Mobile Agents - The Right Vehicle for Distributed Sequential Computing

Lei Pan, Lubomir F. Bic, and Michael B. Dillencourt

Technical Report No. 01-68

Information and Computer Science, University of California, Irvine, CA 92697-3425
{pan,bic,dillenco}@ics.uci.edu

December, 2001

Mobile Agents - The Right Vehicle for Distributed Sequential Computing

Lei Pan, Lubomir F. Bic, and Michael B. Dillencourt

Information and Computer Science, University of California, Irvine, CA 92697-3425
{pan,bic,dillenco}@ics.uci.edu

Abstract. Distributed sequential computing uses the collective memory of a network of workstations to reduce paging overhead. In contrast to the "page farm" approach, in which a stationary program remotely accesses data, distributed sequential computing moves the code to the data. In this paper, we show that mobile agents are the most natural and effective way to implement this approach. This is because mobile agents preserve the algorithmic integrity of sequential programs, while a message passing implementation requires a complete restructuring of the code.

Keywords: mobile agents, MESSENGERS, distributed computing, distributed sequential computing, algorithmic integrity, paging, Crout factorization, Network of Workstations

1 Introduction

While mobile agents have been successfully employed in many special situations, they have not been widely used for general-purpose computing. It has been said that what mobile agents need is a "killer application" [6]. In this paper, we discuss a programming paradigm called *distributed sequential computing*, introduced in [5], and we argue that mobile agents are uniquely suited to this mode of programming.

In essence, distributed sequential computing is a means of applying the power of a network of workstations to improve the performance of data-intensive sequential programs without reprogramming them. This is based on the observation that under certain circumstances partitioning the data onto different machines and reducing the disk paging overhead by using the collective memory of a network of workstations can result in considerable performance increase, without converting the underlying algorithm to a parallel implementation. One means of doing this is the "page farm" approach described in [11], in which a process runs on a single machine and accesses data by using remote memory accesses. A major disadvantage of this approach is that it uses a large amount of data bandwidth when the total amount of data to be accessed is large. Another approach is to use the distributed sequential computing paradigm introduced in [5]. The data is distributed over the workstations in the network just as in the "page farm" approach. The difference is that rather than having the code run on a single machine and remotely access the data, the code moves to the data. This approach is useful when there is large amount of data but the program state information is relatively small, so that moving the code to the data requires less bandwidth than moving the data to the code.

In [5] we described several examples of this approach, and we showed that it has two major advantages: *algorithmic integrity* and *performance improvement*. Algorithmic integrity refers to the fact that the distributed sequential algorithm is identical to the sequential algorithm. Additional statements (annotations) may be added to tell the code how to access the data, but the sequential algorithm remains intact. The performance improvement comes from the reduction or elimination of paging overhead, and the decrease in network traffic from moving code to data rather than data to code. In addition, the agent doing the computation can inject other agents to preload data that will be needed in the future or to post-write data that has already been computed and needs to be written to the disk.

In the present paper, we argue that mobile agents are not only a good way to do distributed sequential programming, but they are *the* right way to do it. Of course, anything that can be done with mobile agents can be done with message passing: after all, at low level mobile agents are a special case of message passing, in the sense that an agent is ultimately a stream of bytes and hence a message. But, at a high level which is closer to the level of the application programmer, mobile agents provide a new layer of abstraction that helps the programmer in two ways. First, a distributed sequential program implemented as a mobile agent has a single locus of computation that captures the entire program state. This means that a programmer using mobile agents does not have to distribute the program state over all the participating nodes; this must be done explicitly in a message-passing implementation. This observation, that mobile agents simplify the programming task by eliminating the necessity of explicitly maintaining the state of the process, has been previously articulated in [10]. Second, in a mobile agent implementation, the attention of the programmer is switched from stationary nodes to moving agents. This change of focus is significant because now data that gets communicated between nodes is not seen as being *moved* at all; rather, it is *carried by the agent* during the move.

For comparison purposes, we show in this paper that if we attempt to do distributed sequential programming using a message-passing paradigm, the programming task becomes considerably more complicated. So while we do not claim to have found a "killer application" for mobile agents, we do claim that we have identified an important class of programs with the property that the only natural way of expressing these programs is through the use of mobile agents.

The paper is organized as follows. Section 2 contains a brief discussion of the message-passing and agent-based approaches to distributed computing. Section 3 presents a simple example of distributed sequential computing, and compares the different ways mobile agents and message passing would be used to implement the algorithm. Section 4 describes a numerical application, Crout factorization, and its different implementations. The last section contains some conclusions and final remarks.

2 Message Passing and Mobile Agents

Message passing is a commonly used approach for distributed and parallel computing on network of computers. Messages are used to communicate problem data, execution status, and process synchronization among computing nodes on a network. Today, the message passing standard defined by the Message Passing Interface Forum ([7]), or MPI, is a *de facto* standard for message passing on both distributed-memory supercomputers and networks of workstations. The most fundamental communication mechanism provided by MPI is the send-receive primitive. A data source performs a *send*() to transmit a message to a destination, which must perform a *receive*() to accept it.

Recently mobile agents are starting to be considered for distributed computing. Mobile agents are programs that move dynamically among networked machines, carrying their data and execution states with them. A mobile agent, with "strong mobility" ([9]), can halt its execution, encapsulate the values of its variables and execution stack, move to another machine, restore the state, and continue executing. Although ultimately based on message passing at low level, this "mobile" capability, as will be shown later in this paper, is the right vehicle for application programmers who want to make good use of a distributed environment with less efforts.

All mobile-agent systems have the same general architecture: a server on each machine accepts incoming agents, and for each agent, starts an appropriate execution environment, loads the agent's state information into the environment, and resumes agent execution.

The most important feature that distinguishes agent-based systems from conventional message-passing systems is that all functionality of the application is embedded in individual agents, i.e., the programs are carried by agents as they navigate through space. This is in contrast to message-passing, where messages are only passive carriers of data.

The MESSENGERS system ([2–4]) is an environment for distributed computing in which applications are developed as collections of autonomous self-migrating computations, called Messengers. In the MESSENGERS system, there are three levels of networks. The lowest level is the physical network(e.g. a LAN or WAN), which constitutes the underlying computational nodes. Superimposed on the physical layer is the daemon network, where each daemon is a server process that receives, executes, and dispatches Messengers. The logical network is an application-specific computation network created on top of the daemon network. Messengers may be injected (from the shell or by another Messenger) into any of the daemon nodes and they may start creating new logical nodes and links on the current or any other daemons.

The important concepts and features of MESSENGERS are as follows:

- All participating physical nodes have MESSENGERS daemons running on them. The logical network is thus established on top of this daemon network, which in turn runs on the physical network.

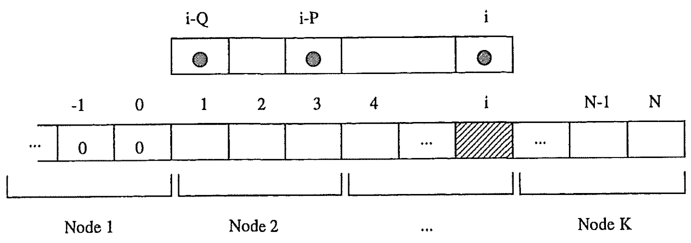

Fig. 1. Access pattern and array distribution for a simple 1-D array access program.

- The MESSENGERS scripting language is a subset of C, so it is easy for C/C++ programmers to develop MESSENGERS applications. Furthermore, the MESSENGERS script is first preprocessed into C code which is then compiled into machine native code, and therefore the execution is very efficient;.
- There are two types of variables: Messenger variables and node variables. A Messenger variable, often taken as a medium for communication, is one that belongs to a particular Messenger and travels with that Messenger to different logical nodes, whereas a node variable is one that is stationary to a logical node.
- The most important navigational statement for Messengers are *create*() and *hop*(). The *create*() statement generates a link along which a Messenger moves, and creates a logical node on the physical node. The *hop*() statement causes the Messenger to navigate to a node, along a link or using the node's logical address.

3 A Simple Example: Updating A 1-D Array Sequentially

In this section, we present a simple, contrived example to illustrate the idea behind distributed sequential programming and also to demonstrate why mobile agents (as opposed to message passing) are the right way to do distributed sequential programming. The example consists of an algorithm that sequentially updates the items in a one dimensional array according to a certain fixed pattern.

Figure 1 shows the access pattern. and Figure 2(a) contains the sequential program defining the algorithm.

P and Q are positive constant integers, $f_a()$, $f_p()$ and $f_q()$ are functions. It is assumed that $a[j] = 0$ for $j \leq 0$, and that the number of bytes required to represent each element of the array $a[\cdot]$ is considerably larger than the number of bytes required to represent the values returned by the functions $f_p()$ and $f_q()$. For example, each element of $a[\cdot]$ could be a large structure or a $k \times k$ array, while each of the values returned by the two functions might be a scalar or a column vector with k entries.

As N and the size of each element of $a[\cdot]$ become large, the array $a[\cdot]$ will no longer fit into the main memory of a single computer, and disk paging will cause

```
(1) for ($i = 1$; $i <= N$; $i++$) {
(2)       $x = f_p(a[i - P])$;
(3)       $y = f_q(a[i - Q])$;
(4)       $a[i] = f(x, y)$;
(5) }
```

(a)

```
(1)     for ($i = 1$; $i <= N$; $i++$) {
(1.1)       $hop(node\_map(i - P))$;
(2)         $x = f_p(a[i - P])$;
(2.1)       $hop(node\_map(i - Q))$;
(3)         $y = f_q(a[i - Q])$;
(3.1)       $hop(node\_map(i))$;
(4)         $a[i] = f(x, y)$;
(5)     }
```

(b)

Fig. 2. Pseudocode for a simple 1-D array access program. (a) the sequential implementation. (b) the distributed sequential implementation using MESSENGERS.

the program's performance to degrade badly. This disk paging can be avoided by distributing the array onto several machines, as illustrated in Figure 1. Figure 2(b) shows the distributed sequential implementation using MESSENGERS. The code is simply the original sequential code, augmented with three *hop*() statements. A *hop*() statement suspends the execution, moves the mobile code to a node whose number is passed as an argument, and resumes execution on that node. A command to hop to the node on which the mobile program is already running is a no-op, and the cost is negligible. The function *node_map*(j) returns the node on which the element $a[j]$ resides. Thus in Figure 2(b), line (1.1) causes the mobile code to hop to the machine where array item $a[i - P]$ resides. The statement at line (2), causes the value $f_p(a[i - P])$ to be loaded into the MESSENGERS variable x, which is then carried around by the Messenger. Since $f_p(a[i - P])$ is much smaller than $a[i - P]$ in size, carrying x around is much cheaper than shipping $a[i - P]$ first and computing $f_p(a[i - P])$ later.

If we compare the MESSENGERS code in Figure 2(b) with the original code in Figure 2(a), we see that in the MESSENGERS implementation, the algorithm sequence and code structure are not changed. The only change is the insertion of three *hop*() statements. This code change preserves the algorithmic integrity; in essence, the *hop*() statements are annotations describing how to obtain the data, rather than statements that affect the computation or change the algorithm. Statically, the MESSENGERS code is very similar to the original code, which is a big advantage for programming and code maintenance. On execution, the MESSENGERS code moves through the distributed environment, taking full advantage of distributed computing.

To develop the distributed sequential program from the sequential program, the programmer first needs to decide how to distribute the data. This is a necessary step for any distributed programming paradigm, such as message passing. Notice that a by-product of this step would be constructing the function *node_map*(), which is the only detail missing from the pseudocode in Figure 2(b). The program must then be augmented to "drive" through the network, performing computation at the appropriate locations and making intermediate stops, as necessary, to "pick up" or "drop off" data. For real applications, this requires careful thinking, but the simple example presented above illustrates the basic idea.

We can of course implement the same algorithm using message passing. Since message passing is happening among processes on different nodes, a natural way of quantifying the complexity of this approach is to count the number of "roles" or "node states" that all the nodes can have. Let us consider one node, and see how many roles it can take. In any iteration of the main loop, three array entries are referenced, namely $a[i]$, $a[i-P]$, and $a[i-Q]$. Call the nodes containing these three entries the "participating nodes." The node containing $a[i]$ (call it the "master node") can either contain or not contain $a[i-P]$, and it can independently either contain or not contain $a[i-Q]$. (Note: we are assuming here that we do not know whether $P > Q$; if we knew this, we could eliminate one possibility.) In the case where neither $a[i-P]$ nor $a[i-Q]$ is on the master node, these two values could be on a common second node or on two different nodes. So there are five different cases. In each case, there is a different combination of states that the master and the 0, 1 or 2 other participating nodes (call them "slave nodes") have. In each of these cases, a different set of "threads of control," representing the node states runs on the participating nodes. Each thread of control communicates with the threads of control on other nodes using message passing. So code must be written for each of the five cases:

1. i, $i-P$, and $i-Q$ all on the same node (code listed in Figure 3(a))
2. i and $i-P$ on the same node, and $i-Q$ on a different node (code listed in Figure 3(b))
3. i on one node, and $i-P$ and $i-Q$ both on a different node (code listed in Figure 3(c))
4. i and $i-Q$ on the same node, and $i-P$ on a different node (code listed in Figure 3(d)))
5. i, $i-P$, and $i-Q$ all on different nodes (code listed in Figure 3(e))

Notice that the proposed code is only one way of doing things. For example, the programmer could choose to have the master always be on one node that does not contain any entries of the array $a[\cdot]$ and treats all the participating nodes as slaves, and then the code could be written from a different point of view. There are also ways to reduce the number of code pieces by merging master or slave code using "if statements" here and there. Smarter and cleaner message passing code than the one presented is certainly possible.

Nevertheless, one simple fact cannot be changed: with a message-passing implementation that avoids sending the values of array entries, the original code (Figure 2(a)) must be broken down into several code pieces. This results in a significant departure of the implementation from the original algorithm, destroying the algorithmic integrity. The reason why the code must be broken down is that the state of the entire computation is distributed over the various nodes, so the application programmer needs to worry about what states the nodes are in, and how and when to coordinate their state changes.

```
// Master1
(1) $x = f_p(a[i-P])$;
(2) $y = f_q(a[i-Q])$;
(3) $a[i] = f_a(x, y)$;

// Slave1
//  (none)
```

(a)

```
// Master2
(1) $x = f_p(a[i-P])$;
(2) $send(node\_map(i-Q), i-Q)$;
(3) $receive(node\_map(i-Q), y)$;
(4) $a[i] = f_a(x, y)$;

// Slave2
(1) $receive(node\_map(i), i-Q)$;
(2) $y = f_q(a[i-Q])$;
(3) $send(node\_map(i), y)$;
```

(b)

```
// Master3
(1) $send(node\_map(i-P), i-P, i-Q)$;
(2) $receive(node\_map(i-P), x, y)$;
(3) $a[i] = f_a(x, y)$;

// Slave3
(1) $receive(node\_map(i), i-P, i-Q)$;
(2) $x = f_p(a[i-P])$;
(3) $y = f_q(a[i-Q])$;
(4) $send(node\_map(i), x, y)$;
```

(c)

```
// Master4
(1) $send(node\_map(i-P), i-P)$;
(2) $receive(node\_map(i-P), x)$;
(3) $y = f_q(a[i-Q])$;
(4) $a[i] = f_a(x, y)$;

// Slave4
(1) $receive(node\_map(i), i-P)$;
(2) $x = f_p(a[i-P])$;
(3) $send(node\_map(i), x)$;
```

(d)

```
// Master5
(1) $send(node\_map(i-P), i-P)$;
(2) $receive(node\_map(i-P), x)$;
(3) $send(node\_map(i-Q), i-Q)$;
(4) $receive(node\_map(i-Q), y)$;
(5) $a[i] = f_a(x, y)$;

// Slave5.1
(1) $receive(node\_map(i), i-P)$;
(2) $x = f_p(a[i-P])$;
(3) $send(node\_map(i), x)$;

// Slave5.2
(1) $receive(node\_map(i), i-Q)$;
(2) $y = f_q(a[i-Q])$;
(3) $send(node\_map(i), y)$;
```

(e)

Fig. 3. Pseudocode for the message-passing implementation of the simple 1-D array access program shown in Figure 2(a). The various cases depend on which nodes contain the entries $a[i]$, $a[i-P]$, and $a[i-Q]$.

Mobile agents eliminate this problem by providing a layer of abstraction that gives the application programmer a single locus of computation that captures the state of the entire computation. In the next section, we will provide an example illustrating how this new layer of abstraction can help with a numerical application.

4 A Real Application - Crout Factorization for Solving Linear Systems

The paper [5] describes several examples of distributed sequential programming for numerical computing, along with performance data supporting the usefulness of distributed sequential programming. In this section, we describe one of those examples; here the focus is on why, once we have decided to use distributed sequential programming, mobile agents are better than message passing. The particular example chosen here, Crout Factorization [8], is noteworthy because it illustrates how distributed sequential programming can be adapted to situations in which the amount of data is much larger than the combined memories of all the machines in the network. Moreover, it is a real-world example of an algorithm that is run on very large data.

In essence, Crout Factorization is a method of factoring a symmetric positive-definite matrix K into the product of three matrices

$$K = U^T DU,$$

where U is an upper triangular matrix with unit diagonal entries and D is a diagonal matrix. This decomposition can then be used to solve a linear system of equations. The algorithm works in place on the matrix K; when it concludes, the diagonal entries are the entries of D, and the entries above the diagonal are the entries of U. Typically, K is a sparse banded matrix, meaning that entries that are more than a fixed distance b from the diagonal are set to 0. The value b is called the *half-bandwidth* of the matrix. Because K is sparse, it is stored using standard techniques for space-efficient storage of a sparse matrix.

Figure 4(a) shows the pseudocode for Crout factorization [8]. In line (3), the summation over l corresponds to a dot product of two sub-vectors of columns i and j. The computation of the jth column depends on previously computed columns. Because the matrix is banded, the computation of column j only requires portions of the b previous columns, where b is the half-bandwidth of the matrix. The "working set" of matrix entries required to compute column j is shown shaded in Figure 5(a).

When the size of the working set exceeds the size of the main memory on a single workstation, extensive paging overhead occurs. This paging can be eliminated by using the distributed sequential implementation of the algorithm described in [5]. The idea is to split the matrix into pieces, where each piece is a contiguous set of columns. The size of the piece is chosen so that each piece can fit into the main memory of one workstation. The algorithm runs on k workstations, where $k-1$ is the number of pieces that comprise a working set. Figure

```
(1)  For $j = 1 \ldots N$

(2)      For $i = 2 \ldots j-1$

(3)          $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$

(4)      End For

(5)      For $i = 1 \ldots j-1$
(6)          $T \leftarrow K_{ij}$
(7)          $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(8)          $K_{jj} \leftarrow K_{jj} - TK_{ij}$
(9)      End For

(10) End For
```

(a)

```
(1)      For $j = 1 \ldots N$
(1.1)        hop(to column j)
(1.2)        load column j
(2)          For $i = 2 \ldots j-1$
(2.1)            hop(to column i)
(3)              $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$
(3.1)            load $K_{ii}$
(4)          End For

(4.1)        hop(to column j)

(5)          For $i = 1 \ldots j-1$
(6)              $T \leftarrow K_{ij}$
(7)              $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(8)              $K_{jj} \leftarrow K_{jj} - TK_{ij}$
(9)          End For
(9.1)        unload column j
(9.2)        inject I/O Messenger if required
(10)     End For
```

(b)

Fig. 4. Pseudocode for Crout factorization. (a) the sequential implementation. (b) the distributed sequential implementation using MESSENGERS.

5(b) shows an example for which the working set is subdivided into three pieces (i.e., for which $k = 4$). The arrows indicate how an agent, carrying the jth column which it is computing, would move through the pieces of the working set. This is a very efficient use of the network because computing a column requires only three hops, each of which requires moving only the code and one column of the matrix to the machine containing a remote piece of the working set. Moving the entire working set to a single stationary process would require much more data to be transferred.

Figure 6 shows the logical network, again assuming that the working set is decomposed into three pieces. The logical network consists of four nodes. Three of the nodes are used to hold the three working set pieces shown in Figure 5, and a fourth node is used to post-write the computed piece that is no longer used for the factorization and to preload the next piece that will be computed later. All four logical nodes are fully connected, so that an agent can hop to any node from anywhere in one step. Figure 6 shows how we would use the logical network ring as a "running wheel" rotating forward (clockwise) while it processes the matrix pieces in sequence. Figure 6(a) shows the state of the logical network when columns belonging to piece k are being computed. The working set, consisting of pieces k, $k-1$, and $k-2$ of the matrix, is distributed over nodes 1, 2, and 3. While the columns in piece k are being computed, node 4 is

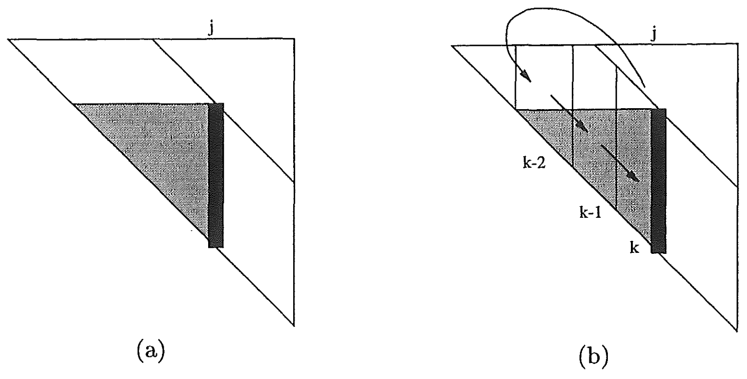

Fig. 5. Working sets in the Crout factorization algorithm: (a) Working Set for the *jth* Column. (b) Decomposition of the working set into three pieces.

preloading piece $k+1$. When the computation progresses to the point where a column in piece $k+1$ is being computed, piece $k-2$ is no longer part of the working set. So now the working set, consisting of pieces $k+1$, k, and $k-1$ of the matrix, is distributed over nodes 4, 1, and 2, as shown in Figure 6(b). Node 3 is now free to write the updated contents of piece $k-2$ that will no longer be used to the disk, and then to preload piece $k+2$.

The distributed sequential implementation of Crout factorization using MESSENGERS is shown in Figure 4(b). The only difference between this code and the sequential code is that three hop statements, three load/unload statements, and one inject statement are added. Although there are a large number of hops, most of them will be local hops. These are essentially no-ops, so their cost will be negligible. The load statements involve copying a single column (at line 1.2) or a single matrix entry (at line 3.1) into MESSENGER variables. Once the new values of column j have been computed, by the Messenger visiting the nodes that contain the pieces of the working set, they are copies back into the appropriate location on the machine storing column j by the unload statement at line 9.1. The inject statement (line 9.2) initiates the post-writing of the piece of the output that will no longer be used and the preloading of the next piece of the matrix K, and is performed when the computation of a piece is completed.

As can be seen by comparing Figures 4(a) and (b), none of the statements in the sequential code are altered when moving from the sequential implementation to the distributed sequential implementation. Some new statements are added, telling the computation when to hop, when to load and unload data from its "briefcase" (i.e., when to copy node data into or out of Messenger variables), and when to start transferring data to and from the disk. These statements are really annotations that coordinate the mapping between the computation and the data, but they do not modify the logic of the existing sequential computation. Hence, the mobile agent implementation using MESSENGERS preserves the integrity of the sequential algorithm.

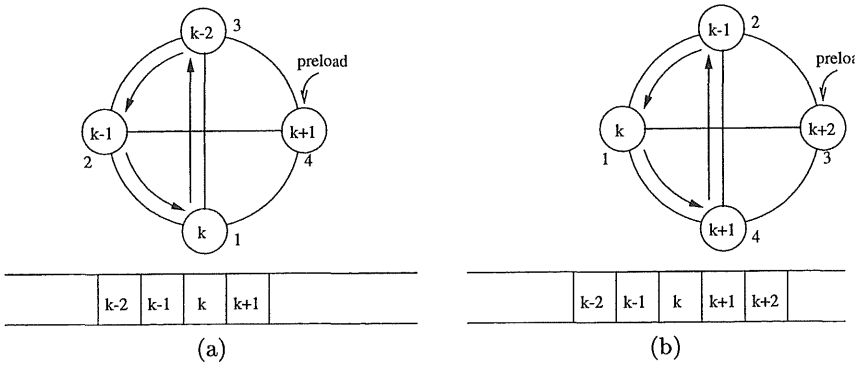

Fig. 6. Logical network for Crout factorization with the working set divided into three pieces. (a) currently computing piece k. (b) currently computing piece k+1.

It is of course possible to mimic our implementation using message passing by assigning each node a "role" or a state, which changes as the application progresses. Figure 6 suggests how we might do this. For example, we can see from Figure 6(a) that at the time when we are processing matrix piece k, node 1 takes the role as master, node 3 is the slave, and node 2 is the slave of node 3, which makes it the slave of the slave (the subslave). At this time, node 4 does not directly take part in the actual computing; rather it is performing the post-writing of piece $k-3$ and the preloading of piece $k+1$. The master's task is to send out a matrix column to its slave, and then to wait until it receives the partially processed column and the diagonal terms from the subslave. When this data is received, the master will compute the column terms that it is responsible for, and do the scaling for the entire column with the diagonal terms. The master repeats this task until all the columns that belong to it are processed, at which time all nodes change state (the master becomes the subslave, the subslave becomes the slave, the slave becomes the preloader, and the preloader becomes the new master). The logic for the slave and the subslave are simpler: all they need to do is to receive a column (from the master and the slave, respectively) compute the corresponding column terms, and pass on the column together with the corresponding diagonal entries (to the subslave and the master, respectively). In addition to the master, the slave, and the subslave, a controller is also necessary to tell the processes when to switch states and to initiate the I/O processing (the post-write and the preload). Figure 7 shows the pseudocode for the controller, master, slave, and subslave.

Notice that the pseudocode provided here is incomplete. It does not describe in detail how to control state changes (i.e. how to stop one thread of control, and start another on a node), nor does it completely specify exactly which column terms $\{K_i\}$ and which diagonal terms $\{K_{ii}\}$ are communicated and computed. Although lines (3) and (4) in Figures 7(c) and (d) look exactly the same, they actually update different set of column terms and send different set of diagonal terms. So they are different code lines if we expand them.

```
// Controller
(1) for $(j = 1;\ j <= num\_pieces;\ j++)$ {
(2)     send(all, "state change");
(3)     send(M, "start", j);
(4)     send(I/O, "I/O");
(5)     receive(M, "piece done");
(6) }
```

(a)

```
// Slave
(1) while(1) {
(2)     receive(M, $\{K_i\}$);
(3)     compute($\{K_i\}$);
(4)     send(SS, $\{K_i\}$, $\{K_{ii}\}$);
(5) }
```

(c)

```
// Master
(1) receive(Cntl, "start", j);
(2) for $(i = I_j;\ i <= I_{j+1} - 1;\ i++)$ {
(3)     send(S, $\{K_i\}$);
(4)     receive(SS, $\{K_i\}$, $\{K_{ii}\}$);
(5)     compute($\{K_i\}$);
(6)     scale($\{K_i\}$);
(7) }
(8) send(Cntl, "piece done");
```

(b)

```
// SubSlave
(1) while(1) {
(2)     receive(S, $\{K_i\}$);
(3)     compute($\{K_i\}$);
(4)     send(M, $\{K_i\}$, $\{K_{ii}\}$);
(5) }
```

(d)

Fig. 7. Pseudocode for Crout factorization implementation using message passing. (a) Controller. (b) Master. (c) Slave. (d) Subslave.

Figure 8 depicts the message-passing patterns among the three computing nodes for each configuration of states, together with the state change pattern for all the nodes. In the figure, M stands for master, S for slave, and SS for subslave. The horizontal arrows represent messages, and the vertical arrows represent state changes.

If we compare the task of producing the agent-based implementation shown in Figure 4(b) with that of producing the message-passing code shown in Figure 7, the differences are considerable. To produce the agent-based code, we need to first size the application (i.e., determine how big the working set is, and from this and the amount of memory available on each workstation compute the number of nodes and the size of each piece). We then have to write the function that decides which columns belong to which piece and which pieces are mapped to which nodes. These two tasks have to be performed as part of any distributed implementation. But once they are done, we have all the ingredients necessary to turn the sequential implementation into the agent-based distributed sequential implementation. In contrast, producing the message-passing version requires carefully analyzing the roles played by the various nodes. This can of course be done, but it is tedious and error-prone. Moreover, the high-level structure of the original algorithm shown in Figure 4(a) has been abstracted out of the code, so

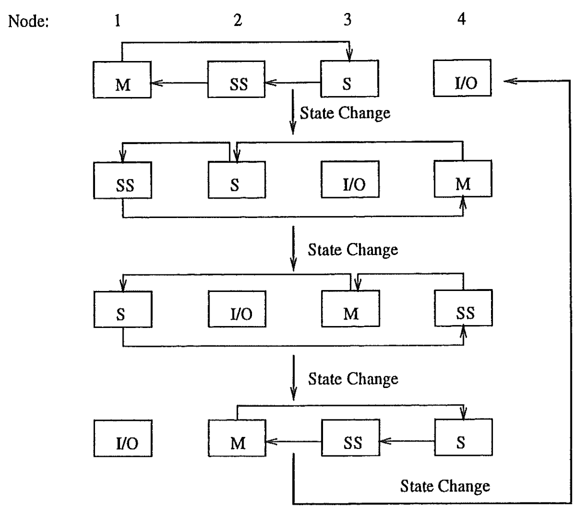

Fig. 8. Node state change pattern for Crout factorization

it would be a very difficult task to reconstruct the original algorithm from the message-passing implementation.

5 Conclusions and Final Remarks

In this paper we have presented two examples of distributed sequential computing. We have seen that the MESSENGERS implementations preserve the algorithmic integrity, as the algorithms remain unchanged. The only difference between the sequential algorithm and the MESSENGERS implementation is the addition of statements to allow the MESSENGERS to navigate through the distributed environment, to load and unload data to and from its state variables, and to inject auxiliary Messengers that preload and post-write data.

Distributed computing requires managing three tasks: data distribution, code distribution, and inter-process coordination (communication and synchronization). Data distribution consists of analyzing the data access pattern, decomposing the data into pieces, distributing the pieces, and managing mapping of data to nodes. These steps must be performed irrespective of whether the application is developed using message passing, mobile agents, or any other distributed computing paradigm.

The mobile agent approach hides the details about how the code is actually distributed. As far as the application programmer is concerned, there is only one copy of the code, which distributes itself dynamically. This is very different from the message-passing approach, in which there are multiple code copies, not all of them necessarily the same. The application programmer must be aware of which code is running on which nodes. Adding new nodes may require new code to be written, as in the case of the Crout factorization example discussed in

Section 4. As the number of node states that the programmer has to explicitly handle increases, the complexity of the programming task increases, and so does the complexity of the higher level controller code necessary for coordinating state changes for all the nodes. The code copies depart significantly from the original algorithm. It would take a very highly skilled code reviewer to put these code pieces back together to obtain the original algorithm. This makes maintenance and further development difficult. On the other hand, since mobile agent code is augmented from the original algorithm by inserting only hop and load/unload statements, it is quite easy to recognize the old algorithm from the mobile code. This is the practical value of algorithmic integrity. It is made possible by the fact that in the mobile agent approach, program states are handled implicitly by the mobile code.

For inter-process coordination, the message passing approach handles both data communication and process synchronization explicitly using the send-receive commands. This places a burden on the application programmer. For example, the programmer needs to know what to expect when programming the receiver. Mobile agent systems remove this burden from the programmer and pass it down to a lower level function. When a mobile agent hops to a different node, the data and program state are both carried by the agent itself. The new node is simply a resource providing CPU cycles and storage space, and the application programmer does not need to worry about any coordination from the view point of the node. At the application level, all coordination and communication are encapsulated in the augmented code (the original code plus the hop, load/unload statements), and the mobile agent system figures out the low level details.

We briefly consider a third alternative to implementing distributed sequential computing, namely Remote Procedure Calls (RPC). Like mobile agents RPC is a higher-level approach to distributed computing, built on top of message passing. The main disadvantage of using RPC for distributed sequential computing is its lack of *navigational autonomy*, the ability to navigate freely through the system. Because RPC consists of a form of procedure calls, implementations using RPC must return to the point from which they were called. This presents a difficulty when the navigational pattern is a cycle like that in the Crout factorization example of Section 4. In terms of the logical network shown in Figure 6(a), the natural pattern is to repeatedly cycle through nodes 1,3,2,1,3,2,1.... This could be simulated in RPC by, for example, having node 1 call node 3, node 3 call node 2, and then returning through node 3 to node 1 and repeating. However, this is an artificial approach, and it adds unnecessary network traffic (the return from node 2 to node 3) to each pass through the cycle. Alternatively, a separate controller could be introduced that mimics the navigation pattern of the mobile agent implementation. But this doubles the network traffic because each hop becomes a remote call followed by a remote return. Moreover, each remote call to a note would have to explicitly take into account the state of the computation as it arrived at the node, so this approach is really just a simulation of message passing by RPC.

We have demonstrated in this paper that distributed sequential applications are a natural fit with mobile agents and a very poor fit with message passing, which is a lower level approach to distributed computing. An obvious analogy is with higher level languages and machine languages. This suggests the following general question: under what circumstances is it possible to automatically generate a message-passing implementation of a distributed sequential algorithm (something like the pseudocode shown in Figure 3) from the mobile agent implementation and appropriate additional information, such as a specification of the data access pattern and the mapping of data to nodes?

References

1. L.F. Bic, M. Fukuda, M.B. Dillencourt: Distributed computing using autonomous objects. Computer, vol.29, no.8, IEEE Comput. Soc, Aug. 1996. 55-61
2. M. Fukuda, L.F. Bic, M.B. Dillencourt: Messages versus messengers in distributed programming. Journal of Parallel and Distributed Computing **57**, (1999) 188-211
3. M. Fukuda, L.F. Bic, M.B. Dillencourt, F. Merchant: Distributed coordination with MESSENGERS. Science of Computer Programming, vol.31, (no.2-3), Elsevier, July 1998. 291-311
4. M. Fukuda, L.F. Bic, M.B. Dillencourt, F. Merchant: MESSENGERS: Distributed Programming Using Mobile Autonomous Objects. Journal of Information Sciences, to appear.
5. Lei Pan, Lubomir F. Bic, Michael B. Dillencourt: Distributed Sequential Numerical Computing Using Mobile Code: Moving Computation to Data. 2001 International Conference on Parallel Processing, Valencia, Spain, Sept. 2001.
6. Kotz, D., Gray, R.S.,: Mobile agents and the future of the Internet. Operating Systems Review, vol.33, (no.3), ACM, July 1999. 7-13
7. MPI: A Message-Passing Interface Standard. The Message Passing Interface Forum (1995). Available online at http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html
8. Thomas, J. R., Hughes: The Finite Element Method, Linear Static and Dynamic Finite Element Analysis. Prentice Hall, 1987
9. Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna: Designing Distributed Applications with Mobile Code Paradigms. http://citeseer.nj.nec.com/carzaniga97designing.html
10. David Chess, Colin Harrison, and Aaron Kershenbaum: Mobile Agents: Are They a Good Idea? IBM Research Report, IBM Research Division, 1995.
11. Goerge Dramamitos and Evangelos P. Marktos: Adaptive and Reliable Paging to Remote Main Memory. Journal of Parallel and Distributed Computing **58**, (1999), 357-388.