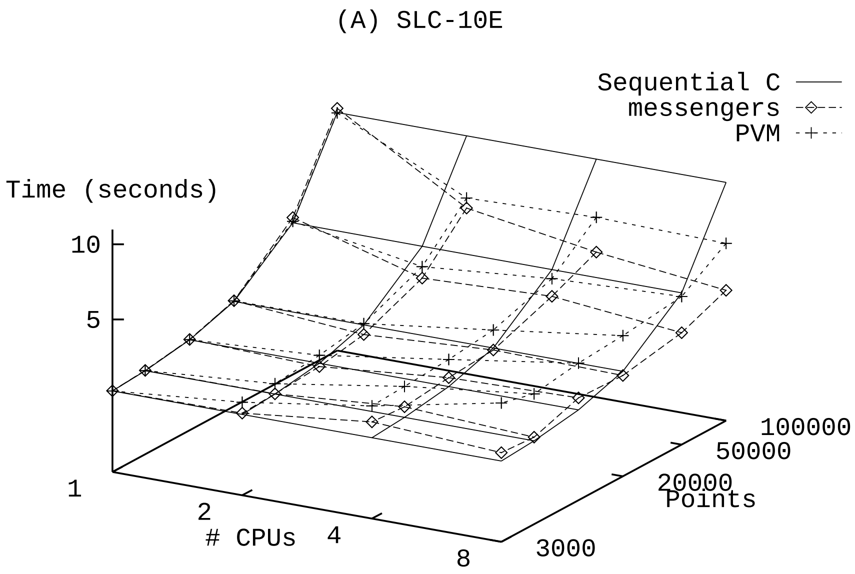


Figure 4.20. Bisection program



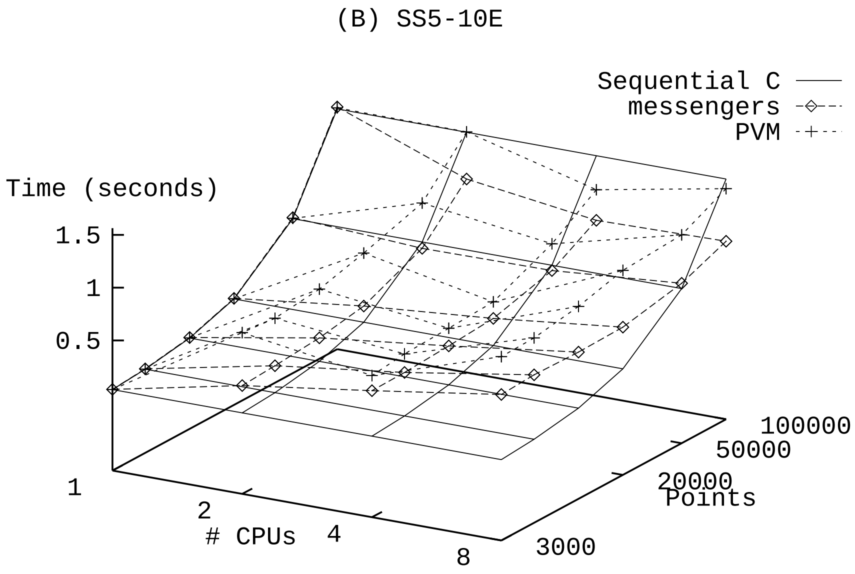


Figure 4.21. MESSENGERS vs PVM when running a bisection program with dummy floating operations

is decomposed into a series of block multiplications and additions. The main advantage of this algorithm is that it increases locality of references and thus improves the utilization of caches.

The experiments were conducted using two processor configurations: (A) a 2×2 grid of 170 MHz SS-5 workstations (SS5-10S) and (2) a 3×3 grid of 110 MHz SS-5 workstations (SS5-10E), where each workstation has 32MB of memory. The results for these two processor configurations are shown in Figure 4.22(A) and (B), respectively. First, the block-oriented sequential algorithm yields a significant speedup over the naive algorithm. Second, the figures also show a significant speedup for both PVM and MESSENGERS. For the larger block sizes this speedup is almost linear over the block-oriented sequential algorithm and in some cases is superlinear over the naive sequential algorithm (due to the poor caching performance of the naive sequential algorithm.) Specifically, the speedup for a 1000×1000 matrix multiplication on 4 processors using MESSENGERS is 3.7 over the block-oriented sequential algorithm and 4.5 over the naive sequential algorithm. On 9 processors, the speedup for a 1500×1500 matrix multiplication is 5.8 over the block-oriented algorithm and 6.7 over the naive algorithm. Finally, MESSENGERS further achieves speedup over PVM beyond approximately 150 and 50 blocks when using 2×2 and 3×3 processors respectively.

There are two main reasons for MESSENGERS performance superiority over PVM. First, no congestion occurs at the master node although Messengers from all slave nodes need to come back to the master for barrier synchronizations. This is because both the computation granularity and the amount of data broadcast and rotated for each stage are large enough to distribute the arrivals of Messengers at the master node.

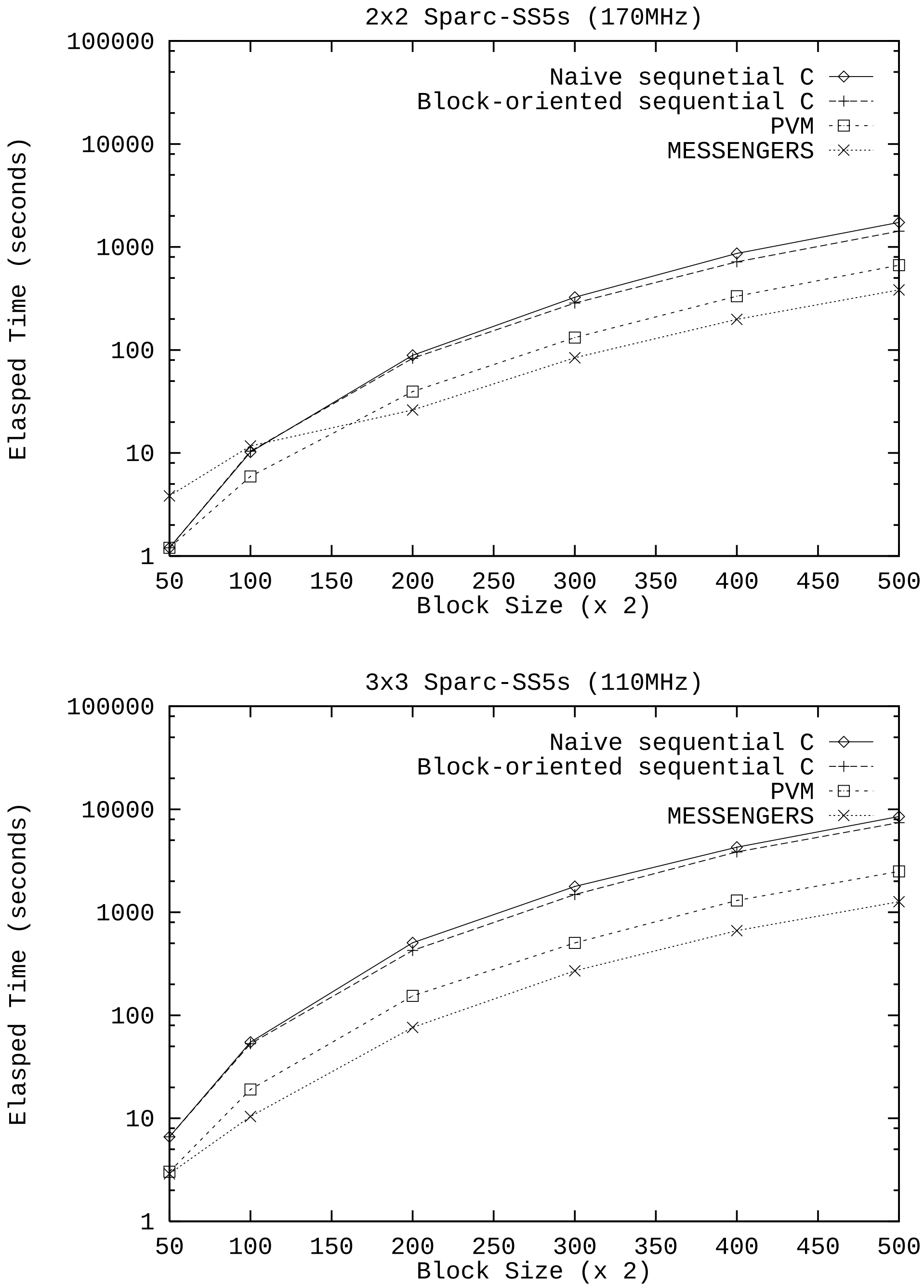


Figure 4.22. Matrix multiplication

Second, PVM's data copying overhead upon a broadcast of block A is proportional to the number of processors in a row. In contrast, MESSENGERS does not need any data copying operations at the destination although a Messenger carrying A needs to copy the data from node variables into its messenger variables at the source node. This is because the Messenger computes the multiplication by directly accessing A in its messenger variables and B in the node variables at the destination. There is no difference in a rotation of block B between PVM and MESSENGERS.

4.2.6 Individual-Entity-Based Simulation

In this section, we concentrate on MESSENGERS' performance only. The reasons, as discussed in Section 3.2.6, is that we would have to rewrite the program in PVM whenever we change the CPU configurations. As shown in Figure 3.15, all functionalities to manage particles in lists and exchange them with specific processes must be coded carefully, while MESSENGERS takes care of such particle management automatically.

First, we observe the performance of the particles random work, which is not tuned to reduce the number of unnecessary local and remote *hop()* statements. We especially investigate the problem size, (i.e., the grid size and the number of particles) and the effect of inserting local events. Then, we examine Wa-tor, a more sophisticated simulation, which reduces unnecessary *hop()* statements and provides the node boundary information.

1. Particles Random Walk

The performance tests have been conducted for two cases: (1) 36 particles start their random walk from the central 6×6 square of a 10×10 2D grid, and (2) 144 particles start from the central 12×12 square of a 20×20 grid. The second test case has a four times larger problem size in space and entities than the first case. We used 1, 4, and 9 slave CPUs. For the last two CPU configurations, the master CPU is separate from the slaves, and focuses on the global virtual time (GVT) computation.

Figure 4.23(A) and (B) shows the results of simulating 36 and 144 particles respectively. The curves show that MESSENGERS running on a single slave CPU catches up with the sequential program at approximately 10,000 floating-point operations in the 36 particles simulation, while the parallel versions do not achieve speed-up. Similarly, 144 particles simulation on MESSENGERS needs the same size of computation granularity to break even with the sequential C program, however its parallel executions also catch up with the sequential program.

We have also investigated the performance affected by inserting a certain number of dummy *func()* statements between any two consecutive *hop()* statements, each of which simply schedules the calling particle Messenger's activity at the next simulation time step. The purpose of this investigation is to determine the number of local events that would have to be scheduled for each simulation entity in order to overcome the overhead of local or remote *hop* operations. Figure 4.24 shows this result. With only one local event inserted between *hop()* statements, MESSENGERS execution on multiple CPUs already achieves speedup over the sequential version. Furthermore, the result shows that there is little change in the performance beyond eight local

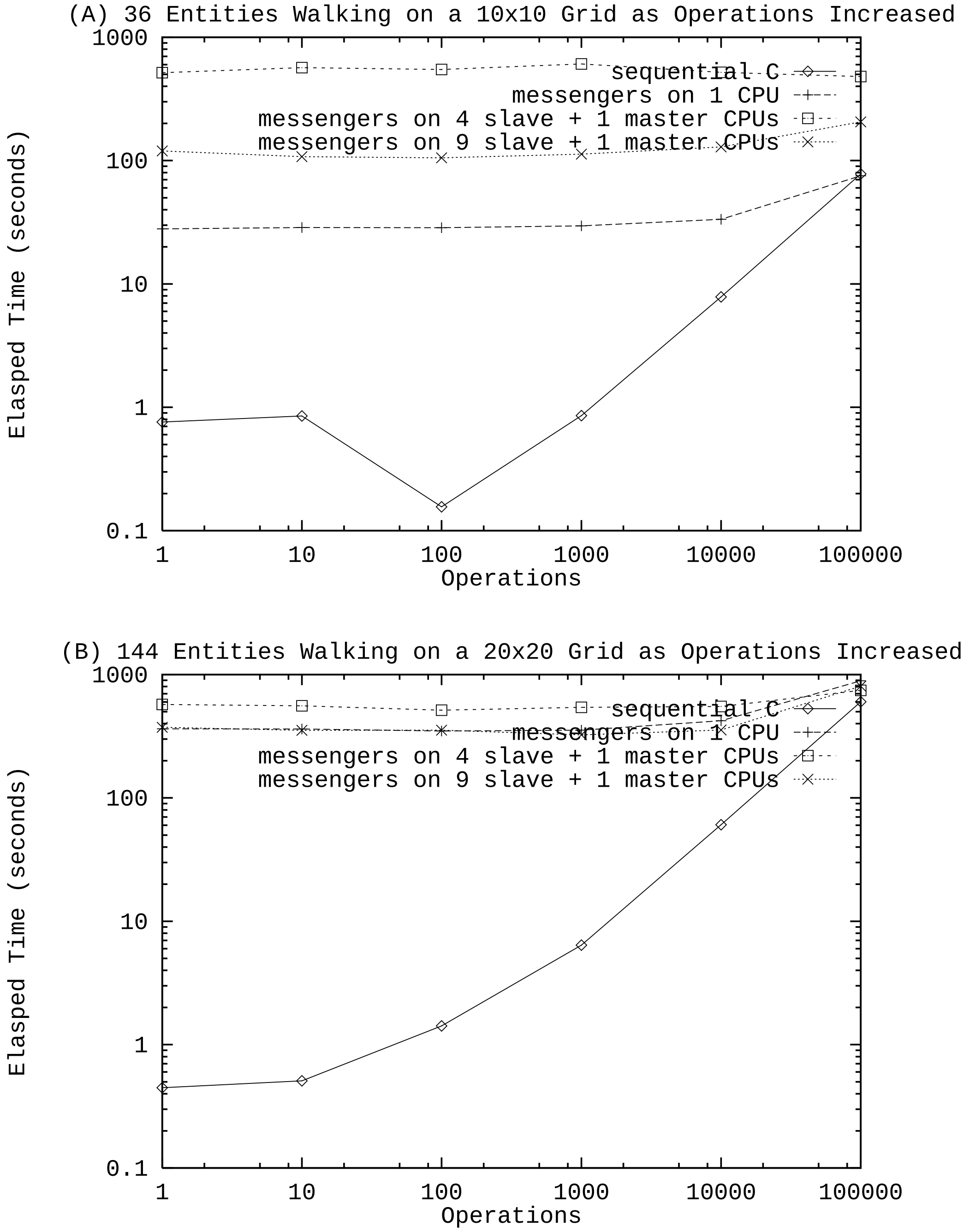


Figure 4.23. Particles random walk in 2D space (2000-virtual-time simulation on SS5-10E)

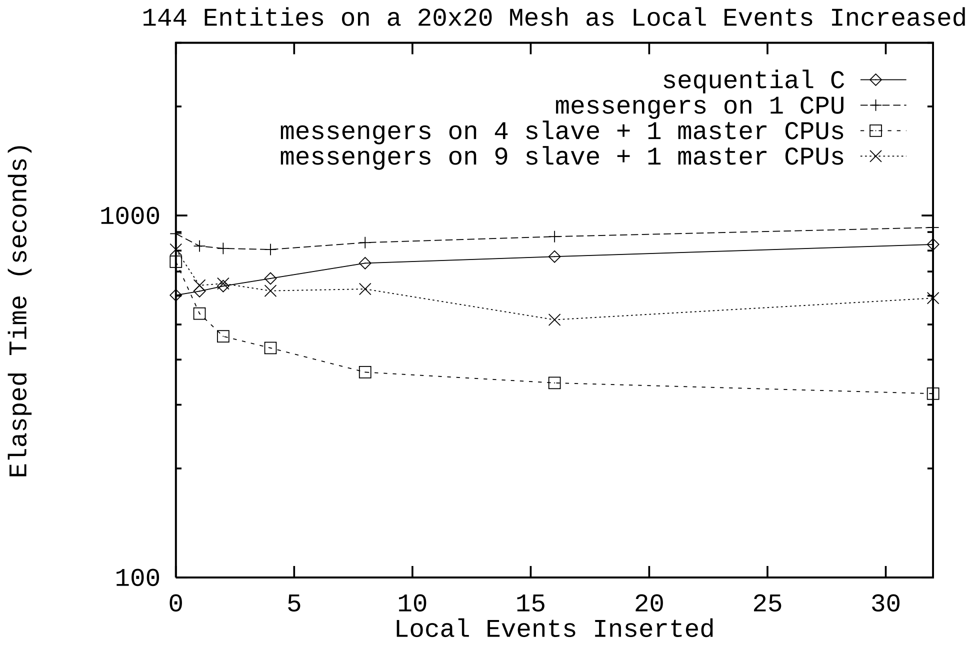


Figure 4.24. Particles random walk in 2D space as local events increased (2000-virtual-time simulation on SS5-10E)

events. Therefore, for the individual-based simulations, the key in achieving better performance is to reduce the unnecessary hops, which in turn increase the appropriate number of local events. As discussed in Section 3.2.6, our implementation of Wa-tor is based on one-to-one logical-to-daemon node mapping, and exchanges the boundary information of the simulation sub-space each daemon takes care of. Therefore, it reduces the unnecessary hops of simulation entities. In the following, we discuss the performance of Wa-tor.

2. Wa-tor

Figure 4.25 shows the performance results of the Wa-Tor program under increased computational granularity of each individual. The simulation space is represented as a 32×32 matrix and striped into 2, 4, and 8 segments. There

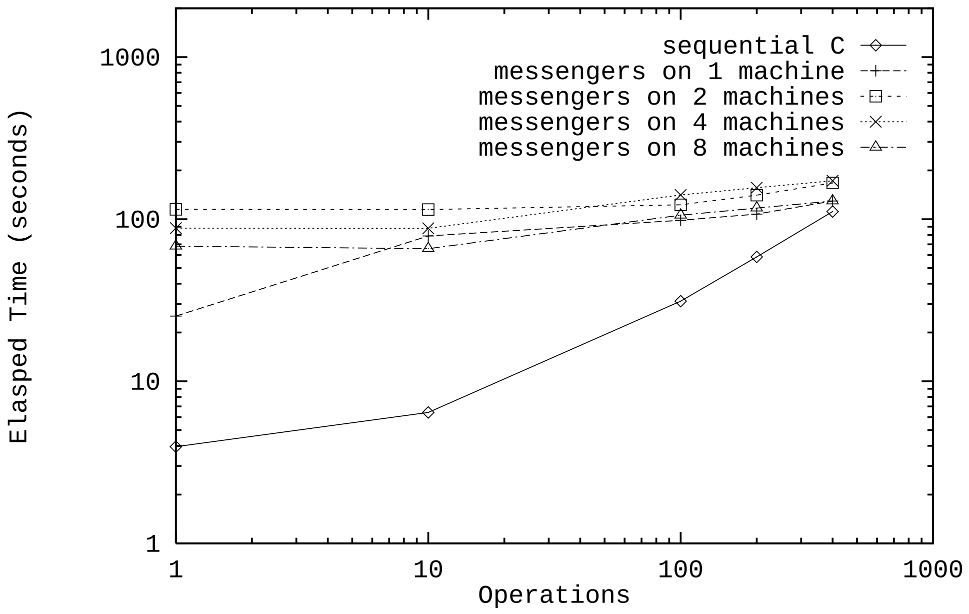


Figure 4.25. Wa-Tor program with 500 entities (100-virtual-time simulation SLC-10E)

are 500 Messengers representing fish and sharks. The results show that a relatively low number of operations (approximately 400) is sufficient to compensate for the interpretation and communication overhead. Furthermore, there is some speedup between different numbers of machines despite the relatively small number of individuals being simulated.

4.3 Summary

We have demonstrated that the Messengers performance is very competitive with practical message-passing systems, notably, *threads* and PVM, for almost all the example applications. Messengers has been shown to even have performance superiority over these two systems for several problems: (1) divide-and-conquer (2) information diffusion, (3) matrix multiplication, and (4) all other applications except circulatory simulation when sending sufficiently large amounts of data.

Chapter 5

Virtual Time Computing

This chapter discusses the necessity for and the role of the virtual time concept in autonomous object computing. Virtual time is the value of a monotonically increasing software counter. Using a priority queue, a single machine can simply order events with causal relationship along its *local virtual time* (referred to as LVT in the following discussion). However, in distributed systems, each machine maintains its own LVT which increases independently of the others. Thus, LVTs must be synchronized over the system to maintain the overall causal ordering of events [Lam78, BM93]. The globally minimal time obtained from this system-wide synchronization, which is referred to as *global virtual time (GVT)* [Jef85], is guaranteed to monotonically increase over the system. Events scheduled at the current GVT can be safely processed without violating causality. Thus, the virtual time concept is taken advantage of by event-driven applications, such as parallel discrete event simulations (PDES) [Fuj90b], fault recovery using event-logging mechanisms, and time-stamped transaction systems [CDK94]. It is also useful in solving a variety of distributed coordination problems, such as distributed termination. In the following discussion, we show how the virtual time concept simplifies the coding of MESSENGERS scripts compared to not only explicit synchronization schemes but even to analogous sequential C programs.

In order to minimize the number of messages exchanged and the blocking periods, *GVT* synchronizations must be implemented and optimized at system level rather than in each different application. Several schemes have been developed and tested so far, especially for PDES. However, there have been no experiments in the context of autonomous-object-oriented systems, especially using a LAN. We have developed a new implementation scheme that takes advantage of MESSENGERS' network architecture. We examine under what conditions our scheme shows better performance than other approaches to *GVT* synchronization.

5.1 Virtual-Time Maintenance

This section first demonstrates why the virtual time concept makes autonomous-objects-based programming easier than programming without the concept. It then reviews previous research on *GVT* synchronization and discusses the problems they have not yet addressed.

5.1.1 Distributed Synchronization Problems

A major concern with designing distributed applications is synchronization among parallel execution instances. Such synchronization problems range from simple hand-shake synchronization between two execution instances, to more complicated barrier synchronizations, and finally to the most complicated problems of distributed termination. In the following, we summarize how the message passing model, (i.e., PVM) and MESSENGERS address each of these synchronization problems, and demonstrate the necessity for the virtual time concept.

1. Simple synchronization:

The simplest synchronization is one requiring the exchange of hand-shake signals between two execution instances (for instance, a producer-and-consumer problem), which is used to coordinate their operation.

- **Message Passing:** As discussed in Section 4.2.5, synchronization between two execution instances is maintained implicitly by communication if the send and/or receive primitives are synchronous. PVM performs this type of synchronization with a synchronous *pvm_recv()*.
- MESSENGERS: Since Messengers are moving over the system, synchronization among them must be realized by a rendezvous. In other words, two Messengers must meet at the same logical node and synchronize on a shared node variable. Most existing mobile-agent systems actually use a similar form of rendezvous synchronization. Alternately, virtual time can be used to achieve the desired sequencing of events in most cases.

2. Barrier synchronization:

This synchronization ensures that all activities within a specific group, (e.g., all loop iterations) complete their computation before any activity following the barrier can commence.

- **Message Passing:** There are several algorithms to realize barrier synchronization: (1) a central server algorithm, where all instances send a completion message to the server, and wait for a permission to proceed. This is given by the server upon receiving all messages, (2) a broadcast algorithm, where each instance broadcasts a message to all others, and then awaits the arrival of a message from all other instances, indicating their completion, and (3) a ring-based algorithm, where a token is passed among all instances in a pre-specified order. The instance originating the token detects a synchronization when the token returns. PVM implements the barrier synchronization in *pvm_barrier()*, that is transparent to the users.

- MESSENGERS: Without the virtual time concept, Messengers can be used in the same way as passive messages to implement any of the three schemes discussed above. Virtual time permits a more elegant solution. By scheduling all concurrent activities during the same time interval and the subsequent activities during a later interval, a barrier effect is automatically created.

3. Distributed termination problem:

This problem is a more complicated form of barrier synchronization, since during a given computation phase, execution instances may be spawned or terminated, or dormant activities may be awakened by receiving a message. The goal is detecting when there are no more active execution instances and no more chances for new instances to be spawned or reactived. Parallel discrete event simulations (PDES) can be regarded as a continuous chain of distributed terminations, each of which corresponds to a virtual time tick.

The algorithms introduced for the barrier synchronizations cannot solve distributed termination since new instances may be spawned and terminated dynamically and existing instances do not know if they may be receiving additional messages in the future. To implement termination, we need special execution instances in each physical node, which observe the local activities and ensures by communicating with each other that there are no more activities (other than these special execution instances) in the system. In the following discussion, we refer to these special instances as synchronization processes in PVM or synchronization Messengers in MESSENGERS.

- **Message Passing:** There are two major schemes that use synchronization processes to detect a distributed termination: (1) a central server algorithm [Sam85] and (2) a ring-based algorithm [Mat93]. The central server algorithm has all synchronization processes send the server a completion message and wait for a response from the server. In order to eliminate in-transit messages, they also need to report the numbers of messages received from and sent to other synchronization processes respectively, so that the server can ensure that there are no more in-transit messages by waiting until the total numbers of messages sent and received over the system are equal. The server then informs all the synchronization processes of a distributed termination. The ring-based algorithm passes a token among all instances in a prespecified order three times. In the first circulation, each synchronization process keeps the received token until it detects that there are no local activities, and then passes the token with the vector of the numbers of messages sent to other processes. In the second circulation, each process keeps the token until it receives the same number of messages as the total number of messages already sent from the others to this process. The third circulation informs all processes of a distributed termination.
- MESSENGERS: The algorithms discussed above for the message-passing model may be emulated using Messengers. Virtual time relieves users from implementing these algorithms in Messenger scripts, and thus greatly simplifies applications. By scheduling all concurrent activities at a given virtual time

interval and a termination signal at the next virtual time interval, the signal will automatically be delayed until all previous activities have terminated.

Advantages of Using Virtual Time for Synchronization

The concept of virtual time permits implementing synchronization at system level by invoking virtual time primitives. Its programming and performance advantages are:

- Applications do not have to implement synchronization schemes independently. Since the system manages all execution instances and keeps track of all in-transit messages, it is natural to implement the synchronization at system level also. This greatly simplifies each application program.
- The optimistic synchronizations (discussed later in Section 5.1.3) can only be realized at system level. This is because process image or Messenger status must be saved and restored for rollback operations, which is handled by the systems but is not visible to user processes or Messengers themselves.
- The system-level implementation achieves higher performance than at user-level. A user-level implementation would likely increase process context switching among daemons, synchronization processes, and other user processes.

For these reasons, MESSENGERS has implemented virtual time synchronization in the interpreter daemon. The rest of this chapter reviews existing virtual time synchronization techniques, presents a new scheme developed for MESSENGERS, and then validates its advantages by demonstrating how Messenger scripts are simplified and how much speed-up the MESSENGERS synchronization scheme achieves over other approaches.

5.1.2 Conservative Synchronization

Conservative synchronization is the simplest virtual time implementation that (1) allows each machine to process only events scheduled at the current GVT, (2) at each time increment, finds the next minimum LVT over the system, and (3) advances GVT to this LVT.

The main problem is the detection of in-transit Messengers as discussed in the central server algorithm for distributed termination detections in Section 5.1.1. Given a master daemon, each daemon has to inform the master of the numbers of its incoming and outgoing Messengers as well as its next LVT, so that the master can confirm that there are no more in-transit Messengers by comparing the number of incoming Messengers and that of outgoing Messengers.

To illustrate the algorithm, consider a daemon network connecting four daemon nodes linearly as shown in Figure 5.1. At each daemon node N, Messengers schedule their activities along a local virtual time line as events $ev(N, i)$ where N and i denote the daemon node and the sequential event number ordered chronologically. (For instance, $ev(A, 1)$ is the first event scheduled at the daemon node A.) Assuming that the current GVT is $t + 1$, conservative synchronization allows the daemons A and D to process events $ev(A, 1)$ and $ev(D, 1)$ in parallel, and then has all daemons report their own minimum LVTs. The daemons A, B, C, and D report $t+2$, $t+3$, $t+5$, and $t+4$ respectively. The algorithm then advances GVT to $t + 2$, and allows the daemon A to process $ev(A, 2)$.

As shown in Figure 5.1, this scheme requires many unnecessary synchronizations. For instance, despite the fact that daemon A can actually proceed up to

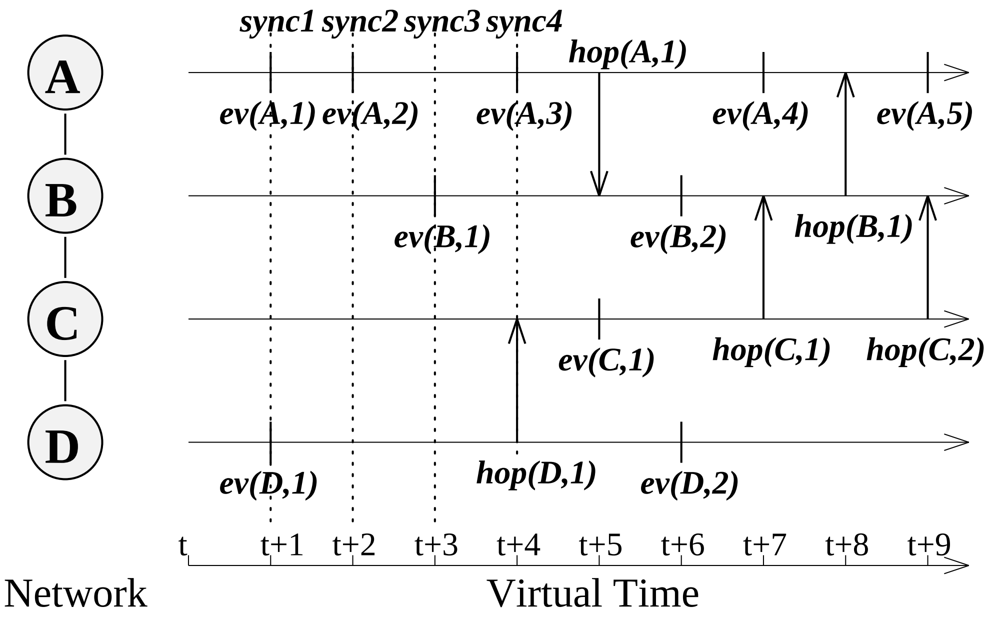


Figure 5.1. Conservative synchronization

$ev(A, 4)$ independently, it is forced to synchronize with the other daemons four times before $ev(A, 4)$. Also, the synchronizations may serialize unrelated events and hops, for instance, $ev(A, 2)$, $ev(B, 1)$, and $hop(D, 1)$.

5.1.3 Optimistic Synchronization

As observed in conservative synchronization, each physical node can process its own events until it receives a Messenger from another node. However, it is generally impossible to predict the arrival of a Messenger, unless the application has a known uniform pattern of Messenger exchanges. To achieve more parallelism, optimistic synchronization [Fuj90a] permits each node to continue processing events that are considered unrelated, and to have the capability to roll back its computation when receiving a straggler Messenger whose time stamp is older than this node's LVT. This synchronization is known as optimistic synchronization. The following summarizes the three major optimistic synchronization schemes studied so far.

1. **Risk-Free Optimistic Synchronization**

   This scheme (1) allows each physical node to process its local events until it encounters the first outgoing Messenger, (2) exchanges its LVT with others (3) finds the minimum LVT and distributed it as the new GVT, and (4) permits the nodes to send Messengers scheduled at this new GVT.

   A node may have proceeded beyond the new GVT before it receives Messengers from other nodes and must therefore roll back its computation to the new GVT to maintain causality. For this anticipated rollback, the main techniques of this algorithm are (1) periodical check-pointing of each node's intermediate computation status, and (2) garbage collection of outdated status at each GVT synchronization. Rollback operations take place inside each node locally but do not affect other nodes, since the actual release of outgoing messages is postponed until a new GVT is decided. This risk-free optimistic synchronization scheme has been employed in SRAD [DR90] as "local rollback" and SPEEDES [Ste92] as "breathing time bucket".

   Figure 5.2 shows a timing chart illustrating this synchronization scheme over the same daemon network as the one used in Section 5.1.2. Assuming that the current GVT is $t+1$, daemons A, B, C, and D are allowed to process all local events up to $hop(A,1)$, $hop(B,1)$, $hop(C,1)$ and $hop(D,1)$, as indicated by the "$\surd$" symbol on each line. Thereafter, they report $t+5$, $t+8$, $t+7$, and $t+4$ as their next minimum LVTs respectively. The algorithm then advances GVT to $t+4$ and allows the daemon D to commit $hop(D,1)$. Upon receiving the Messenger from D, daemon C must locally roll back its computation to

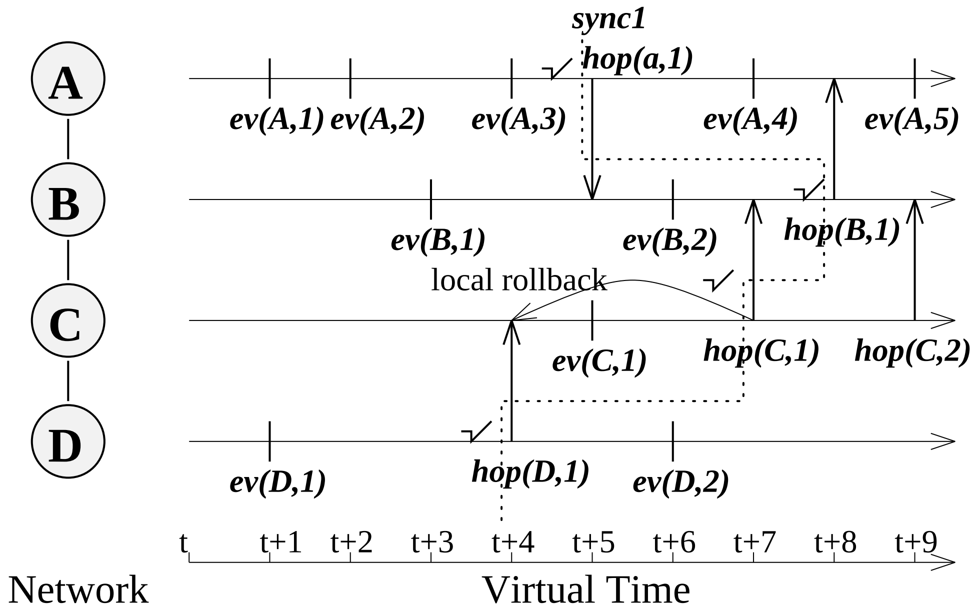


Figure 5.2. Risk-free optimistic synchronization

$t+4$. Daemons A and D have to simply wait until GVT reaches their LVTs, (i.e., $t+5$ and $t+8$ respectively.)

The advantages of this algorithm are two-fold: (1) avoiding unnecessarily conservative synchronizations, and (2) limiting the scope of rollback operations to the node itself. However, the algorithm still incurs unnecessary overhead. In Figure 5.2, only $hop(D,1)$ is committed at the new GVT, although the system does not violate causality even if the daemon A commits $hop(A,1)$. As a result, the algorithm serializes the transfers of all Messengers that do not occur at the same time instant, even if they do not have any causality.

2. **Fully Optimistic Synchronization**

This algorithm allows all physical nodes to process their own events and commit Messenger transfers independently without any restrictions. Each node, however, must roll back its computations upon receiving a straggler Messenger whose time stamp is older than the node's LVT. During rollback operations it may find that Messengers have already been erroneously released to other nodes and thus must be cancelled. Another problem is the timing of garbage collection to release outdated states. Hence, in addition to the techniques of the risk-free optimistic synchronization, the fully optimistic synchronizations needs two more techniques: (1) cancellation of erroneous messages by sending anti-messages, and (2) periodic global synchronization to update GVT for garbage collection. Timewarp [JBW+87] realized this fully optimistic synchronization.

Figure 5.3 describes this algorithm working over the same daemon network as in the previous two synchronization schemes. Assume that daemons A, B, C, and D process their LVTs to $t+9$, $t+8$, $t+9$, and $t+4$ respectively, (as indicated by the "$\surd$" symbol.) Upon receiving $hop(D,1)$, daemon C has to roll back its computation to $t+4$. During this rollback operation, C finds that $hop(C,1)$ has been committed and thus sends an anti-message to daemon B, which then rolls back its computation to $t+7$ and sends an anti-message to daemon A to cancel $hop(B,1)$. Finally, daemon A turns back its LVT to $t+8$.

The underlying idea of this algorithm is to have each node process as many events as possible, expecting that most events will occur in the order of their causalities, thus avoiding unnecessary blocking. However, the algorithm has a major drawback known as a "domino effect", which may cause a certain rollback to produce a chain of anti-messages, (for instance those for $hop(C,1)$ and $hop(B,1)$ in Figure 5.3), resulting in a system-wide cascading rollback.

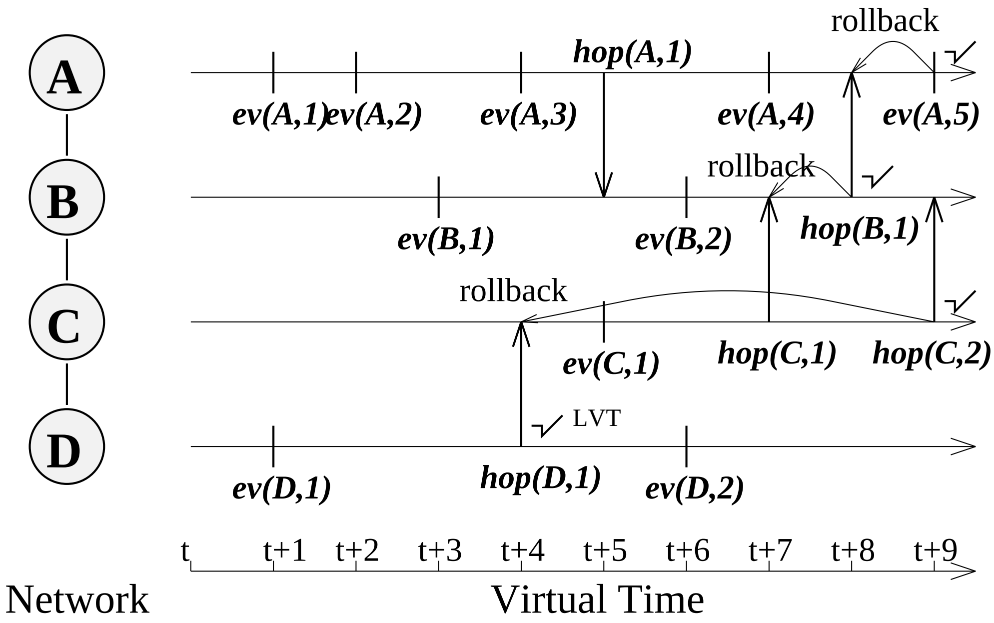


Figure 5.3. Fully optimistic synchronizations

3. **Optimistic Time Window**

Given a certain time window size, this algorithm (1) allows all physical nodes to process their own events and Messenger transfers independently within this window, and (2) has all physical nodes synchronize with each other when they reach the upper limit of their LVTs. Like the fully optimistic synchronization, a straggler Messenger causes the receiving node not only to roll back its computation but also to cancel erroneous Messengers. Both, however, are bounded by the fixed window size. The algorithm is a balanced combination of the previous two optimistic synchronization schemes. Within each given time window, it works as fully optimistic synchronization. The end of time window corresponds to risk-free synchronization. This idea was developed as a revised form of the breathing time bucket in SPEEDES and was called "breathing timewarp" [Ste93].

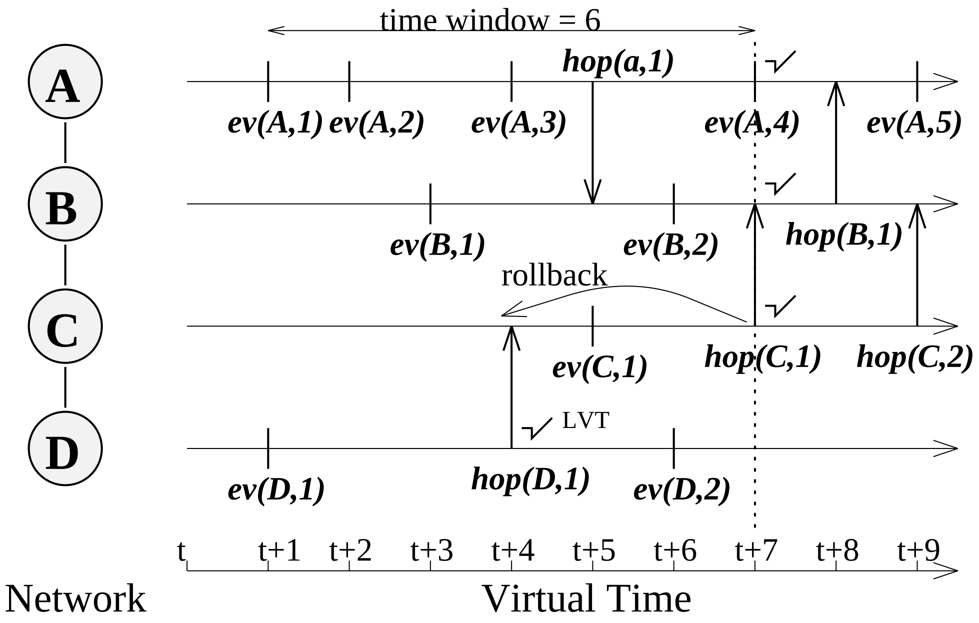


Figure 5.4. Optimistic time windows

Figure 5.4 illustrates this algorithm working on the same daemon network used in the previous three synchronization schemes. Assuming that the current GVT is $t+1$ and a time window is set to 6, daemons A, B, and C have already advanced their LVTs to $t+7$, while D is about to commit $hop(d,1)$ at $t+4$. The daemon C needs to roll back its computation to $t+4$ as well as send an anti-message for $hop(c,1)$. However, unlike the fully optimistic synchronizations, daemon B does not have to turn back its LVT and thus further cascading rollbacks will not be triggered.

The underlying idea is derived from the observation that there are few causal interactions among different nodes within any short time interval. By adding a temporal restriction, (i.e., a time window), this scheme shrinks the time lag between the fastest and the slowest nodes, and therefore prevents cascading rollbacks. However, the problem is how to decide the appropriate window size which is generally application-dependent.

5.1.4 Discussion

The introduction of the virtual time concept offers efficient solutions to various synchronization problems. We reviewed several virtual time synchronization schemes. The conservative synchronization is easy to implement. It may however produce numerous unnecessary synchronizations and serialize many independent events. The optimistic synchronization can alleviate this problem by having each node optimistically process its local events in parallel. However, without any restrictions, this type of synchronization may cause cascading rollbacks. The optimistic time window scheme may be effective to prevent such rollback explosions. However, the best window size is application-dependent and even varies during execution because the occurrence of causal messages is not constant. Nevertheless, since existing systems do not consider which physical nodes exchange messages frequently, a temporal restriction is the best known way to address cascading rollbacks.

5.2 MESSENGERS Synchronization Scheme

This section describes the MESSENGERS virtual time synchronization scheme. The main idea is to use the knowledge of the daemon network topology in order to prevent rollback explosions. While the optimistic time window restricts the time lag between the fastest and the slowest nodes over the entire network, the MESSENGERS' scheme limits the lag between any set of neighboring nodes. These two ideas are orthogonal and may be combined in order to provide an even stronger restriction on the optimistic computation. In the following, we detail our scheme and discuss its pros and cons.

5.2.1 Use of Knowledge of Daemon Network Topology

Most virtual-time systems today consist of message-passing processes which do not maintain a system-wide logical network structure and thus communicate with each other without distinguishing between neighboring or non-neighboring processes. This is the main reason why the optimistic time window has been the only effective scheme to address the domino effect.

The MESSENGERS' daemons on the other hand, are linked into a separate topology (Section 2.2). The resulting daemon network generally represents the heavy traffic routes of Messengers. The implicit mapping (Section 2.4.3) causes new logical nodes to be mapped to neighbors of the current daemon node according to its threshold value, and therefore most logical links coincide with daemon links. Messengers generally navigate along logical links more frequently rather than jumping directly to remote destinations, and hence, neighboring daemons interact with each other much more frequently than non-neighboring daemons. From this observation we can infer that a daemon node has a greater chance of receiving straggler Messengers from its neighbors than its non-neighbors. Therefore, equalizing the LVTs of neighboring daemons is likely to reduce local rollbacks, which in turn is likely to better prevent rollback explosions.

5.2.2 Proposed Protocol

The normalization of neighboring daemons' LVTs consists of four steps: (1) exchanging LVTs among neighboring daemons, (2) reporting LVTs to the master daemon to decide a new GVT, (3) rolling back to the locally minimum LVT, and (4) receiving a new GVT from the master daemon. Figure 5.5 illustrates these four steps:

1. **LVT Exchange:** Each node first multicasts its LVT to its neighbors when it needs to increase free memory space by releasing outdated events or it receives a timer interrupt. Upon receiving LVTs from all its neighbors, the daemon node finds out the minimum and the maximum among these LVTs. For example, the daemon B in Figure 5.5 has two neighbors, A and C, and recognizes that $t + 9$ and $t + 6$ are the local maximum and minimum respectively.

2. **Report to Master:** Each daemon reports a set of its neighboring LVTs to the master daemon which is in charge of deciding the next GVT. However, all daemons do not have to report to the master, since neighboring daemons may provide redundant LVTs reports. Therefore, the master daemon can determine LVTs of all the daemons by receiving LVT reports from only these daemons whose LVT is not locally maximum, (i.e., B, C, and D in Figure 5.5.)

3. **Pre-rollback:** Each daemon whose LVT is not locally minimum starts a pre-rollback to the local minimum LVT while it is waiting for the next GVT given by the master daemon.

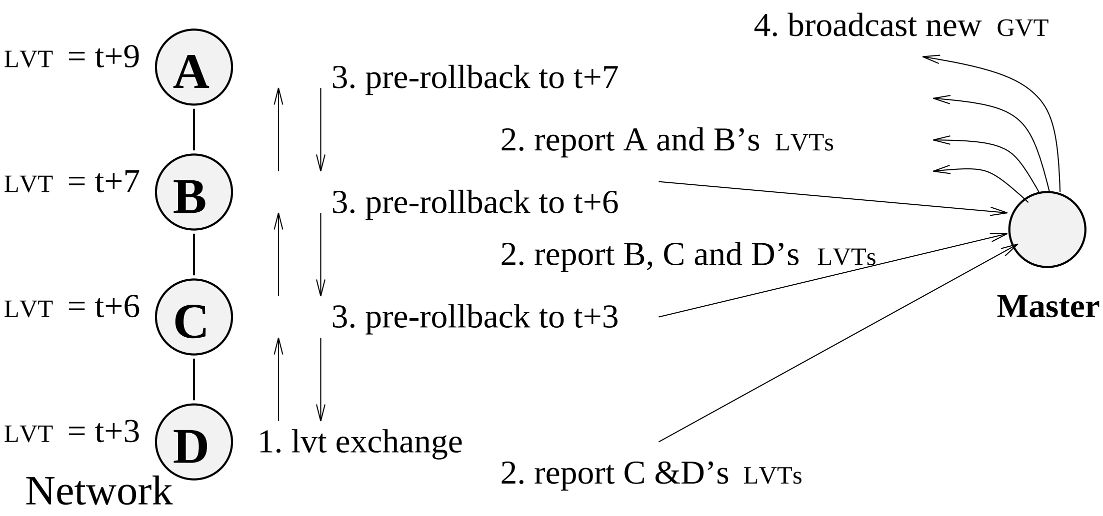


Figure 5.5. Normalization of neighboring daemons' LVT

4. **New GVT:** As soon as the master obtains LVTs for all daemons, it broadcasts the new GVT to these daemons. The master daemon may already know all LVTs before it receives a LVT message from all the daemons that are supposed to report LVT. In Figure 5.5, for instance, the master finds out all LVTs with messages from B and D, and thus it does not have to wait for a message from C. In that case, the master discards the message from C after confirming that this message is not sent to the master for the next GVT synchronization phase.

Figure 5.6 shows a timing chart when this synchronization scheme takes place over the same daemon network as the one used in Figure 5.5. Assume that the daemons A, B, C, and D process their LVTs to $t+9$, $t+7$, $t+6$, and $t+3$ when they need a GVT synchronization to execute garbage collection. After a LVT exchange phase, A determines that it has the locally maximal LVT, and thus does not report its LVT to the master. While it is waiting for a new GVT from the master, it starts a pre-rollback to $t+7$. On the other hand, B and C determine that

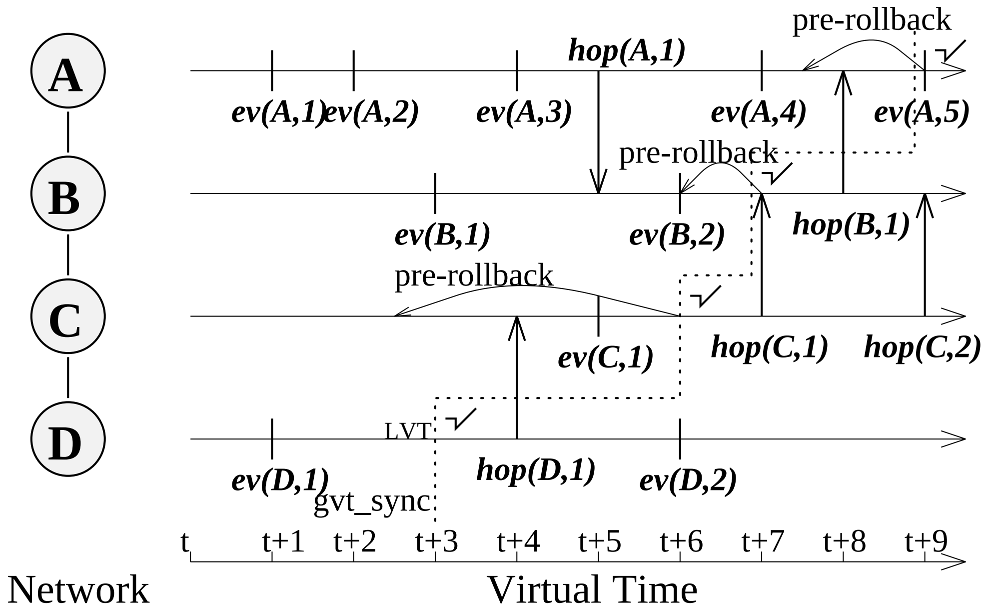


Figure 5.6. Timing chart of MESSENGERS synchronization scheme

their LVTs are neither locally maximum nor minimum, and therefore they report their LVTs to the master as well as rolling back to $t+6$ and $t+3$ respectively. D finds out that its LVT is the local minimum and thus only reports its LVT to the master but does not perform a rollback. The master then broadcasts the new $GVT(t+3$ to all nodes. Thereafter, A, B, and C have a lower possibility of further rollback operations when Messengers arrive through $hop(b,1)$, $hop(c,1)$, and $hop(d,1)$ in the near future.

5.2.3 Discussion

Advantages

The advantages of the MESSENGERS' virtual time synchronization scheme are four-fold:

1. While MESSENGERS synchronization scheme minimizes the temporal lag between neighboring daemons, it allows more temporal lag between non-neighboring daemons because they have more geographic distance. Therefore, Messengers with greater geographic separation can process their events more in parallel. On the other hand, the optimistic time window defines this temporal lag globally. Wherever Messengers are running, their executions are globally restricted by the same window size.

2. In the optimistic time window, the optimal window size varies during the course of execution. On the other hand, MESSENGERS synchronization scheme is based on the daemon links which coincide with the logical network.

3. The MESSENGERS synchronization scheme allows each daemon to keep exchanging Messengers at any point in time. On the other hand, the optimistic time window pauses each daemon once it proceeds beyond the window.

4. MESSENGERS synchronization scheme is orthogonal to the optimistic time window. An application can combine these two schemes when an application needs to allow Messengers to jump to non-neighboring daemons and wants to restrict the temporal lag over the system as well as the temporal lag between neighboring daemons.

Disadvantages

The disadvantages of the MESSENGERS synchronization scheme are two-fold:

1. The number of messages required for each GVT phase is proportional to that of the daemon links because each daemon must exchange its own LVT with the neighbors. Assuming that the existing synchronization schemes

use a master-slave broadcast scheme to compute a new GVT, the number of necessary messages is $2N$, where N is the number of daemon nodes excluding the master. On the other hand, given a linear network of daemon nodes as shown in Figure 5.5, $2(N-1)$ messages are needed during a LVT exchange phase, and a minimum of $\frac{N}{2+N}$ to a maximum of $2N-1$ messages are required during a GVT decision phase. The total number of messages ranges from $(3.5N-2)$ to $(4N-3)$. However, each daemon initiates this type of GVT synchronization only when it needs to discard outdated events or when a timer interrupt occurs. (This takes place in a time frame of every several hundred milli-seconds to every few seconds.) Thus the number of messages for each GVT phase can be regarded as small compared to the numbers of Messengers and anti-messages exchanged over the network.

2. Pre-rollbacks occur at some daemon nodes every GVT synchronization, and hence the total number of rollbacks may occasionally be larger than with the fully optimistic synchronization. However, each such pre-rollback is expected to occur over a short time interval, and thus generates few anti-messages. Furthermore, short pre-rollbacks are not likely to result in cascading rollbacks.

In the quantitative evaluation of Section 5.4, we measure how the advantages and disadvantages affect the performance of the MESSENGERS synchronization scheme.

5.3 Qualitative Evaluation

This section demonstrates how virtual time programming addresses distributed synchronization problems. To provide concrete illustrations, we use the actual code of four applications: a naive shortest path program, Dijkstra's algorithm, a particle random walk simulation, and a matrix multiplication.

5.3.1 Naive Shortest Path Program

In the following, we show that virtual time programming simplifies a shortest path program, which needs to detect its "distributed termination" before the answer is displayed.

Given a graph where links carry non-negative weights, this program finds the shortest path between any two points. Figure 5.7 shows a two-dimensional grid as such a graph example. A naive algorithm is a diffusion over the graph to exhaustively examine all possible routes. After the search, the program displays the shortest path in reverse order, (i.e., from the destination to the source nodes.)

Figure 5.8 shows two Messenger scripts for this algorithm. The *find_path* Messenger finds the shortest path and the *show_spath* Messenger displays the result. The *find_path* starts from the source node, (i.e., *n_00* in the case of the two-dimensional grid shown in Figure 5.7). Whenever the Messenger arrives at a node, it examines two conditions: (1) if the current node has been visited by this Messenger for the first time or (2) if its previously computed distance *dis_recent* is larger than the distance *dist_accum* of the path along which this Messenger has

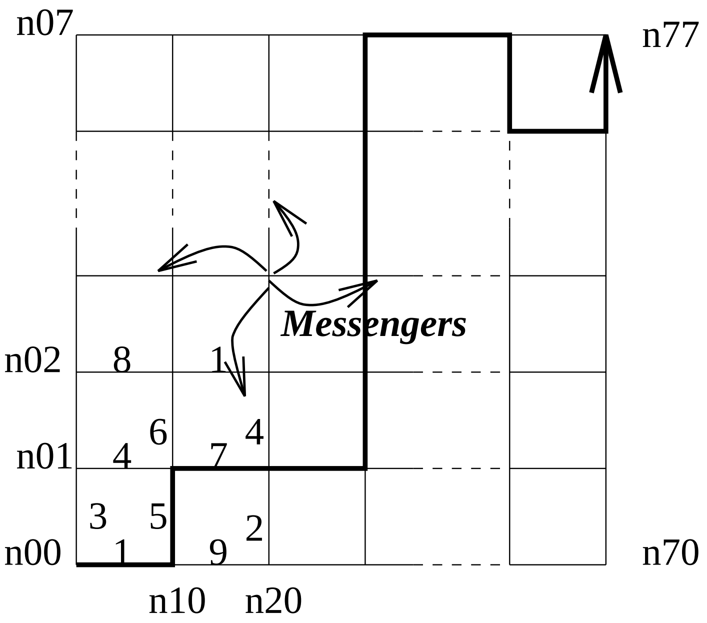


Figure 5.7. Shortest path program over a 2D grid

traveled (line 4). If none of these two conditions are satisfied, the Messenger just exits. Otherwise, the Messenger marks this node as *VISITED* (line 8), updates its distance *dist_recent* to the source node (line 9), records the predecessor node in *addr_prdr* (line 10) and propagates to all the neighbors. After all *find_path* Messengers are terminated, the shortest path is recorded as a pointer chain of all the nodes on this path. The *show_path* Messenger may thereafter begin navigating along the shortest path in reverse order from the destination to the source and display all the nodes it visits (lines 16-19).

The most important statement of this code is line 15 which schedules line 16 and the subsequent statements to be executed in the current virtual time plus one. Since all the propagated *find_path* Messengers are terminated in the current virtual time, line 15 guarantees that the *show_path* Messengers display the result after the complete termination of *find_path* Messengers. Without line 15, the application

```
 (1) find_path(i, j) {
 (2)   hop(node = "n_00");
 (3)   final_node = "n_" & i & j;
 (4)   while (status != VISITED || dist_accum < dist_recent) {
 (5)     dist_accum += $weight0;
 (6)     if ($node == final_node && status != VISITED)
 (7)       func(name = "M_inject"; in = "show_path", i, j);
 (8)     status = VISITED;
 (9)     dist_recent = dist_accum;
(10)     addr_prdr = addr_prev;
(11)     addr_prev = $address;
(12)     hop();
(13) } }

(14) show_path(x, y) {
(15)   func(name = "M_sched_time_dlt"; in = 1);
(16)   func(name = "print_node"; in = $node, $weight0);
(17)   while (addr_prdr != NULL) {
(18)     hop(node = addr_prdr);
(19)     func(name = "print_node"; in = $node, $weight0);
(20) } }
```

Figure 5.8. Exhaustive shortest path program using MESSENGERS with virtual time

needs to keep track of active *find_path* Messengers. One solution is that each *find_path* Messenger reports the number of Messengers created at every propagation and its termination to the logical node *n_00*. The final Messenger reporting to *n_00* assures complete termination and injects a *show_path* Messengers. With virtual time computing, such complicated behavior is simplified into a single line of code (line 15 in Figure 5.8.)

Figure 5.9 shows the corresponding sequential program code. The distance computation at each node (lines 6-11 in Figure 5.9) is similar to that of the Messenger script (lines 4-9 in Figure 5.8). Since the MESSENGERS daemon interprets duplicate Messengers in a round-robin fashion, the sequential version must achieve the graph propagation using a breadth-first search. It therefore stores nodes visited at the current search stage into the *next_level* queue (line 13). At

```
(1) breadth_first(i, j) {
(2)   current_level = &n[0][0];
(3)   while (current_level != NULL) {
(4)     while ((node = dequeue(current_level)) != NULL) {
(5)       while ((neighbor = dequeue(neighbor_queue, node)) != NULL) {
(6)         if (neighbor->status != VISITED
(7)         ||  node->dist_recent + neighbor->weight < neighbor->dist_recent) {
(8)            neighbor->dist_recent = node->dist_recent + neihgbor->weight;
(9)            neighbor->status = VISITED;
(10)           neighbor->addr_prdr = node;
(11)           enqueue(next_level, neighbor);
(12)    } } }
(13)    current_level = next_level; }
(14)  show_path(i, j);
(15) }

(16) show_path(x, y) {
(17)  node = &n[x][y];
(18)  printf("node = %s, weight = %d\n", node->name, node->weight);
(19)  while (node->addr_prdr != NULL) {
(20)    node = node->addr_prdr;
(21)    printf("node = %s, weight = %d\n", node->name, node->weight);
(22) } }
```

Figure 5.9. Sequential program using breadth-first search for shortest path

the next stage, it picks up these nodes one-by-one (line 4), and examines all their neighbors (line 5).

The Messenger script's superiority over the sequential version is three-fold:

1. The sequential program needs to implement a breadth-first search, whereas the Messenger code does not have to deal with this, because the MESSENGERS daemon automatically interprets over all Messengers.

2. The sequential program needs to implement two queues, *next_line* to store nodes visited at the current search stage, and *neighbor_queue* to store the neighbors of each node. The MESSENGERS code has no explicit queues.

3. The sequential program needs to know the network topology of a given graph *a priori* in order to make *neighbor_queue*, while the *find_path* Messenger script is independent from and reusable for any network topology.

All three of these points make the MESSENGERS program easier to understand, despite the fact that it is a parallel program, and thus can take advantage of multiple processors.

5.3.2 Dijkstra's Algorithm

As a second example, we show how virtual time programming easily changes the naive shortest path algorithm into the more efficient Dijkstra's algorithm by adding only a single statement.

Given the same weighted graph as used in the naive algorithm, Dijkstra's algorithm checks all neighbors of the node currently visited, but always selects the "lightest" neighbor as the next node it should visit. Therefore, the first path the algorithm traverses is the shortest path, whereas the naive algorithm has to traverse all possible paths exhaustively.

Figure 5.10 shows a *find_path* Messenger script implementing Dijkstra's algorithm. Note that the Messenger script for the naive algorithm as shown in Figure 5.8 is modified into Dijkstra's algorithm by just adding the virtual time scheduling function in line 13. This function has the calling Messenger wait for a period of time proportional to the weight of the last traversed link, which paces the propagation of *find_path* Messengers along the virtual time line. Therefore, the shortest path is traversed earlier than the other routes.

```
find_path(i, j) {
  hop(node = "n_00");
  final_node = "n_" & i & j;
  while (status != VISITED || dist_accum < dist_recent) {
    dist_accum += $weight0;
    if ($node == final_node)
      func(name = "M_inject"; in = "show_path", i, j);
    status = VISITED;
    dist_recent = dist_accum;
    addr_prdr = addr_prev;
    addr_prev = $address;
    hop();
    func(name = "M_sched_time_dlt"; in = $weight);
} }
```

Figure 5.10. Dijkstra's algorithm using Messengers with virtual time

Figure 5.11 shows a sequential program implementing Dijkstra's algorithm. The program maintains a fringe list which includes nodes on the fringe of a traversed tree. This fringe list is a priority queue sorting all its nodes in distance order. From this list, the program picks up the node whose distance from the source node is the smallest (line 15) and checks all its neighboring nodes (line 6). If any of the neighbors has not yet been visited (line 8) or if it is on the fringe and its distance can still be decreased (line 7-8), it is enqueued into the fringe list with its new distance (line 10-13). When the program picks up the destination node (line 5) or the fringe list becomes empty (line 15), the program finds out and displays the shortest path (line 18).

The Messengers programming advantages for Dijkstra's algorithm is threefold:

1. The sequential version needs to sort the fringe nodes, while such scheduling operations are supported by the Messengers virtual time features.
2. The sequential version must implement two queues: *fringe_list* and *neighbor_queue*, while the Messengers version does not have to implement any.

```
breadth_first(i, j) {
  node = &n[0][0];
  final_node = &n[i][j];
  fringe_list = NULL;
  while (node != final_node) {
    while ((neighbor = dequeue(neighbor, node)) != NULL) {
      if (neighbor->status == UNVISITED ||
         (neighbor->status == FRINGE &&
           node->dist_recent + neighbor->weight < neighbor->dist_recent)) {
        neighbor->dist_recent = node->dist_recent + neighbor->weight;
        neighbor->addr_prdr = node;
        neighbor->status = FRINGE;
        enqueue_and_sort(fringe, neighbor);
    } }
    if ((node = dequeue(fringe)) == NULL)
      break;
  }
  show_path();
}
```

Figure 5.11. Sequential program using Dijkstra's algorithm for shortest path

3. The sequential version must be given a fixed network topology *a priori*, while MESSENGERS supports any network topologies.

5.3.3 Individual-Based Simulation

We use the particle random walk simulation over a 2D grid to show how simulation programmers can focus on models of their own simulation entities without having to implement complex schemes for temporal coordination. We discussed modeling the particle's behavior in Section 3.2.6. The *random_walk* Messenger in Figure 3.15 moves randomly every time the variable *gvt* is incremented (line 16), however it does not include the code to increment *gvt*. Without the virtual time features, a user must implement a synchronization Messenger to distribute the virtual time over the network for each iteration. Figure 5.12 illustrates this Messenger's script. (The algorithm is identical to the one discussed in Section 5.1.1.)

Each node maintains the numbers of messengers incoming to and outgoing from this node in *node.in_msgr* and *node.out_msgr*, and the next event's LVT in *node.min_future_lvt*. It also keeps GVT in its *node.gvt* variable. These values are zero-initialized by the daemon when they are allocated. The synchronization Messenger propagates itself to each node (line 16), where it updates *node.gvt* and stores the values of *node.in_msgr*, *node.out_msgr*, and *node.min_future_lvt* in its corresponding *Messenger variables*; *msgr.in_msgr*, *msgr.out_msgr*, and *msgr.lvt* (lines 18-20). Each Messenger then hops back to the master node (line 21) where it sums up the number of Messengers sent and that of Messengers received at each node (lines 22 and 23). Only the last Messenger returning from a slave node continues working (lines 26 and 27) and determines that there are no in-transit Messengers when the contents of *node.in_msgr* and *node.out_msgr* are identical (line 28). If they are identical, this last Messenger updates GVT (line 29), and broadcasts the new GVT value (line 16). Otherwise, it simply hops out again to collect the new contents of *node.in_msgr* and it node.out_msgr from each node.

The virtual time features keep simulation programmers from having to provide such a time-control program. In addition, the spin wait on *gvt* by the original *random_walk* (lines 16-19 in Figure 3.15) can be greatly simplified as shown in Figure 5.13.

The *M_sched_node* function suspends the calling Messenger until *resident_id* becomes unequal to *my_id* or until GVT is incremented. (Section 2.4.4 explains how this function works.)

```
/* node variables ... declared in another file in actual implementation */
struct {int in_msgr;        /* # msgrs incoming to this node            */
        int out_msgr;       /* # msgrs outgoing from this node          */
        int min_future_lvt; /* # the next event's lvt at this node      */
        int gvt;            /* gvt                                      */
} node;

/* messenger variables */
struct {int in_msgr;     /* used to carry node.in_msgr to master        */
        int out_msgr;    /* used to carry node.out_msgr to master       */
        int lvt;         /* used to carry node.min_future_lvt to master */
        int gvt;         /* used to carry gvt from master to others     */
} msgr;

gvt(sim_time, total_nodes) {
  master = $address;
  node.gvt = 0;
  msgr.gvt = gvt;
  while (msgr.gvt < sim_time) {
    hop(node = *);
    node.gvt = msgr.gvt;
    msgr.out_msgr = node.out_msgr;
    msgr.in_msgr = node.in_msgr;
    msgr.lvt = node.min_future_lvt;
    hop(node = master);
    node.out_msgr += msgr.out_msgr;
    node.in_msgr += msgr.in_msgr;
    if (node.min_future_lvt > msgr.lvt)
      node.min_future_lvt = msgr.lvt;
    if (--total_nodes > 0)
      break;
    if (node.in_msgr == node.out_msgr)
      node.gvt = node.min_future_lvt;
    node.in_msgr = 0;
    node.out_msgr = 0;
} }
```

Figure 5.12. GVT maintenance at user level

```
/* an original spin wait on gvt */
        while (gvt == i + 1) {
          func(name = "M_stall");
          if (resident_id != my_id)
            break; }

/* the corresponding virtual time function */
        func(name = "M_sched_node"; in = resident_id, "!=", my_id, 1);
```

Figure 5.13. Simplification from spin wait to virtual time scheduling

```
(1) matrix_mult(m) {
(2)   for (k = 0; k < m; k++) {
(3)      hop(physical = *)           /* go to all slave nodes */
(4)      if (node.j = node.i + k) {
(5)         func(name="copy"; in=node.A; out=msgr.A); /* multicast A to all */
(6)         hop(node = ,$address; link = "row",);     /* nodes in the row   */
(7)      }
(8)      func(name = "multiply");    /* Cij = Aij * Bij */
(9)      func(name = "M_sched_time_dlt"; in = 1);   /* synchronization */
(10)     func(name = "copy"; in = node.B; out = msgr.B);
(11)     hop(link = -"column");      /* rotate B to column i - 1 */
(12)     func(name = "copy"; in = msgr.B; out = node.B);
(13)     func(name = "M_sched_time_dlt"; in = 1);   /* synchronization */
(14) } }
```

Figure 5.14. MESSENGERS script of matrix multiplication using virtual time

5.3.4 Matrix Multiplication

As the last example of virtual time computing, we revisit the matrix multiplication discussed in Section 3.2.5. The main disadvantage of the Messengers script (Figure 3.13) was that the programmer needs to explicitly implement barrier synchronizations (lines 12-14 and 20-22). These synchronizations can be replaced with virtual time scheduling functions to yield much more elegant solutions. Figure 5.14 shows the updated version of the Messenger script. Lines 9 and 13 realize barrier synchronizations by delaying the subsequent computations (lines 10 and 14, respectively) to the next virtual time interval $delta = 1$. In other words, the algorithm alternates between the rotations of the B matrix and the multicasting of the A values by scheduling each at a new point in virtual time.

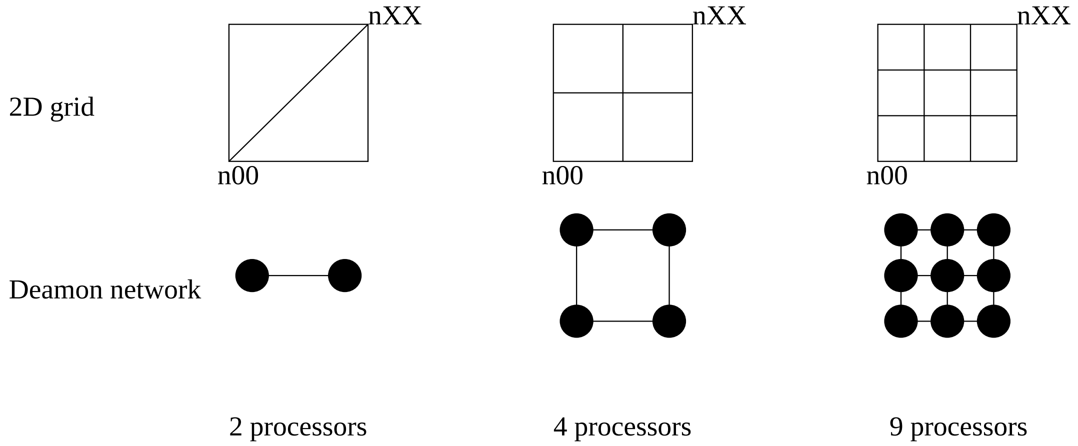


Figure 5.15. Logical-to-daemon network mapping for 2D grid

5.4 Quantitative Evaluation

This section compares the performance of MESSENGERS synchronization scheme with that of the other existing schemes. We use the three applications discussed in the qualitative evaluation: a naive shortest path algorithm, Dijkstra's algorithm, and a particle random walk, all of which use the same two-dimensional grid. Figure 5.15 shows how to construct a daemon network and map this grid on the daemon network when 2, 4, and 9 processors are used. We investigate four performance criteria for each application: (A) computation granularity necessary to achieve speed-up over the sequential version, (B) the change of performance when shifting the grid space from 10×10 to 20×20 and 30×30, (C) the best synchronization scheme, and (D) the speed-up when switching the network speed from 10Mbps to 100Mbps.

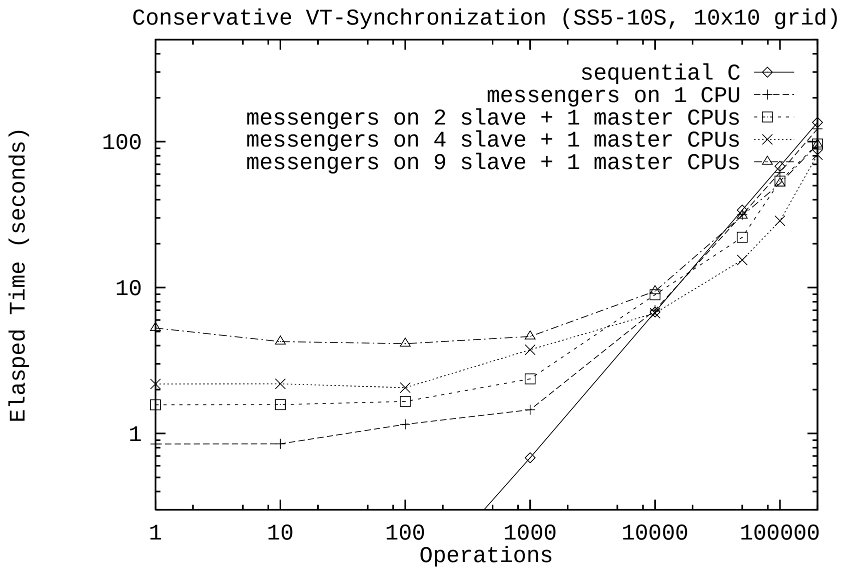


Figure 5.16. Shortest path as computation granularity increases

5.4.1 Naive Shortest Path Program

We first investigated the performance of the naive shortest path program as an example containing distributed termination.

(A) Computation Granularity

Since it just uses the virtual time to detect the completion of all propagated *find_path* Messengers, there is no advantage to using the optimistic schemes. Therefore, we focus on the computation granularity necessary for the conservative synchronization to achieve speed-up. Figure 5.16 shows that the threshold on four processors is 10,000 floating-point operations.

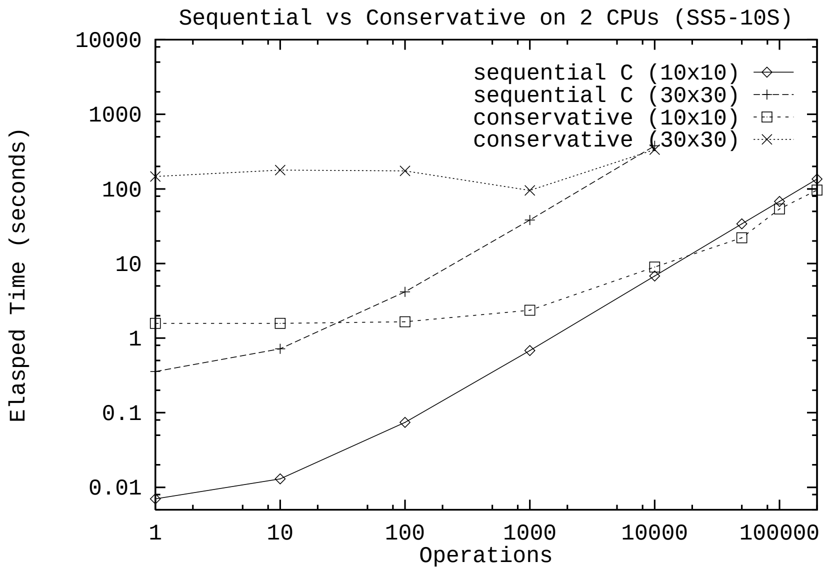


Figure 5.17. Performance of shortest path program on 10×10 and 30×30 grid spaces

(B) Performance on 10×10 and 30×30 Grid Spaces

Figure 5.17 shows the performance of the conservative synchronization when it takes place on 10×10 and 30×30 grid spaces mapped on two CPUs. The threshold to break even with the sequential version is lowered from 20,000 to 10,000 floating-point operations, i.e., to approximately half, due to the nine-fold larger gird, which inherently involves a larger number of Messengers.

(C) Best Synchronization Scheme

Figure 5.18 compares four versions of the program shown in Figure 5.8: one with the user-level synchronization Messenger shown in Figure 5.12, (i.e., the non-virtual-time version), one with conservative synchronization, one with fully optimistic synchronization, and one with the MESSENGERS optimistic synchronization.

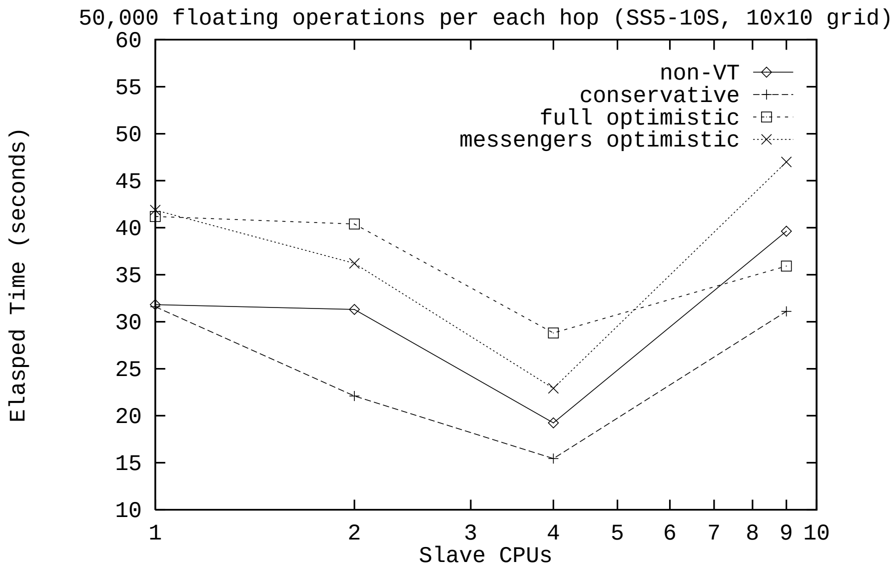


Figure 5.18. Comparison among various synchronizations for shortest path

Conservative synchronization shows the best performance and executes 1.4 times faster than the non-virtual-time version.

(D) Performance on 10Mbps and 100Mbps Network

Table 5.1 compares which scheme takes advantage of the faster network. Only the non-virtual-time implementation achieves a marginal speed-up. All others experience a slow-down. In the naive algorithm, Messengers move over the grid in the same virtual time interval non-deterministically unlike Dijkstra's algorithm. The frequency of communication becomes higher using the 100Mbps network and thus there are more chances to induce the congestion of Messengers at some nodes where the interpreter daemon frequently switches between its internal threads to handle the socket read/write operations.

	10x10 grid (100,000 operations, 2 CPUs)		
Synchronization schemes	10Mbps (seconds)	100Mbps (seconds)	Speed up (magnitude)
sequential C	67.797	67.797	not applicable
non-VT	50.596	49.859	1.015
conservative VT	53.697	57.890	0.928
full optimistic	61.370	128.404	0.478
MESSENGERS scheme	58.465	122.953	0.476

Table 5.1. Performance of shortest path program on 10Mbps and 100Mbps networks

(E) Summary

The results indicate that the virtual time synchronizations achieve speed-up over the versions that do not use such system-level synchronizations, however, only the conservative scheme is effective for the simple distributed termination problem. As expected, performance depends on the problem size rather than network speed. The more computation granularity per hop and the larger the network space, the higher is the observed speed-up. However, network speed does not improve performance.

5.4.2 Dijkstra's Algorithm

(A) Computation Granularity

Figure 5.19 shows the performance of the MESSENGERS and the sequential versions implementing Dijkstra's algorithm as computation granularity increases up to 200,000 floating-point operations. The threshold to benefit from parallelism is 100,000 operations. It is one order of magnitude larger than that of the naive shortest path algorithm. In the sequential version, Dijkstra's algorithm runs 5.8 times faster than the naive program, whereas, in the MESSENGERS version,

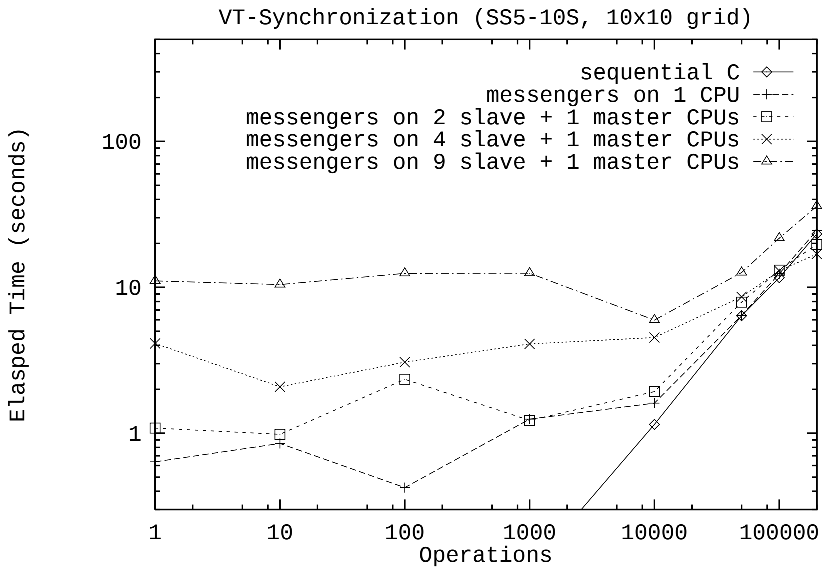


Figure 5.19. Dijkstra's algorithm as computation granularity increases

Dijkstra's algorithm runs only 1.7 times faster at 10,000 floating-point operations and 3.7 times faster at 100,000 floating-point operations.

(B) Performance on 10×10 and 30×30 Grid Spaces

Figure 5.20 compares the performance of the conservative synchronization scheme against the sequential execution when it takes place on 10×10 and 30×30 grid spaces mapped on two CPUs. Similar to the naive shortest path algorithm, the threshold to surpass the sequential version is reduced from 100,000 to 50,000, i.e., by half when the grid size is enlarged nine times.

(C) Best Synchronization Scheme

Figure 5.21 compares four versions of the program shown in Figure 5.10: one with the user-level synchronization Messenger shown in Figure 5.12, (i.e., the non-virtual-time version), one with the conservative synchronization, one with the fully

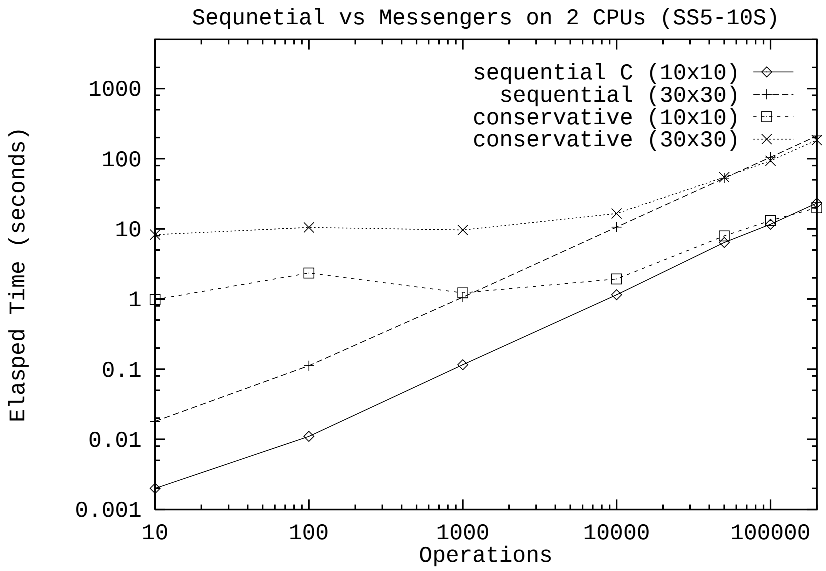


Figure 5.20. Performance of Dijkstra's algorithm on 10×10 and 30×30 grid spaces

optimistic synchronization, and one with the MESSENGERS optimistic synchronization. The best synchronizations scheme depends on the number of processors used for this algorithm. With two, four, and nine processors, the best schemes are MESSENGERS, conservative, and fully optimistic schemes, respectively. The MESSENGERS scheme shows the best performance when the daemon network is simple, but its performance declines when the network is more complicated.

(D) Performance on 10Mbps and 100Mbps Network

Table 5.2 examines which scheme takes advantage of the faster network. The non-virtual-time version achieves the highest speed-up on both grid sizes. Since this version implements *GVT* using the explicit *gvt* Messenger, its interpretation overhead prevents network congestion. Similarly, the optimistic synchronizations avoid the congestion slightly better due to their state-saving overhead. On the

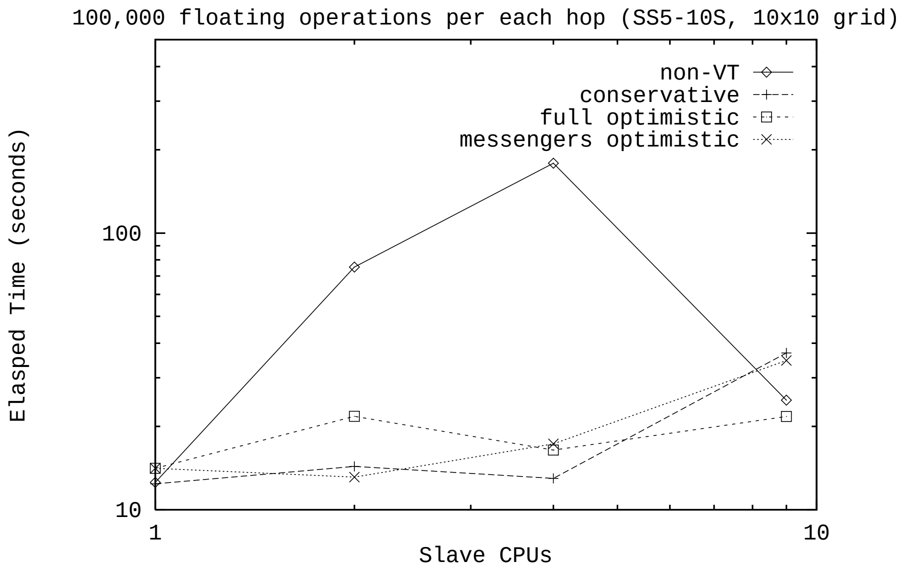


Figure 5.21. Comparison among various synchronizations for Dijkstra's algorithm

other hand, the conservative scheme has no overhead and thus is the most likely to cause a congestion.

In terms of absolute performance, in most cases the MESSENGERS scheme is the best with the smaller grid. With the larger grid the optimistic scheme is marginally better. Given the slower network (10Mbps) and the larger grid, the conservative scheme outperforms the MESSENGERS scheme.

(E) Summary

The evaluations show that in most cases the virtual-time synchronizations achieve speed-up over the non-virtual-time versions. The MESSENGERS scheme is superior on two processors, however it is not effective for more complicated daemon networks. Similar to the naive shortest path algorithm, performance improves with a larger computation network. The faster network significantly improves

	10x10 grid (100,000 operations, 2 CPUs)		
Synchronization schemes	10Mbps (seconds)	100Mbps (seconds)	Speed up (magnitude)
sequential C	11.634	11.634	not applicable
non-VT	75.405	14.610	5.161
conservative VT	14.345	25.934	0.553
full optimistic	21.776	20.934	1.040
MESSENGERS scheme	13.120	15.659	0.838

	30x30 grid (100,000 operations, 2 CPUs)		
Synchronization schemes	10Mbps (seconds)	100Mbps (seconds)	Speed up (magnitude)
sequential C	105.601	105.601	not applicable
non-VT	280.497	132.226	2.121
conservative VT	93.694	139.927	0.670
full optimistic	122.555	110.007	1.114
MESSENGERS scheme	128.849	126.483	1.019

Table 5.2. Performance of Dijkstra's algorithm on 10Mbps and 100Mbps networks

only the non-virtual-time version whose interpretation overhead avoids network congestion. The performance difference is very marginal for the optimistic and the MESSENGERS schemes and actually decreases markedly for the conservative approach.

5.4.3 Individual-Based Simulations

(A) Computation Granularity

Non-virtual-time and conservative virtual-time versions of the particle random walk on a 20×20 grid have been compared for two cases: (1) no local events are scheduled between any hops of a particle, and (2) two local events are scheduled every time a particle hops. Due to their state-saving overhead, the optimistic schemes cannot demonstrate better performance without any local events or with smaller computation granularity. The comparison between the optimistic

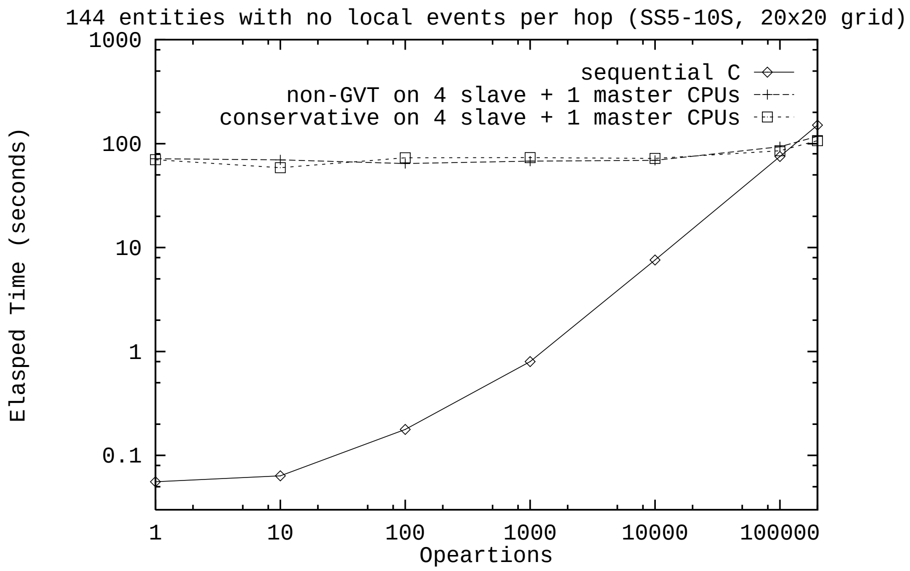


Figure 5.22. Particle random walk with no local events as computation granularity increases (200-virtual-time simulation)

and conservative schemes are shown later in the discussion of Best Synchronization Scheme. Figures 5.22 and 5.23 show the results of these two cases respectively. In case (1), the conservative virtual-time version achieves little improvement over the non-virtual-time version and requires more than 100,000 floating-point operations in order to surpass the sequential version. This is because there are some particles hopping between daemons during every virtual time increment. Such hop statements are the biggest source of overhead as discussed in Section 4.2.6. In case (2), (i.e., two local events per hop), the conservative version achieves between 1.3 and 1.4 speed-up over the non-virtual-time version. The virtual-time scheme requires 50,000 floating-point operations to surpass sequential execution.

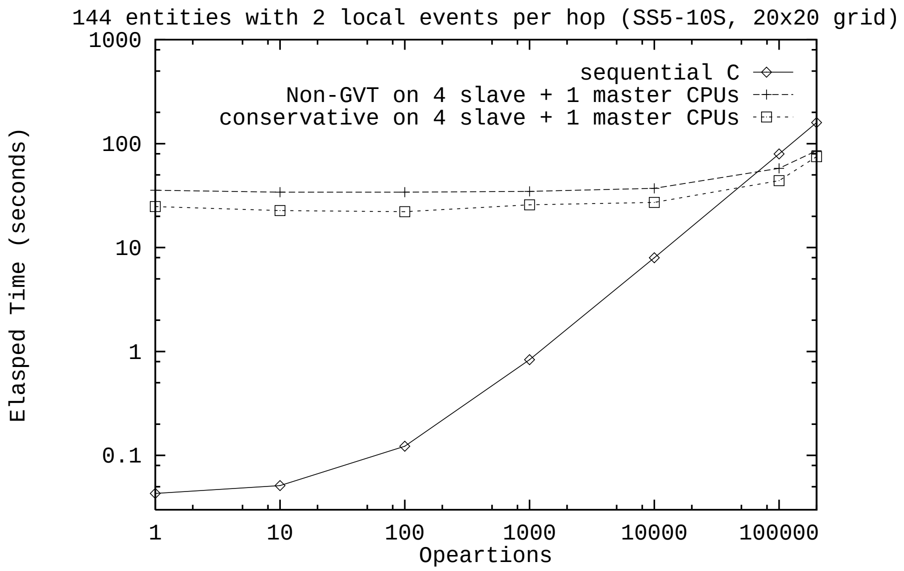


Figure 5.23. Particle random walk with 2 local events as computation granularity increases (200-virtual-time simulation)

(B) Performance on 10×10 and 20×20 Grid Spaces

Figure 5.24 shows the performance of the conservative synchronization scheme when it takes place on 10×10 and 20×20 grid spaces mapped to four CPUs. The quadrupled grid significantly drops the threshold required to surpass the sequential version, i.e., from 200,000 to 40,000 floating-point operations.

(C) Best Synchronization Scheme

Figure 5.25 and Figure 5.26 show the results of various synchronization schemes using four CPUs for two cases: (1) random walk of 36 particles on a 10×10 grid, and (2) random walk of 144 particles on a 20×20 grid. In the former case, all virtual-time synchronization schemes are inferior to the sequential program until each particle schedules eight or more local events per hop. At

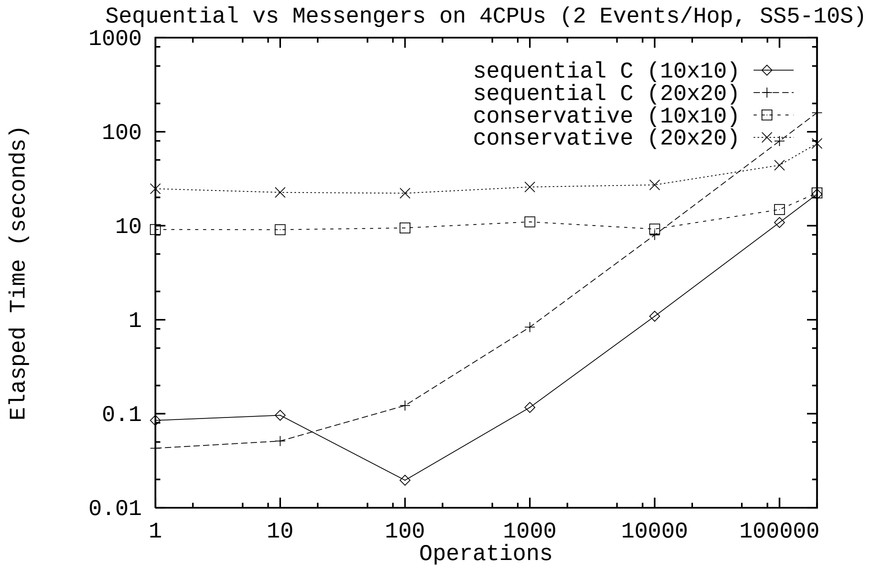


Figure 5.24. Performance on 10×10 and 20×20 grid spaces (200-virtual-time simulation)

that point, the best scheme is the fully optimistic synchronization, followed by the MESSENGERS optimistic synchronization. Prior to that point, the simulation space and the number of simulation entities are too small. However, even in such a case, MESSENGERS optimistic scheme is better than the other optimistic schemes with up to four local events per hop. In the second experiment (Figure 5.26), almost all the synchronization schemes achieve speed-up over the sequential program with only one or more local events per hop. Similar to the former case, MESSENGERS optimistic scheme is better than the other schemes with up to two events per hop, however it slows down with four or more events.

(D) Performance on 10Mbps and 100Mbps Network

Table 5.3 shows that none of the schemes achieves speed-up on either grid size. This result is in contrast with Dijkstra's algorithm where at least the non-virtual-time version achieves a speed-up. The main difference between the particles

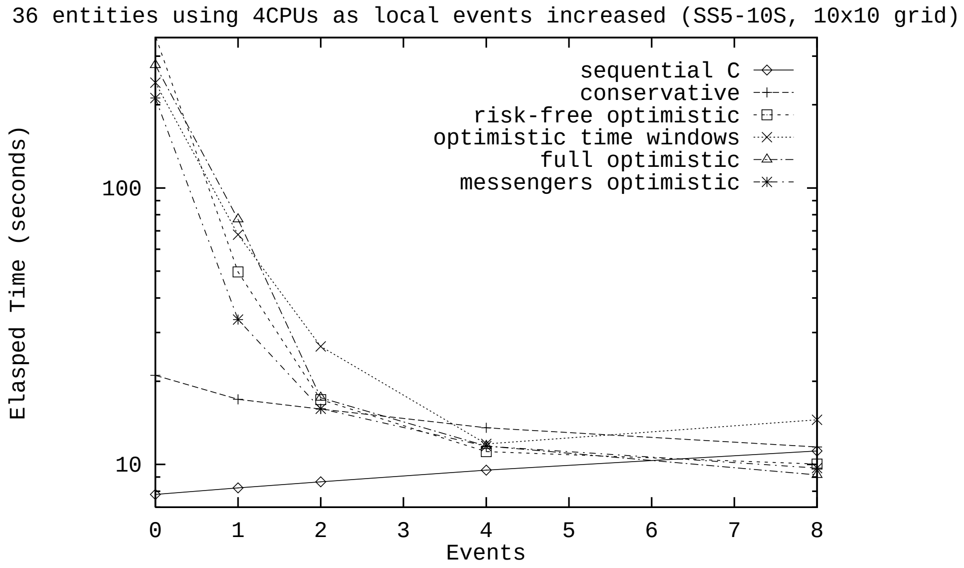


Figure 5.25. Comparison among various synchronizations for 36 entities random walk (200-virtual-time simulation)

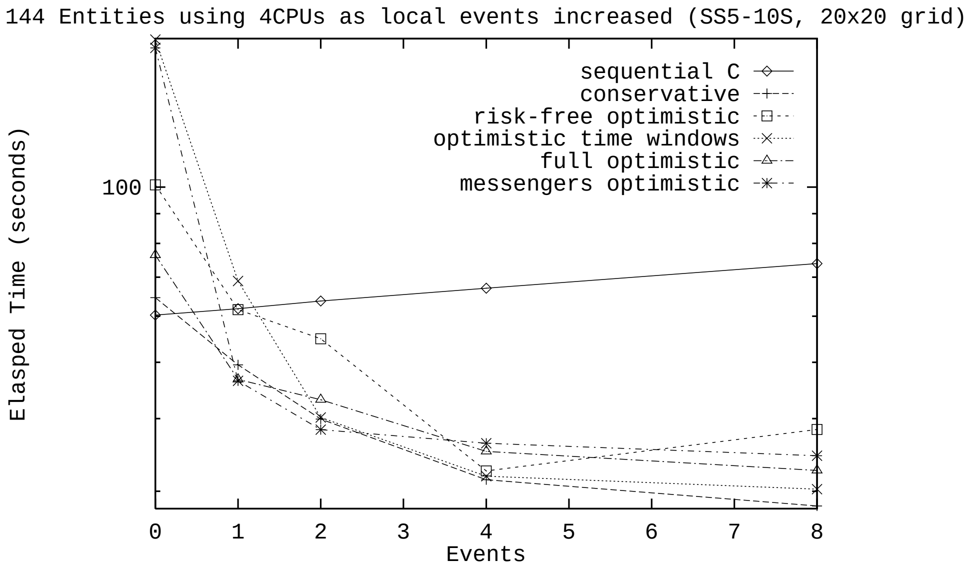


Figure 5.26. Comparison among various synchronizations for 144 entities random walk (200-virtual-time simulation)

	10x10 grid (100,000 operations, 2 CPUs)		
Synchronization schemes	10Mbps (seconds)	100Mbps (seconds)	Speed up (magnitude)
Sequential C	8.650	8.650	not applicable
non-VT	19.598	29.849	0.657
conservative VT	11.075	21.397	0.518
full optimistic	13.010	21.198	0.614
MESSENGERS scheme	12.238	20.797	0.588

	20x20 grid (100,000 operations, 2 CPUs)		
Synchronization schemes	10Mbps (seconds)	100Mbps (seconds)	Speed up (magnitude)
Sequential C	63.721	63.721	not applicable
non-VT	63.985	82.139	0.779
conservative VT	45.146	54.289	0.832
full optimistic	58.026	109.537	0.530
MESSENGERS scheme	66.940	145.972	0.459

Table 5.3. Performance of particles random walk on 10Mbps and 100Mbps networks (200-virtual-time simulation)

random walk and Dijkstra's algorithm is that Messengers representing particles continuously move randomly over the 2D grid, while all the Messengers except one in Dijkstra's algorithm terminate themselves on their way to the destination node. The higher frequency of communication by the 100Mbps network and the continuous random walk by Messengers are likely to induce Messengers congestion.

(E) MESSENGERS Scheme vs Optimistic Time Window

In addition to the above four experiments, this test demonstrates the performance of the MESSENGERS synchronization versus the optimistic time window schemes. Figure 5.27 shows how the performance of the optimistic time window scheme changes when its window size varies from 0 to 250 virtual time units. If each particle schedules its behavior every 10 units and inserts two local events per hop, it hops every 30 units. Therefore, the optimal window size is 30 units. Figure 5.27 empirically illustrates the optimal size. The optimistic time window

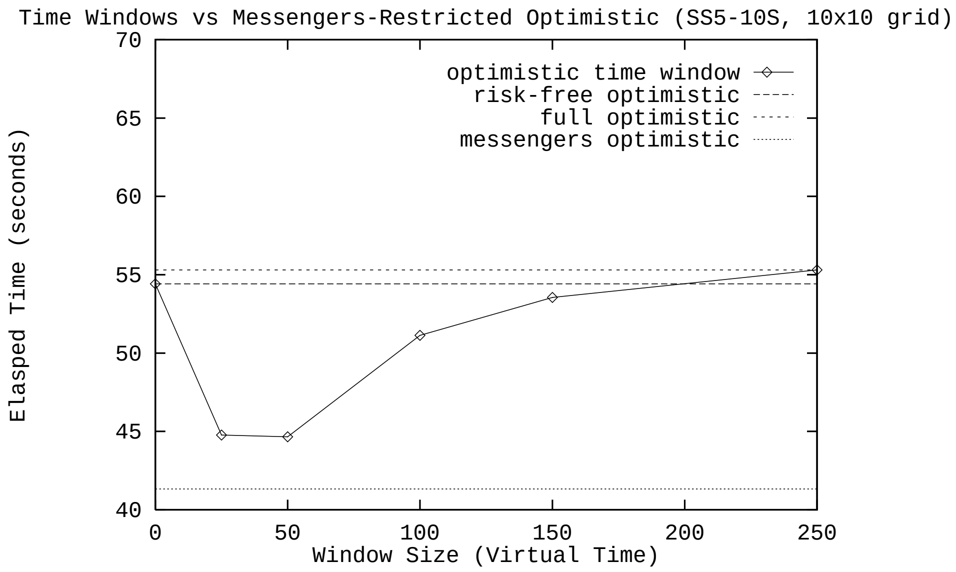


Figure 5.27. Messengers vs time windows on 4 CPUs (200,000 operations/hop, 250-virtual-time simulation)

scheme slows down as its size becomes smaller or larger than approximately 30 units.

(F) Summary

The results show that local events between hops play an important role for virtual time synchronizations to achieve speed-up. Also, the virtual time synchronizations show better performance on a large simulation space and with a large number of simulation entities. Eight or more events are necessary for a 10×10 grid, while only one event is sufficient for a 20×20 grid to obtain speed-up. MESSENGERS optimistic scheme shows better performance than the other optimistic schemes with only a small number of local events, (i.e., one or two event), however it slows down as more local events are inserted between hops. Higher network speed does not improve the performance of this application. The results also demonstrate that the performance of the optimistic time window scheme depends on the window size.

5.5 Discussion

Barrier synchronizations and distributed terminations are difficult to implement at the user level, since autonomous objects are dynamically spawned, terminated, or are moving over the network. This chapter shows that virtual time programming addresses such problems and simplifies Messengers scripts, some of which are even simpler than the corresponding sequential versions.

Virtual time synchronizations achieve a certain degree of speed-up over the non virtual-time versions, however they require one or more local events inserted between hop statements for the speed-up to be significant. The larger the size of the computational network, (i.e., the 2D grid in the above examples), the more speed-up virtual time computation achieves.

MESSENGERS synchronization scheme was proven to be better than the other optimistic schemes under some conditions: (1) a simple daemon network and (2) one or two local events per hop. As more daemon links are incident on each daemon node, this scheme becomes more restrictive and produces more LVT exchanges. However, if each Messenger schedules more local events, there is little difference between the MESSENGERS scheme and the other existing schemes.

MESSENGERS virtual time computing has an additional distinctive feature: simulation programmers do not have to use explicit wake-up calls if a sleeping Messengers is to wake up earlier than its scheduled time. Messengers schedule on both virtual time and arbitrary node variable conditions and thus can be awaken by a change of the node variable's content before the logical clock reaches the scheduleed time. In contrast, message-passing-based simulation systems do not generally allow simulation entities to suspend themselves on conditions associated with shared variables, so a user cancellation message must be sent to an entity when it needs to wake up before the logical clock reaches the scheduled time.

Chapter 6

Related Work

In this section, we compare MESSENGERS with three areas of related research: mobile agents (Section 6.1), coordination paradigms (Section 6.2), and object migration (Section 6.3). Although some aspects of their functionalities are similar to MESSENGERS, their main objectives are different. This section makes clear what is similar to and different from MESSENGERS.

6.1 Mobile Agents

"Intelligent" mobile agents are programs capable of physically moving through WANs (notably the Internet), and performing a variety of service tasks on behalf of their users [IEE96, MA97]. Notable systems include Telescript [Whi94], IBM Aglet [LC96], Tacoma [JvS95], Agent Tcl [Gra96], and University of Geneva's Messenger system [Tsc93] (referred to as UGM to distinguish it from MESSENGERS.)

6.1.1 Description of Existing Mobile Agents Systems

Telescript provides its own object-oriented language for expressing agents' behavior and facilitates electronic commerce with a complex security model. IBM Aglets are Java-based mobile agents whose navigational behavior is supported through special classes called the Java Aglet Application Programming Interface. Tacoma provides significant fault tolerance tools to Tcl-based agents. Agent Tcl is another mobile-agent system relying on Tcl, however unlike Tacoma, Agent Tcl provides navigational primitives that do not require the programmer to explicitly capture agents' state. UGM replaces traditional messages and communication protocols with mobile agents. Nodes in UGM establish communication in the following way: a node wishing to communicate with another first dispatches an agent to the target node. This carries the logic of an arbitrary communication protocol, which it establishes on the target node. This protocol is then used for subsequent message exchange between the two nodes and is discarded when it is no longer needed.

6.1.2 Comparison with MESSENGERS

Although the above systems have explored different aspects of mobile agents, their main objective is to facilitate the use of various distributed services available on WANs or managing the networks using the agents' navigational autonomy as a vehicle. In contrast, MESSENGERS uses the same basic principles of autonomous navigation to harness the power of LANs for general purpose distributed computations. This difference in objective clearly distinguishes MESSENGERS from mobile-agent systems in many aspects of data organization, navigational autonomy, and execution sequencing.

MESSENGERS provides a logical network, whose size and topology are completely independent from the underlying daemon or physical networks. This is used by the application to organize and manage its data, which is accessed using powerful navigational commands that include implicit replication based on a navigational calculus [FBD97]. Since Messengers resume their execution at the next statement following a navigational command upon each migration, programmers can code Messengers from the "driver's" point of view.

In contrast, mobile-agent systems do not have the concept of a logical network, although some of them allow a physical node to provide multiple working places for agents, (e.g., cabinets in Tacoma and contexts in IBM aglets.) Note that these places are regarded as a fully connected rather than structured in a logical network. Therefore, mobile agents need to specify explicit node destinations (and their internal working places) in their navigational commands. In many cases, they also require programmers to capture agents' status and even provide subsequent code, (e.g., *folder create*, *store*, and *meet* in Tacoma, and *submit* in UGM.)

6.2 Coordination Paradigms

This area of research is concerned with providing the synchronization, communication, and creation/destruction of computational activities required to orchestrate the individual computations into a coherent system [AHM96, CE96].

Examples of coordination paradigms include Gamma [BL93], IWIM [Arb96], Linda [CG89], and PoliS [Cia94] as a variant of Linda.

6.2.1 Description of Existing Coordination Systems

Gamma is based on a chemical reaction metaphor. Its programs consist of matched (*reaction-conditions*, *action*) pairs. Execution proceeds by replacing a collection of objects that satisfy a reaction condition by the result of applying the corresponding action. As programs are executed, they may cause multi-set transformations that create the reaction conditions necessary to allow other programs to execute. IWIM consists of a hierarchical master-workers structure model. At each layer, a master, (which can be a worker at a higher layer,) treats its own workers as black boxes and handles only the coordination among these workers. On the other hand, a worker only focuses on its computation by reading from/writing to its communication ports, but has no knowledge about how these ports are connected to the other workers. Linda allows processes to operate on a shared space of tuples addressed by their contents, using four operators: *out* to add a tuple, *read* to read a tuple, *in* to read and delete a tuple, and *eval* to insert and evaluate a tuple. PoliS is a variant of Linda and supports explicit partitioning of the tuple space. It allows multiple named tuple spaces, called *places*, such that each tuple belongs to exactly one tuple space. The collection of all tuple spaces is called the *polispace*. The execution threads in PoliS are autonomous active tuples, called *agents*. Because an agent is a tuple, an agent belongs to exactly one tuple space. An agent can read tuples within its own tuple space and can write tuples to any tuple space.

6.2.2 Comparison with MESSENGERS

Both MESSENGERS and coordination paradigms are capable of migration and dynamic decomposition of computational activities. However the main difference lies in how to provide these capabilities. The coordination paradigms specialize in the interaction among computation entities, each of which focuses on whom to interact with and what to do. Thus, object migration and decompositions are provided with combinations of interaction-oriented primitives rather than simple navigational primitives. On the other hand, the MESSENGERS point of view is strongly navigation-oriented. Each object, namely Messenger, focuses on where to go and what to do. Therefore object migration and decompositions are simply achieved by navigation and task-coordinating statements.

In MESSENGERS, the state space is a logical network, and object migrations or duplications can be implicitly achieved along logical links. In addition, programmers do not have to capture objects' migration or duplication status. Dynamic decomposition is supported by allowing C functions to be stored at nodes and carried from node to node by Messengers. In contrast, coordination paradigms need to explicitly copy objects. For instance, Polis has an *agent* write a copy of itself into another tuple space and terminating itself in the current tuple space. There is no direct support for dynamic decomposition in these paradigms, but the effect could be achieved through combinations of other operations.

6.3 Process Migrations

Process migration has been investigated by a number of projects. This can be supported at the object level or by the underlying operating system, i.e., by supporting the migration of address spaces and process states between machines.

Emerald [RTL+91] provides a number of specific primitives to efficiently locate and move fine-grained objects (including processes). Similarly, Obliq [Car95] permits threads to transfer themselves between physical sites while retaining their original semantics.

These forms of migration differ from MESSENGERS in two significant ways. First, with a few notable exceptions, migration typically takes on a *passive* form: the migrating process has no autonomy. It is being moved by a command executed by some other object. A Messenger, in contrast, moves as the result of its *own* actions. Second, the purpose of migration in most systems is to improve performance. With MESSENGERS, navigating through the network is a basic concept of the programming paradigm and is an important and explicit part of any application, even when running on a single processor.

Chapter 7

Conclusion

We have discussed the software architecture and features of MESSENGERS necessary to support general-purpose distributed computing. We have demonstrated both MESSENGERS's programming superiority and performance competitiveness with conventional message-passing programming. In particular, our proposed scheme for virtual-time synchronization among autonomous objects has achieved speed up over other synchronization schemes under certain conditions.

MESSENGERS is public-domain software available from its home page: `http://www.ics.uci.edu/~bic/messengers`. The system has been used at UCI by ICS198/199 undergraduate students, ICS247 graduate students, and a toxicology research group in the College of Medicine. It has been also used by four universities: Karlsruhe (Germany), British Columbia (Canada), Purdue (USA), and Kanazawa (Japan). Other universities and agencies have made inquires via the Web. In order to further improve the capabilities of MESSENGERS in distributed computing, several important research items should be pursued:

1. **Compilation to full native code:**

   Once an application is completely developed and runs on homogeneous machines, it may no longer require the MESSENGERS interpreter's flexibility, which the user might like to trade for better performance. One solution is to compile Messenger scripts into native code and make the MESSENGERS daemon's navigation, decomposition, and temporal coordination features accessible from the native code in the form of ordinary system function calls.

2. **Navigational calculus:**

   How to express navigational autonomy is another research item related not only to MESSENGERS itself but to all mobile agents. It is important to consider various cases in which applications need navigational autonomy and to mathematically define the autonomy as a navigational calculus.

3. **Node (re-)mapping:**

   MESSENGERS currently provides only one mapping strategy using a creation threshold value and a daemon network topology as described in Section 2.4.3. The development of new node-mapping strategies is essential to map a more complex logical network over a physical network that considers the communication speed of each physical link and the load status of each physical node. Also, dynamic node remapping is necessary for load balancing.

4. **Debugging environment:**

   Currently, the only available MESSENGERS debugging tool is a tracing program displaying the execution sequence of the Messenger that a daemon is interpreting. Better debugging tools are necessary to keep track of Messengers' movements, display the logical network graphically, and show a consistent snapshot of system-wide execution status.

5. **Fault tolerance:**

   Fault tolerance is another problem that we will address in our future work. The interpreter daemon will be enhanced to detect whether any of its neighbors in the daemon network experience any faults. Once such faulty neighbors are detected, new Messengers can be dispatched or existing ones rerouted to avoid these faulty neighbors so that computations can continue.

In view of the many qualitative and quantitative benefits derived from using an autonomous-objects-based approach and the future opportunities listed above, this research has proved that MESSENGERS is a promising tool for general-purpose distributed computing.

Bibliography

[AHM96] J.-M. Andreoli, C. Hankin, and D. Le Métayer, editors. *Coordination Programming: Mechanisms, Models, and Semantics*. Imperial College Press, London, UK, 1996.

[And91] G.R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, 1991.

[Arb96] F. Arbab. The IWIM model for coordination of concurrent activities. In *Coordination Languages and Models*, pages 34–56, April 1996.

[BFD96] L.F. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, 29(8), Aug. 1996.

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

[BL87] L.F. Bic and C. Lee. A data-driven model for a subset of logic programming. *ACM TOPLAS*, 9(4), Oct. 1987.

[BL93] J-P. Banâtre and D. Le Métayer. Programming by multiset transformation. *CACM*, 36(1):98–11, January 1993.

[BM93] Özalp Babaoğlu and Keith Marzullo. Consistent global states of distriubted systems: Fundamental concepts and mechanisms. In Sape

Mullender, editor, *Distributed Systems*, pages 55 – 149. Addison-Wesley, second edition, 1993.

[Bor92] P. Borst. The first implementation of the WAVE systems for UNIX and TCP/IP computer networks. Technical Report 18/92, Univeristy of Karlsruhe, December 1992.

[BST89] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), 1989.

[Car95] L. Cardelli. Obliq: A language with distributed scope. *Computing Systems*, 8(1):27–59, Winter 1995.

[CDK94] Geourge Coulouris, Jean Dollimore, , and Tim Kindberg. *Distributed Systems, Concepts and Design*, chapter 15, Recovery and Fault Tolerance, pages 449 – 476. Addison-Wesley, second edition, 1994.

[CE96] Paolo Ciancarini and Chris Hankin (Eds.). *Coordination Languages and Models, Proc. First Int'l Conf, COORDINATION'96*. Springer Verlag, Apr. 1996.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Comm. ACM*, 32(4), Apr. 1989.

[Cia94] P. Ciancarini. Distributed programming with logic tuple spaces. *New Generation Computing*, 12(3):251–284, 1994.

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

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 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.

[DR90] D. Dickens and P. Reynolds. Srads with local rollback. In *Proceedings of SCS Multiconference on Distributed Simulation*, volume 22, pages 161–164, January 1990.

[FBD96] Munehiro Fukuda, Lubomir F. Bic, and Michael B. Dillencourt. MESSENGERS: A distributed computing environment for autonomous objects. Technical report, TR 96-20, Information and Computer Science, University of California, Irvine, 1996. `http://www.ics.uci.edu/~bic/messengers`.

[FBD97] Munehiro Fukuda, Lubomir F. Bic, and Michael B. Dillencourt. Distributed coordination with MESSENGERS. Technical report, TR 97-20, Information and Computer Science, University of California, Irvine, 1997.

[FBDM96] M. Fukuda, L.F. Bic, M. Dillencourt, and F. Merchant. Intra- and inter-object coordination with MESSENGERS. In *First Int'l Conf. on Coordination Models and Languages (COORDINATION'96)*, Apr. 1996.

[FMB+96] M. Fukuda, K.L. Morse, L.F. Bic, M. Dillencourt, E. Lee, and D. Menzel. A novel approach to toxicology simulation based on autonomous objects. In *SCS Western MultiConference, Simulation in the Medical Sciences*, Jan 1996.

[Fuj90a] R.M. Fujimoto. Optimistic approaches to parallel discrete event simulation. *Transactions of the Society for Computer Simulation*, Vol.7(No.2):153 – 191, 1990.

[Fuj90b] R.M. Fujimoto. Parallel discrete event simulation. *Comm. ACM*, Vol.33(No.10):30 – 53, October 1990.

[Fuk96a] M. Fukuda. MESSENGERS system library manual. Report MSGR-02, University of California, Irvine, 1996.

[Fuk96b] M. Fukuda. MESSENGERS user's manual. Report msgr-01, University of California, Irvine, 1996.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994.

[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.

[Gra96] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, Monterey, California, July 1996.
`http://www.cs.dartmouth.edu/~agent/papers/index.html`.

[HDP88] M. Huston, D. DeAngelis, and W. Post. New computer models unify ecological theory. *BioScience*, 38(10):682–691, November 1988.

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

[HW92] A. Huth and C. Wissel. The simulation of the movement of fish schools. *Journal of Theoretical Biology*, 156:365–385, 1992.

[IEE96] IEE. *IEE Colloquium on Intelligent Agents and Their Applications*, No.1996/101, London, UK, Apr. 1996.

[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.

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

[JvS95] D. Johansen, R. van Renesse, and F. B. Schneider. An introduction to the TACOMA distributed system version 1.0. Technical Report 05-23, Department of Computer Science, University of Tromsø, June 1995. `http://www.cs.uit.no/DOS/Tacoma/index.html`.

[KSS96] Steve Kleiman, Devang Shah, and Bart Samaalders. *Programming with THREADS*, chapter 13, Threads and Processes: Advanced Topics, pages 229 – 244. Prentice Hall, 1996.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, Vol.21(No.7):558 – 565, 1978.

[LC96] D.B. Lange and D.T. Chang. IBM Aglets workbench, programming mobile agents in Java. White paper, IBM Tokyo Research, Japan, September 1996.

[Li88] Kai Li. A shared virtual memory system for parallel computing. In *Proc. of the 1988 Int'l Conf. on Parallel Procesing*, 1988.

[MA97] *MA '97: First International Conference on Mobile Agents*, Springer-Verlag, Lecture Notes in Computer Science, Berlin, Germany, Apr. 1997.

[Mat93] Friedmann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, Vol.18(No.4):423 – 434, August 1993.

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

[Pan84] V. Pan. *How to multiply matrices faster.* Lecture Notes in Computer Science 179. Springer-Verlag, Berlin, 1984.

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

[PS88] H.-O. Peitgen and D. Saupe, editors. *The Science of Fractal Images.* Springer-Verlag, 1988.

[RA84] J. C. Ramsey and M. E. Andersen. A physiologically based description of the inhalation pharmacokinetics of styrene in rats and humans. *Toxicology and Applied Pharmacology*, 73:159–175, 1984.

[RD89] P.W. Richardson and T. Danielsen. Intelligent messages or when messages come alive. In K. Boyanov and R. Angelinov, editors, *Network Information Processing Systems*. North-Holland, 1989.

[Res94] M. Resnick. Changing the centralized mind. *Technology Review*, pages 33–40, July 1994.

[Rey87] C.W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, July 1987.

[RTL+91] R.K. Raj, E Tempero, H.M. Levy, A.P. Black, et al. Emerald - a general-purpose programming language. *Software-Practice & Experience*, 21(1), Jan. 1991.

[Sam85] B. Samadi. *Distributed Simulation, Algorithms and Performance Analysis*. PhD thesis, University of California, Los Angeles, 1985.

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

[Ste90] W. Richard Stevens. *UNIX Network Programming*, chapter 6, Berkeley Sockets, pages 258 – 341. Prentice Hall, 1990.

[Ste92] Jeff S. Steinman. Speedes: A multiple-synchronization environment for parallel discrete-event simulation. *International Journal in Computer Simulation*, 2:251–286, 1992.

[Ste93] Jeff S. Steinman. Breathing time warp. In *Proceedings of 7th Workshop on Parallel and Distributed Simulation – PADS'93*, pages 109–118, May 1993.

[Tsc93] C. F. Tschudin. *On the Structuring of Computer Communications*. PhD thesis, University of Geneva, Centre Universitaire d'Informatique,

Geneva, Switzerland, 1993.
`http://cuiwww.unige.ch/tios/msgr/home.html`.

[Vil92] F. Villa. New computer architectures as tools for ecological thought. *Trends in Ecology and Evolution (TREE)*, 7(6):179–183, June 1992.

[Wal82] D.W. Wall. Messages as active agents. In *Proc. 8th Symposium on the Principles of Programming Languages*, pages 34–39. ACM, 1982.

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