International Journal of Parallel Programming, Vol. 32, No. 1, February 2004 (© 2004)

Distributed Parallel Computing Using Navigational Programming

Lei Pan,[1] Ming Kin Lai,[1] Koji Noguchi,[1]
Javid J. Huseynov,[1] Lubomir F. Bic,[1] and
Michael B. Dillencourt[1]

Received April 4, 2003; revised October 1, 2003; accepted October 21, 2003

Message Passing (MP) and Distributed Shared Memory (DSM) are the two most common approaches to distributed parallel computing. MP is difficult to use, whereas DSM is not scalable. Performance scalability and ease of programming can be achieved at the same time by using navigational programming (NavP). This approach combines the advantages of MP and DSM, and it balances convenience and flexibility. Similar to MP, NavP suggests to its programmers the principle of pivot-computes and hence is efficient and scalable. Like DSM, NavP supports incremental parallelization and shared variable programming and is therefore easy to use. The implementation and performance analysis of real-world algorithms, namely parallel Jacobi iteration and parallel Cholesky factorization, presented in this paper supports the claim that the NavP approach is better suited for general-purpose parallel distributed programming than either MP or DSM.

KEY WORDS: Distributed parallel computing; navigational programming; message passing; distributed shared memory; incremental parallelization.

1. INTRODUCTION

The two fundamental approaches to distributed computing, message passing (MP), and distributed shared memory (DSM), each have a drawback: MP is hard to use, and DSM is not scalable. We propose a general-purpose distributed programming approach called *Navigational Programming* (NavP), defined as the programming of self-migrating threads either explicitly with migration statements or implicitly through data distribution.

[1] School of Information & Computer Science, University of California, Irvine, California 92697-3425. E-mail: {pan, mingl, knoguchi, javid, bic, dillenco}@ics.uci.edu

In distributed computing, computations are spread out to multiple processing elements in order to utilize the power of these elements collectively. One way to disseminate computations, as provided by the NavP approach, is to have the programmers insert navigational statements (e.g., *hop*()) into the program text. A sequential program becomes distributed after such insertion, and it now performs *distributed sequential computing* (DSC) with distributed data using a single locus of computation. Among the benefits of such NavP-based DSC are improved performance on large problems resulting from eliminating disk thrashing at a cost of modest network communication, and increased programmability compared to MP.(1, 2) The limitation of this DSC is that it is still sequential computing. To gain more from distributed computing, multiple concurrent DSC threads are orchestrated to perform *distributed parallel computing* (DPC).

The NavP approach combines the advantages of MP and DSM, and balances convenience and flexibility:

1. With MP, distributing data usually means restructuring code. MP parallel programs suffer from code tangling. In contrast, NavP-based DSC code preserves the code structure of the original sequential algorithm, and composing a DPC program from NavP-based DSC threads avoids code tangling. The NavP approach uses shared variable programming as the DSM does, and supports incremental parallelization. As a result, NavP programs are easy to develop and maintain.
2. DSM programs tends to move large data and suffer from false sharing. Synchronizations in a DSM program are usually done with *barrier*s—global operations that are in most cases too strict and expensive. In contrast, similar to MP, the NavP approach follows the principle of **pivot-computes**, which is defined as the principle under which a computation takes place on the node that owns the large-sized data. This node is called the **pivot node**. A NavP program does not move more data than needed, and it only uses local events for synchronizations. As a result, NavP programs are efficient and scalable.

In this paper, we describe the NavP approach to distributed parallel computing, and describe the steps necessary to turn a sequential algorithm into a NavP-based distributed parallel program. We compare the ease of programming and efficiency of our NavP approach with the DSM and MP approaches. This is done by using two real-world algorithms: Jacobi iteration and Cholesky factorization. We present a performance comparison between ours and the MPI(3) implementations. For Cholesky factorization, we also compare the performance with that of a ScaLAPACK(4) implementation. These examples support the claim that the NavP approach is better suited for general purpose distributed parallel programming than either MP or DSM.(5)

The rest of the paper is organized as follows. Section 2 introduces the important concepts and features of the mobile agent system MESSENGERS, which is the underlying system supporting NavP. Section 3 presents the NavP approach to distributed sequential computing and compare it with MP. Section 4 describes the NavP approach to distributed parallel computing and its advantages. Section 5 provides the typical steps of the NavP approach. Section 6 is case studies of two real-world applications, with the comparisons of the approaches listed in Section 7. The last section contains our conclusions.

2. THE MESSENGERS SYSTEM

The MESSENGERS system,(6–9) developed in the School of Information & Computer Science at the University of California, Irvine, is an environment for general-purpose distributed computing. In the MESSENGERS system, applications are developed as collections of self-migrating threads, called Messengers. Like many implementations of mobile agents with *strong mobility*,(10–16) a Messenger can halt its execution, encapsulate the values of its variables, move to another node, restore the state, and continue executing. In MESSENGERS, this sequence of operations is carried out by a *hop*() statement.

There are two types of variables in the MESSENGERS language: agent variables and node variables. An agent variable is private to a particular Messenger and travels with that Messenger as it migrates through the network. A node variable is stationary, and is public and accessible by all Messengers currently on the node to which the variable belongs. Hence agent variables can be used to carry data between nodes, while node variables typically hold large amount of data and can be used for "inter-thread" communication.

A Messenger's programmer tells it to migrate with the navigational statement *hop*(). A destination node's address or a link between the source and the destination nodes can be used as the argument for the *hop*() statement. When a Messenger hops, it takes the data in its agent variables with it to wherever it migrates.

A Messenger can spawn another Messenger using the statement *inject*(). Synchronization among Messengers uses "events." The statements of *signalEvent*() and *waitEvent*() implement the classical operations of process blocking and wake-up. Since no remote data accessing is allowed, the events are only local and so is synchronization. A Messenger's execution is not preempted between any two navigational statements. A Messenger must explicitly relinquish control to other Messengers using statements such as *hop*() or *inject*(). We call this feature "non-preemptive scheduling."

In MESSENGERS, the concept of a mobile agent is used as a programming model. This is in contrast to many Java mobile agent systems, where the emphasis is on actual code mobility. Strong mobility in MESSENGERS system means that computation, but not code, navigates through the network. To do so, a two-phase compilation is used by the MESSENGERS compiler. In the first phase, a MESSENGERS program is broken into smaller C functions at navigational statements. The execution of these functions is "chained" by a logical program counter called "the pointer to next function."(7) This logical program counter and the agent variables are communicated between the nodes using socket-level message passing. The overhead due to the bookkeeping (i.e., a Messenger control block (MCB) carrying a Messenger's internal information(17)) of a Messenger is about 200 bytes. That is, a *hop*() will send about 200 bytes in addition to the actual data, which would also have to be sent in an MP program. In the second phase, the C functions are further compiled into machine native code and loaded on the nodes in the form of dynamic shared libraries for execution. One subtle but important feature of the MESSENGERS system is that it allows code to either be loaded from a shared disk or, in a non-shared file system, to be sent across the network at most once, irrespective of how many times the locus of computation moves across the network.(18)

3. DISTRIBUTED SEQUENTIAL COMPUTING

Distributed sequential computing (DSC) is computing with distributed data using a single locus of computation. The original motivation for introducing DSC was improving performance on large problems by eliminating paging overhead without the need of developing parallel algorithms.(1, 2) Using NavP, sequential programs are easily augmented into scalable DSC programs. A second advantage of DSC is that it serves as a good starting point for incremental parallelization. In distributed parallel computing (DPC), concurrent DSCs are orchestrated to perform one task together.

A simple example reveals why the NavP-based DSC has better programmability and hence is easier to use than message passing, and why it is more efficient and scalable than DSM. The example is a sequential loop over an 1-D array $A[1:n]$, with pseudocode shown in Fig. 1(a). Because of the dependency between $A[i]$ and $A[i-1]$, this loop is hard to

```
(1) for $i = 2 : n$
(2)     $A[i] = A[i-1] + 1$
(3) end
```

(a)

```
(1)   for $i = 2 : n$
(1.1)   $x = A[i-1]$
(1.2)   if $i == (n/2+1)$ $hop(2)$
(2)       $A[i] = x + 1$
(3)   end
```

(c)

```
(0.1)if $\mu == 1$
(1)      for $i = 2 : n/2$
(2)          $A[i] = A[i-1] + 1$
(3)      end
(3.1)    $Send(A[n/2], 2)$
(3.2)elseif $\mu == 2$
(3.3)    $Recv(x, 1)$
(3.4)    $A[1] = x + 1$
(1')     for $i = 2 : n/2$
(2')         $A[i] = A[i-1] + 1$
(3')     end
(3.5)end
```

(b)

Fig. 1. A loop over distributed data. (a) Sequential or DSM. (b) MP. (c) NavP.

parallelize.[19] For simplicity, n is assumed to be even, and the first $n/2$ elements of $A[.]$ are assumed to be on node 1 and the second $n/2$ elements on node 2.

DSM is easy to use to carry out a DSC task; the DSM code is exactly the same as the sequential code shown in Fig. 1(a). Since the algorithm is sequential, a stationary process would run the code on one of the two nodes (e.g., node 1) throughout the computation. Nevertheless, there is a cost to be paid for the ease of use because now the computation is on distributed data. That is, the stationary process would "pull in" half of the array from the remote node to its local node for computation to continue. This is neither efficient nor scalable. The DSM program violates the principle of pivot-computes.

In order to follow the principle of pivot-computes, the locus of computation has to migrate from one node to the other. In the sequential MP code shown in Fig. 1(b), the shift of locus of computation is done using message passing. The loop is broken into two parts, based on where data and computation are, one in the **if** construct corresponding to node 1 (lines (1), (2), and (3)), another in the **else if** construct for node 2 (lines (1′), (2′), and (3′)). With MP, distributing data thus means restructuring code according to execution location. At the array boundary with $A[n/2]$ on node 1 and $A[n/2+1]$ on node 2, explicit communication of $A[n/2]$ and synchronization are needed to transfer the locus of computation (lines (3.1) and (3.3)), even if the program is still sequential. The two restructured code blocks (lines (1)–(3), and lines (3.4)–(3′)) now only compute with local data (received messages are buffered locally).

In the NavP implementation, with pseudocode shown in Fig. 1(c), the loop index i and the temporary variable x are agent variables. The NavP program does not require explicit data transfer and synchronization because the communication of i and x between the nodes is "intra-thread" as the self-migrating thread hops across the array boundary, and the synchronization is subsumed in the flow control. The NavP implementation preserves the original loop structure. The burden of code restructuring according to where the data and computations are is taken away from the application programmers, and passed down to the MESSENGERS compiler. The NavP implementation also follows the principle of pivot-computes because the locus of computation moves with the thread, which dynamically makes it the owner of the large sized data (i.e., $A[.]$) involved in the computation.

The array $A[1:n]$ is a **distributed shared variable** or **DSV**, which is defined as logically one single variable composed of multiple node variables. In this example, the array $A[1:n]$ is logically one single variable constructed using two node variables $A[1:n/2]$ each having half the size of the array. The NavP approach makes shared variable programming possible beyond shared memory.(5) A global view of the array (i.e., global indexing) is preserved by a DSV through a shifted array pointer(5) or a global-local array index map (an example is given in Section 6.2). In contrast, processes in MP use local views of the local data they own (for instance lines (1) and (1′) in Fig. 1(b), and the example shown in Section 6.2). Table I gives a taxonomy of variables in NavP code. In general, whatever can be passed as messages in MP code can be put in agent variables. Also, the data (e.g., a loop index) that is locally scoped to a migrating computation (e.g., a loop that spans over multiple nodes) can be put in agent variables.

Table II lists the problems that arise in distributed sequential programming and compares how the MP and NavP approaches address these problems.

First, in order to follow the principle of pivot-computes, the locus of computation needs to be shifted from node to node. Normally, processes

Table I. A Taxonomy of Variables

	local	distributed
private	(none)	agent variable
public	node variable	DSV

Table II. Problems and Solutions in Distributed Sequential Computing

Requirements	Problems	MP Solution	NavP Solution
1. shift locus of computation	execution cannot continue	restructure code	hop
2. shift locally scoped data	data does not follow	explicit transfer or recompute	carry
3. treat data in global view	index reset to 0	no attempt to solve	global-local map

cannot migrate across machines at arbitrary points in the middle of execution because their program counters would become invalid. The MP solution to this problem is to break the global loop into smaller local loops. Hence partitioning of the data requires restructuring of the code. The NavP approach, on the other hand, enables the application programmer to see the problem from a different view. Self-migrating threads are able to migrate across node boundaries at the application level. Second, when the locus of computation shifts, some evolving data that is scoped locally to the computation (e.g., the loop index) needs to follow. This does not happen automatically. The MP solution is to either explicitly transfer, or locally recompute this data. In the NavP approach, this type of locally scoped data is carried in agent variables. Third, an original sequential algorithm is usually developed without considering any data distribution issue. All data is treated in a global view, i.e., any data can be accessed from anywhere in the algorithm pseudocode with the same index. This may not be true anymore for the algorithm's implementation in a distributed memory environment. For example, the starting index of a distributed array is reset to 0 or 1 across machine boundaries. In the MP implementation, because the code is restructured around execution locations, and the focus of the programmer is placed on local process computing on local data, the global view of data is lost completely. In NavP code, a global view is preserved so that the data accessing looks the same as in the original algorithm. DSVs with array pointer shifts, or global-local array index maps are the mechanisms used to preserve a global indexing, and thread migration is the "bridge"(5) linking the distributed memories.

The transformation from a sequential algorithm to its NavP-based DSC implementation preserves the original code structure and introduces only minor changes. We say that this kind of transformation preserves *algorithmic integrity*.(1, 2) This is why the NavP approach is easier than MP. A NavP program also follows the principle of pivot-computes, which is why it is efficient and scalable.

4. DISTRIBUTED PARALLEL COMPUTING USING NAVP

The simple example shown in Fig. 1 has no concurrency in the computation. Yet it is still interesting and useful to study the programming of sequential algorithms in a distributed environment. This is because DSC not only can improve the performance of large sequential problems by eliminating disk paging with only minor re-programming efforts,(2) but also help distributed parallel computing (DPC).

A parallel programming model can be based on *data parallelism* or *task parallelism.*(20) In data parallelism, identical computations are simultaneously applied to different data elements of the entire problem, while in task parallelism concurrent executions of different computations are applied to the same or different data elements. In either of these classifications, sequential computations are fundamental building blocks. These sequential computations can be running concurrently, or they can be waiting for one sequential task to finish before they can continue. There will be at least one such sequential task that computes on distributed data, which makes it DSC, unless the computations can be divided into a number of completely independent sub-tasks with no communication and synchronization among them. Algorithms that can be perfectly parallelized are called "embarrassingly parallel," but they are rare, easy to deal with, and hence of little interest in this work. In fact, most parallel algorithms have inherently sequential portions that become bottlenecks for speedup; this phenomenon is the basis of Amdahl's law and its variants.

The NavP approach to DPC, as it is today, belongs to *explicit parallel programming* in which a programmer is required to code an explicitly parallel algorithm.(21) There are two major issues in parallel programming: *data locality* and *concurrency.*(21) The performance of a parallel program depends on both. As observed in our previous work,(2) the NavP approach exploits data locality well while preserving algorithmic integrity for distributed sequential computing. If a computation is subdivided into DSC sub-tasks with some of them running concurrently in certain periods of time during the entire execution, ease of programming and good data locality can then be achieved through the programming and composing of these NavP-based DSC sub-tasks. The fundamental idea behind the NavP-based distributed parallel computing is to decompose the computation into NavP-based DSC threads and orchestrate them together to solve one problem.

The NavP-based DPC differs from the DSM and MP approaches in several important aspects. With DSM, the problem is decomposed into DSCs, and each of these DSC sub-programs preserves algorithmic integrity. Nevertheless, as demonstrated by the simple example shown in

Fig. 1(a), each of these DSCs may be less efficient and scalable as the principle of pivot-computes may be violated and hence data locality is not exploited fully. Also, concurrent DSCs running on page-based DSMs may be accessing data on the same page resulting in false sharing and hence poor performance. Furthermore, since remote data accessing is provided, synchronization in DSM programs are usually done using a *barrier*—a global operation that is too strict and expensive. The MP approach is more efficient and scalable, but it has rather poor programmability. In MP, processes run in parallel on their home nodes and communicate with one another by sending and receiving messages. The processes do not have a common memory address space. This forces problem decomposition to be centered around executing locations (i.e., nodes). In particular, with MP, computations are decomposed into strictly sequential computing sub-tasks (i.e., non-distributed) assigned to execution locations using **if** or **else if** constructs. As a result, code is restructured for distributed sequential computing. A simple example is shown in Fig. 1(b). Furthermore, for parallel computing the code corresponding to different sub-tasks is tangled within one **if** or **else if** block, each corresponding to a node location where the code block is executed, "polluting" each other. An example of this code tangling is presented in Fig. 7 in Section 6.2. This code restructuring and code tangling makes MP programming significantly harder than DSM or the NavP-based DPC.

Using NavP-based DSCs to compose DPC programs has the following advantages. First, as we observed before,(1, 2) each NavP-based DSC program preserves algorithmic integrity and exploits data locality well. Second, NavP code encapsulates the state and behavior of each sequential sub-task, and the coordination among the threads is done using injections or events. Thus the code tangling that occurs in MP—when portions of the code belonging to different tasks must be intermingled because they are executed on the same node—is avoided. These two advantages make the NavP-based DPC easy to use. Third, similar to DSM, the NavP-based DPC uses shared variable programming,(5) but since no remote data accessing is allowed, no global synchronization is needed. Synchronizations are through local events that are perceivable by self-migrating threads running on the same local node. This local synchronization and the fact that each DSC sub-computation exploits data locality well are the reasons why NavP-based DPC is efficient and scalable. We will use real-world applications and analysis to demonstrate these advantages in Section 6.

5. THE STEPS OF THE NAVP APPROACH

Starting from a sequential algorithm, there are three steps in developing a NavP-based DPC program.

1. *data distribution:* A programmer first maps the data to different machines. In doing so, two rules should be followed. The first is that the programmer needs to make sure the data distribution and the algorithm are such that the computation is at a coarse granularity level. This is an important factor for good performance. Fortunately, many algorithms exhibit some degree of locality of access and are coarse grained. The second is that the size of the data set on each machine should not exceed the main memory of the machine in order to avoid heavy disk paging.[2] These two rules are not unique to NavP. Any programming approach would require such data mapping.
2. *DSC:* In this intermediate step, the programmer augments the sequential algorithm with *hop*() and load/unload statements (to load data to/from agent variables) to make a DSC program. It is important to follow the principle of pivot-computes in the sub-computations of the program. A sub-computation can be any of the basic programming constructs, such as a loop.
3. *parallelization:* In this step, the programmer transforms a DSC program into a DPC program. There are two sub-steps here, the order of which is not important. *Step 3.1 Parallelization*: In this sub-step, the programmer determines whether any dependencies exist across *hop*() statements. Note that with the NavP approach this is considerably simpler than general data dependency analyses, because all that is needed is to check whether the computation at the time of the hop carries any intermediate results from the previous node to the next node. If not, the computations on the two different nodes are independent and the code may be split at the point of the *hop*() statement. The *hop*() statement is removed and two computations become two concurrent DSCs. An *inject*() or a *clone*() statement is needed to spawn a separate thread for each node. The Cholesky factorization shown in Section 6.2 is an example of such parallelization. *Step 3.2 Pipelining*: In this sub-step, the programmer looks for opportunities for pipelining. In some cases, the computations across a *hop*() statement depend on

[2] To be precise, the "working set" on each node should not exceed the main memory. In this paper, we use "data set" for simplicity.

each other, but the DSC can be pipelined in the network of nodes linked with *hop*() statements. The programmer can then slice the sequential computation into a sequence of smaller computations and assign them to different threads. Each thread hops to the next node as soon as it finishes its computation on the current node, and all the threads build a pipeline. Further improvement may be possible when the computations assigned to different threads can shift phase to achieve complete parallelism without changing the result of the entire computation. The Jacobi iteration shown in Section 6.1 provides an example of pipelining and phase shift. The above two sub-steps can be iterated as needed. *Step 3* can be applied repeatedly to achieve incremental parallelization.

The above three steps may need to be reiterated one or more times. This is because the data distribution scheme developed in *Step 1* emphasizes only data accessing locality for DSC, and in *Step 3* this distribution may need to be adjusted for load balancing, without losing the locality. We do this based on the observation that exploiting data locality has greater benefits over load balancing, and hence should take the priority.(22) Our approach provides incremental parallelization with *Step 2* as a starting point and with a repeated use of *Step 3*. A programmer can choose to use the DSC program as a result of *Step 2* if parallelization is impossible or impractical, or incrementally parallelize it as new opportunities are found during the repeated iterations of the steps. In contrast, MP is not amenable to incremental parallelization.(20) In the three steps, the two concerns in distributed parallel programming, namely data locality and concurrency, are separated in each iteration.

The above steps may be reordered or combined by some experienced programmers. As long as the steps are repeated and feedback is provided to each step in future iterations, it does not particularly matter which step we choose as the starting point.

6. CASE STUDIES

In this section, we present two examples, namely parallel Jacobi iteration and parallel Cholesky factorization. We chose Jacobi iteration because it lends itself naturally to implementations using both MP and NavP. In fact, because the code structures of these two implementations are so similar, we only provide the NavP pseudocode. In contrast, Cholesky factorization is an application in which parallel steps alternate with steps that are inherently sequential. As a result, the NavP and MP implementations have very different structures. Hence both pseudocodes are presented.

For both examples we present performance data for the MPI and MESSENGERS implementations. For Cholesky factorization, performance of ScaLAPACK(4) is also presented for comparison. A detailed comparison of programmability and scalability of the different implementations will be presented in Section 7. All performance data were obtained from SUN Ultra 60's with 256MB of main memory, 1GB of virtual memory, and 100Mbps of Ethernet connection. These workstations run Solaris 8, and have a shared file system (NFS). The MPI system we used for performance comparison was LAM 6.5.9 from Indiana University.(23, 24) The mobile agent system used was MESSENGERS.(9) The C compiler used was `gcc 3.2.2`, the Fortran compiler (for ScaLAPACK) used was `g77 3.2.2`. All matrices in the codes were single-precision real. This corresponds to the data types "float" in C and "real*4" in Fortran, respectively.

6.1. Parallel Jacobi Iteration

Jacobi iteration(25) is an iterative solution scheme used in solving systems of linear equations. The basic steps of Jacobi iteration are listed below.

Let

$$Au = f \tag{1}$$

be a system of linear equations, where A is an $N \times N$ matrix, u and f are vectors of size N. Matrix A can be decomposed into

$$A = D - L - U, \tag{2}$$

where D is the diagonal of A, and $-L$ and $-U$ are the strictly lower and upper triangular parts of A. Equation (1) can then be re-written as:

$$Du = (L + U)\, u + f, \tag{3}$$

or

$$u = D^{-1}(L + U)\, u + D^{-1} f. \tag{4}$$

Note that since D is a diagonal matrix, computing the inverse of D is trivial.

We define the Jacobi iteration matrix by

$$P = D^{-1}(L + U), \tag{5}$$

where a generic element of P, p_{st}, is given, with respect to an element of A, a_{st}, by

$$p_{st} = \begin{cases} 0 & \text{if} \quad s = t; \\ -\dfrac{a_{st}}{a_{ss}} & \text{otherwise.} \end{cases} \tag{6}$$

We can introduce iteration in Eq. (4), and express the Jacobi iterative method as:

$$u^{n+1} \leftarrow Pu^n + D^{-1} f, \tag{7}$$

where u^n is the solution vector at iteration step n, and $u^0 = \{0\}$. Equation (7) reveals that parallelizing Jacobi iteration is essentially parallelizing matrix-vector multiplications at each iteration step.

If we partition the matrix P and the vectors into blocks and subvectors, respectively, we can re-write Eq. (7) in block fashion as:

$$u_i^{n+1} \leftarrow \sum_{j=0}^{p-1} P_{ij} u_j^n + D_{ii}^{-1} f_i, \tag{8}$$

for $i = 0 : p-1$, where p is the number of segments into which the solution vector u is sub-divided. Matrices P and D are sub-divided into p^2 pieces. u_i^{n+1} and u_j^n are sub-vectors, and P_{ij} and D_{ii} are sub-matrices.

Figure 2(a) lists the pseudocode for parallel Jacobi iteration using DSM. Each process executes this code. In the pseudocode, μ is the ID of the current process and $\mu = 0 : p-1$. The **while** loop (lines (3)–(14)) runs until the global error *err* satisfies a user defined tolerance *TOL*. The **for** loop (lines (5)–(9)) and line (10) compute u_μ—the sub-solution vector for the next iteration (see Eq. (7)). Each process computes a "local" error $r[\mu]$ corresponding to the sub-solution it is responsible for at line (11). A "global" error *err* is computed, for the previous iteration, at line (6). A synchronization is done by the *barrier* statement at the end of each iteration at line (13).

For the MESSENGERS implementation, we decompose the square matrix A vertically into p "slices," where p is the number of participating nodes. We pick p to be such that the matrix slices fit into their corresponding local memories completely so that no disk paging will occur anywhere at any time. The vertical slice on each participating node is further decomposed into p sub-matrices, which become the basic matrix blocks of the block-fashion Jacobi iteration. The vectors holding solutions for the nth and $(n+1)$st steps and right-hand-side vector f are also subdivided into p subvectors.

```
(1)    $r[\mu] = 1.$



(2)    $j = \mu; err = 1.; v_\mu = \{0.\}$
(3)    while $err > TOL$
(4)        $u_\mu = \{0.\}; err = 0.$
(5)        for $c = 0 : p-1$
(6)            $err += r[c]$
(7)            $u_\mu += P_{\mu j} v_j$
(8)            $j = (j+1)\%p$

(9)        end
(10)       $u_\mu += D^{-1}_{\mu\mu} f_\mu$
(11)       $r[\mu] = \|u_\mu - v_\mu\|_2$
(12)       $v_\mu = u_\mu$
(13)       barrier
(14)   end
```

(a)

```
(0.1)   for $\mu = 0 : p-1$
(0.2)       hop(node_map($\mu$))
(1)         $r = 1.$
(1.1)       inject(Jacobi($\mu$))
(1.2)   end

(1.3)   Jacobi(int $\mu$)
(2)         $j = \mu; err = 1.; v_\mu = \{0.\}$
(3)         while $err > TOL$
(4)             $u_\mu = \{0.\}; err = 0.$
(5)             for $c = 0 : p-1$
(6)                 $err += r$
(7)                 $u_\mu += P_{\mu j} v_j$
(8)                 $j = (j+1)\%p$
(8.1)               hop(node_map($j$))
(9)             end
(10)            $u_\mu += D^{-1}_{\mu\mu} f_\mu$
(11)            $r = \|u_\mu - v_\mu\|_2$
(12)            $v_\mu = u_\mu$
(13)
(14)        end
(14.1)  end
```

(b)

Fig. 2. Pseudocode for parallel Jacobi iteration. (a) DSM. (b) NavP.

Figure 3 shows an example with $p = 3$. Each large square box in dashed lines represents the memory of a node, while each small box in solid lines represents a thread. The subvector u_i^{n+1}, $i = 0 : p-1$ inside a small box is stored in an agent variable, and is carried by the thread to wherever it migrates. The three node variables representing matrices A_{ij}, $i, j = 0 : p-1$ together make a distributed shared variable, or DSV. Similarly, u_i^n, f_i, $i = 0 : p-1$ are each a DSV respectively. The arrows depict how the three self-migrating threads will hop following one another in the ring linking all the nodes. We note that in conventional MP implementations of Jacobi iteration, the matrix is partitioned and distributed in horizontal slices, rather than vertical slices. Changing our code to use horizontal slices

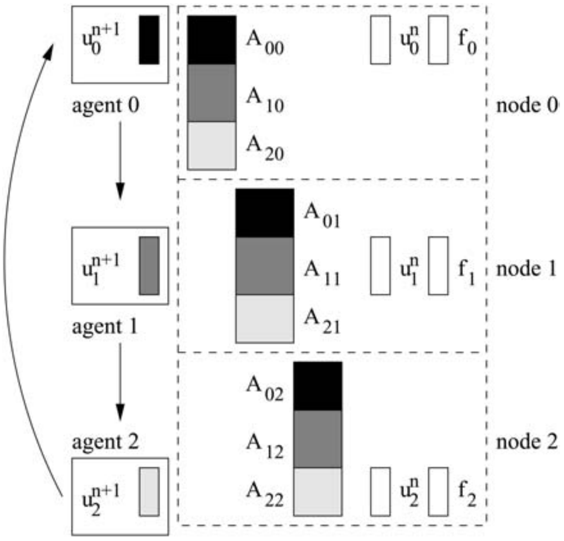


Fig. 3. Memory use in NavP parallel Jacobi iteration.

would require only a minor modification, namely having the self-migrating threads carry the u_i^n's and storing the u_i^{n+1}'s as node variables.

To compute the summation in Eq. (8), u_i^{n+1} and $P_{ij}, u_j^n, j = 0 : p-1$, need to be together on the same node for each value of j. Since only A_{ij}'s are stored (as shown in Fig. 3), and P_{ij}'s are implicitly computed from A_{ij}'s using Eq. (6), the above requirement is equivalent to having u_i^{n+1} and $A_{ij}, u_j^n, j = 0 : p-1$ together for each value of j. Notice that since in terms of data storage and access pattern P_{ij} and A_{ij} are the same, in the following we may inter-change the use of P_{ij} and A_{ij} (i.e., to describe the algorithm correctly we use P_{ij} (Eq. (8)), but to discuss data accessing pattern we may use A_{ij} (Fig. 3)). The efficient way is to have the large data A_{ij}'s stay where they are on the pivot node, and let small data pieces u_i^{n+1}'s go "meet" with them for computation. This follows the principle of pivot-computes. Figure 3 shows that subvectors and matrix blocks marked with same greyscales must be together at some stage during the computation of the summation, and this is done with self-migrating threads carrying u_i^{n+1} and hopping among participating nodes.

Figure 2(b) lists the pseudocode of block-fashion NavP-based parallel Jacobi iteration. Lines (1.3) to (14.1) are code for a thread named *Jacobi*. Between lines (2)–(14) this code is not much different from the corresponding DSM code shown in Fig. 2(a); the only change is the insertion of one *hop*() statement. This is an advantage referred to as algorithmic integrity.[1, 2] The *node_map*() shown at lines (0.2) and (8.1) is a matrix-piece-to-node map. Self-migrating threads are able to access shared but remote data pieces (e.g., A_{ij}'s) they need with migrations. To self-migrating threads, any distributed shared variable is just "hops away." When threads migrate, they carry with them their locally scoped data. In this example, the locally scoped data includes the sub-vector u_μ that is being computed, the loop counters j and c, and the global error *err* that is being collected from all node variables r. Notice that the **for** loop at lines (0.1)–(1.2) in Fig. 2(b) is not superfluous: the DSM code is written in an SPMD (single program multiple data) style and is executed on all nodes, while the NavP approach needs a dispatcher to inject concurrent threads on appropriate nodes, and orchestrate the execution of the parallel computation.

In the NavP-based parallel Jacobi program, each thread is doing distributed sequential computing[(1)] with one locus of computation at any time. With proper synchronization, concurrent threads can be organized to collectively work around large data pieces to further achieve parallelism. In this example, no explicit synchronization is needed because the threads are non-preemptive (discussed in Section 2), and no one can pass another to break the sequence of shared variable accessing. Fig. 4(a) depicts how three self-migrating threads each representing a DSC coordinate with each other in time and space to achieve parallel computing.

Table III and Fig. 5 show the performance comparison between NavP and MPI implementations with various matrix sizes on different numbers of workstations. Sequential elapsed times were generated by running a C

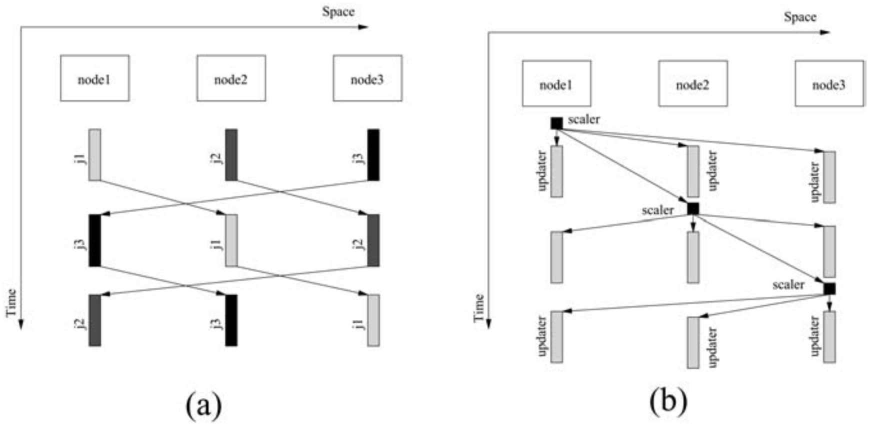


Fig. 4. Concurrent threads in distributed environment. (a) Jacobi. (b) Cholesky.

Table III. Performance of Parallel Jacobi Iteration

Matrix Size	8000		12000		16000		20000	
Number of Workstations	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
				Sequential				
1	29.71	1.00	66.81	1.00	118.88	1.00	185.50	1.00
				MESSENGERS				
2	16.91	1.76	-	-	-	-	-	-
4	8.18	3.63	18.71	3.57	-	-	-	-
6	5.56	5.34	12.43	5.38	-	-	-	-
8	4.42	6.72	9.36	7.14	16.53	7.19	-	-
10	3.42	8.70	8.12	8.23	13.36	8.90	-	-
12	3.11	9.56	6.63	10.08	11.56	10.28	17.19	10.79
				MPI				
2	16.51	1.80	-	-	-	-	-	-
4	8.24	3.60	18.39	3.63	-	-	-	-
6	5.44	5.46	11.94	5.59	-	-	-	-
8	4.45	6.68	9.23	7.24	16.22	7.33	-	-
10	3.34	8.90	7.24	9.22	13.19	9.01	-	-
12	2.79	10.66	6.65	10.05	11.54	10.30	17.17	10.80

program that implements a block-fashion of Jacobi iteration. The number of blocks was chosen to be 16. This gives a better performance than that of a non-block-fashion sequential implementation. The scaled speedup of the NavP program is almost the same as that of the MP program, and their trends as the number of machines increases are the same which indicates same scalability.

The parallel Jacobi program is "homogeneous" in that on all participating nodes, exactly the same computation, e.g., matrix-vector multiplication, is executed at any time. For this type of parallel programs, message-passing programming is able to do almost as nicely as NavP programming is, because all MP processes run the same code and no code structure is broken. In Section 6.2, we will present a more complicated algorithm in which parallel steps are interleaved with sequential ones.

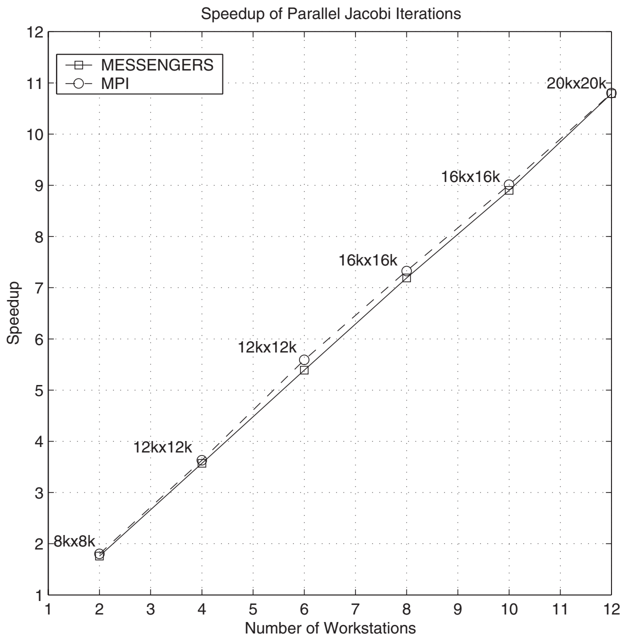


Fig. 5. Performance of parallel Jacobi iteration.

6.2. Parallel Cholesky Factorization

Cholesky factorization is an algorithm for factorizing symmetric positive definite matrices. In this subsection, we briefly describe this algorithm, and then present three implementations, using the DSM, NavP, and MP approaches. The MP and DSM implementations of Cholesky factorization are based on those in a classic book on matrix computations.[26]

A positive definite matrix A can be factored into the product of two matrices

$$A = GG^T, \tag{9}$$

where G is a lower triangular matrix called the Cholesky triangle. This decomposition can then be used for different purposes, such as to solve a linear system of equations of $Ax = b$. The Cholesky factorization algorithm takes A as its input and produces the matrix G. It works in place on the matrix A; when it concludes, the entries on and below the diagonal are the entries of G. For simplicity we will assume here that A is a full $n \times n$ matrix.

Depending on the order used to update the matrix A, there are two different sequential implementations, namely, inner and outer product

```
(1)  for $k = 1 : n$
(2)    if $\mu == 1$
(3)      $v_{loc}(k:n) = A(k:n,k)$
(4)      $v_{loc}(k:n) /= \sqrt{v_{loc}(k)}$
(5)      $A(k:n,k) = v_{loc}(k:n)$
(6)    end
(7)    $barrier$

(8)    $updating(\mu,k,n)$

(9)    $barrier$


(10) end

(11) $updating(int\ \mu, int\ k, int\ n)$
(12)   $v_{loc}(k+1:n) = A(k+1:n,k)$


(13)   for $j = k+\mu : p : n$
(14)     $w_{loc}(j:n) = A(j:n,j)$
(15)     $w_{loc}(j:n) -= v_{loc}(j)v_{loc}(j:n)$
(16)     $A(j:n,j) = w_{loc}(j:n)$
(17)   end

(18) end
```

(a)

```
(1)      for $k = 1 : n$
(2)
(3)
(4)        $A(k:n,col(k)) /= \sqrt{A(k,col(k))}$
(5)
(6)
(7)
(7.1)      for $\mu = 1 : p$
(8)          $inject(updating(\mu,k,n))$
(8.1)      end
(9)
(9.1)      $hop(node\_map(k+1))$
(9.2)      $waitEvent(Evt,\ k+1)$
(10)     end
(11)     $updating(int\ \mu, int\ k, int\ n)$
(12)       $v_{loc}(k+1:n) = A(k+1:n,col(k))$
(12.1)     $hop(node\_map(k+\mu))$
(12.2)     $waitEvent(Evt,\ k)$
(13)       for $j = k+\mu : p : n$
(14)
(15)         $A(j:n,col(j)) -= v_{loc}(j)v_{loc}(j:n)$
(16)
(17)       end
(17.1)     $signalEvent(Evt,\ k+1)$
(18)     end
```

(b)

Fig. 6. Pseudocode for parallel Cholesky factorization. (a) DSM. (b) NavP.

versions. Figure 6(a) contains pseudocode for a parallel implementation of outer product Cholesky factorization in shared memory or DSM, adapted from the code given in the book.[26] Notice that the book assumes that the DSM system used requires that data in the shared memory be copied to local variables before any computations can be applied to it.

In the algorithm shown in Fig. 6(a), there are two types of computations performed on the columns of the matrix A:

1. *scaling:* A column is scaled using its diagonal term. This happens in each iteration over the matrix columns, and has a time complexity of $\Theta(n)$ for each iteration. We can gain little if we parallelize the scaling at each iteration, but we are unable to parallelize the scaling as an entire task because the scaling iteration steps are sequential.

   The columns that have been scaled are called **G columns**. These columns will no longer be modified but will be used in later computation. Scaling processes all columns sequentially from left to right, i.e., a column is ready to be scaled only after all the columns to its left have been scaled and therefore turned into G columns, and after itself is updated using the information from all these G columns;

2. *updating:* A column is updated using the values in all the G columns to its left. This is the expensive part of the algorithm: the work done in each iteration of k is $\Theta(n^2)$. This portion of the algorithm is parallelized.

The DSM implementation assumes that there are p nodes, each executing the pseudocode shown in Fig. 6(a). The index k loops over all the columns of A (line (1)). The first node with node ID $\mu == 1$, is responsible for scaling column k (lines (2)–(6)), after which all the nodes, including node 1, will update in parallel the columns they are responsible for and that are to the right of column k (line (8)). In particular, node μ will update the columns $(k+\mu):p:n$ (lines (13)–(17)); that is, the columns starting at $k+\mu$, ending at up to n, with an increment of p. After all the nodes are done updating, node 1 can start scaling on the next column $(k+1)$, and the computation thus continues. Again it is assumed that computations cannot be done directly to shared variables,(26) so lines (3), (5), (12), (14), and (16) are used to copy data using and from the shared variable A. Not all DSMs have this limitation, in which case these extra lines could be removed from the pseudocode. We include these lines to emphasize that the actions they represent do take place when the terms of matrix A accessed resides in remote memory.

In order to balance load, a scheme for computation distribution is proposed in the book.(26) The basic idea is to update the columns in a round-robin fashion. This is illustrated by the following simple example. Suppose the number of columns in A is $n = 11$, the number of nodes is $p = 3$, and the ith column of A is denoted by a_i. In the most straightforward way, the updating of columns would be assigned to nodes as follows:

$$\begin{array}{c|c|c} [a_1\ a_2\ a_3\ a_4 & a_5\ a_6\ a_7\ a_8 & a_9\ a_{10}\ a_{11}]. \\ \text{node 1} & \text{node 2} & \text{node 3} \end{array}$$

In the round-robin scheme, on the other hand, the assignment of column computations to nodes is as follows:

$$\begin{array}{c|c|c} [a_1\ a_4\ a_7\ a_{10} & a_2\ a_5\ a_8\ a_{11} & a_3\ a_6\ a_9]. \\ \text{node 1} & \text{node 2} & \text{node 3} \end{array}$$

In the contiguous allocation scheme, node 1 would be idle after columns 1 to 4 have been computed, even though much work remains. In the round-robin scheme, node μ carries out the construction of $G(:, \mu : p : n)$ where the column index starts from μ, ends at up to n, with an increment of p. This scheme distributes the computation of matrix A evenly to all participating nodes, and ensures that all of the nodes are busy most of the time and they finish at about the same time.

The DSM pseudocode Fig. 6(a) describes the algorithm of Cholesky factorization using the round-robin computation distribution scheme, but without specifying how data is distributed. With unstructured DSM where programmers have no control over data distribution, the DSM generated data distribution is a well-known reason for inefficiency. We will return to this issue in Section 7.1.

In the NavP implementation, data distribution follows the round-robin fashion. A mechanism similar to what's provided by the qualifier *shared* in HPF or UPC can be used by a NavP programmer to define the data distribution pattern. The NavP program is written in MESSENGERS, and the pseudocode is shown in Fig. 6(b). There are two types of threads: a single scaling thread named *Scaler* (with code lines (1)–(10)), and multiple updating threads named *Updaters* (with code lines (11)–(18)). *Scaler* carries the loop index k, an agent variable, that loops through all columns of matrix A—a distributed shared variable. On the kth iteration, *Scaler* scales column k (line (4)). The function $col(k)$ maps the global column index k to a local column index; this function is needed because each node stores only a portion of the entire global matrix A. After scaling the column, *Scaler* injects p *Updaters* (lines (7.1)–(8.1)), and then it hops to the node that owns the next column of A (line (9.1)). The ID of this node is found using a column-to-node map function *node_map*(). *Scaler* then waits for the next round of computation. Each of the p *Updaters* loads the newly computed G column k (again the local column index is $col(k)$) into its agent variables (line (12)), and then hops to the appropriate node (line (12.1)). In parallel, these p threads update the A columns for which they are responsible on all p nodes, using the G column stored in their agent variables and the matrix entries stored in the distributed shared variable A (line (15)). Two maps are used in the NavP code (lines (4), (9.1), (12), (12.1), and (15)) and they are application dependent. In particular, here the column-to-node map is $node\_map(k) = (k-1)\% p+1$, and the global-to-local-column-index map is $col(k) = (k-\mu)/p+1$, where k is global column index, p is number of nodes, and μ is current node ID. Because matrix columns are not assigned to nodes using a linear map, local memory accessing cannot be done with shifted pointers,[5] but the maps we use here are simply by-products of a user defined data distribution scheme.

Scaling is performed sequentially by a single thread, while updating is done in parallel by p concurrent threads. Figure 4(b) depicts how the two types of threads coordinate to achieve interleaved sequential and parallel computing.

In Fig. 6(b), *signalEvent*$(Evt, k+1)$ at line (17.1) signals the $(k+1)$st event of the variable Evt, and *waitEvent*$(Evt, k+1)$ and *waitEvent*(Evt, k) at lines (9.2) and (12.2) wait till the $(k+1)$st or the kth events or higher is signaled, respectively. After *Scaler* executes the *inject*() command at line (8), it hops away immediately, and then the injected threads start executing (line (12)). Thus *Scaler* hops to the next node and continues its computation without having to wait for the injected *Updaters* to hop away. The *waitEvent*() and *signalEvent*() primitives (lines (9.2), (12.2), and (17.1), respectively) are used to protect the distributed shared variable A from being updated in an incorrect order. In particular, *waitEvent*() at line (9.2) makes *Scaler* wait until the *Updater* working on the same node finishes updating the distributed shared variable A and signals an event Evt at line (17.1). *waitEvent*() at line (12.2) makes sure that the *Updaters* from earlier iterations have all finished, so the current *Updater* can start doing its work.

The NavP pseudocode (Fig. 6(b)) preserves algorithmic integrity with respect to the DSM original (Fig. 6(a)), and also with respect to the sequential original.(26) The synchronization events (lines (9.2), (12.2), and (17.1) in Fig. 6(b)) do the same job as the synchronization barriers do (lines (7) and (9) in Fig. 6(a)). In addition to the event-related lines, two hops (lines (9.1) and (12.1)) and one load statement (line (12)) are inserted to tell the threads where to migrate and what to carry for later sharing and computing. Two maps, namely *node_map*() and *col*( ), are used to tell the code about how data is distributed. Notice that the **for** loop at lines (7.1)–(8.1) in Fig. 6(b) is not superfluous: the DSM code is written in an SPMD (single program multiple data) style and is executed on all nodes, while the NavP approach represents a single thread that orders the computational steps in its natural sequence, executes all the sequential portions of the computation on the appropriate node, and orchestrates the execution of the parallel portion of the computation by injecting multiple self-migrating threads that hop to the appropriate nodes and perform their work independently.

Pseudocode for an MP solution of Cholesky factorization, adapted from the implementation presented in the book,(26) is presented in Fig. 7. Each process executing this code runs a **while** loop (line (2)), with loop index q, over all local columns this node owns. A global column index k, which is the same as the loop index k in Fig. 6(a) and (b), is being computed by all processes (lines (8) and (25)). The local column index q is mapped to its corresponding global position in the matrix *A*, and is then tested against the global index *k* (line (3)). If the test result in line (3) is true, the process owns the column that needs to be scaled. Therefore, it scales the column to get a new G column (line (4)), and passes the new G column to its right neighbor in the node ring (line (6)), before it uses the new G column to update the local *A* columns (line (11)). If the test result in line (3) is false, this process will receive the new G column from its left neighbor (line (15)), forward it to its right neighbor if needed (line (19)), and then update its local *A* columns (line (23)).

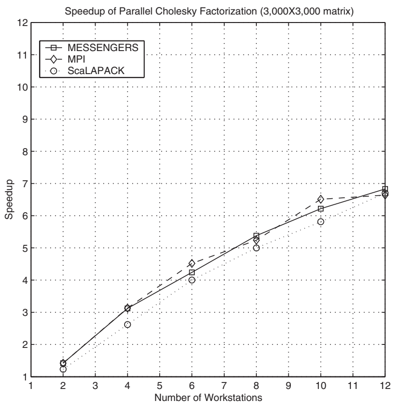
```
 (1) $k = 1; q = 1; col = \mu : p : n; L = length(col)$
 (2) while $q <= L$
 (3)   if $k == col(q)$
 (4)     $A_{loc}(k : n, q) /= \sqrt{A_{loc}(k, q)}$
 (5)     if $k < n$
 (6)       $Send(A_{loc}(k : n, q),\ right)$
 (7)     end
 (8)     $k = k + 1$
 (9)     for $i = q + 1 : L$
(10)       $r = col(i)$
(11)       $A_{loc}(r : n, i) -= A_{loc}(r, q) A_{loc}(r : n, q)$
(12)     end
(13)     $q = q + 1$
(14)   else
(15)     $Recv(g_{loc}(k : n),\ left)$
(16)     $\alpha = proc\ which\ sent\ k^{th}\ G\ col$
(17)     $\beta = index\ of\ right's\ final\ col$
(18)     if $right \neq \alpha\ and\ k < \beta$
(19)       $Send(g_{loc}(k : n),\ right)$
(20)     end
(21)     for $i = q : L$
(22)       $r = col(i)$
(23)       $A_{loc}(r : n, i) -= g_{loc}(r) g_{loc}(r : n)$
(24)     end
(25)     $k = k + 1$
(26)   end
(27) end
```

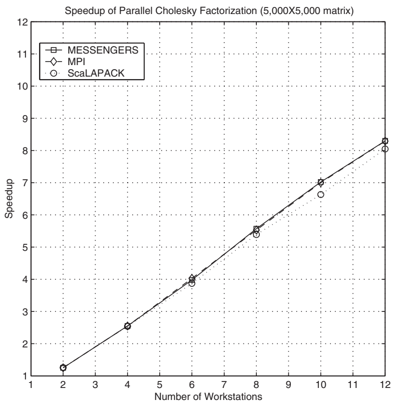
Fig. 7. Pseudocode for parallel Cholesky factorization in MP.

Table IV compares the elapsed time and speedup among the NavP, the MP, and the ScaLAPACK implementations. ScaLAPACK implements a block-fashion Cholesky factorization algorithm. Sequential timings were generated by running ScaLAPACK's implementation on one node and choosing a matrix block size of 64×64. This gives a sequential performance that is better than those using other block sizes, and is better than that of our sequential code in C. For the parallel run of ScaLAPACK, we chose a matrix block size of 1×1 so that the underlying algorithm is the same as what the NavP and MPI programs implement. Figure 8 depicts the

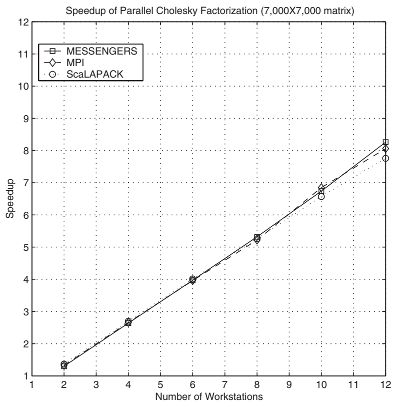
Table IV. Performance of Parallel Cholesky Factorization

Matrix Size	3000		5000		7000	
Number of Workstations	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
			Sequential			
1	95.04	1.00	453.15	1.00	1373.50	1.00
			Messengers			
2	66.72	1.42	361.67	1.25	1052.91	1.30
4	30.47	3.12	177.56	2.55	519.86	2.64
6	22.39	4.24	113.92	3.98	346.20	3.97
8	17.66	5.38	81.31	5.57	258.11	5.32
10	15.27	6.22	64.51	7.02	203.69	6.74
12	13.91	6.83	54.60	8.30	166.30	8.26
			MPI			
2	67.08	1.42	363.07	1.25	1031.49	1.33
4	30.34	3.13	177.85	2.55	517.02	2.66
6	21.01	4.52	112.46	4.03	346.84	3.96
8	18.14	5.24	82.02	5.53	263.03	5.22
10	14.61	6.51	64.70	7.00	200.60	6.85
12	14.31	6.64	54.66	8.29	170.31	8.06
			ScaLAPACK			
2	77.05	1.23	358.63	1.26	1000.71	1.37
4	36.34	2.62	178.32	2.54	508.66	2.70
6	23.75	4.00	117.17	3.87	342.84	4.01
8	19.02	5.00	84.08	5.39	261.06	5.26
10	16.36	5.81	68.37	6.63	209.17	6.57
12	14.18	6.70	56.30	8.05	177.05	7.76





(a) (b)



(c)

Fig. 8. Performance of parallel Cholesky factorization. (a) $3K \times 3K$, (b) $5K \times 5K$, (c) $7K \times 7K$ matrices.

speedup data. Again, the speedup of our NavP implementation is almost the same as those of the MPI, and their trends as the number of machines increases are the same which indicates same scalability. There is no performance degradation in spite of the greatly improved programmability.

6.3. Summary of Steps

In this section, we describe how the three typical steps discussed in Section 5 map into the two real-world examples described in Sections 6.1

and 6.2. *Data distribution:* We first distribute columns of a matrix in block fashion in both Jacobi iteration and Cholesky factorization. The number of machines is chosen such that the size of the data set on each machine fits in the main memory completely. *DSC:* We decide which sub-computations are to be implemented. In Jacobi iteration, the sub-computation to be implemented is the loop between lines (5)–(9) in Fig. 2(a), which is the sub-computation of a new solution vector. In Cholesky factorization the two nested loops (lines (1)–(10) and (13)–(17) in Fig. 6(a)) represent the two sub-computations of scaling and updating, respectively. In both cases, we augment the code with *hop*() and load/unload statements to get a DSC program. To follow the principle of pivot-computes in each sub-computation, we let the loop indices and the vectors meet with the larger matrix blocks. *Parallelization:* In Cholesky factorization, the updating steps on all the nodes are independent of each other. Therefore, we take *Step 3.1* and assign them to concurrent self-migrating threads. This transformation is depicted in Fig. 9. We add events and injections to coordinate the threads from different loops. We further realize that distributing the columns in a round-robin fashion will provide better load balancing, so we reiterate back to *Step 1* and adjust the data distribution to a round-robin fashion. In Jacobi iteration, we look for pipelining opportunity. We observe that the

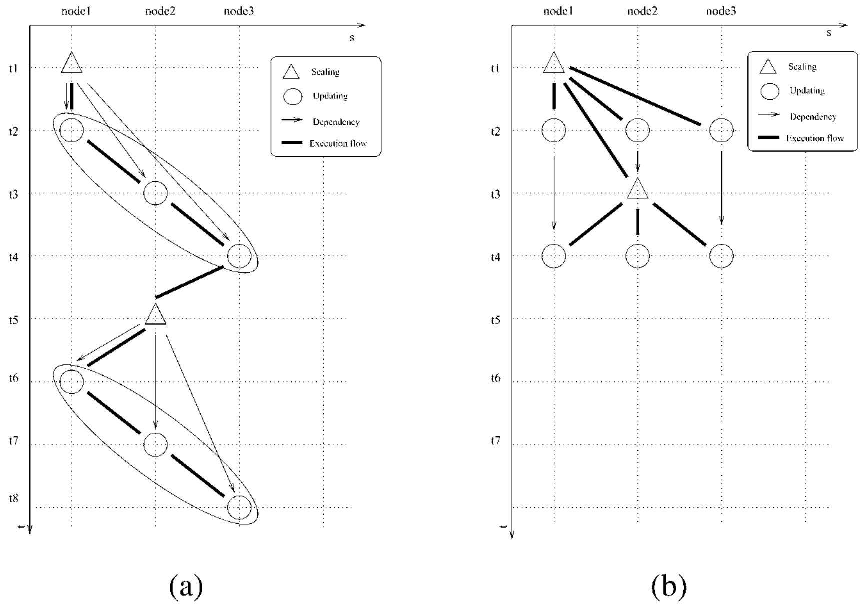
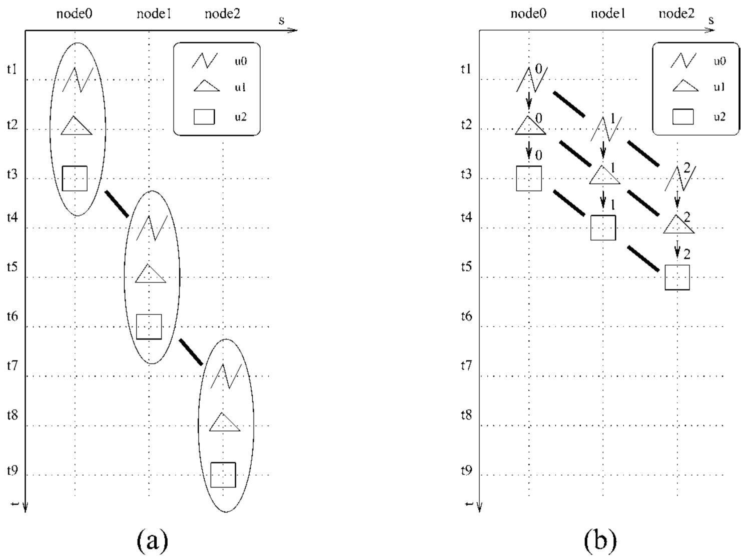


Fig. 9. Cholesky factorization. (a) DSC. (b) DPC after Step 3.1.



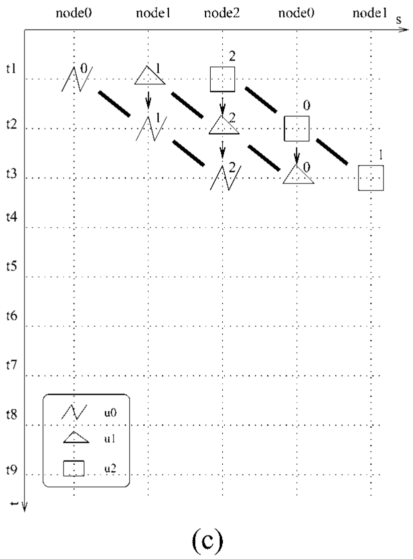


Fig. 10. Jacobi iteration. (a) DSC. (b) DPC after Step 3.2. (c) DPC after a phase shift.

computation of different new sub-solution vectors can really be done at the same time; so we take *Step 3.2* and assign the computations of different sub-solution vectors to different threads, and have them run in a pipeline. Furthermore, we phase shift these threads to achieve complete parallelism, and this does not affect the correctness of the result. The above transformations are depicted in Fig. 10.

7. COMPARISON OF APPROACHES

In this section we compare the NavP approach with the two classical solutions, namely MP and DSM. Table V summarizes the comparison of the approaches, with explanations as follows.

7.1. Data Distribution

There are two ways to handle data distribution. In a classical DSM system in which data distribution is completely transparent, it is unlikely that data distribution happens to be consistent with computation distribution, because this would require information that is application dependent. This is obvious in Cholesky factorization in which the round-robin data distribution is based on the decision of distributing computation in the same fashion. As a result, in classical DSM systems the nodes that perform certain computations may not own the corresponding data being computed, resulting additional communication overhead. In some variations of DSM systems, a programmer is able to specify the data distribution pattern. For example, HPF and UPC[(27–30)] allow a programmer to define the data distribution pattern with a key word *shared*, which can be used to define "block," "cyclic" (i.e., round-robin in Cholesky factorization), or "cyclic block" type of data distribution. But now data distribution

Table V. Comparison of Approaches

Aspect	Unstructured DSM	MP	NavP
1. data distribution	transparent	explicit	explicit
2. data sharing	transparent, shared var	explicit, message	explicit (hop), DSV + agent var
3. data locality	does not follow pivot-computes	follows pivot-computes	follows pivot-computes
4. parallel program composition	stationary DSCs	strictly sequential processes	navigational DSCs
5. parallel program synchronization	explicit, global barrier	explicit, remote message	explicit, local event

is no longer completely transparent. In MP and the NavP approach, a programmer controls the data distribution pattern explicitly.

7.2. Data Sharing

Both DSM and NavP use shared variable programming, but they have different mechanisms.[5] In DSM, shared variables are put on distributed shared memory (also called shared virtual memory), and are accessible from any node, while in MESSENGERS distributed shared variables are constructed logically from node variables that are only accessible locally. An example of a DSV is the matrix A in Cholesky factorization, shown in Fig. 6(b). In a page-based DSM, data entries belonging to computations of different nodes can happen to be put in the same page, resulting in false sharing as multiple nodes are updating the parts of which they own the computations. In the NavP approach, false sharing is impossible. The large sized data is put to stationary DSVs, and the only way for the threads to shared them is to hop to the node where the data resides. It is important to realize that a shared variable does not have to be globally accessible. Inter-node communication is carried out by thread migration, and the agent variables are private to their owner threads and are not shared at all. In MP, communication is through messages, and processes do not share anything directly among themselves, hence false sharing is not a possibility.

Another well known problem with DSM is its lack of memory coherence. Because of the larger-than-necessary data movement, data replication is used in order to reuse the data moved and amortize the cost of communication over multiple repeated accesses. Data replication is done also to increase the degree of concurrent accessing. But replication leads to the problem of incoherence, and memory coherence protocols are then needed in order to guarantee correctness. Now the new problem is that the traffic caused by consistency requirements is by itself a big overhead in many situations. The protocols are complicated, and they do not always provide efficient solution for dynamic data access patterns. In the NavP approach, similar to MP, no data is replicated, therefore no memory coherence protocol is needed. Also, in the NavP approach, since communication between nodes is intra-thread, there is no need to have a "receiver" in the application code to pair up with the hop. This means the NavP approach uses one-sided communication naturally.[20, 31]

7.3. Data Locality

In the DSM implementation of Cholesky factorization, each process executes a parallel data-independent sub-task (i.e., updating). The sequential part (i.e., scaling) is carried out by a single process, at which time all other processes wait. The DSM code looks almost the same as the original algorithm. The problem with the DSM program is its efficiency. The scaling part is not always done by the pivot, or the owner node of the column being scaled. Rather, it is all done on the node with ID $\mu == 1$, as shown in Fig. 6(a). This violates the principle of pivot-computes, and the result is that almost the entire matrix will be pulled to the scaling node, which is more expensive than needed. This example shows why not knowing the data distribution, or in other words, not knowing who the pivots are for each sub-computation, could result in less efficient code. Indeed, experiments have shown that exploiting data locality has greater benefits over load balancing, and hence should take the priority.(22) One of the ways to ensure that the pivot does scaling, is to put the data distribution in the programmer's hands so that in the code the pivot nodes can be specified to compute.

The data being moved is stored in agent variables. In Jacobi iteration, the agent variables are: u_μ, which stores the sub-solution-vector of the next iteration; *err*, which is the global error; and j and c, which are loop counters, all shown in Fig. 2(b). In Cholesky factorization, the communicated data moved by the *Updaters* is all the G columns stored in the agent variable v_{loc} (loaded in line (12), and carried away by the *hop*() statement in line (12.1) in Fig. 6(b)). The data being moved by the *Scaler* is the loop index k. The NavP implementation moves almost the same amount of data as does the MP program. The only additional data are (1) The Messenger control block (MCB) that is about 200 bytes in size; (2) Loop indices. Since small sized data is moved to meet with large sized data in both MP and NavP code, or in other words, these two approaches both follow the principle of pivot-computes, they are both efficient and scalable.

7.4. Parallel Program Composition

The NavP approach decomposes a problem into DSCs, some or all of which are concurrent. In Jacobi iteration, multiple concurrent threads are each a DSC but altogether doing data parallel computing. This is clearly seen in Fig. 4(a). In Cholesky factorization, the decomposition is more heterogeneous. As depicted in Fig. 4(b), the thread *Scaler* conducts the distributed sequential scaling, and it injects multiple concurrent data parallel *Updaters*, which are all DSCs because they compute with the G column and the columns to be updated, and in all but one case these columns reside on two different nodes. Decomposing a problem into DSCs each implemented using NavP helps programmability. This is best seen by comparing the NavP and MP pseudocodes shown in Fig. 6(b) and Fig. 7 respectively. The explanation is as follows.

As discussed in Section 3, the NavP approach preserves algorithmic integrity of a DSC, while the MP approach does not. The three problems and solutions shown in Table II are exemplified in Cholesky factorization as follows. First, in the original algorithm the scaling of all columns of A is put in a loop over the global index k. This computation is required by the principle of pivot-computes to happen across machine boundaries as the columns that are being scaled become remote. The MP solution breaks the global loop over k into p smaller local loops over index q (line (2) in Fig. 7). In the NavP code shown in Fig. 6(b), the *hop*() statement at line (9.1) solves the problem. Second, the loop index k is locally scoped to the loop, and when the locus of computation shifts, it needs to follow. The MP solution is to locally recompute k at lines (8) and (25) shown in Fig. 7. In the NavP code, k is an agent variable. This data transfer in the NavP approach is implicit because to a self-migrating thread, although the locally scoped data is carried across machine boundaries, the communication is still intra-thread. Third, in MP code shown in Fig. 7, lines (4), (6), (11), and (23) all use the local column indices (the second index in $A(\cdot,\cdot)$). In the situation where the global correspondence of a local index is needed (e.g., at line (3)), the local data view is mapped back to global. In the NavP code, we can preserve the global data view by using the DSV A, and the global-local data map $col(\cdot)$.

Section 4 pointed out that composing DPC with DSCs can avoid code tangling, a problem with the MP approach. In the NavP implementation of Cholesky factorization, the problem is decomposed into two types of DSCs, namely *Scaler* and *Updaters*, and their coordination are done using injections and events, as shown in Fig. 6(b). Since the interfaces among different DSCs are only injections and events, the composition incurs very small code tangling, and no code for computation is tangled at all. In MP parallel programs, code belonging to different sequential computations might be grouped together according to execution location. For example, in the MP implementation of Cholesky factorization shown in Fig. 7, the code responsible for scaling and updating respectively is tangled on the scaling node (lines (4)–(12)), but not on the other nodes (lines (15)–(24)).

In DSM, parallel programs are composed also using DSCs. Nevertheless, the DSCs in DSM are all stationary and therefore may not exploit data locality by following the principle of pivot-computes. An example is the scaling which pulls all the matrix columns to the node with ID $\mu == 1$, as shown in Fig. 6(a).

The disadvantages of MP programming are rooted in the fact that MP forces its programmers to handle non-trivial location-related details in their code. In contrast, the NavP approach leaves those details of implementation to a compiler which is good at restructuring code based on locations. To be fair, we should point out that the MP approach does not always restructure code significantly causing difficulties in programming. For example, in Jacobi iteration, since all nodes conduct the same computations, and each of them is matrix-vector multiplication in block fashion that completely finishes on one node, no matter how we decompose the problem (i.e., into DSCs or into computations that are centered around executing nodes), the code looks pretty much the same. In general, MP handles "homogeneous" data parallel computing well.

7.5. Parallel Program Synchronization

In the DSM code shown in Fig. 6(a), two *barriers* are used for synchronization. A *barrier* involves global communication and is in most cases a restriction that is stronger than necessary. A useful feature of the NavP approach is that since all data accesses to variables are local, the only synchronization required is among different threads on the same node; in other words, no inter-node synchronization is required. The performance advantage in Cholesky factorization resulting from only requiring local synchronization can be seen in Fig. 11(b): the next round of scaling can start as soon as the previous local updating is done, regardless of whether or not the remote updatings are finished. In contrast, the global *barriers* (lines (7) and (9) in Fig. 6(a)) are less efficient. As depicted in Fig. 11(a), the next iteration can only start after all *Updaters* from the previous iteration have finished. In a distributed environment with relatively high network latency, the performance improvement from this overlapping using local synchronizations can be significant. Both DSM and the NavP approach use shared variable programming, but DSM needs *barriers* while NavP approach does not. The NavP approach decouples communication from synchronization, as can be seen in the pseudocode shown in Fig. 6(b). Between nodes, communication is done by carrying agent variable. For example, the G columns are being carried by the *Updaters* at line (12). This is an intra-thread activity, and is done, at application level, by each individual thread and therefore no synchronization is needed (or in other words, synchronization is subsumed in the flow control), similar to the one-sided communication in MPI-2.[31] Multiple self-migrating threads communicate among themselves using shared variables. The synchronization of these threads is through local events or injections (see lines (8), (9.2), (12.2), and (17.1)). In contrast, with MP messages sent across nodes are used for communication, synchronization, or both purposes.

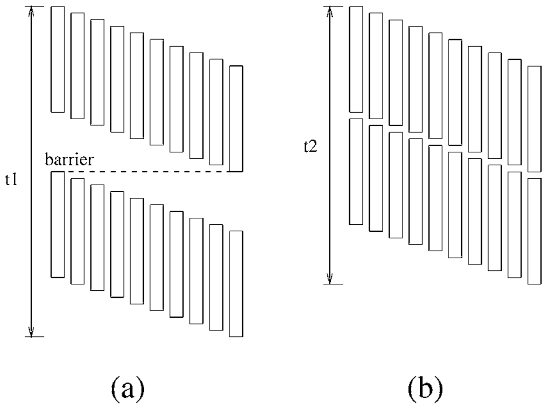


Fig. 11. Synchronization. (a) Global in DSM. (b) Local for NavP.

8. CONCLUSIONS

In this paper, we have developed a new way of doing distributed parallel programming. We call it the navigational programming (NavP) approach, and described the steps of the approach so that it can be applied to the parallelization of various applications. We applied the NavP approach to two real-world parallel algorithms, namely Jacobi iteration and Cholesky factorization. Our NavP and MPI implementations of the algorithm yield almost identical performance. Since there is a general agreement that the performance of MP is considerably better than that of DSM in many non-trivial applications, our results imply that the performance of NavP is also superior to that of DSM. From the efficiency point of view, the NavP approach is as suited for general purpose high performance computing as MP is. In contrast to MP, the advantage of ease of programming is clearly shown in the NavP implementations. The reason for NavP parallel programs to be efficient and scalable is their data locality and synchronization, while their data sharing and parallel program composition contribute to the ease of programming, as shown in Table V. While the examples in this paper are drawn from the domain of numerical analysis, the NavP approach is a general one to developing general purpose parallel distributed programs.

Both the MP and the NavP approaches have one thing in common: data distribution and data sharing are explicit, rather than transparent as in DSM (see Table V).

1. *data distribution:* The programmer must develop an application-dependent strategy for data distribution, and then construct, as by-products of the strategy, the data-node map and data-global-local map. These maps are then explicitly used in the code. In contrast, none of these maps shows up explicitly in DSM code, because the mapping is taken care of by the underlying DSM system. However, this by no means says that DSM programmers are worry free. In fact, as shown in the Cholesky factorization, for the purposes of load balancing, a DSM programmer does need a data accessing strategy, from which constructing the maps is only a small step.
2. *data sharing:* A DSM system provides the benefits of transparent data sharing for its programmers. However, this seeming advantage can be the cause of poor performance and scalability of DSM programs because a programmer can easily make a wrong decision of moving larger amount of data than necessary in a sub-computation. In the NavP approach, variable sharing is accomplished by explicitly inserting *hop*() and load/unload statements. This approach provides a balance between convenience and efficiency: the code is easy to develop from the original algorithm, and the programmer has full control over exactly when and where data movement happens.

There are two basic approaches to parallel programming: *explicit parallel programming*—the programming of explicitly parallel algorithms, or *implicit parallel programming*—feeding a parallelizing compiler with sequential programs.(21) Achieving completely implicit parallel programming is very difficult for the compiler builders because the parallelizing compiler must analyze and understand the dependencies in different parts of the sequential code to ensure an efficient mapping to a parallel computer. Recent research trends have shown compromises in dealing with these difficulties: in distributed memory, data distribution is put into the hands of the programmers and hence is no longer completely transparent in systems such as UPC and HPF.(27–30) In physically shared memory, where data distribution is less of a problem, parallel regions can be specified by the programmers in order for the compiler to better find concurrency, as in OpenMP.(32) Still implicit and explicit parallel programming styles have their strengths and weaknesses, and hence have their own applicable areas. In particular, implicit parallel programming is easy to use for the programmers, and it explores instruction-level parallelism, whereas explicit parallel programming requires more input from the programmers, and is good at higher level task-parallel problems.

Compromises are necessary not only between implicit and explicit approaches, but also among approaches within the explicit parallel programming itself. DSM and MP have a tendency to converge in some aspects. One example is the introduction of explicit data distribution in HPF and UPC. The other is the one-sided communication introduced in, e.g., MPI-2. But these changes do not remove the difficulties rooted in stationary processes. Explicit data distribution in DSM is not really useful unless the locus of computation is explicitly transferred. If this is done using stationary processes, the disadvantage in programmability is the same as with MP: algorithmic integrity is lost. One-sided communication represents an attempt to augment the MP model by adding memory-sharing capability. If this is done using only stationary processes, it can lead to a violation of pivot-computes, because the single stationary process involved may compute with a large amount of remote data. Thus, these innovations only partially achieve their goals unless processes have the ability to navigate. NavP allows the programmers to take full advantage of explicit data distribution while preserving algorithmic integrity. NavP also allows accessing remote data—the goal of one-sided communication in MP—while following the principle of pivot-computes.

The NavP approach, as presented here, belongs to explicit parallel programming, and so do the DSM and MP approaches used as comparisons. As such, this approach, similar to its two counterparts, leaves the problems of finding efficient data mapping, maximizing parallelism, achieving load balancing, and adapting to changing network environments, to its programmers. Nevertheless, there is nothing that prevents the NavP approach from supporting implicit parallel programming in the future. For example, the NavP programs, which preserve their original code structures, could be used to serve as an intermediate step at which data locality is exploited by a future new parallelizing compiler which supports strong mobility. Our future work includes making the NavP approach more implicit by building tools to automate its three steps.

ACKNOWLEDGMENTS

The authors wish to thank Eugene Gendelman for helpful discussions about the MESSENGERS system.

REFERENCES

1. L. Pan, L. F. Bic, and M. B. Dillencourt, Distributed Sequential Computing Using Mobile Code: Moving Computation to Data, L. M. Ni and M. Valero (eds.), *Proceedings of the 2001 International Conference on Parallel Processing (ICPP 2001)*, IEEE Computer Society, Los Alamitos, California, pp. 77–84 (September 2001).

2. L. Pan, L. F. Bic, M. B. Dillencourt, and M. K. Lai, Mobile Agents—The Right Vehicle for Distributed Sequential Computing, S. Sahni, V. K. Prasanna, and U. Shukla (eds.), *Proceedings, 9th International Conference on High Performance Computing—HiPC 2002*, *Lecture Notes in Computer Science*, Vol. 2552, Springer-Verlag, Berlin, Germany, pp. 575–584 (December 2002).
3. I. Pramanick, MPI and PVM Programming, R. Buyya (ed.), *High-Performance Cluster Computing*, Vol. 2, Programming and Applications, Prentice–Hall PTR, Upper Saddle River, N.J., pp. 48–86 (1999).
4. L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania (1997).
5. L. Pan, L. F. Bic, and M. B. Dillencourt, Shared Variable Programming Beyond Shared Memory: Bridging Distributed Memory with Mobile Agents, H. Ehrig, B. Kramer, and A. Ertas (eds.), *Proceedings of the 6th International Conference on Integrated Design & Process Technology (IDPT-2002)*, Society for Design & Process Science, Grandview, Texas (June 2002).
6. L. F. Bic, M. Fukuda, and M. B. Dillencourt, Distributed Computing using Autonomous Objects, *IEEE Computer*, **29**(8):55–61 (August 1996).
7. C. Wicke, L. F. Bic, M. B. Dillencourt, and M. Fukuda, Automatic State Capture of Self-Migrating Computations in MESSENGERS, K. Rothermel and F. Hohl (eds.), *Proceedings, Second International Conference on Mobile Agents, MA '98*, *Lecture Notes in Computer Science*, Vol. 1477, Springer-Verlag, Berlin, Germany, pp. 68–79 (September 1998).
8. M. Fukuda, L. F. Bic, and M. B. Dillencourt, Messages versus Messengers in Distributed Programming, *J. Parallel Distr. Com.*, **57**:188–211 (1999).
9. E. Gendelman, MESSENGERS *User's Manual (version 2.1)*, Information & Computer Science, University of California, Irvine, Irvine, California (2001).
10. D. Chess, C. Harrison, and A. Kershenbaum, Mobile Agents: Are They a Good Idea?, J. Vitek and C. Tschudin (eds.), *Selected Presentations and Invited Papers, Second International Workshop on Mobile Object Systems, MOS '96*, *Lecture Notes in Computer Science*, Vol. 1222, Springer-Verlag, Berlin, Germany, pp. 25–47 (July 1997).
11. A. Fuggetta, G. P. Picco, and G. Vigna, Understanding Code Mobility, *IEEE Transactions on Software Engineering*, **24**(5):342–361 (May 1998).
12. D. B. Lange and M. Oshima, Seven Good Reasons for Mobile Agents, *Commun. ACM*, **42**(3):88–89 (March 1999).
13. D. Milojicic, Trend Wars: Mobile Agent Applications, *IEEE Concurency*, **7**(3):80–90 (July-Sept. 1999).
14. R. S. Gray, D. Kotz, G. Cybenko, and D. Rus, *Mobile Agents: Motivations and State-of-the-Art Systems*, Technical Report TR2000-365, Dartmouth College, Hanover, New Hampshire (April 2000).
15. L. Bettini and R. De Nicola, Translating strong mobility into weak mobility, G. P. Picco (ed.), *Proceedings, 5th International Conference on Mobile Agents, MA 2001*, *Lecture Notes in Computer Science*, Vol. 2240, Springer-Verlag, Berlin, Germany, pp. 182–197 (December 2001).
16. D. Kotz, R. Gray, and D. Rus, Future Directions for Mobile Agent Research, *IEEE Distributed Systems Online*, **3**(8) (2002).
17. C. Wicke, *Implementation of an Autonomous Agents System*, Master's thesis, Dept. of Information and Computer Science, University of California, Irvine, Irvine, California (September 1998).

18. E. Gendelman, L. F. Bic, and M. B. Dillencourt, Fast File Access for Fast Agents, G. P. Picco (ed.), *Proceedings, 5th International Conference on Mobile Agents, MA 2001*, *Lecture Notes in Computer Science*, Vol. 2240, Springer-Verlag, Berlin, Germany, pp. 88–102 (December 2001).
19. R. Morgan, *Building an Optimizing Compiler*, Butterworth-Heinemann, Boston, Massachusetts (1998).
20. C. Leopold, *Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches*, John Wiley & Sons, New York (2001).
21. V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin Cummings, Redwood City, California (1994).
22. E. Markatos and T. LeBlanc, *Load Balancing vs. Locality Management in Shared-Memory Multiprocessors*, Technical Report 399, Computer Science Department, University of Rochester, Rochester, New York (October 1991).
23. G. Burns, R. Daoud, and J. Vaigl, LAM: An Open Cluster Environment for MPI, J. W. Ross (ed.), *Proceedings of Supercomputing Symposium*, pp. 379–386 (1994).
24. J. M. Squyres and A. Lumsdaine, A Component Architecture for LAM/MPI, *Proceedings, 10th European PVM/MPI Users' Group Meeting*, *Lecture Notes in Computer Science*, Vol. 2840, Springer-Verlag, Berlin, Germany (2003).
25. W. L. Briggs, *A Multigrid Tutorial*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania (1987).
26. G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd edn., Johns Hopkins University Press, Baltimore, Maryland (1996).
27. W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, *Introduction to UPC and Language Specification*, Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Bowie, Maryland (May 1999).
28. T. A. El-Ghazawi and S. Chauvin, *Getting Started with UPC*, High Performance Computing Laboratory, George Washington University, Washington DC (June 2001).
29. R. S. Schreiber, An Introduction to HPF, *Lecture Notes in Computer Science*, Vol. 1132, pp. 27–44 (1996).
30. C. H. Koelbel, *The High Performance Fortran Handbook*, MIT Press, Cambridge, Massachusetts (1994).
31. MPI-2: Extensions to the Message-Passing Interface, The MPI Forum (July 1997).
32. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*, Morgan Kaufmann Publishers, San Francisco, California (2001).