

From Distributed Sequential Computing to Distributed Parallel Computing

Lei Pan, Lubomir F. Bic, Michael B. Dillencourt, and Ming Kin Lai
School of Information and Computer Science
University of California, Irvine, CA 92697-3425, USA
{pan,bic,dillenco,mingl}@ics.uci.edu

Abstract

One approach to distributed parallel programming is to utilize self-migrating threads. Computations can be distributed first, and parallelized second. The first step produces a distributed sequential thread, which can be incrementally parallelized by the second step. This paper prescribes three transformations that turn distributed sequential programs into distributed parallel programs. Real-life examples and performance data are presented, and the advantages of our approach are discussed.

Keywords: *distributed sequential computing, distributed parallel computing, program transformation, parallel Jacobi iteration, parallel Cholesky factorization*

1. Introduction

Navigational Programming (NavP) is the programming of self-migrating threads. Currently, the self-migrating threads we use are in the form of mobile software agents. Explicit *hop()* statements are inserted in the NavP programs in order for the computation (not code) to navigate through the network. The data that needs to follow the agents is put into *agent variables*, whereas the data that is stationary to a node is stored in *node variables*.

Distributed sequential computing (DSC), defined as computing with distributed data using a single locus of computation, plays an important role in NavP. Implemented using a self-migrating thread, a DSC program increases the performance of its non-distributed sequential counterpart on large problems by eliminating disk paging at a cost of efficient network communication, meanwhile preserves *algorithmic integrity* and hence good programmability [1]. Moreover, DSC is not just about sequential programming, it also serves *distributed parallel computing (DPC)*. There are various ways of developing parallel programs, one of which starts from a sequential algorithm and looks for par-

allelism or pipeline opportunities. This strategy is used in solving many parallel matrix computation problems. Starting from a sequential algorithm, our approach consists of two steps: the first converts the sequential algorithm into a DSC program, and the second transforms the DSC program into a DPC program. The question of how to go from DSC to DPC is the focus of this paper. The first step is being studied in our other works [1].

We prescribe three transformations that help to exploit concurrencies in a DSC program and construct a DPC program using multiple DSCs. This is in section 2. We then use these transformations to help solve two real-world problems, namely parallel Jacobi iteration and parallel Cholesky factorization, in section 3. The implementations and performance data are compared with that of MPI programs. In the last section, we discuss the advantages of our approach and outline our future work.

2. From DSC to DPC

A sequential algorithm can be augmented into a DSC program by distributing the data, choosing the right data to put in agent variables, and inserting *hop()* statements [1]. A DSC program is by itself useful to solve large problems in a distributed environment. It can also be incrementally parallelized.

The computations performed by a DSC may be parallelized or pipelined using the three transformations discussed in this section. After one or more transformations, a DSC program is turned into a DPC program employing multiple DSC threads running sequentially or concurrently. Figures 1(a), 2(a), and 3(a) depict sequential computations running in a 2D space with spatial and temporal dimensions. These computations may be executing in loops, which means they could continue to spread along both dimensions (only two nodes are shown in the figures). Each box in the figures represents a computation. The computation represented by a box marked R produces intermedi-

ate result that will be required by the following computation on the next node, whereas the computation represented by a box marked NR is independent of the computation on the next node (i.e., the node on which the computation will continue). The three transformations converting a DSC to a DPC program are:

1. **Transformation I**, depicted in Fig. 1, schedules the computation on the next node as soon as the dependency condition allows it. That is, since in the DSC the computation on the next node only depends on the intermediate result from part of the computation on the current node, as depicted in Fig. 1(a), the thread can clone itself as soon as the computation of R is done, and have the clone hop to the next node carrying the intermediate result and continue the computation at the same time when it continues the computation of NR on the current node (Fig. 1(b)). A hop is represented by a thick line in all figures. A special case is when $R = \emptyset$ in which case the computations on the two nodes are completely independent of each other and can be performed in parallel.

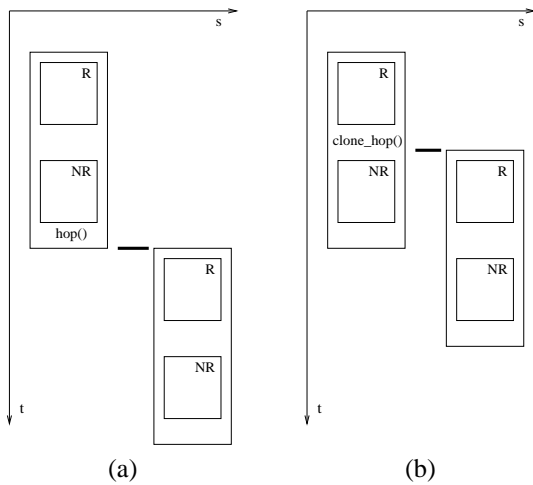


Figure 1. Transformation I. (a) DSC. (b) After transformation.

2. **Transformation II**, depicted in Fig. 2, explores pipeline opportunities. The computation of R1 on the current node only depends on the intermediate result from the computation of R1 on the previous node (Fig. 2(a)). The thread is split into two. The first thread, after performing R1 on the current node, will hop to the next node carrying the intermediate result and continue the computation of R1 there. The second thread will be transformed using either Transformation I (Fig. 2(b)), or Transformation II recursively (i.e., by splitting R2 further). The two threads will need to be synchronized on the nodes. That is, upon finishing its computation, the first thread will need to signal an event to allow

the second thread to move on to its computation. An event is represented by an arrow in Fig. 2(b).

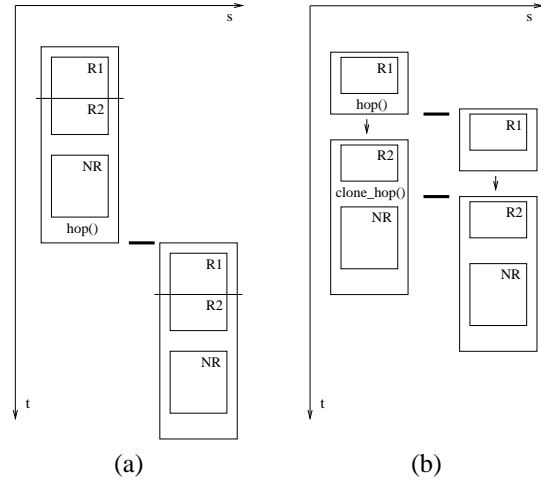


Figure 2. Transformation II. (a) DSC. (b) After transformation.

3. **Transformation III**, shown in Fig. 3, is for parallel reduction. The computations on the two nodes each compute a partial result which is added to or multiplied by the total result (Fig. 3(a)). If computations of the partial results do not depend on each other, they can be performed in parallel, and the total result can be collected by a separate DSC thread (Fig. 3(b)).

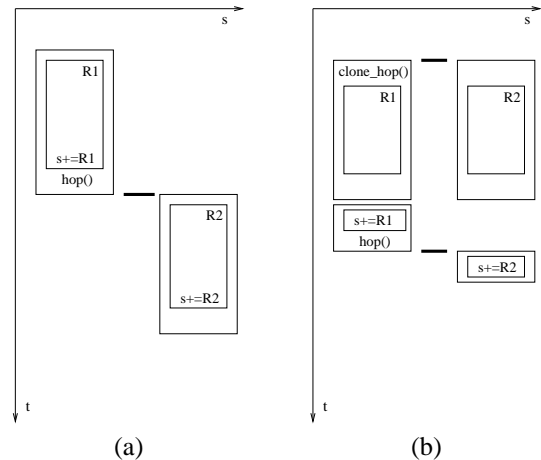


Figure 3. Transformation III. (a) DSC. (b) After transformation.

3. Case Studies

Parallel Jacobi iteration and parallel Cholesky factorization are used in our case studies.

3.1. Parallel Jacobi iteration

Jacobi iteration [2] is an iterative solution scheme used in solving systems of linear equations. Let $Au = f$ 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 $A = D - L - U$, where D is the diagonal of A , and $-L$ and $-U$ are the strictly lower and upper triangular parts of A . We define the Jacobi iteration matrix by $P = D^{-1}(L + U)$, and introduce iteration to express the Jacobi iterative method as: $u^{n+1} \leftarrow Pu^n + D^{-1}f$, where u^n is the solution vector at iteration step n , and $u^0 = \{0\}$. This equation reveals that parallelizing Jacobi iteration is essentially parallelizing matrix-vector multiplication. If we partition the matrix P and the vectors into blocks or sub-vectors, we can rewrite Jacobi iteration in block fashion as:

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

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.

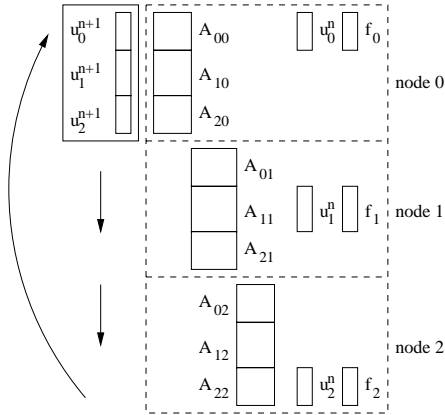


Figure 4. Memory use in DSC Jacobi iteration.

For the DSC implementation, we assume that the square matrix A is vertically partitioned into p “slices,” and each node is allocated with one such slice. p is chosen such that the matrix slices fit into their corresponding local memories completely and hence no disk paging will occur anywhere at any time. The vertical slice on each participating node

is further decomposed (logically but not physically) into p sub-matrices, which become the basic matrix blocks of the block-fashion Jacobi iteration. The vectors holding solutions for the n^{th} and $(n + 1)^{st}$ steps and right-hand-side vector f are also subdivided into p subvectors. These are depicted in Fig. 4 in which $p = 3$ as an example.

With this data partition, the sequential block-fashion pseudocode in Fig. 5(a) can be augmented into the DSC code listed in Fig. 5(b) by inserting two `hop()` statements and properly choosing agent and node variables. The memory use of the DSC code is depicted in Fig. 4, in which each large square box in dashed lines represents the node variable area, while the small box in solid lines represents the agent variable area. The variables $i, j, u_{0:p-1}$, and `err` are agent variables. In Fig. 5, $u_{0:p-1}$ are used to store the result from the previous iteration, and $u_{0:p-1}$ are for the current iteration. They correspond to $u_{0:p-1}^n$ and $u_{0:p-1}^{n+1}$ in Fig. 4, respectively. The arrows depict how the self-migrating thread will hop in the ring linking all the nodes to perform the computation. The `node_map()` shown at lines (7.1) and (12.1) of Fig. 5(b) is a matrix-piece-to-node map.

The migration of the thread brings the small data (i.e., the vector being computed) to be together with the large data (i.e., the matrix slices). In particular, to compute the summation in equation (1), 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. 4), and P_{ij} ’s are computable from A_{ij} ’s, 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 interchange the use of P_{ij} and A_{ij} (i.e., to describe the algorithm correctly we use P_{ij} , but to discuss data accessing pattern we may use A_{ij}).

The DSC computation is depicted schematically in Fig. 6(a), again with $p = 3$. Corresponding to each value of j , the values of u_0, u_1 , and u_2 are computed on a node. Each thick line represents a `hop()` statement. Observe that this DSC program can be transformed to three DSC threads running on the node in a pipeline fashion using Transformation II (with $NR = \emptyset$) discussed in section 2. This transformation is possible because, for example, the computing of u_2 on a node does not depend on the computing of u_0 or u_1 on that node. But rather, it depends on the intermediate result from the computing of u_2 on the previous node. The three DSC threads are depicted in Fig. 6(b). The thread computing u_0 will signal an event (events are represented by arrows in the figure) to allow the thread computing u_1 to start on the same node, before the former hops to the next node. It is further observed that the computing of u_1 or u_2 does not have to all start from node0. In other words, the DSC threads can “shift phase” (imagine, e.g., that u_1 starts on node1 and u_2 starts on node2) without affecting

```

(1)  $err = 1; v_{0:p-1} = \{0\}$ 
(2) while  $err > TOL$ 
(3)    $u_{0:p-1} = \{0\}; err = 0$ .
(4)   for  $j = 0 : p - 1$ 
(5)     for  $i = 0 : p - 1$ 
(6)        $u_i + = P_{ij} v_j$ 
(7)     end for
(8)   end for
(9)   for  $i = 0 : p - 1$ 
(10)     $u_i + = D_{ii}^{-1} f_i$ 
(11)     $err + = \|u_i - v_i\|_2$ 
(12)     $v_i = u_i$ 
(13)  end for
(14) end while

```

(a)

```

(1)  $err = 1; v_{0:p-1} = \{0\}$ 
(2) while  $err > TOL$ 
(3)    $u_{0:p-1} = \{0\}; err = 0$ .
(4)   for  $j = 0 : p - 1$ 
(5)     for  $i = 0 : p - 1$ 
(6)        $u_i + = P_{ij} v_j$ 
(7)     end for
(7.1)    $hop(node\_map((j+1)\%p))$ 
(8)   end for
(9)   for  $i = 0 : p - 1$ 
(10)     $u_i + = D_{ii}^{-1} f_i$ 
(11)     $err + = \|u_i - v_i\|_2$ 
(12)     $v_i = u_i$ 
(12.1)   $hop(node\_map((i+1)\%p))$ 
(13)  end for
(14) end while

```

(b)

Figure 5. Pseudocode for Jacobi iteration. (a) Sequential. (b) DSC.

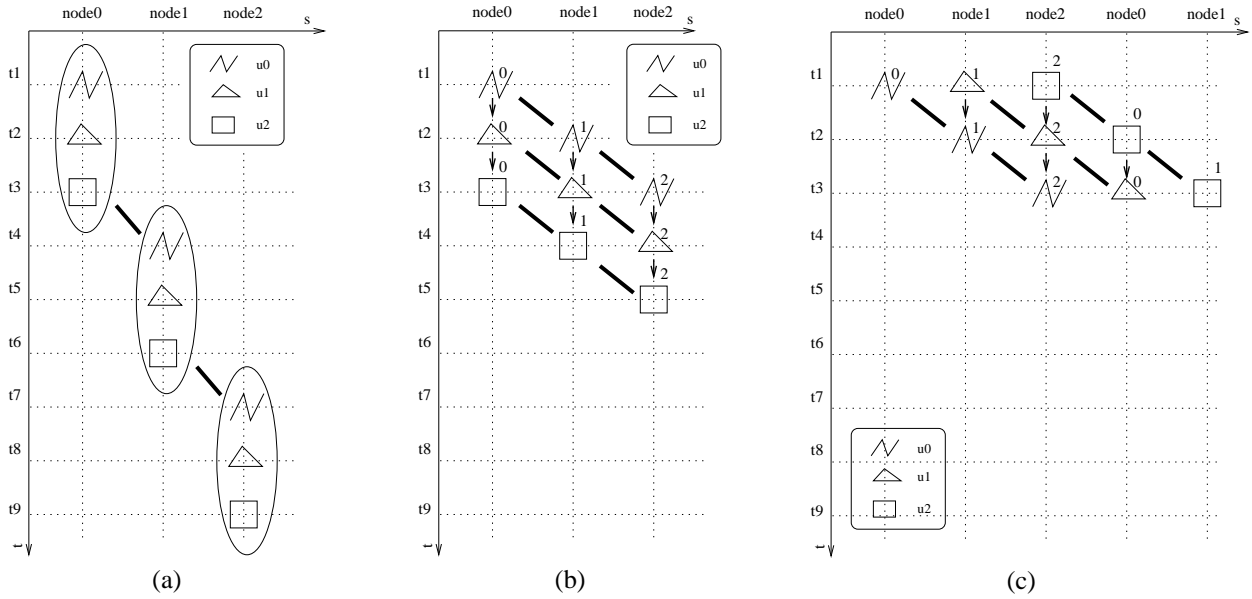


Figure 6. Jacobi iteration. (a) DSC. (b) DPC using pipelined DSCs. (c) DPC using parallel DSCs.

the results. This makes the computations of u_0 , u_1 , and u_2 perfectly parallel, rather than pipelined, as depicted in Fig. 6(c).

The parallel code using multiple concurrent DSCs is listed in Fig. 7. Lines (1.3) to (15) are code for a self-migrating thread named *Jacobi*. In this code, no explicit synchronization using events is needed because the agents use a non-preemptive policy, and no one can pass another to break the sequence of shared variable (i.e., the matrix pieces) accessing. Transformation III is used to reduce the

global error. Each DSC thread collects (line (6)) the partial error computed on a node from the previous iteration (line (11)) as it navigates through the nodes. The partial errors are computed in parallel.

Fig. 8 depicts the memory use for parallel Jacobi iteration. The subvectors u_0 , u_1 , and u_2 are now separated into three agent variables carried by three different agents. The subvectors and matrix blocks marked with the same gray-scales must be together at some stage during the computation, and this is done with the three migrating threads carry-

```

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

(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 for
(10)     $u_\mu += D_{\mu\mu}^{-1} f_\mu$ 
(11)     $r = \|u_\mu - v_\mu\|_2$ 
(12)     $v_\mu = u_\mu$ 
(13)  end while
(14)  end
(15) end

```

Figure 7. Pseudocode for parallel Jacobi iteration using concurrent DSCs.

ing them, hopping among the participating nodes, and computing in parallel. Notice that if the matrix is partitioned and distributed as horizontal slices, the only change needed in the current implementation is to have agent variables carry $u_{0:p-1}^n$, and put $u_{0:p-1}^{n+1}$ to node variables.

3.2. Parallel Cholesky factorization

Cholesky factorization is an algorithm for factorizing symmetric positive definite matrices [3]. 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 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.

The sequential algorithm is listed in Fig. 9(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 (line (2) in Fig. 9(a)). This happens in each iteration over the matrix columns, and has a time complexity of $\Theta(n)$ for each iteration. 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

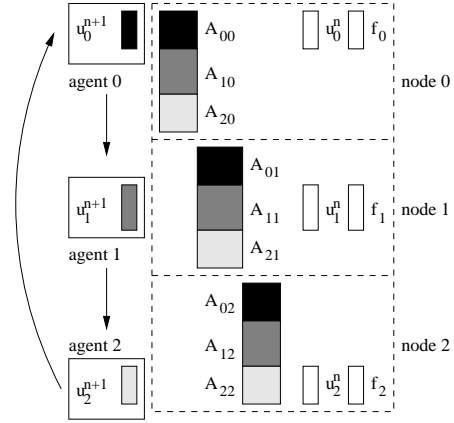


Figure 8. Memory use in parallel Jacobi iteration.

all columns sequentially from left to right ($k = 1 : n$), 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.

We distribute the data (i.e., the matrix A) to the participating nodes. The columns of matrix A are distributed to the nodes in a round robin fashion, in order to achieve better load balancing [3]. If the the final objective is a DSC program, a block fashion should be chosen. In order to transform the sequential algorithm into DSC (code not shown), we can insert two *hop()* statements to the sequential code, one in each loop. The first *hop()*, inserted between lines (1) and (2) in Fig. 9(a), migrates the computation to the node that hosts the column being scaled, whereas the second *hop()*, inserted between lines (7) and (8) in Fig. 9(a), drives the computation to all the nodes in sequence carrying the G column from the latest scaling to perform updating. Before the second *hop()*, one assignment is used to load the most recent G column into an agent variable. Also, the loop indices k and j are stored in agent variables. The execution of the DSC program is visualized in Fig. 10(a), in which the number of nodes p is assumed to be 3. Each thick line represents a hop; a hop can be from a node to itself.

Now we transform the DSC program into a DPC program. The updatings on different nodes are independent of each other; but they all depend on the previous scaling, as depicted by the arrows in Fig. 10(a). Therefore, Transformation I (with $R = \emptyset$) can be applied to the DSC. After the transformation, concurrent updating DSCs are employed after each scaling step, as shown in Fig. 10(b).

```

(1) for  $k = 1 : n$ 
(2)    $A(k : n, k) / = \sqrt{A(k, k)}$ 
(3)
(4)   updating ( $k, n$ )
(5)
(6) end for
(7) updating (int  $k$ , int  $n$ )

(8)   for  $j = k + 1 : n$ 
(9)      $A(j : n, j) - = A(j : n, k)A(j, k)$ 
(10)   end for
(11) end

```

(a)

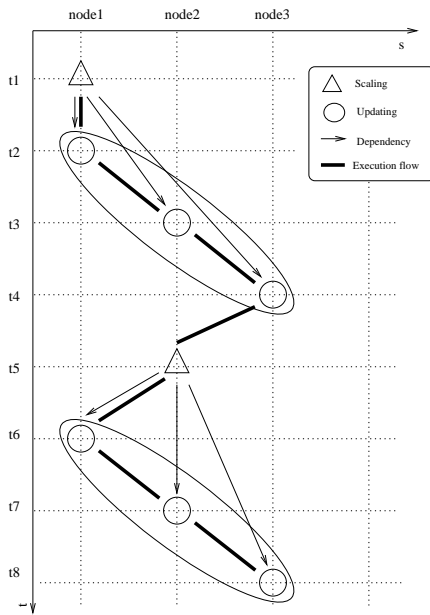
```

(1) for  $k = 1 : n$ 
(2)    $A(k : n, col(k)) / = \sqrt{A(k, col(k))}$ 
(3)
(3.1) for  $\mu = 1 : p$ 
(4)   inject(updating ( $\mu, k, n$ ))
(4.1) end for
(5)
(5.1) hop (node_map( $k + 1$ ))
(5.2) waitEvent (Evt,  $k + 1$ )
(6) end for
(7) updating (int  $\mu$ , int  $k$ , int  $n$ )
(7.1)  $v_{loc}(k + 1 : n) = A(k + 1 : n, col(k))$ 
(7.2) hop (node_map( $k + \mu$ ))
(7.3) waitEvent (Evt,  $k$ )
(8)   for  $j = k + \mu : p : n$ 
(9)      $A(j : n, col(j)) - = v_{loc}(j)v_{loc}(j : n)$ 
(10)   end for
(10.1) signalEvent (Evt,  $k + 1$ )
(11) end

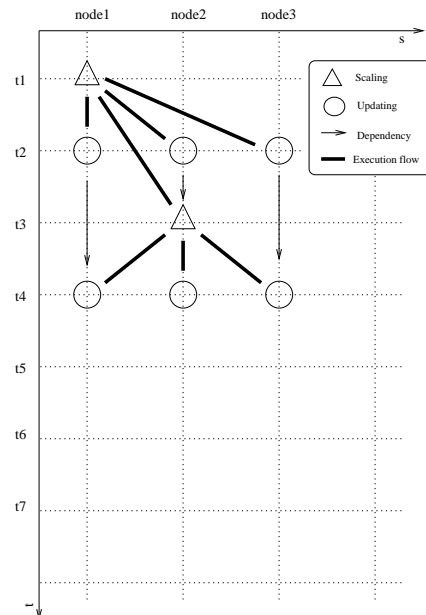
```

(b)

Figure 9. Pseudocode for Cholesky factorization. (a) Sequential. (b) DPC using concurrent DSCs.



(a)



(b)

Figure 10. Cholesky factorization. (a) DSC. (b) DPC employing concurrent DSCs.

The DPC program is listed in Fig. 9(b). There are two types of composing DSC programs: a single scaling DSC named *Scaler* (with code lines (1)–(6)), and multiple updating DSCs named *Updaters* (with code lines (7)–(11)). *Scaler* carries the loop index k , an agent variable, that

loops through all columns of matrix A . On the k^{th} iteration, *Scaler* scales column k (line (2)). 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 (3.1)–(4.1)), and then it hops to the node that owns the next column of A (line (5.1)). 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 (7.1)), and then hops to the appropriate node (line (7.2)). 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 of A (line (9)). Two maps are used in the DPC code (lines (2), (5.1), (7.1), (7.2), and (9)). 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. The two types of DSCs, namely scaling and updating, interact with each other in the DPC program using only local injections or events. There are three lines of code that use $signalEvent()$ and $waitEvent()$ primitives (lines (5.2), (7.3), and (10.1)) for synchronizing threads that are running on the same node. The semantics of these primitives is described in the MESSENGERS manual [4]. After *Scaler* executes the $inject()$ command at line (4), it hops away immediately, and then the injected *Updaters* start executing (line (7.1)). 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 are used to make sure that the dependency relations are being respected. In particular, $waitEvent()$ at line (5.2) makes *Scaler* wait until the *Updater* working on the same node finishes updating A and signals an event Evt at line (10.1). $waitEvent()$ at line (7.3) makes sure that all the *Updaters* from earlier iterations have finished, so the current *Updater* can start.

The MPI code adapted from reference [3] is listed Fig. 11. It is obvious that the MPI code exhibits a huge departure from the original code structure. The two nested loops are both broken. The outer **for** loop over k is broken into smaller **while** loops over all the local columns that a node owns. Each process executing this code runs the **while** loop (line (2)) with loop index q . A global column index k , which is the same as the loop index k in Fig. 9(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 at 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 at line (3) is false, this process will receive the new

```

(1)  $k = 1; q = 1; col = \mu : p : n; L = length(col)$ 
(2) while  $q \leq L$ 
(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 if
(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 for
(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 if
(21) for  $i = q : L$ 
(22)  $r = col(i)$ 
(23)  $A_{loc}(r : n, i) - = g_{loc}(r)g_{loc}(r : n)$ 
(24) end for
(25)  $k = k + 1$ 
(26) end if
(27) end while

```

Figure 11. Pseudocode for parallel Cholesky factorization using MPI.

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)). The inner **for** loop over j in Fig. 9(a) is broken into two each appearing in an **if** or an **else if** code block. These smaller loops each update the local columns that a node owns. The access to the new G column is now through passed messages, rather than through shared variables in the sequential or NavP programs. Furthermore, as the **for** loop over j is broken down into smaller loops for each node, this data communication is relocated out of the loops, resulting in a fairly different and complicated termination condition (lines (16)-(18)).

3.3. Performance data

We have implemented the two examples. The mobile agent system used was the MESSENGERS system developed in the School of Information & Computer Science, University of California Irvine. The message passing system used was LAM 6.3.1/MPI 2 C++. The performance data was obtained from SUN Ultra Sparc 1 model 170's with 64MB of main memory, 1GB of virtual memory, and 10Mbps of Ethernet connection. Fig. 12 and Fig. 13 show the speedup

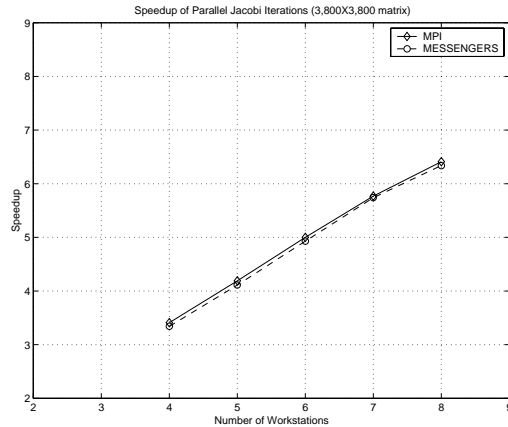


Figure 12. Performance of parallel Jacobi iteration.

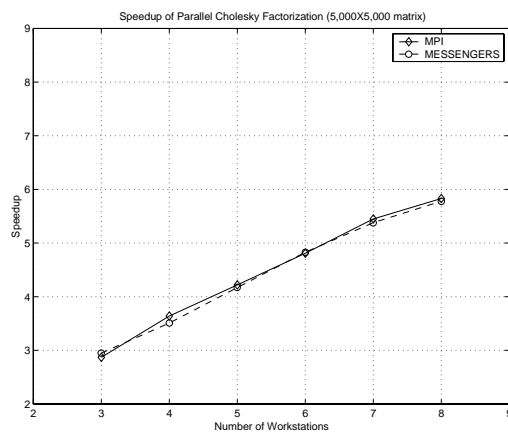


Figure 13. Performance of parallel Cholesky factorization.

obtained from running the DPC and the MPI programs. The speedups of our DPC programs are almost the same as that of the MPI programs, and their trends as the number of machines increases are the same which indicates same scalability. In order to make NavP practically useful, the *hop()* statements in our mobile agent code must be very efficient. This is addressed in our earlier work [5], and the key idea is to avoid moving code as the locus of computation migrates.

4. Final Remarks

Exploiting parallelism in the dependency graph of a sequential algorithm is a relatively straightforward way of obtaining parallel programs. The novelty of our approach lies in the fact that we employ self-migrating threads to cover the computations in the graph respecting the dependency

relations. Our approach consists of two steps: the first goes from a sequential algorithm to a distributed sequential program using self-migrating threads, the second transforms the DSC program into a DPC program composed from multiple concurrent DSC threads. In this paper, we provide three transformations that can be applied in the second step.

Our approach has several advantages. First, both the DSC and DPC programs preserve algorithmic integrity. This avoids unnecessary changes in code structure. In contrast, MPI code often exhibits considerable unnecessary departure from the original algorithm, in the form of reordered lines of code or broken loops. This is clearly seen in the Cholesky factorization example. In the cases when parallel algorithms that do not resemble their sequential origins must be developed to achieve better performance, our approach helps to ensure that the implementations (i.e., code) resemble the parallel algorithms. Second, the DPC programs are as efficient and scalable as their MPI counterparts. This is because our thread migration (i.e., hops) is made almost as efficient as message passing. Third, our approach provides incremental parallelization, which is a weak point of message passing. Fourth, since manual transformation from sequential algorithms to parallel programs is made easy with the use of DSC programs as an intermediate step, one can naturally expect that tools can be easily built to automate the transformations. Developing such tools will be part of our future work.

References

- [1] L. Pan, L. F. Bic, and M. B. Dillencourt, "Distributed sequential computing using mobile code: moving computation to data," in *Proceedings of the 2001 International Conference on Parallel Processing (ICPP 2001)*, L. M. Ni and M. Valero, Eds. Los Alamitos, Calif.: IEEE Computer Society, Sept. 2001, pp. 77–84.
- [2] W. L. Briggs, *A Multigrid Tutorial*. Philadelphia, Pa.: Society for Industrial and Applied Mathematics, 1987.
- [3] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. Baltimore, Md.: Johns Hopkins University Press, 1996.
- [4] E. Gendelman, *MESSENGERS User's Manual (version 2.1)*, Information & Computer Science, University of California, Irvine, 2001.
- [5] E. Gendelman, L. F. Bic, and M. B. Dillencourt, "Fast file access for fast agents," in *MA 2001: 5th International Conference on Mobile Agents*, Atlanta, GA, Dec. 2001, pp. 88–102.