

# Distributed Parallel Computing Using Navigational Programming: Orchestrating Computations Around Data

Lei Pan, Lubomir F. Bic, Michael B. Dillencourt, Javid J. Huseynov and Ming Kin Lai

Information & Computer Science

University of California, Irvine

Irvine, CA 92697-3425, USA

{pan,bic,dillenco,javid,mingl}@ics.uci.edu

## ABSTRACT

Message Passing (MP) and Distributed Shared Memory (DSM) are the two most common approaches to programming on distributed memory systems. MP is difficult to use, while DSM is not scalable. Performance scalability and ease of programming can be achieved at the same time by using “shared variable programming” and following the principle of “computation locus following data,” which is embodied in our Navigational Programming approach. The implementation of a real-world algorithm, parallel Cholesky factorization, presented in this paper supports our claim that Navigational Programming is better suited for general purpose distributed programming than either MP or DSM.

## KEY WORDS

Navigational Programming, distributed parallel computing (DPC), message passing (MP), distributed shared memory (DSM), shared variable (SV) programming, Cholesky factorization

## 1 Introduction

Two fundamental approaches to distributed programming, message passing (MP), and distributed shared memory (DSM) each has a drawback: MP is hard to use, and DSM is not scalable. We have proposed a general-purpose distributed programming approach [1, 2, 3], based on the use of a mobile agent system. We call this approach Navigational Programming [4]. Navigational Programming combines the advantages of MP and DSM: (1) Navigational Programming uses shared variable (SV) programming for communication and synchronization as DSM does, and hence the code is easy to develop and maintain (this is referred to as “algorithmic integrity”); (2) Similar to MP, Navigational Programming allows the programmers to take full control of data distribution, and to make smart decisions by binding small amount of data to the locus of computation and having them follow the larger data pieces. As a result, the programs are efficient and scalable. Navigational Programming balances convenience and flexibility, thus combining the advantages of earlier approaches such as MP and DSM.

Navigational Programming can be applied in dis-

tributed sequential computing (DSC) [1, 2]. Among the benefits are improved performance on large problems (from eliminating disk thrashing), and increased programmability. The limitation of DSC is that those programs are still sequential. To gain more from distributed computing, the next logical step is to develop programs that employ multiple concurrent agents working in parallel.

In this paper, we present a real-world algorithm: the Cholesky factorization, and its parallel implementation using Navigational Programming. We compare the ease of programming and efficiency of our approach with the other implementations. We also present a performance comparison between our approach and the MPI [5] implementation. The experimental evidence from this parallel example supports our claim that Navigational Programming is better suited for general purpose distributed programming [3].

The DSC we introduced takes advantage of “computation locus following data” [1, 2]. This advantage in terms of efficient use of network bandwidth is kept in our parallel programs here by all individual mobile agents. Furthermore, these mobile agents are organized around (large) data pieces to perform one task in parallel.

The rest of the paper is organized as follows. Section 2 presents parallel implementations of parallel Cholesky factorization using DSM, MP, and Navigational Programming approaches. Section 3 contains a detailed comparison of the implementations, along with performance data. The last section contains some final remarks.

## 2 Parallel Cholesky Factorization

Cholesky factorization is an algorithm for factorizing symmetric positive definite matrices. In this section, we briefly describe this algorithm, and then present three implementations, using the DSM, Navigational Programming, and MP approaches. The MP and DSM implementations of Cholesky factorization are based on those in a classic textbook on matrix computation [6].

A positive definite matrix  $A$  can be factored into the product of two matrices  $A = GG^T$ , where  $G$  is a lower triangular matrix called the Cholesky triangle. This decomposition can then be used to solve a linear system of equations. 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 versions. Figure 1(a) contains pseudocode for a parallel implementation of outer product Cholesky factorization in shared memory or DSM, adapted from the code given in Ref. [6].

In Figure 1(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 accounts for a very small fraction of the total work: each column is scaled once, so the total cost of all the scaling is  $O(n^2)$ . Hence there is little to be gained by parallelizing this portion of the algorithm. 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 total work done in the updating steps is  $\Theta(n^3)$ . Hence this is the portion of the algorithm that is parallelized.

The DSM implementation assumes that there are  $p$  processors, each running the pseudocode shown in Figure 1(a). The scaling is performed at line (4), and the updating is performed at line (15). It is assumed that computations cannot be done directly to shared variables [6], so lines (3), (5), (12), (14), and (16) are used to copy data to and from the shared variable  $A$ . In some DSM's, this limitation does not exist, in which case these extra lines could be removed from the pseudocode. We include these lines to emphasize that the actions they represent take place when the  $A$  matrix term involved resides in remote memory. This copying creates excessive data movement when the matrix  $A$  is not distributed properly for the application.

In order to balance load, a round-robin data distribution scheme for updating the columns is proposed in Ref. [6]. This scheme is illustrated by the following simple example. Suppose the number of columns in  $A$  is  $n = 11$ , the number of processors is  $p = 3$ , and the  $i$ th column of  $A$  is denoted by  $a_i$ . In the most straightforward scheme, contiguous allocation of columns, the updating of columns would be assigned to processors as follows:

$$\begin{array}{c|c|c} [a_1 & a_2 & a_3 & a_4 & | & a_5 & a_6 & a_7 & a_8 & | & a_9 & a_{10} & a_{11}] \\ \text{PE1} & & & & & \text{PE2} & & & & & \text{PE3} & & \end{array}$$

In the round-robin scheme, the assignment of columns to processors is as follows:

$$\begin{array}{c|c|c} [a_1 & a_4 & a_7 & a_{10} & | & a_2 & a_5 & a_8 & a_{11} & | & a_3 & a_6 & a_9] \\ \text{PE1} & & & & & \text{PE2} & & & & & \text{PE3} & & \end{array}$$

In the contiguous allocation strategy, processor 1 would be idle after columns 1 to 4 have been computed, even though much work remains. In the round-robin strategy, processor  $\mu$  carries out the construction of  $G(:, \mu : p : n)$  where the column index starts from  $\mu$ , ends at up to  $n$ , with an increment of  $p$ . This strategy distributes matrix  $A$  evenly to all participating processors, and ensures that all of the processors are busy most of the time. The pseudocode in Figure 1(a) reflects this round-robin strategy. The index  $k$  loops over all the columns of  $A$  (line (1)). The first processor (processor ID  $\mu == 1$ ) is responsible for scaling column  $k$  (lines (2)-(6)), after which all the processors, including processor 1, will update the columns they are responsible for and that are to the right of column  $k$  (line (8)). In particular, processor  $\mu$  will update the  $A$  columns  $(k + \mu) : p : n$  (lines (13)-(17)). After all the processors are done updating, processor 1 can start scaling on the next column  $(k + 1)$ , and the computation continues.

Our Navigational program implemented in MESSENGERS [7, 8], shown in Figure 1(b), consists of two types of mobile agents: a single scaling agent, and multiple updating agents. The single scaling agent carries the loop index  $k$  which loops through all columns of matrix  $A$ , which is a shared node variable [8]. On the  $k$ th iteration, the scaling agent scales column  $k$  (line (4)). The function  $\text{col}(k)$  maps the global column index  $k$  to a local column index; this function is needed because each processor stores only a portion of the entire global matrix  $A$ . After scaling the column, the scaling agent injects  $p$  updating agents (lines (7.1)-(8.1)), and then it hops to the processor that owns the next column of  $A$  (line (9.1)). The ID of this processor  $\mu$  is found using a column-to-processor map. The scaling agent then waits for the next round of computation. Each of the  $p$  updating agents loads the newly computed  $G$  column  $k$  (again the local column index is  $\text{col}(k)$ ) into its agent variables (line (12)), and then hops to the appropriate processor (line (12.1)). In parallel, these  $p$  agents update the  $A$  columns for which they are responsible on all  $p$  processors, using the  $G$  column stored in agent variables [8] and the  $A$  entries stored in shared node variables (line (15)). Two maps are used in the Navigational Programming implementation (lines (4), (9.1), (12), (12.1), and (15)) and they are application dependent. In particular, here the column-to-processor map is  $\text{proc\_map}(k) = (k - 1) \% p + 1$ , and the global-to-local-column-index map is  $\text{col}(k) = (k - \mu) / p + 1$ , where  $k$  is global column index,  $p$  is number of processors, and  $\mu$  is current processor ID. Because matrix columns are not assigned to processors using a linear map, local memory accessing cannot be done with shifted pointers [3], but the maps we use here are simply by-products of a user defined data distribution strategy.

Notice that scaling is performed sequentially by a single agent, while updating is done in parallel by  $p$  concurrent agents. This is because scaling accounts for only a

```

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

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

(9)   barrier;

(10) }

(11) updating(int  $\mu$ , int k, int n) {
(12)   vloc(k + 1 : n) = A(k + 1 : n, k);

(13)   for(j = k +  $\mu$ ; j <= n; j += p) {
(14)     wloc(j : n) = A(j : n, j);
(15)     wloc(j : n) -= vloc(j)vloc(j : n);
(16)     A(j : n, j) = wloc(j : n);
(17)   }

(18) }

```

(a)

```

(1)   for(k = 1; k <= n; k++) {
(2)
(3)
(4)     A(k : n, col(k)) /=  $\sqrt{A(k, col(k))}$ ;
(5)
(6)
(7)
(7.1)   for(c = 1; c <= p; c++) {
(8)     inject(updating(c, k, n));
(8.1)   }
(9)
(9.1)   hop(proc_map(k + 1));
(9.2)   waitEvent(scaleEvt, k + 1);
(10) }

(11) updating(int c, int k, int n) {
(12)   vloc(k + 1 : n) = A(k + 1 : n, col(k));
(12.1) hop(proc_map(k + c));
(13)   for(j = k + c; j <= n; j += p) {
(14)     A(j : n, col(j)) -= vloc(j)vloc(j : n);
(15)
(16)
(17)   }
(17.1) signalEvent(scaleEvt, k + 1);
(18) }

```

(b)

Figure 1. Pseudocode for Parallel Cholesky Factorization using (a) Distributed Shared Memory (b) Navigational Programming.

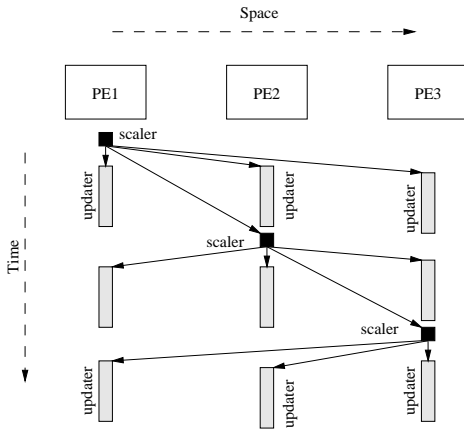


Figure 2. Interleaved Parallel and Sequential Steps

very small fraction of the entire computation, and there is no need to make the extra effort to parallelize it. Figure 2 depicts how the two types of mobile agents coordinate to achieve interleaved sequential and parallel computing.

In the pseudocode of Figure 1(b), there are two lines of code that use `signalEvent` and `waitEvent` primitives for synchronizing all agents that are local to a logical node

[8]. The semantics of these primitives is described in MESSAGES manual [8]. After the scaling agent executes the `inject` commands at line (8), it hops away immediately after all the injections are done, and then the injected agents start executing (line (12)). Thus the scaler hops to the next processor and continues its computation and communication without having to wait for the injected updating agents to hop away. The `waitEvent` and `signalEvent` primitives (lines (9.2) and (17.1), respectively) are used to protect the shared variable  $A$  from being updated in an incorrect order. A nice feature of Navigational Programming is that since all data accesses to variables are local, the only synchronization required is among agents on the same machine; in other words, no inter-machine synchronization is required. The performance advantage resulting from only requiring local synchronization can be seen in Figure 2: the next round of scaling can start as soon as the previous local updating is done, regardless of whether or not the remote updating is finished. In contrast, the global barriers (lines (7) and (9) in Figure 1(a)) are less efficient. In a distributed environment with relatively high network latency, the performance improvement from overlapping and local synchronization can be significant.

The Navigational Programming pseudocode (Figure 1(b)) preserves algorithmic integrity with respect to the DSM original (Figure 1(a)), and also with respect to

```

(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)     }
(8)     k = k + 1;
(9)     for (i = q + 1; i <= L; i++) {
(10)      r = col(i);
(11)       $A_{loc}(r : n, i) - = A_{loc}(r, q) * A_{loc}(r : n, q)$ ;
(12)    }
(13)    q = q + 1;
(14)  }
(15)  else {
(16)    recv( $g_{loc}(k : n)$ , left);
(17)     $\alpha$  = proc which sent  $k^{th}$  G col;
(18)     $\beta$  = index of right's final col;
(19)    if (right !=  $\alpha$  and  $k < \beta$ ) {
(20)      send( $g_{loc}(k : n)$ , right);
(21)    }
(22)    for (i = q; i <= L; i++) {
(23)      r = col(i);
(24)       $A_{loc}(r : n, i) - = g_{loc}(r) * g_{loc}(r : n)$ ;
(25)    }
(26)    k = k + 1;
(27)  }
(28) }

```

Figure 3. Pseudocode for MP parallel Cholesky Factorization.

the sequential original [6]. The synchronization events (lines (9.2) and (17.1) in Figure 1(b)) do the same job as the synchronization barriers do (lines (7) and (9) in Figure 1(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 agents where to migrate and what to carry for later sharing and computing. Two maps are used to tell the code about how data is distributed. The creation of these maps does not represent extra work for the programmer, since a data distribution strategy must also be developed in the DSM implementation in order to achieve reasonable load balancing. Notice that the `for` loop at lines (7.1)-(8.1) in Figure 1(b) is not superfluous: the DSM code is written in an SPMD style and is executed on all processors, while the Navigational program represents a single agent that orders the computational steps in its natural sequence, executes all the sequential portions of the computation on the appropriate processor, and orchestrates the execution of the parallel portion of the computation by injecting multiple agents that hop to the appropriate processors and perform their work independently.

Pseudocode for an MP solution of Cholesky factorization, adapted from the implementation presented in

Ref. [6], is presented in Figure 3. Each process executing this code runs a `while` loop (line (2)), with loop index  $q$ , over all local columns this processor owns. A global column index  $k$ , which is the same as the loop index  $k$  in Figures 1(a) and (b), is being computed by all processes (lines (8) and (26)). 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 processor 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 (16)), forward it to its right neighbor if needed (line (20)), and then update its local  $A$  columns (line (24)).

### 3 Comparison of the Solutions

In this section we compare our Navigational Programming solution with the two classical solutions (MP and DSM). We also compare the performance of our implementation with an MPI implementation. All performance data is obtained from SUN Ultra Sparc 1 model 170's with 64MB of main memory, 1GB of virtual memory, and 10Mbps of Ethernet connection. These workstations have a shared file system (NFS). The MPI system we use for performance comparison is LAM 6.3.1/MPI 2 C++ from University of Notre Dame. The mobile agent system used is the MESSENGERS system developed at University of California Irvine [8].

The MP pseudocode differs significantly from the DSM or Navigational Programming pseudocode. In a distributed environment, computation is required to go across machine boundaries, and this creates several problems, which are listed in Table 1 together with their MP and Navigational Programming solutions.

1. In Cholesky factorization, computation of all  $A$  columns is put in a loop over the global index  $k$ . This computation is required to happen across machine boundaries as the columns that are being computed become remote. However, normally processes cannot be shifted across machines because their program counters would become invalid. The MP solution to this problem, is to break the global loop over  $k$  into  $p$  smaller local loops over index  $q$ . With MP, partitioning data means restructuring code. The code lines are regrouped based on the locations where they will be executed. In Figure 3, the two groups are the `if` and `else` blocks. This code restructuring completely changes the temporal order of the original code lines. The Navigational Programming solution, however, is based on a higher level of abstraction. That is, with a mobile agent system, agent threads are able to "jump" across machine boundaries at application level.

Requirements	Problems	MP Solution	Navigational Programming 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

Table 1. Problems and Solutions

- When the locus of computation shifts, some evolving data that is scoped locally to the computation (e.g.,  $k$ ) needs to follow. However, this does not happen automatically. The MP solution is to explicitly transfer, or locally recompute this data. Lines (8) and (26) in Figure 3 are for this recomputing. In Navigational Programming, this type of locally scoped data is identified and carried in agent variables (e.g.,  $k$  is an agent variable). This data transfer in Navigational Programming is implicit because to a mobile agent, although the locally scoped data is carried across machine boundaries, the communication is still intra-thread.
- In a “pure” algorithm developed without considering the details of memory arrangements (shared memory, DSM, or PRAM [9] algorithms can all be taken as pure algorithms in this sense), 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 across machine boundaries. In MP implementation, because the focus of the programmer is placed on local process executing on local data, the global view of data is not preserved at all. A good example is that lines (4), (6), (11), and (24) in Figure 3 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., line (3) in Figure 3), the local data view is mapped to global. In Navigational Programming, we take the opposite approach. Because shared variables stored across distributed memories are “bridged” [3], or logically linked and physically made available with mobile agents, in the Navigational Programming implementation we can preserve the global data view. A global-local data map, as a by-product of the application specific data distribution strategy, is used to help local data accessing with global indexing (e.g., the map `col(.)` in Figure 1(b) serves as the mechanism).

By comparing the solutions of MP and Navigational Programming, we can see why in the MP implementation the code structure is completely altered from the original pure algorithm. All the disadvantages in MP programming are rooted in the fact that MP is at a low level at which

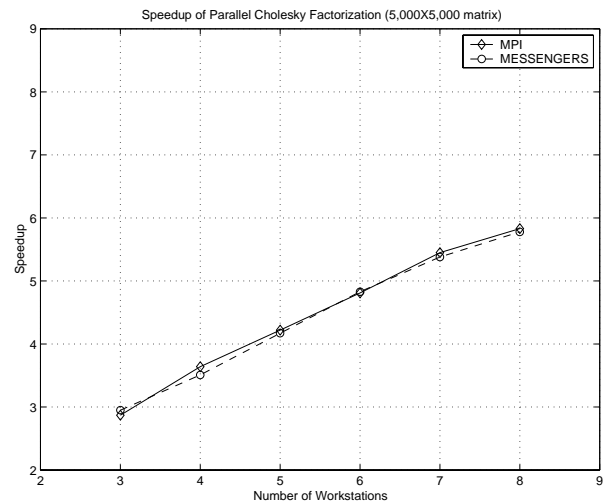


Figure 4. Performance of Parallel Cholesky Factorization.

programmers handle non-trivial details in their handcrafted code. In contrast, the Navigational Programming approach leaves those details of implementation to a compiler which is good at, e.g., restructuring code based on locations.

As far as performance is concerned, the Navigational Programming implementation moves the same amount of data as does the MP program; the communicated data is all the  $G$  columns (loaded in line (12), and sent out by `hop()` statement in line (12.1) in Figure 1(b)). The DSM implementation is likely to create excessive communication if the shared data  $A$  is not distributed in the right way, which is likely to happen since the DSM system does not have the knowledge of the application, and the user of the system does not control where memory is allocated. Furthermore, processor 1 would bring in all the  $A$  columns into local memory for scaling, this is much more expensive, in terms of network use, than the scaling agent hopping to the columns being scaled carrying only small amount of data such as the loop index  $k$ .

Figure 4 shows the speedup data obtained by the Navigational program and MP programs. It is important to observe that the speedup of our implementation 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.

Compared with DSM, two things are explicit in Navigational Programming:

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-processor 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 home free. In fact, as shown in our Cholesky example, for the purposes of, say load balancing, a DSM programmer does need a data accessing strategy, from which constructing the maps is only a small step further.
2. **mobile agent migration:** With Navigational Programming, this is done explicitly using hop() statements, together with explicit data load/unload to/from agent variables. Variable sharing and communication is accomplished by execution of these inserted hop() and load statements. A DSM system provides all these implicitly for DSM programs. 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 to move larger amount of data than needed. In this sense, Navigational Programming provides balanced convenience and flexibility: the code is easy to develop from the original algorithm, and the programmer has full control over exactly when and where data movement happens. Navigational Programming allows the programmer to bridge distributed memory at the application level: by using the aforementioned two maps and the hop() statement, the programmer can view any shared variable allocated in distributed memory as if it were in contiguously shared memory with no boundaries.

## 4 Final Remarks

In this paper, we have applied Navigational Programming to a real-world parallel algorithm, Cholesky factorization. Our Navigational Programming and MPI implementations of the algorithm yield almost identical speedups. This provides evidence that from efficiency point of view, Navigational Programming is as suited for general purpose high performance computing as MP is. One reason for this is that because of the careful implementation of the underlying mobile agent system [10], no code needs to be moved. When the locus of computation migrates, only the program state is moved across the network.

In contrast to MP implementation, the advantage of algorithmic integrity is clearly shown in our Navigational Programming implementation of parallel Cholesky factorization. While the example in this paper is drawn from the domain of numerical analysis, Navigational Programming is a general approach to developing general purpose distributed programs.

## Acknowledgement

The authors wish to thank Koji Noguchi and 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," in *ICPP2001: 30th International Conference on Parallel Processing*. Valencia, Spain: IEEE, Sept. 2001, pp. 77–86.
- [2] L. Pan, L. F. Bic, M. B. Dillencourt, and M. K. Lai, "Mobile agents – the right vehicle for distributed sequential computing," in *HiPC2002: 2002 International Conference on High Performance Computing*, Bangalore, India, Dec. 2002.
- [3] L. Pan, L. F. Bic, and M. B. Dillencourt, "Shared variable programming beyond shared memory: Bridging distributed memory with mobile agents," in *IDPT2002: The Sixth International Conference on Integrated Design and Process Technology*, Pasadena, CA, June 2002.
- [4] L. Pan, L. F. Bic, M. B. Dillencourt, and M. K. Lai, "Navigational programming," Sept. 2002, in preparation.
- [5] I. Pramanick, "MPI and PVM programming," in *High-Performance Cluster Computing: Programming and Applications*, R. Buyya, Ed. Prentice Hall PTR, 1999.
- [6] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 3rd ed. Baltimore: Johns Hopkins University Press, 1996.
- [7] C. Wicke, L. F. Bic, M. B. Dillencourt, and M. Fukuda, "Automatic state capture of self-migrating computations in MESSENGERS," in *MA1998: 2nd International Conference on Mobile Agents*, ser. Lecture Notes in Computer Science, vol. 1477, Sept. 1998, pp. 68–79.
- [8] E. Gendelman, *MESSENGERS User's Manual (version 2.1)*, University of California, Irvine, 2001.
- [9] J. Keller, C. W. Kebler, and J. L. Traff, *Practical PRAM Programming*. New York: Wiley, Dec. 2000.
- [10] E. Gendelman, L. F. Bic, and M. B. Dillencourt, "Fast file access for fast agents," in *5th International Conference on Mobile Agents, MA 2001*, ser. Lecture Notes in Computer Science, G. P. Picco, Ed., vol. 2240. Springer, Dec. 2001, pp. 88–102.