Messages versus Messengers in Distributed Programming*

Munehiro Fukuda
Lubomir F. Bic
Michael B. Dillencourt
Jason M. Cahill

Department of Information and Computer Science
University of California, Irvine

December 10, 1998

Abstract

Messengers are autonomous objects, each capable of navigating through the underlying network and performing various tasks at each node. Messengers applications are written using navigational *commands rather than the* send/receive *primitives of conventional message-passing approaches. In this paper we contrast the two programming styles. From a software engineering viewpoint, the navigational style generally results in a smaller semantic gap between abstract algorithm descriptions and their actual implementations, which makes programs easier to construct, understand, and maintain. In terms of performance,* MESSENGERS *programs are highly competitive with message-passing. We demonstrate these advantages using two concrete applications programmed using* MESSENGERS *and PVM.*

1 Introduction

The objective of this paper is to contrast two different styles of programming in distributed systems. The first, which is by far the most prevalent among all distributed programming languages, is based on *message-passing*. The application is viewed as a collection of concur-

*The authors gratefully acknowledge the support of National Science Foundation Grants CDA-9617349 and CCR-9703572, and US Department of Education Grant P200A70306.

rent tasks (processes, threads), which communicate with one another by exchanging passive data messages. This is accomplished using low-level send/receive primitives or some higher-level abstraction, like remote procedure calls or rendezvous. The second approach is based on the concept of *autonomous "self-migrating" messages*, that is, messages that have their own identity and behavior, which permit them to actively navigate through the underlying computational network, perform various tasks at the nodes they visit, and coordinate their activities with other such messages constituting the distributed computation. To distinguish such autonomous messages from simple passive data messages, we refer to them as *Messengers*.

Autonomous messages, also referred to as mobile agents or self-migrating threads, have been used for a variety of applications, as will be discussed in Section 4. In this paper we concentrate on *general-purpose* distributed computation, as developed in WAVE [SB94, SB96], BPEM [BL87], and most recently, our MESSENGERS[1] system [BFD96, FBDM98]. In these systems, autonomous messages serve as active mobile entities (or agents) that coordinate the computation in both time and space.

There are several fundamental characteristics that distinguish Messengers-based systems from message-passing systems:

- Messengers are capable of creating and using a *logical* network. Nodes may contain arbitrary variables or data structures, while links may be used by a Messenger for navigation, including replicating itself by following multiple links. The logical network thus represents a data structure *external* to and independent of any ongoing activity. To use a biological metaphor, the logical network is like the *exogenous* skeleton of crustaceans and other low-level organisms, which defines the organism's external form. In contrast, the structures that bind together conventional message-passing tasks are *internal* to the tasks. Each task will typically contain information about other tasks with which it may communicate. This information may be passed to the task at creation or determined dynamically. In either case, it is similar to an *endogenous* skeleton of higher-level organisms. This dichotomy has important implications for the

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

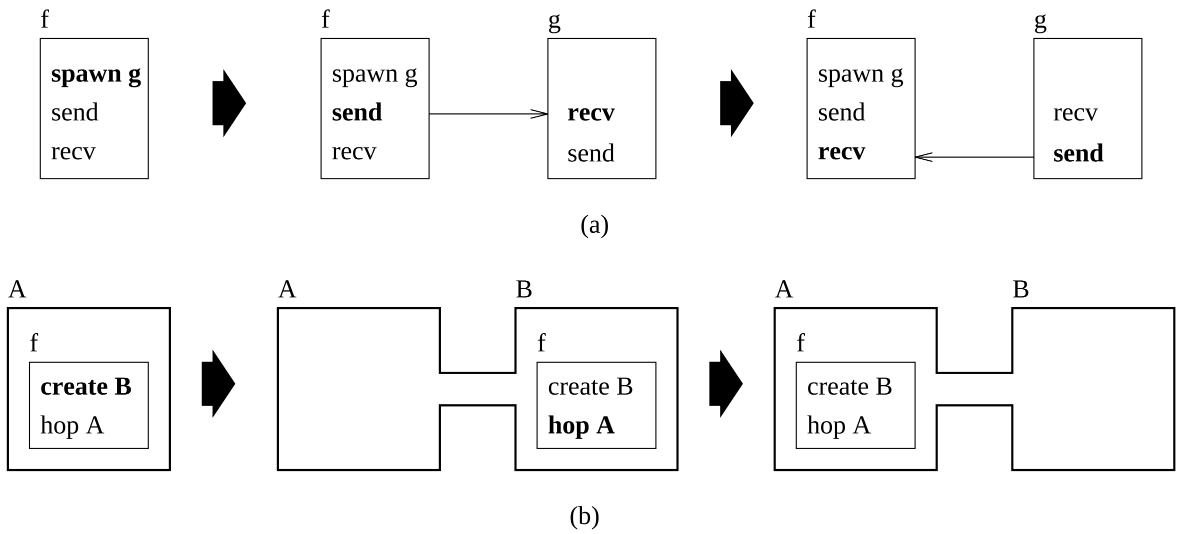


Figure 1: (a) Task creation and message-passing. (b) Logical node creation and navigation

spawning of new activities. Under message-passing, the focus is on *task* creation—a new task is spawned by an existing task and the parent-child relationship is made known to the respective tasks. Under Messengers, the focus is on *node/link* creation. The nodes and links form the places where activities can occur.

- The MESSENGERS logical network is *persistent*. Unless explicitly destroyed, it will continue to exist after the Messengers have moved to other nodes or have terminated. The structures used to connect conventional message-passing tasks are typically *ephemeral*, lasting only as long as the tasks that are using them to communicate with each other. We will see one example of the usefulness of the persistent nature of the MESSENGERS logical network in Section 3.1 of this paper; we also note that it is very useful for systems in which many different processes use the same communication structure (e.g., individual-based systems, distributed interactive simulations [DIS94].)

- All functionality of the application is embedded in the individual Messengers, i.e., the programs carried by Messengers as they navigate through space. This is in stark contrast to message-passing, where the communicating tasks contain all the functionality, while messages are only passive carriers of data. Hence *send/receive* operations are replaced with *navigational* operations that move a Messenger between nodes.

Figure 1 illustrates the above philosophical differences graphically. Part (a) shows a task f, which first spawns a child task g, as indicated by the bold font. The two concurrent tasks then exchange messages with one another using send/receive operations. Part (b) shows a logical node A, currently containing a Messenger f. This Messenger creates another logical node, B, connected to A by a logical link. As part of the create operation, the Messenger automatically moves to the new node B. It may continue executing there or, as shown in the figure, it may move back to A using a navigational operation (hop). It may also create other Messengers, which then share the spaces and corridors created by earlier Messengers and which they can extend or modify as necessary to perform their tasks. Alternately, arbitrary new Messengers may also be injected by the user from the outside (the command shell) at runtime.

We note the following fundamental differences between the two programming styles for the example of Figure 1. (1) *Single program:* Under message-passing, two distinct program components (e.g. functions), f and g, must be developed while only one is necessary under MESSENGERS. (2) *Single thread:* Under message-passing, there are two concurrent activities to deal with at runtime. The execution of these two activities and the exchange of activities between them must be carefully synchronized to ensure a proper alignment of the corresponding send/receive pairs. Under MESSENGERS, the only synchronization necessary is the arrival of the Messenger and the data it contains at node B. (3) *Persistent data structures:* Under message-passing, there is no data structure that could exist without (and outside of) a process or a thread. Under MESSENGERS, on the other hand, the logical network (nodes A and B, and their connecting link) continues to persist even when the Messenger f terminates or moves to some other logical node. The logical network is available for use by other Messengers that may be created as progenies of f or injected independently by either the user or another Messenger.

The remainder of the paper is organized as follows. Following an overview of our system in Section 2, we illustrate the new programming style imposed by Messengers by presenting two concrete application examples (Section 3). These demonstrate that this new paradigm offers elegant solutions to problems that would be much more difficult to achieve using more traditional approaches, while maintaining performance that is highly competitive with message-passing.

2 MESSENGERS

2.1 Principles of Operation

MESSENGERS [BFD96, FBDM98] is a system that supports the development and use of distributed applications structured as collections of autonomous self-migrating computations, called Messengers. To allow Messengers to navigate autonomously through the network and carry out their tasks, the MESSENGERS system is implemented as a collection of daemons instantiated on all physical nodes participating in the distributed computation. A daemon's task is to continuously receive Messengers arriving from other daemons, interpret their behaviors, described as programs carried as part of each Messenger, and send them on to their next destinations as dictated by their behaviors.

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

Messenger programs, referred to as *Messenger scripts*, are written in a subset of C and are compiled into a form of byte code for more efficient transport and parsing [Bid96]. Each script is carried in its entirety by the Messenger as it propagates through the network and is replicated each time the Messenger needs to follow more than one logical link. This requires a programming style different from the commonly used approaches to distributed computing. In particular, the user does not supply any *node programs* to execute on the various network nodes. Rather, all node programs and their communications are part of the underlying infrastructure of daemons. The application itself consists of only the logical network and the corresponding Messengers navigating this network. The programmer's point of view is that of a *navigator*, sitting in the "driver's seat" of a Messenger, and guiding it on its way through the computation. We also refer to this programming style as *data-centric*, since each Messenger is typically responsible for a piece of data or a data structure, which it carries around the network for its own use or to deliver it to a specific node for other Messengers to work on.

A Messenger script is a sequence of statements, which can be of one of the following types: (1) Computational statements enable the Messenger to perform arbitrary computations. They include all standard C assignment and control statements, involving arbitrary variables and constants; (2) Navigational statements endow the Messenger with mobility, permitting it to create and destroy logical nodes and/or links, and to move within the logical network; (3) Function invocation statements provide an interface to the system's environment. They permit the dynamic loading and invocation of precompiled C functions to be executed in native mode.

Since the primary goal of the MESSENGERS system is to allow mutually trusted autonomous computations (Messengers) to collaborate on the solution of a common task, the MESSENGERS system implements a modified non-preemptive scheduling policy. A daemon will interrupt a Messenger only when it issues a navigational command or at the moment when it begins to execute a native-mode function. In particular, the daemon will not interrupt a Messenger in the middle of a native-mode function, or when it is executing any other computational command. Hence a critical section in a Messenger script can be written as a sequence of computational commands, using no additional constructs, provided no navigational commands or invocations of native-mode functions are embedded in the critical section. Alternatively, it can be written as a native-mode function. As will be seen in Section 3, this considerably simplifies the task of developing a MESSENGERS application.

There are three types of variables accessible by the Messengers statements. *Messenger variables* are private to and carried by each Messenger as it propagates through the logical computational network. *Node variables* are resident in nodes of the logical network and shared by all Messengers currently visiting the same logical node. *Network variables* are predefined at each logical node and give each Messenger access to the network information local to the current node. They are prefixed by a "$" sign to distinguish them from the freely definable messengers and node variables. In particular, *$address* contains the address of the current host and *$last* contains the name of the last traversed link. This permits a Messenger to determine along which link it entered the current node.

The distinction between Messenger variables and node variables provides a clear separation between the variables that travel with a Messenger and comprise its state, and the variables that it shares with other Messengers resident at the same node. This has a software engineering advantage, because it distinguishes variables used for computation (by a single Messenger) from variables used for communication and coordination (between Messengers). It also has a performance advantage. When a Messenger hops from one node to another, it always takes the same data items (namely its Messenger variables) with it, along with the code. Consequently, there is no need for copying of data into/out of buffers by the source/receiving daemon. With message passing, by contrast, data must be copied into the message buffer by the sending program and copied out of the message buffer by the receiving program. This extra copying can result in performance degradation in message-passing systems.

Given the focus of this paper, we will concentrate on only the navigational statements. The remainder of this section will introduce the basic syntax and semantics of the statements *hop*, *create*, and *delete*. These are based on a navigational calculus described in [FBDM98].

The *hop* Statement. The *hop* statement permits a Messenger to move around the logical network. Its syntax is as follows:[2]

$$hop(ln = n; ll = l; ldir = d)$$

where ln stands for "logical node", ll stands for "logical link", and $ldir$ stands for the link's direction. Together, the triple (n, l, d) is a destination specification in the logical network where n can be an address, a variable, a constant (including the special node *init*), or a wild card ($*$) that matches any name; l can be a variable, a constant, a wild card, or a "virtual link" (corresponding to a direct jump to the designated node); finally, d can be one of the

[2]The syntax shown here is slightly simplified. In its full generality, a single *hop* statement supports multiple hop specifications, similar to the *create* statement discussed below.

symbols $+, -$, or $*$, denoting "forward," backward," or "either," respectively. The default for all three parameters is $*$ and thus may be omitted.

The semantics of the *hop* statement are as follows: From the current node c, replicate the Messenger to all nodes that match n and are connected to c by links matching l and d. The Messenger executing the *hop* in node c then ceases to exist.

Examples (showing the complete syntax and the equivalent default forms):

- $hop(ln = *; ll = x; ldir = *) \equiv hop(ll = x)$

  from the current node c replicate the Messenger to all nodes connected to c by link x (regardless of link direction)

- $hop(ln = *; ll = x; ldir = -) \equiv hop(ll = x; ldir = -)$

  from the current node c replicate the Messenger to all nodes connected to c by a backward-oriented link x

- $hop(ln = *; ll = *; ldir = *) \equiv hop()$

  from the current node c replicate the Messenger to all nodes connected to c, i.e., all neighboring nodes

The *create* Statement. The *create* statement permits a Messenger to create new logical nodes and/or links. Its syntax is as follows:

$$create(ln = n_1, ..., n_k; ll = l_1, ..., l_k; ldir = d_1, ..., d_k;$$
$$dn = N_1, ..., N_k; dl = L_1, ..., L_k; ddir = D_1, ..., D_k; [ALL])$$

where each triple (n_i, l_i, d_i) specifies a *new* logical node, n_i, connected to the current node by a (possibly directed) link, l_i. The node n_i is created on the daemon node specified by the triple (N_i, L_i, D_i), which is a destination specification in the daemon network.

The semantics of *create* are then as follows: For each destination specification (N_i, L_i, D_i) determine the set of daemon nodes that match the given daemon name (N_i), daemon link (L_i), and the daemon link direction (D_i). When the optional parameter ALL is omitted, choose one of the possible daemons and create the logical node n_i on it. (The choice is made by a set of rules that are beyond the scope of this paper [FBDM98].) When the optional parameter ALL is included, the new node n_i is created on *all* the daemons matching the destination specification and a copy of the Messenger continues executing in each of the newly created nodes n_i.

The newly created logical node n_i can be named (using a variable or constant) or unnamed ($\sim$), and can be connected to the current node c by the link l_i, which could be named (using a variable or constant) or unnamed ($\sim$).

The defaults for the logical network parameters n_i, l_i, d_i are "$\sim$" and those for the daemon network parameters N_i, L_i, D_i are "$*$".

Examples. Assume that a MESSENGERS is executing in a logical node c mapped onto a daemon node C.

- $create(ln =\sim; ll =\sim; ldir =\sim;\ dn = *; dl = *; ddir = *; ALL) \equiv create(ALL)$
  create an unnamed node connected by an unnamed and undirected link to c on all neighboring daemons, i.e., those connected to C by any daemon link.

- $create(ln = a, b; ll = x, y; ldir =\sim, \sim;\ dn = *, *; dl = *, *; ddir = *, *)$
  $\equiv create(ln = a, b; ll = x, y)$
  create a connected to c by x on a neighboring daemon, and create b connected to c by y on another neighboring daemon

The *delete* Statement. The *delete* statement has the same syntax as *hop* and performs the same navigational operations. However, in addition to moving among nodes, it also deletes all logical links it traverses. If a node becomes a singleton, it is also deleted.

2.2 Global Virtual Time

Virtual time is an ordering of dynamically created events, where each event is the execution of a particular program or function. Each such event is scheduled to occur at a particular point along the virtual time line as used by applications to coordinate and synchronize their various subcomputations. The concept of virtual time has been developed in the context of discrete event simulations but is also useful in solving other common coordination problems, such as enforcement of a specific sequence of function executions, barrier synchronization, or termination detection in distributed systems.

On a single-processor machine, virtual time is implemented simply as a priority queue, such that events are time-stamped with the virtual time at which they are to execute and inserted into the queue according to their time-stamps. The system then repeatedly selects and processes the event with the smallest time-stamp, which may cause the insertion of new events. It then advances the virtual time value to the next event on the queue. Hence, conceptually, virtual time can be viewed as the value of a monotonically increasing software counter that emulates the passage of time.

In a distributed system, each machine maintains its own local virtual time, which is increasing independently of the others. To maintain the causal ordering of events, resulting from messages being exchanged among processors, the individual local virtual times must be synchronized. The globally minimal time obtained from this system-wide synchronization, which is referred to as *global virtual time (GVT)*, must be guaranteed to monotonically increase over the entire system, thus giving the illusion of a single global clock that applications can use to coordinate their actions.

A number of possible implementations of global virtual time have been proposed. MESSENGERS supports both a *conservative* and an *optimistic* approach [Jef85, Fuj90]. The conservative approach requires a continuous periodic exchange of timing information among all participating daemons, which results in a significant communication overhead. Optimistic approaches permit processors to advance their local virtual times at their own pace but require that a computation be rolled back if a "straggler" Messenger arrives, i.e., one whose time stamp is smaller than the node's current local virtual time. This, in turn, may require the sending of "anti-Messengers" to cancel Messengers that departed during the time that is being rolled back. A possible domino effect of cascading cancellations may result in a significant overhead for such purely optimistic strategies. Hence the choice between the different implementation strategies generally depends on the type of applications.

To utilize the global virtual time facility, MESSENGERS provides several library functions that permit a Messenger to suspend itself until a certain point in the virtual time has been reached. The two most important functions are the following:

- *M_sched_time_dlt(delta_time)* suspends the calling Messenger for a virtual time interval specified in *delta_time*.
- *M_sched_time_abs(abs_time)* suspends the Messenger until an absolute virtual time *abs_time*.

We will illustrate the use of these functions in temporal coordination using the matrix multiplication example in Section 3.2.

3 Applications

We contrast two different applications programmed using MESSENGERS and message-passing approaches. In both cases we compare the MESSENGERS implementation with an implementation using PVM [GBD+94] and with a sequential implementation using C. PVM was chosen over other popular message-passing systems (notably, MPI), because it is not only a library package of message-passing functions. Rather, it provides a complete execution environment (an abstract machine), which is much closer to MESSENGERS in its underlying philosophy as well as implementation. We used the publicly available version 3.3 of PVM for all experiments.

3.1 Smart Workers vs. Smart Managers

The manager-worker (or master-slave) paradigm is a well-known method for performing a large number of independent tasks in a distributed environment [And91]. In the conventional message-passing implementation of this paradigm, each worker repeatedly receives a computational task, performs the computation, and reports the result. The assignment of tasks to workers is performed by a separate manager. In this section, we contrast this conventional implementation with a MESSENGERS implementation. As we will see, the MESSENGERS implementation is simplified because the workers are able to coordinate themselves and hence a separate manager is unnecessary. We study the performance of the two implementations on a specific application, namely distributed computation of the Mandelbrot set.

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

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

Figure 2: Manager-Worker using message-passing

Consider the program in Figure 2, which shows a message-passing implementation of the common manager-worker 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 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 spawn 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).

```
 (1) manager_worker(){
 (2)  create(ALL)
 (3)  hop(ll = $last)
 (4)  while((task = next_task()) != NULL){
 (5)    hop(ll = $last)
 (6)    res = compute(task)
 (7)    hop(ll = $last)
 (8)    deposit(res)
 (9)  }
(10) }
```

Figure 3: Manager-Worker using Messengers

The corresponding Messengers version of this program is shown in Figure 3. This shows the script of a single Messenger, which is injected into the *init* node of some daemon. Its first course of action is to create the logical network. This is accomplished by the `create(ALL)` statement on line 2, which creates logical nodes connected to the current logical node on every neighboring daemon. This statement 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 these worker Messengers hops back to the original node by following the most recently traversed logical link, which can be accessed using the predefined network variable `$last` (line 3). It then attempts to get a new task to work on (line 4). If successful, it hops back to its logical node—again by following the most recently traversed link (line 5)—where it computes the result (line 6), carries it back to the central node (line 7), and deposits it there (line 8). 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. Note that because of the modified non-preemptive scheduling policy of MESSENGERS discussed in Section 2.1, no explicit synchronization is needed; a worker Messenger will not be interrupted while executing the call to `next_task()` on line 4 or `compute()` on line 6.

3.1.1 Programming Style

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 program components 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. Once the logical network has been created by the `create(ALL)` on line 2, each worker is capable of shuttling autonomously between the central node containing the tasks and its own node where it carries out its work. The logical network provides the "exogenous skeleton" for the application. The logical nodes can be thought of as work areas connected to the central node by corridors, which remain in place even though the work areas and the central node are not continuously occupied. Each worker can 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 is despite the fact that the message-passing version is only written in pseudo code—a fair amount of detail (e.g., packing and unpacking of message data) would have to be added to make this program run under PVM or any other message-passing system. The Messengers code, on the other hand, is very close to executable (the main difference is in the syntax of the function calls). 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.

3.1.2 Performance

To study the performance of the manager/worker paradigm, we chose a practical example, namely the generation of the Mandelbrot set, used to produce fractal curves for a 2D screen [PS88]. Computing the Mandelbrot set requires computing, for each pixel, the sequence of values $\{z_n\}$ given by

$$z_{n+1} = z_n^2 + c,$$

where $z_0 = 0$ and c is a complex number such that the x and y coordinates of the pixel correspond to the real and the imaginary parts of c. A color is assigned to each pixel depending on the first value of n for which $|z_n| > 2$. The resulting image is then displayed.

The computation of each pixel's color is independent of the color of any other pixel, so the problem is highly suitable to parallel processing. Furthermore, the amount of computation necessary for any given pixel is unknown *a priori*; the number of iterations can vary anywhere from 1 to the number of available colors. Hence the manager/worker paradigm, where the distribution of work is driven by the availability of workers, is a good match for this problem.

The performance evaluation was conducted using the same region of Mandelbrot imagery, located at $(-2.0, -1.2, 0.4, 1.2)$, with a fixed number of colors (512) and varying resolutions ($320 \times 320, 640 \times 640$, and 1280×1280). Each image was divided into grids of $8 \times 8, 16 \times 16$, and 32×32 blocks. Each worker program repeatedly picks up the next unprocessed block and computes the sub-problem at its own processor. All experiments were conducted on an Ethernet-based LAN of Sun SPARCstations 5. For comparison, we also implemented the same problems using a sequential algorithm in C running on a single

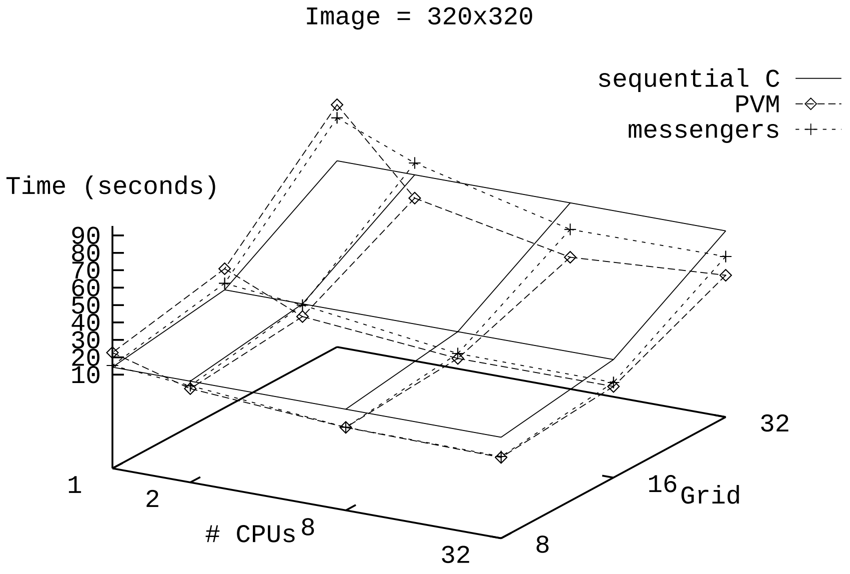


Figure 4: Mandelbrot set computation with image size = 320x320

workstation.

Figures 4, 5, and 6 show the performance of MESSENGERS, PVM, and sequential C, for the three different image resolutions of 320×320, 640×640, and 1280×1280, respectively. For each image resolution and each of the three grid sizes we vary the number of processors from 1 to 32.

The performance of MESSENGERS and PVM are very similar. PVM is slightly better when the grid is finer, i.e., when each subproblem to be solved by a worker becomes smaller. When the granularity is sufficiently large, MESSENGERS performance surpasses that of PVM. Figure 7 shows the results of the case most favorable to MESSENGERS, that is, solving the largest image (1280×1280) using the coarsest grid (8×8). MESSENGERS is five times faster than PVM on 32 processors.

Both systems achieve a speedup over sequential C in most cases, even when only two processors are used. In the most favorable case, MESSENGERS achieves an almost linear speedup on as many as 32 processors.

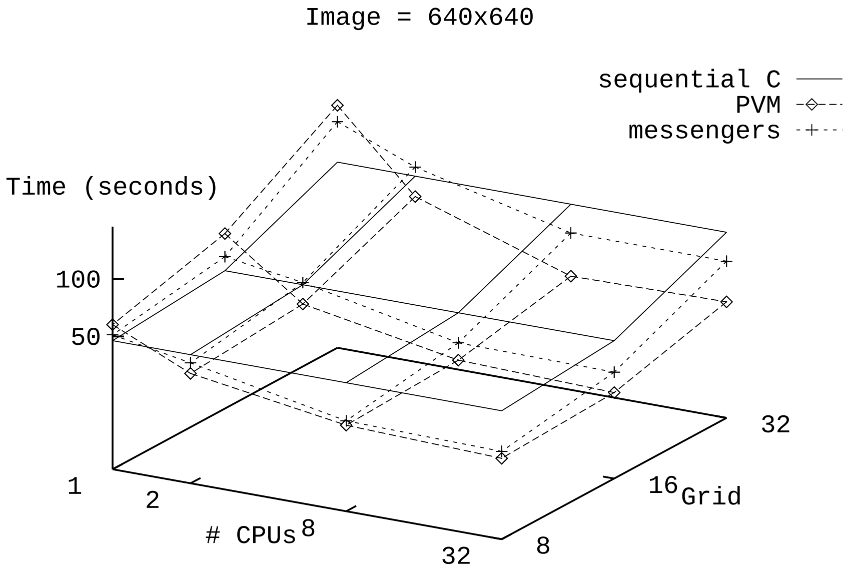


Figure 5: Mandelbrot set computation with image size = 640x640

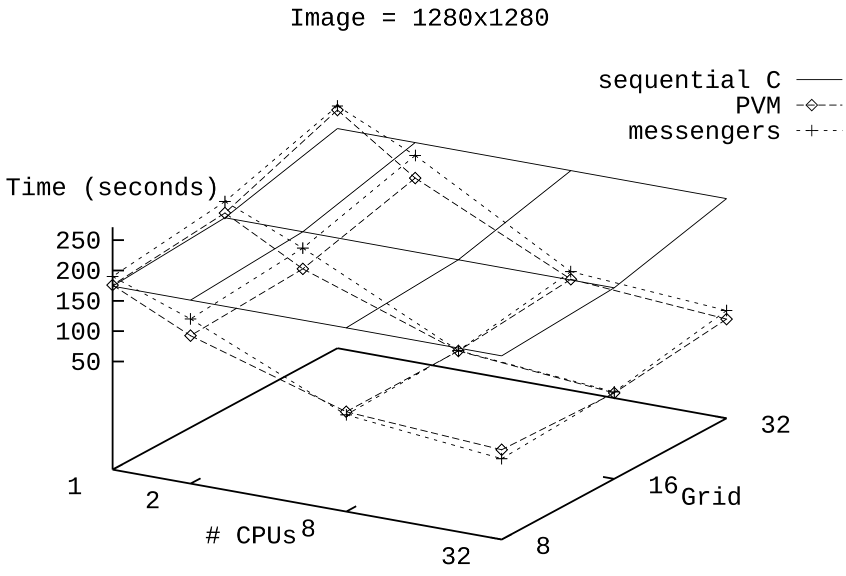


Figure 6: Mandelbrot set computation with image size = 1280x1280

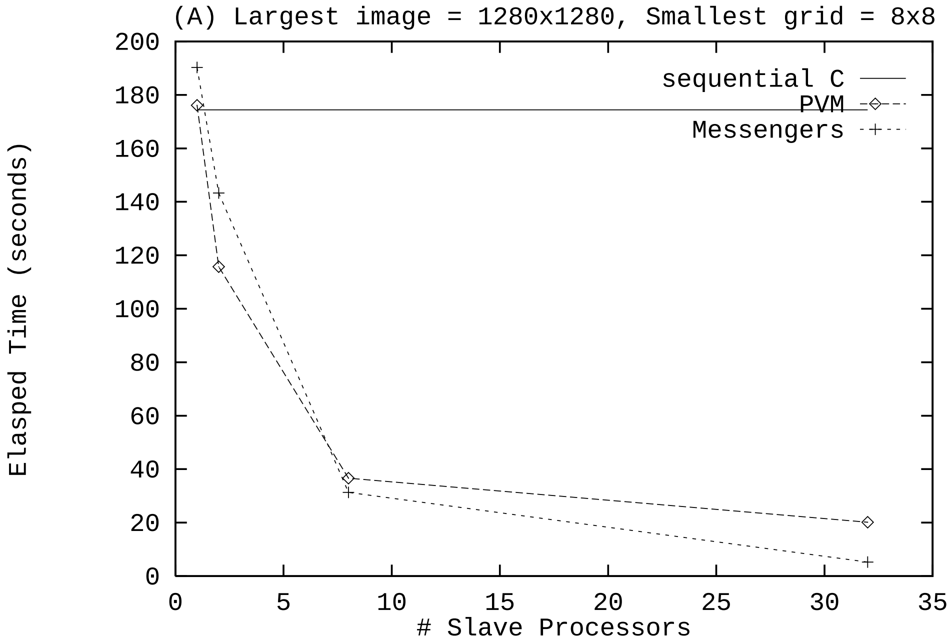
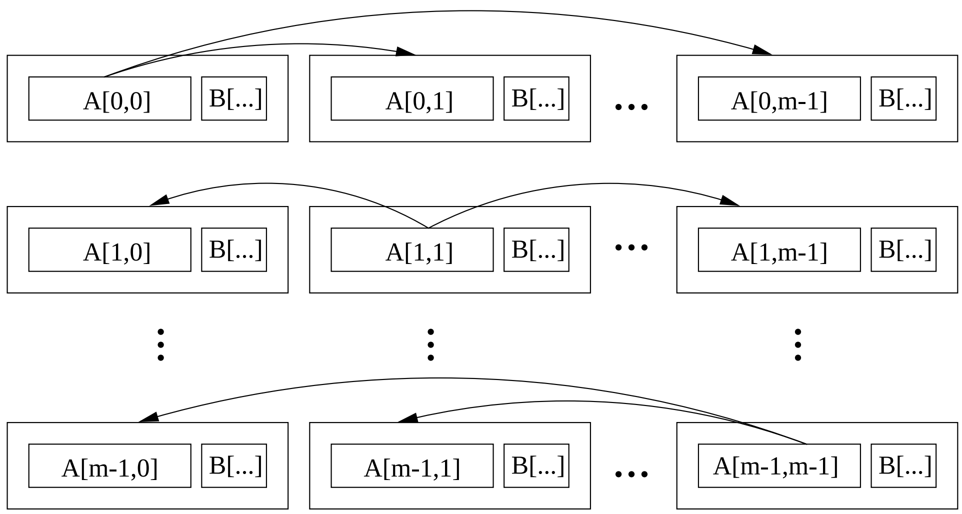


Figure 7: The best case of Mandelbrot set generation

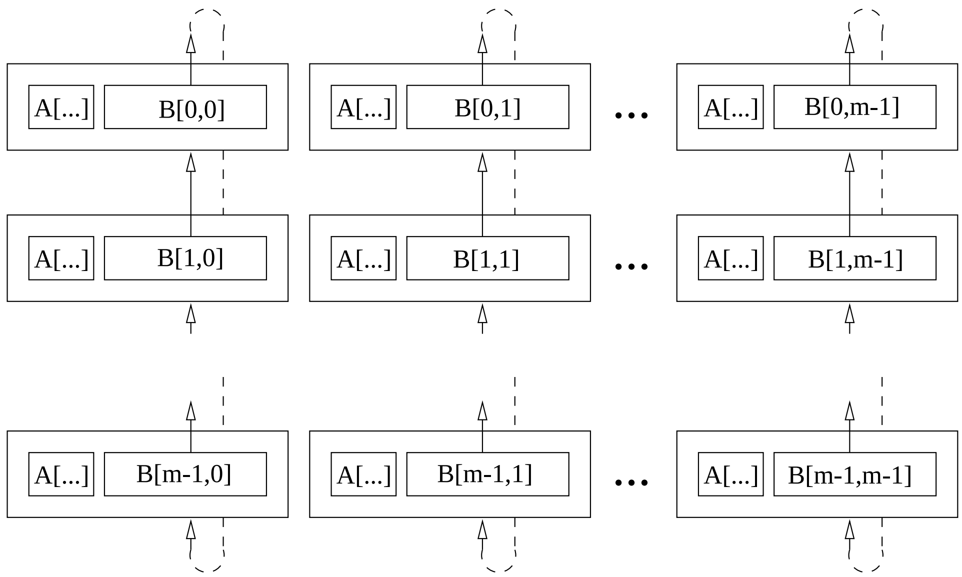
3.2 Matrix Multiplication

The application studied in the previous section involved coordinating tasks with varying computational workloads that could not be predicted in advance. In this section we study an application with a much more regular structure, namely matrix multiplication. We have chosen this applications for two main reasons. First, matrix multiplication is an essential component of a very large class of numerical computations and hence is an important problem in its own right. Second, Messengers is intended as a coordination paradigm for general-purpose computing. Many scientific problems either use matrix multiplication directly or must solve similar numeric problems that, like matrix multiplication, are highly regular in their structure and require significant movement of data between computations. Hence matrix multiply represents a good test case for studying Messengers's capabilities in this arena. This includes both its ability to describe the problem succinctly and intuitively in a distributed manner and the resulting performance.

The simplest possible implementation of matrix multiplication on a single processor is



(a)



(b)

Figure 8: (a) Distribution of A; (b) Rotation of B

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. Considerable effort has gone into developing sequential algorithms that are faster for large matrices but are more complex to program; see for example [Pan84, CW90]. Much of the emphasis in this research has been on improving the asymptotic performance beyond the $\Theta(n^3)$ running time of the naive algorithm for $n \times n$ matrices. But even modifications that do not improve the asymptotic performance beyond $\Theta(n^3)$ can provide some useful speedup. For example, partitioning a matrix into smaller blocks and then decomposing the multiplication into a series of additions and multiplications of these blocks can result in some speedup, because multiplying the smaller sub-matrices obtained by partitioning increases cache utilization and reduces paging overhead. The precise amount of speedup depends on the specific machine architecture. For example, our experiments indicate that on a 110 MHz SPARCstation 5 with 32MB of memory, partitioning a 1500×1500 matrix into 9 blocks of size 500×500 results in a speedup of roughly 13%.

The main point is that the sequential algorithm can be improved by carefully orchestrating data accesses to the matrices. In general, more speedup requires more careful orchestration, which significantly increases the complexity of implementing the algorithm. When compared with parallel matrix multiplication algorithms, we observe that these are no more difficult to write than those optimized for a single processor, yet we gain the additional benefit of speedup by utilizing multiple processors. In the remainder of this section we consider a well-known algorithm for parallel matrix multiplication [GBD+94], which we implement using two very different approaches to parallel program construction—one using message-passing and the other using MESSENGERS.

The algorithm is block-oriented in that the two square $n \times n$ input matrices A and B, and the resulting matrix C are all partitioned into $m \times m$ rectangular blocks of size $s \times s$. That is, $n = m * s$. We will use $A[i,j]$ to denote the corresponding $s \times s$ block of matrix A, i.e., all elements of A with indices ranging from $(i * s)$ to $(i * s + s - 1)$ and $(j * s)$ to $(j * s + s - 1)$. Similarly, $B[i,j]$ and $C[i,j]$ denote the corresponding $(s \times s)$ blocks of B and C. Each block $A[i,j]$, $B[i,j]$, and $C[i,j]$ is assigned to a different processor, addressed using the same coordinates $[i,j]$. The blocks of C remain stationary, that is $C[i,j]$ always resides on processor $[i,j]$, while the blocks of A and B are moved between the processors according to the following three distinct phases:

1. *Distribution of* A: During each iteration, k, all blocks $A[i,j]$ whose indices satisfy the condition $j = (i+k) \bmod m$ are multicast to all processors in the same row. This is illustrated graphically in Figure 8(a). During iteration 0 all elements on the diagonal $A[0,0]$, $A[1,1]$, ..., $A[m-1,m-1]$ are distributed to each row. The diagonal to be distributed is then shifted one to the right. That is, during iteration 1 the elements $A[0,1]$, $A[1,2]$, ..., $A[m-1,0]$ are distributed, etc.

2. *Block multiplication*: The block of A received during the previous step is multiplied with the block B currently residing in the processor and is added to the block of C assigned to the processor.

2. *Rotation of* B: Each block of B is moved to the neighboring processor in the same column using a circular shift. Specifically, the block currently residing on processor $[i,j]$, is shifted to processor $[(i-1) \bmod m, j]$. This is illustrated in Figure 8(b) for iteration 0.

After repeating the above three phases m times, the (distributed) matrix C contains the result of the complete multiplication of the matrices A and B.

Figure 9 shows the implementation of the above algorithm using message-passing. The program first acts as a manager by spawning $m \times m$ copies of itself (lines 3-6). Each copy is assigned a distinct block number $[i,j]$ and is told the block size (s) and the number of blocks per dimension (m). Each of the newly spawned worker processes then joins a multicast group (line 2) in order to communication with the other workers, and it computes the IDs of all other workers in the same row (lines 8-9). Next, each worker enters the for-loop during which all data movement and the individual block multiplications are performed. Lines 11-14 accomplish the multicast of the appropriate elements of A. That is, during each iteration, one of the processors in each row (i.e., one of the diagonals) will satisfy the condition on line 11 and will multicast its block to all other processors in that row (line 12); all others will receive it (lines 13-14). Thereafter, the block multiplication is performed

```
 (1) matrix_mult(s, m, i, j) {
 (2)   join_group("mmult", get_pid());
 (3)   if (parent_id() == VOID) {      /* manager process */
 (4)     for (i = 0; i < m; i++)
 (5)       for (j = 0; j < m; j++)
 (6)         child = spawn(matrix_mult, s, m, i, j);
 (7)   } else {                        /* worker processes*/
 (8)     for (k = 0; k < m; k++)       /* make a multicast group */
 (9)       myrow[k] = pid_in_group("mmult", i*m+k);
(10)     for (k = 0; k < m; k++) {
(11)       if (j == (i + k) mod m)
(12)          multicast(myrow, block_A);  /* multicast A to all nodes in row */
(13)       else
(14)          block_A = receive();
(15)       multiply(A, B, C);          /* Cij = Aij * Bij */
(16)       send(pid_in_group("mmult", (i-1)*m+j), block_B); /* rotate B */
(17)       block_B = receive();
(18) } } }
```

Figure 9: Block Matrix Multiplication using PVM

(line 15). Finally, the matrix B is rotated by each processor sending the current block to its "northern" neighbor in the same column (line 16) and receiving the corresponding block from its "southern" neighbor (line 17).

For the MESSENGERS solution, we first need to build a logical network that will serve as the "exogenous skeleton" for the movement of Messengers and their computations. The topology of this network is shown in Figure 10, where each row is a fully connected subnetwork while each column is a ring. All horizontal links are labeled "row" and are undirected, while all vertical links are labeled "column" and are directed "upward". In general, any static logical network is constructed by describing its topology in a file (either manually or using a graphics tool) and then starting a specialized service Messenger called *net_builder*, which reads the topology file and creates the corresponding logical network. In the case of the network of Figure 10, the effort to produce it is trivial. Since the topology is regular in both dimensions, the user only needs to specify the size and connectivity along each dimension. Both of these are simple parameters to the *net_builder* tool.

Similar to the PVM solution, we assume that the matrices are already distributed over the network (as a result of previous computations) such that each block $[i, j]$ of A, B, and C resides initially on the corresponding node (processor) named $[i, j]$.

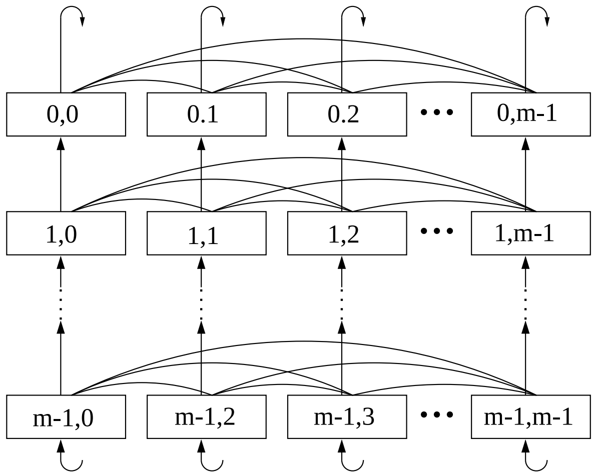


Figure 10: Logical Network for Matrix Multiplication

```
distribute_A(s, m, i, j) {
  M_sched_time_abs((j-i) mod m);
  msgr_A = copy_block(resid_A);
  hop(ll = "row")
  curr_A = copy_block(msgr_A);
}

rotate_B(s, m, i, j) {
  msgr_B = copy_block(resid_B);
  for (k = 0; k < m; k++) {
     M_sched_time_dlt(.5);  /* synchronization */
     C = block_multiply(msgr_B, curr_A, C);  /* Cij = Aij * Bij */
     hop(ll = "column"; ldir = +);  /* rotate B to row i-1 */
  }
}
```

Figure 11: Block Matrix Multiplication using MESSENGERS

The MESSENGERS code for the matrix multiplication is then shown in Figure 11. It consists of two distinct Messengers; an instance of each is injected into every node of the logical network. The *distribute_A* Messenger implements the movement of the array A in that each instance of this Messenger is the embodiment of one of A's blocks. Its task is to get to the appropriate node whenever it is needed there. This temporal coordination is accomplished using the global virtual time discussed in Section 2.2. Each of the *distribute_A* Messengers schedules itself to wake up at the time corresponding to its position in the logical network (line 2). That is, all instances on the initial diagonal ($A[0,0]$, ..., $A[m-1,m-1]$) will wake up at time 0, since their $(j-i)$ mod m value is 0. The instances on the next diagonal to the right ($A[0,1]$, $A[1,2]$, ..., $A[m-1,0]$) will wake up at virtual time 1, and so on.

Whenever an instance of the *distribute_A* Messenger wakes up, it copies the block of A currently stored in the node variable $resid\_A$ into its own messengers variable $msgr\_A$ (line 3), it replicates itself to all nodes in the same row by hopping along all links labeled "row" (line 4), and deposits the block in that node by copying it into the appropriate node variable $curr\_A$. Once it completes the task of distributing its A block, the Messenger terminates.

Each instance of the *rotate_B* Messenger is the embodiment of one of the blocks of the B matrix, which it copies from the node variable area $resid\_B$ to its private messenger variable area $msgr\_B$ when it is first instantiated in a node (line 8). It then enters a loop during which it keeps moving the block it is responsible for up along its respective column (i.e., along the direction of the column link, line 12). The actual block multiplication, which could in principle be performed by a separate set of Messengers, has been included with the *rotate_B* Messenger. This is possible because each shift of a block of B is alway preceded by a corresponding multiplication and hence the overhead of implementing a separate Messenger can be eliminated. The synchronization of the *rotate_B* Messengers is achieved on line 10—each of these Messengers schedules itself to wake up at the half-way point between any two full time ticks, that is, at time 0.5+k. Hence the two Messengers *distribute_A* and *rotate_B* always alternate between their respective executions. Each time *rotate_B* wakes up it performs a block multiplication using its own block of B ($msgr\_B$) and the currently resident block of A ($curr\_A$), and adding the result to the (always-resident) block of C (line

11). Thereafter it hops to its "northern" neighbor (line 12).

3.2.1 Programming Style

Even though both programs implement the same algorithms, the MESSENGERS program results in a very different style of programming:

- The message-passing version takes the conventional point of view where each node runs a stationary process, that performs all the local computations and exchanges of data with neighboring processes as necessary. The abstract algorithm, however, is explained in terms of the data movement, that is, it prescribes what happens to blocks of A and B during each iteration and each phase. Hence the communicating processes must be expressed in terms of two primitives—`send` and `receive`—that need to synchronize with each other so that each pair accomplishes the movement of the appropriate data block at the right time. In particular, the control must assure that only one process in each row performs a multicast of an A block while all others performs a matching receive. Similarly, each process must perform a matching receive for every send of a B block to achieve the rotation of the matrix B.

  The Messengers version takes a *data-centric* point of view. It uses the logical network (Figure 10) as the "exogenous skeleton" within which Messengers move. Each block of A and B is represented by a separate Messenger, which is responsible for carrying it to wherever it is needed at the appropriate time. This permits us to follow the algorithmic description much more closely. Notably, each block of A is programmed to wake up at the appropriate time based on its position in the matrix and to replicate itself along its row. Similarly, each block of B must move up by one node during each iteration. This permits us to deal with the movement of A and B separately by writing two different Messengers scripts. These are then coordinated using the virtual time synchronization. That is, we make sure that the set of *distribute_A* Messengers and the set of *rotate_B* Messengers alternate in time by letting the former wake up at each full time tick k while the latter wake up every half tick $k + 0.5$. The decomposition of the problem into two independent scripts synchronized only via the global virtual time follows closely the abstract specification of the block multiplication algorithm and thus makes the MESSENGERS code conceptually simpler to design and understand.

- The MESSENGERS program is also shorter. This holds 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 actual code in the PVM book spans several pages). The Messengers code, on the other hand, is again very close to executable. The reduction in code length is due to the data-centric point of view, which mimics more closely the algorithm's description and reduces its semantic gap.

3.2.2 Performance

We have conducted the performance comparison between MESSENGERS, PVM, and sequential C implementations on a 2×2 and a 3×3 grid of processors, both using an Ethernet-based LAN of SPARCstations 5. (All experiments relating to the 2×2 grid were run on 110 MHz SPARCstations, while all experiments relating to the 3×3 grid were run on 170 MHz SPARCstations.) The results for these two processor configurations are shown in Figure 12(a) and (b), respectively. MESSENGERS achieves speedup over PVM beyond a block size of approximately 150 on the 4-processor configuration (2×2), and a block size of 20 on the 9-processor configuration (3×3).

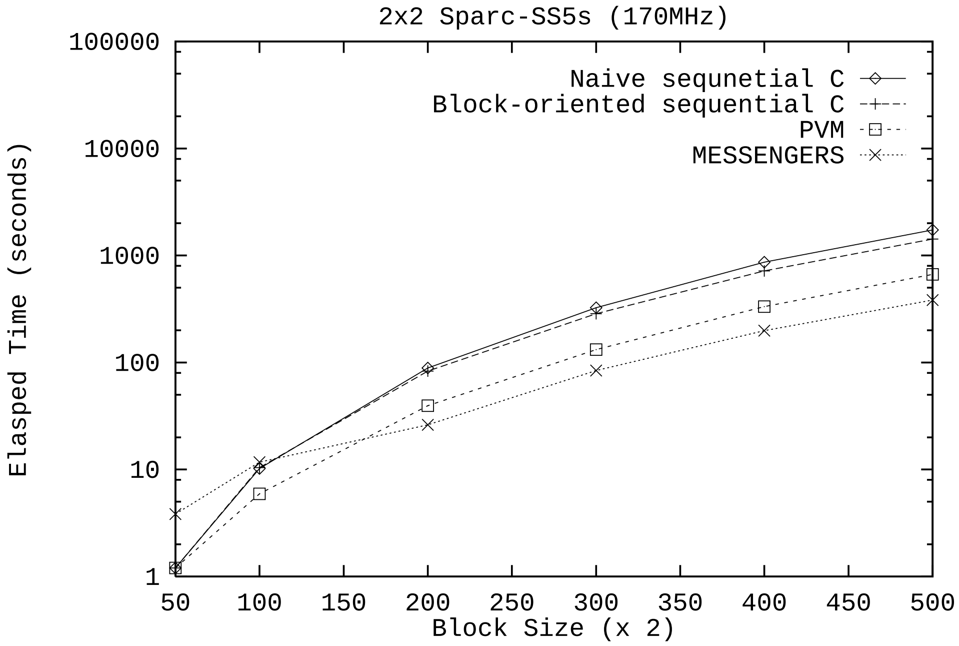
To compare the performance to a sequential program, we implemented two versions of matrix multiplication using C. The first is the simple naive algorithm consisting of the three nested loops as described at the beginning of Section 3.2. The second is a block-oriented matrix multiplication, where the matrices are partitioned according to the same grid as the parallel algorithms (i.e., 2×2 and 3×3) and the multiplication 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. As shown in Figure 12, this yields a significant speedup over the naive algorithm. The figure also shows 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 super-linear 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.

4 Related Research

Self-migrating computations have been employed primarily in the construction of "intelligent" mobile agents, which are programs capable of physically moving through wide-area communication networks (notably the Internet) and performing a variety of service tasks on behalf of their users [IEE96, RPZ97, VT96]. While MESSENGERS and mobile agents share the same basic principle of autonomous navigation, they differ with respect to both intent and structure. Mobile agents typically aim at providing a vehicle for using various distributed services available on wide area networks or to deal with intermittent, unreliable, or low-bandwidth/long-latency connections typical for mobile computing. MESSENGERS, in contrast, is aimed at general-purpose computing. The main goal is to provide a convenient vehicle to harness the computational capacity of existing networks, notably, clusters of workstations or PCs, to speed up some computational task.

Two systems, BPEM and WAVE, were among the first to explore the possibilities of applying the principles of self-migrating objects to general-purpose distributed computing. BPEM [BL87] is a computational model designed to facilitate the parallel processing of knowledge, represented in the form of semantic nets. WAVE [SB94, SB96] is a complete environment of self-migrating computations. It is the closest to MESSENGERS in that both share the following aspects:

- WAVE is the only mobile-agent-based system (other than MESSENGERS) that explored the principles of self-migration for computationally-intensive general-purpose computing. Notably, it has been applied to large scale distributed simulations, the study of collective behavior, various graph algorithms and network control problems,



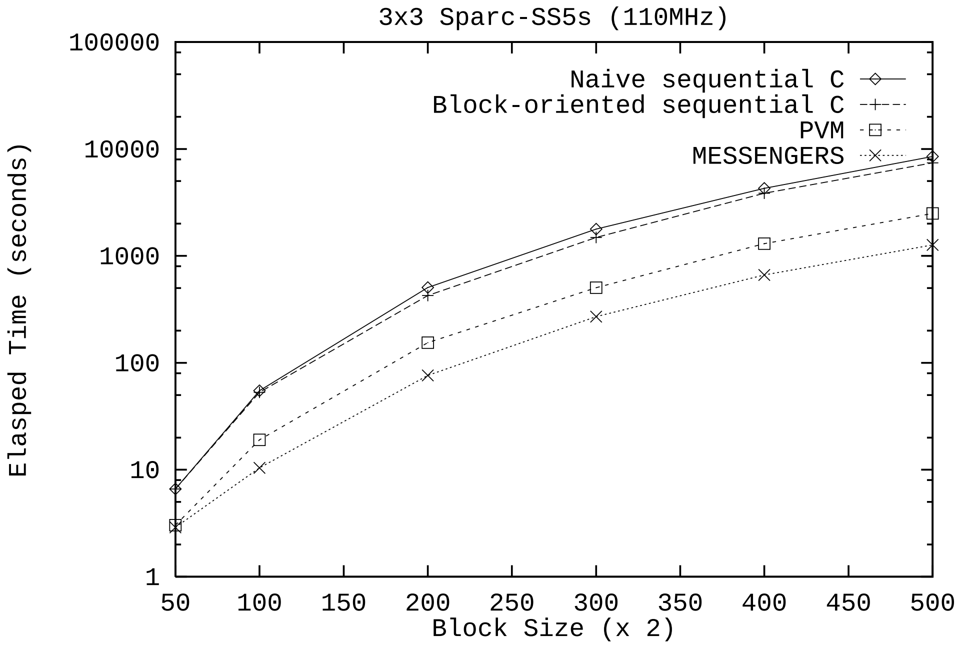


Figure 12: **Matrix Multiplication results.**

virtual reality, and a number of other applications.

- Both WAVE and MESSENGERS permit the creation of an application-specific logical network, whose size and topology are completely independent of the underlying daemon and physical networks. The individual self-migrating computations can use both logical nodes and logical links to express how they wish to replicate and spread through the logical network.

Beyond these common features, the WAVE and the MESSENGERS systems are quite different in both philosophy and implementation:

- WAVE uses a highly specialized language, where each program is a possibly recursive sequence of "moves" where each move specifies the next action to be taken by the program. MESSENGERS, in contrast, uses standard C, extended with three navigational commands implemented as library functions. All standard data types supported by C (other than pointers) are also supported by MESSENGERS.

- WAVE features a highly elaborate semantics of navigation. Notably, each move may be prefixed by a rule that expresses a specific search strategy, such as sequential, OR-sequential, AND-parallel, and a number of others. This, in turn, requires that the runtime infrastructure maintains a trace of every WAVE program so that it can backtrack as necessary. The MESSENGERS navigational philosophy is entirely different. It is based on a navigational calculus (as described in Section 2.1) [FBDM98], which permits each Messenger to specify a set of destination nodes using a set of predicates involving logical and/or physical nodes and links. The Messenger is then replicated to all those destinations that match the specification.

- MESSENGERS provides automatic support for global virtual time, which, as was demonstrated in this paper (Section 2.2), greatly simplifies the construction of certain distributed programs.

- MESSENGERS, unlike WAVE or any other mobile agent system, relies on the availability of a shared file system among all nodes. This limits its applicability to LAN-based clusters of workstations or PCs but it permits a greatly optimized implementation. Notably, Messengers code does not need to be carried between nodes but can be loaded as necessary. Similarly, Messengers are able to use and share files. These design decision are in line with the MESSENGERS intended application area, which is not the Internet but rather high-performance general-purpose computing on clusters of computers.

A third area of related research are coordination paradigms [AHM96, CE96], which provide the synchronization, communication, and creation/destruction of computational activities required to orchestrate the individual computations into a coherent system. Examples of coordination paradigms include Gamma [BL93], Linda [CG89], and the IWIM model [Arb96]. MESSENGERS is similar in some respects to PoliS [Cia94], a variant of Linda intended to simplify the design of distributed systems by incorporating explicit partitioning of the underlying state space (called a "tuple space" in the Linda model). One difference is that in PoliS, computational objects (called "agents") communicate by writing messages in other agents' tuple spaces. Thus an agent would move from one tuple space to another by writing a copy of itself in a new tuple space and then finishing its execution in the old tuple space (so navigation is a byproduct of communication). The MESSENGERS point of view, by contrast, is strongly navigation-oriented. Messengers communicate though a rendezvous at a logical node and an exchange of information through node variables. Messengers scripts, which are written from the point of view of a moving entity, are used for all aspects of coordination, including dynamic creation and destruction of spaces (logical nodes), creation and destruction of activities (Messengers), and inter-Messengers communications.

5 Conclusions

MESSENGERS is a system based on the philosophy of distributed programming using self-migrating computations. This allows the programmer to adapt a completely different point of view: instead of viewing the application as a global collection of concurrent activities interacting with each other via message-passing, the new paradigm puts the programmer into the "driver's seat" of an autonomous object, which it must guide on its journey through the network.

In this paper we have used the MESSENGERS system as a vehicle to study the applicability of self-migrating computations to general purpose computing and to contrast it with conventional message-passing by studying two specific applications from the general-purpose distributed computing area—the generation of a Mandelbrot set using a manager/worker paradigm and matrix multiplication.

The main contributions of this paper are the following:

- Programming style: we have compared the two programming styles—one using message-passing and the other using messengers—by showing the analogous programs for two different applications. This suggests that, for certain classes of applications, Messengers programming results in a smaller semantic gap between the abstract algorithmic specifications and their actual implementations, which makes them easier to construct, understand, and maintain. To our knowledge, no such direct comparisons have yet been conducted and published in the literature.
- Global virtual time: we have demonstrated that global virtual time is a useful feature not only for distributed simulations but as a general synchronization primitive for general purpose computing. Thus having support for global virtual time at the system level greatly simplifies the construction of certain applications.
- Applicability to numeric applications: mobile agents or other forms of self-migrating computations have been applied to many problem domains but not to numeric computations. Our experiments show that purely numeric computations, like matrix multiply, can also benefit from the data-centric view of self-migrating computations.
- Performance: very few performance studies have been conducted using self-migrating computations. One notable exception is a performance model for mobile agent systems, which compares remote procedure calls with mobile agents in the context of the Mole system [SS97]. Another performance evaluation has been done for the Omniware system, which focused on the overhead of its safety features [ATLLW96]. Both of these studies concentrated on Internet-based computing. The present paper is an

continuation of our earlier experiments [FBD97] and, to our knowledge, is the only line of research conducting specific performance evaluations of self-migrating computations for general purpose distributed computing. It shows that, depending on the computational granularity, MESSENGERS applications are capable of achieving significant speedups over sequential C programs and are competitive with message-passing environments.

For additional information, including access to the more detailed technical reports about MESSENGERS referenced in this paper, the interested reader is invited to browse our WWW page: `http://www.ics.uci.edu/~bic/messengers`.

References

[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 P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, pages 34–56, Cesena, Italy, April 1996. Springer Verlag Lecture Notes in Computer Science 1061.

[ATLLW96] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 127–136, Philadelphia, PA, May 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. *Comm. ACM*, 36(1):98–111, Jan. 1993.

[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.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, March 1990.

[DIS94] DIS Steering Committee. The DIS vision: A map to the future of distributed simulation. *Institute for Simulation and Training*, 1994.

[FBD97] M. Fukuda, L.F. Bic, and M.B. Dillencourt. Performance of the messengers autonomous-objects-based system. In *First Int'l Conf. on Worldwide Computing and Its Applications (WWCA97)*, Tsukuba, Japan, March 1997. Springer-Verlag. Lecture Notes in Computer Science 1274.

[FBDM98] M. Fukuda, L.F. Bic, M. Dillencourt, and F. Merchant. Distributed coordination with messengers. *Science of Computer Programming*, 31(2), 1998. Special Issue on Coordination Models, Languages, Applications.

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

[GBD+94] A. Geist, A. Beguelin, J. Dongarra, W. Jieng, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, MA, 1994.

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

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

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

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

[RPZ97] K. Rothermel and R. Popescu-Zeletin, editors. *MA '97: First International Conference on Mobile Agents*, Springer-Verlag, Lecture Notes in Computer Science 1219, Berlin, Germany, Apr. 1997.

[SB94] P.S. Sapaty and P.M. Borst. An overview of the WAVE language and system for distributed processing of open networks. Technical report, Dept pf EEE, University of Surrey, Guildford, UK, 1994. `http://www.ira.uka.de/I32/wave/wave.html`.

[SB96] P. Sapaty and P. Borst. Wave: Mobile intelligence in open networks. In *Proc. 1st Annual Conference of Emerging Technologies and Applications in Communications (etaCOM'96)*, Portland, OR, May 1996.

[SS97] M. Strasser and M. Schwehm. A performance model for mobile agent systems. In *Proc. of the Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, Las Vegas, NV, June 1997.

[VT96] J. Vitek and C. Tschudin, editors. *MOS'96: Second Int'l Workshop on Mobile Object Systems*, Springer-Verlag, Lecture Notes in Computer Science 1222, Berlin, Germany, July 1996.