Mobile Pipelines: Parallelizing Left-Looking Algorithms Using Navigational Programming

Lei Pan[1], Ming Kin Lai[2], Michael B. Dillencourt[2], and Lubomir F. Bic[2] ⋆

[1] Jet Propulsion Laboratory,
California Institute of Technology, Pasadena, CA 91109-8099, USA
Lei.Pan@jpl.nasa.gov
[2] Department of Computer Science,
Donald Bren School of Information & Computer Sciences,
University of California, Irvine, CA 92697-3425, USA
{mingl,dillenco,bic}@ics.uci.edu

Technical Report 05-12

July 22, 2005

Abstract. Parallelizing a sequential algorithm—i.e., manually or automatically converting it into an equivalent parallel distributed algorithm—is an important problem. Ideally, the parallel algorithm should preserve the computational structure of the original sequential algorithm, display a high degree of parallelism, have low communication overhead, and be scalable. The difficulty of accomplishing this for a particular sequential algorithm depends on the nature of the algorithm and the specific model of parallelism under consideration. Generally, the so called "right-looking" algorithms are easy to parallelize because they tend to have a significant amount of data parallelism. In these algorithms, data is "eagerly" propagated to and consumed by subsequent computations immediately after it is produced, and so it does not need to be kept in temporary storage for extended periods of time.
In this paper we consider the class of "left-looking" sequential algorithms [1], which cannot be easily parallelized using the message-passing or distributed shared memory models. These algorithms are characterized by "lazy" propagation of data. We show that these algorithms can be directly parallelized using the Navigational Programming model. We present performance data demonstrating the effectiveness of our approach.

1 Introduction

The right- and left-looking formulations of a class of algorithms (e.g., LU or Cholesky factorization) can be represented schematically [1] as shown in Fig. 1, in which $B1()$ and $B2()$ are blocks of code lines that update the entries of an array. In the right-looking algorithm listed in Fig. 1(a), the newly updated entry i at line (2) is "eagerly" (i.e., immediately) consumed by the computations of the inner loop to update entries $j = i + 1 \,..\, N$. The computations of different j entries are dependent on entry i but independent of each other, thus making the inner loop data parallel and parallelizable using data-parallel language constructs such as *forall* or *doall*. In contrast, in the left-looking algorithm shown in Fig. 1(b), the incorporation of entry i in the inner loop is delayed ("lazy") until the actual computation of entry j needs it. The inner loop now carries dependence and the algorithm hence admits only pipeline parallelism.

⋆ The authors gratefully acknowledge the support of a U.S. Department of Education GAANN Fellowship.

```
(1)  for i = 1 to N
(2)    B1(i)
(3)    for j = i + 1 to N
(4)      B2(i, j)
(5)    end for
(6)  end for
```

(a)

```
(1)  for j = 1 to N
(2)    for i = 1 to j − 1
(3)      B2(i, j)
(4)    end for
(5)    B1(j)
(6)  end for
```

(b)

Fig. 1. (a) Right-looking code; (b) Left-looking code

A simple contrived left-looking algorithm is shown in Fig. 2(a). The j^{th} iteration of the outer loop, which computes $a[j]$, requires the values of $a[i]$ computed by all the previous iterations. Because of this dependence, the algorithm does not exhibit data parallelism, so it cannot be parallelized directly using *forall* or *doall*. In this particular case, the algorithm can be parallelized by first transforming it by switching the order of loop nesting. This works because switching the order of loop nesting changes the algorithm from a "left-looking" algorithm, in which the $a[i]$'s are not consumed until they are actually needed, to a "right-looking" algorithm, in which the $a[i]$'s are aggressively pushed through the computation. However, the general problem of converting a left-looking algorithm to an equivalent right-looking algorithm is not always completely straightforward. A symbolic analysis technique for verifying the legality of program transformations is available, but there is no known transformation sequence to convert one to another [1]. In this paper we examine an alternative approach to

```
(1)  for j = 2 to N

(2)    for i = 1 to j − 1

(3)      $a[j] \leftarrow j * (a[j] + a[i])/(j + i)$
(4)    end for

(5)    $a[j] \leftarrow a[j]/j$
(6)  end for
```

(a)

```
(1)    for j = 2 to N
(1.1)    hop(node_map[j]); $x \leftarrow a[l[j]]$
(2)      for i = 1 to j − 1
(2.1)      hop(node_map[i])
(3)        $x \leftarrow j * (x + a[l[i]])/(j + i)$
(4)      end for
(4.1)    hop(node_map[j]); $a[l[j]] \leftarrow x$
(5)      $a[l[j]] \leftarrow a[l[j]]/j$
(6)    end for
```

(b)

Fig. 2. Pseudocode for a simple algorithm. (a) Sequential; (b) DSC using NavP

parallelizing left-looking algorithms: rather than converting them to right-looking algorithms, we parallelize the original code directly, thus preserving the integrity of the original sequential algorithm. As we show, this can be done quite easily using the paradigm of Navigational Programming (NavP), in which multiple migrating threads carry out the computation. In this model, computations are programmed to migrate among the processors. They follow the locations of large-sized data, while carrying along small-sized data. The individual migrating computations generally follow each other, thus forming a **Mobile Pipeline**. Figure 3 illustrates the principle by comparing a conventional (stationary) pipeline with a mobile pipeline. In the figure, $C1$, $C2$, and $C3$ are computations, and a, b, c, d, and e are the data being computed. In a conventional pipeline, $C1$, $C2$, and $C3$ are stationary, whereas in a mobile pipeline they migrate. The essence of our NavP approach is to use Distributed Sequential Computing (DSC) [2] threads to construct mobile pipelines to exploit pipeline parallelism

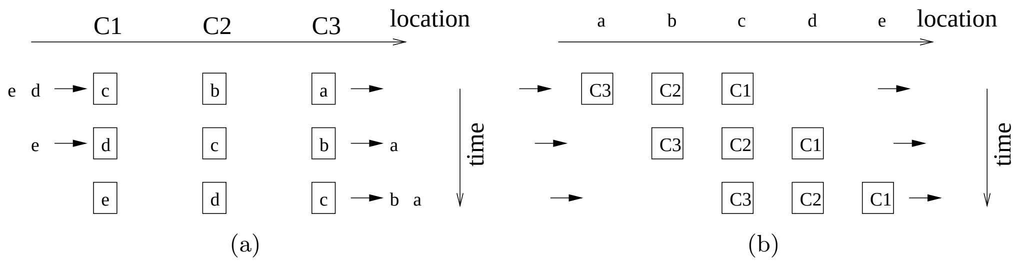


Fig. 3. The comparison of two pipelines. (a) Conventional; (b) Mobile

in the left-looking algorithms. The NavP view naturally describes efficient distributed algorithms, with regular or irregular communication patterns, using code that is structurally the same as the original sequential algorithm [3]. If we attempt to directly parallelize the code using either a Distributed Shared Memory (DSM) or Message Passing (MP) paradigm, we find that we either have to use considerably more memory—enough that the solution is no longer scalable—or asymptotically increase the communication cost. The reason why NavP is superior for this problem can be summarized as follows, in the context of Fig. 2(a). We can think of the computation of $a[j]$ as being a pipeline of $j-1$ stages, with the i^{th} stage being the incorporation of the value of $a[i]$. If each pipeline is stationary and remains on one processor, then each needed value of $a[i]$ must, at some point during the execution of the pipeline, be on that processor. If the values are stored there permanently, additional memory is required; if they are stored there temporarily, additional communication is required. The NavP solution, in contrast, avoids this problem by having a moving pipeline visit the necessary data, so that no element of the array needs to be replicated or re-communicated.

We discuss this in more detail in Sect. 2. In Sect. 3, we discuss the results of applying the same pipelining technique to a numerical kernel, Crout factorization. We present performance data in Sect. 4, and we conclude by discussing some related work and some final remarks.

2 A Simple Example

In this section, we discuss and analyze the parallelization of the sequential algorithm introduced in Sect. 1. To make the discussion more concrete, assume that the parallel computation is being performed on P processors, each of which stores N/P array entries. In Sect. 2.1 we describe a NavP implementation that requires a communication cost of $O(N \cdot P)$ communications and $O(N/P)$ memory on each processor. In Sects. 2.2 and 2.3, we show that any direct parallel implementation of the sequential algorithm using either MP or DSM either requires $\Omega(N^2)$ communication cost or requires $\Omega(N)$ memory usage on at least one processor. The first case represents an asymptotic increase in communication cost, whereas the second essentially requires that the entire input array be stored on one processor, which is not a scalable solution if the number of processors is large.

2.1 NavP Solution

In NavP, we use multiple self-migrating threads to carry out computations for distributed parallel computing. We insert statements of the form *hop*(*dest_node*) into sequential codes to provide computation mobility. The threads carry data to remote nodes using thread-private variables, and they communicate with each other using shared node variables (stationary on a node, and shared by all threads that currently reside on that node). Concurrent self-migrating threads residing on the same node use events, with *signalEvent*() and *waitEvent*(), to synchronize with each other. This is necessary because the daemon on each node uses multiple threads to handle communication and computation. NavP is essentially distributed concurrent shared variable programming. It provides a different view of parallel distributed computing [3] from the classical SPMD (Single Program Multiple Data) view.

Our NavP approach uses the MESSENGERS [4–6] migrating thread environment. This parallel execution environment is efficient because:

1. Code is loaded from a Network File System (NFS) disk or a remote location only once. We do not move code every time the computation locus revisits a node in the network: all we need to move is the current state of the computation.
2. We keep the cost of state book-keeping small, so that the cost of moving the state of the computation in NavP is only slightly more than sending a comparably-sized message in Message Passing Interface (MPI). More precisely, each migration of a thread in the MESSENGERS system requires, in addition to the state information, a control block of about 220 bytes. This overhead is negligible as long as the granularity and the message size are reasonably large. This is borne out by the performance data in Sect. 4 and previous performance studies of our MESSENGERS system.
3. We use user-level multi-threading, and the computations and communications done by the migrating threads are efficiently scheduled to fully utilize the CPU cycles. This is achieved by a queuing mechanism that is mostly transparent to the programmers.

In the NavP approach, the parallelization of a given sequential algorithm proceeds in two steps. The first step is referred to as DSC (Distributed Sequential Computing) [2]. In this step, we start with a data distribution pattern. We then insert *hop*() statements in the sequential code so that the computation follows the data it accesses through the network. The resulting DSC program is a distributed program, but with a single locus of computation.

Figure 2(b) lists the DSC code of the simple example. Three *hop*() and load/unload compound statements are inserted (at lines (1.1), (2.1), and (4.1)). Code structure is not changed. In the pseudocode, x is a thread-private variable that is available to the thread wherever it hops, and $a[.]$ is a distributed shared variable that is logically one big array but physically a distributed collection of sub-arrays. The auxiliary array $node\_map[.]$ provides the node ID of a given array entry, and $l[.]$ contains the local array index of an entry with a given global index. The DSC code works for arbitrary data distribution (e.g., block, cyclic, or block cyclic data distribution).

The next step of NavP is called DPC (Distributed Parallel Computing). In this step, transformations are used to cut the long DSC computation thread into several shorter ones. Each of these threads are "pushed up" or scheduled to run as early as possible, subject to the constraint that all dependences must be respected. These threads spread out parallel computations as they hop out to the remote nodes on the network. The DPC implementation of the example is listed in Fig. 4(b). Each computation of j becomes a thread that is "injected" or spawned by another thread running the outer loop of j (lines (1), (1.2), and (6)). The code for each thread, lines (1.1) through (4.1), remains almost the same as the DSC code listed in Fig. 4(a). The only difference is the insertion of two new lines of event handling, to synchronize the accesses to the entry $a[1]$. Each thread waits at line (2.2) until the previous thread is done accessing $a[1]$, and at line (3.1) it notifies all other threads on the node that it has finished accessing $a[1]$. In this way, the threads organize themselves into a pipeline when they access $a[1]$: the thread computing $a[j]$ runs immediately after the thread computing $a[j-1]$. Because MESSENGERS uses non-preemptive FIFO scheduling, and because threads hopping from the same source node to the same destination node preserve their ordering, the pipeline remains

```
(1)   for j = 2 to N




(1.1)   hop(node_map[j]); x ← a[l[j]]
(2)     for i = 1 to j − 1
(2.1)     hop(node_map[i])

(3)       x ← j ∗ (x + a[l[i]])/(j + i)

(4)     end for
(4.1)   hop(node_map[j]); a[l[j]] ← x
(5)     a[l[j]] ← a[l[j]]/j
(6)   end for
```

(a)

```
(0.1)signalEvent(evt, 1)
(1)   for j = 2 to N
(1.2)   inject(entry_proc(j))
(6)   end for

(1.0)Thread entry_proc(j)
(1.1)   hop(node_map[j]); x ← a[l[j]]
(2)     for i = 1 to j − 1
(2.1)     hop(node_map[i])
(2.2)     if (i = 1) waitEvent(evt, j − 1)
(3)       x ← j ∗ (x + a[l[i]])/(j + i)
(3.1)     if (i = 1) signalEvent(evt, j)
(4)     end for
(4.1)   hop(node_map[j]); a[l[j]] ← x
(5)     a[l[j]] ← a[l[j]]/j
(5.1)end Thread
```

(b)

Fig. 4. Pseudocode for a simple algorithm. (a) DSC using NavP; (b) Pipelining using NavP

intact throughout the entire computation: migrating threads do not pass each other in the mobile pipeline. Each computation migrates through the pipeline, progressively visiting the successive stages (the elements $a[i]$ that it successively incorporates into its computation). Note that the code works correctly irrespective of how the array $a[.]$ is distributed.

There are three advantages to building a mobile pipeline: (1) In programming a DSC, we follow the principle of pivot-computes. This principle says that in resolving a DBlock (a block of code accessing data distributed over multiple machines, such as the inner loop in the simple example), the computation should occur where the largest amount of data is, so that a small amount of data is carried to meet with a large amount of data rather than the other way around. This node is called the pivot node. In the present example, this principle says that the computation of $a[j]$ should happen on the nodes that host the $a[i]$'s. As the computation of $a[j]$ proceeds, the pivot node changes. Assigning the computation of $a[j]$ statically to any single node would cost more than our DSC does because it requires more data communication. (2) We use concurrent threads to explore parallelism. For algorithms that exhibit pipelining opportunities, we simply insert multiple DSC threads to form a mobile pipeline, and synchronize them using events. All synchronization events are local to a node and hence efficient. This is possible because threads are not allowed to access data remotely. (We note that when data parallelism is present, we can use multiple concurrent DSC threads to exploit this data parallelism [6], but this is not the focus of this paper.) (3) The NavP code as listed in Fig. 2(b) and Fig. 4(b) work for arbitrary data distribution. All that changes is the contents of the *node_map*[.] and $l[.]$ arrays. This provides tremendous flexibility, because the programmer can experiment with different data distribution patterns using exactly the same code. For better performance, we can use a block algorithm (listed in Fig. 5(a)) so the granularity is coarse. Figure 5(b) shows the details of the block function $F()$ (called in line (3) of Fig. 5(a)). The functions $start(J, I)$ and $end(J, I)$ choose the right starting and ending loop indices for the cases of $I = J$ and $I < J$, respectively. The block pipeline code is similar to the code listed in Fig. 4(b) and therefore omitted.

Asymptotically, if this algorithm is run on P processors, each processor will hold N/P array entries. The thread that starts on processor k (for $k = 1, \ldots, P$) will hop to processor 1, then processor 2, and so forth, ending up at processor k, for a total of k hops. On each hop it will carry N/P array entries. Hence the total communication costs of all the threads is

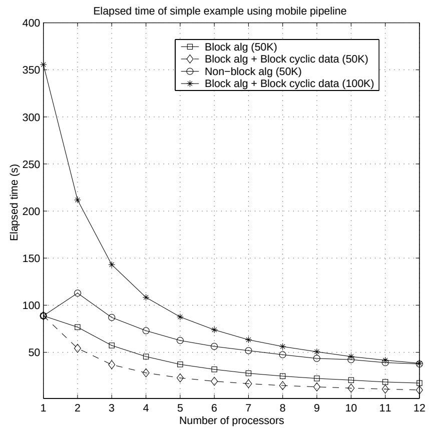
```
(1)   for J = 1 to num_blocks
(1.1) hop(node_map[J]); load blk J to x[]
(2)     for I = 1 to J − 1
(2.1)     hop(node_map[I])
(3)       Call F(I, J, a, x)
(4)     end for
(4.1) hop(node_map[J])
(5)     Call F(J, J, x, x); unload x[] to blk J
(6)   end for
```

(a)

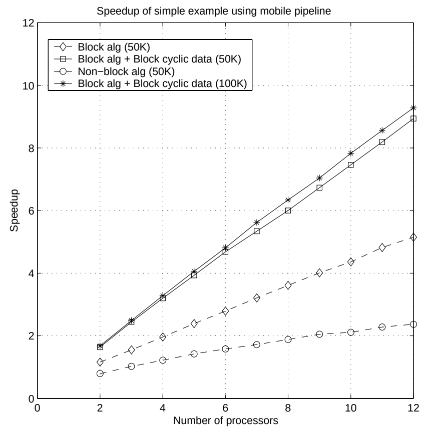
```
(1)Procedure F(I, J, a, x)
(2)  for j = start(J, I) to end(J, I)
(3)    for i = start(I, J) to end(I, J)
(4)      x[l[j]] ← j ∗ (x[l[j]] + a[l[i]])/(j + i)
(5)    end for
(6)    a[l[j]] ← a[l[j]]/j
(7)  end for
(8)end Procedure
```

(b)

Fig. 5. Pseudocode for a simple algorithm (block fashion). (a) DSC; (b) The function F()



(a)



(b)

Fig. 6. Performance of the simple problem. (a) Elapsed time; (b) Speedup

$N/P \cdot \sum_{k=1}^{P} k$, which is $O(N \cdot P)$ as stated at the beginning of Sect. 2. Since on any particular processor, a thread hops away as the next thread is executing, the additional storage required on each processor is $O(N/P)$. We note that a slight modification at startup time is required to achieve this space requirement. The memory allocation for each thread (not shown in the pseudocode) is performed after the hop at line (1.1) of Fig. 4(b). In this fashion, we limit the memory use at processor 1 to $O(P)$, which is $O(N/P)$ provided $P = O(\sqrt{N})$. Alternatively, the injection loop at line (1) can be periodically interrupted to ensure that no more than $O(N/P)$ threads are active at processor 1 at any given time. We can further improve performance by using a block cyclic data distribution. This allows all the processors in the pipeline to get involved in the computation earlier and hence increases parallelism. As shown in Fig. 6, the block algorithm and block cyclic data distribution both help improve performance dramatically.

2.2 Shared Memory Solution

On shared memory or DSM, we can use *doacross* to handle cross-iteration synchronization and hence explore pipeline parallelism [7]. The pseudocode is listed in Fig. 7(b).

If we consider this solution carefully, we find, as stated at the beginning of Sect. 2, that if we follow the original sequential algorithm we either need asymptotically more storage or

```
(1)   for j = 2 to N
(2)     for i = 1 to j − 1

(3)       a[j] ← j ∗ (a[j] + a[i])/(j + i)
(4)     end for
(5)     a[j] ← a[j]/j

(6)   end for
```

(a)

```
(0)   post(EV(1))
(1)   doacross j = 2 to N
(2)     for i = 1 to j − 1
(2.1)     wait(EV(i))
(3)       a[j] ← j ∗ (a[j] + a[i])/(j + i)
(4)     end for
(5)     a[j] ← a[j]/j
(5.1)   post(EV(j))
(6)   end doacross
```

(b)

Fig. 7. Pseudocode for a simple algorithm. (a) Sequential; (b) Shared memory

asymptotically more communication than with the NavP solution. (For simplicity, we assume a block data distribution.) Indeed, suppose each processor stores only N/P additional array entries. Consider the computation for the data on the processor holding the k^{th} block. For each value of j such that $a[j]$ resides on that processor, it will be necessary to fetch each of the $k-1$ previous blocks. Since the original sequential algorithm is being followed, the computation of $a[j]$ is completed before the computation of $a[j+1]$ is started. This means that these $k-1$ blocks must be fetched in the computation of each of the N/P values of $a[j]$ stored on this processor (i.e., each block must be fetched N/P times). Hence the total communication cost of fetching all the blocks that processor k needs is $(N/P)^2 \cdot k$. Summing over k, we find that the total communication cost is $(N/P)^2 \cdot \sum_{k=1}^{P} k$, which is $\Omega(N^2)$.

We note that these repeated fetches can be avoided if we cache the entire array on each processor, but this would require having more than $\Omega(N/P)$ storage on each processor. (Indeed, some of the processors would need $\Omega(N)$ storage, which makes it a non-scalable solution.) The repeated fetches can also be avoided if, when we fetch a block, we propagate the values in the block into the computation of all the values of $a[j]$ being computed on the current processor; however, this represents a change in the algorithm from left-looking to right-looking.

2.3 Message Passing Solution

The pipelining approach underlying the NavP implementation can be directly implemented using Message Passing, using a more conventional pipelining architecture in which the processes are stationary and the elements of the array are passed between processes as they are computed. For each value of j, the process responsible for computing $a[j]$ receives the values $a[1], \ldots, a[j-1]$ and uses them to compute $a[j]$.

The problem with this approach is that there is a mismatch between the flow of data and the direction of the computation. Suppose, for example, that a particular process is responsible for computing the two consecutive values $a[j]$ and $a[j+1]$. It first computes $a[j]$, using the values $a[1], \ldots, a[j-1]$. After this computation is complete, it then computes $a[j+1]$, using the values $a[1], \ldots, a[j]$. So the question arises: what does it do with the values $a[1], \ldots, a[j-1]$ between the two computations? One possibility is to store them locally, which requires storage proportional to N rather than N/P: this is not a scalable solution. A second possibility is to discard the values after the first computation and receive them again for the second computation; this increases the total amount of communication required over the running time of the algorithm to $\Omega(N^2)$. The only remaining possibility is to change the order of computation. This amounts to converting the original left-looking sequential algorithm to an equivalent right-looking algorithm.

2.4 Mimicking the NavP Solution Using Message Passing

Since any distributed programming paradigm, including Navigational Programming, ultimately relies on the exchange of messages between machines, it should be possible to translate our implementations directly into a message-passing system such as MPI. However, this does not appear to be a straightforward task.

We discuss the issues with the MP solution at two different levels of granularity: a fine grained level where each processing element (PE) hosts only one entry of the array and a coarse level which is a more realistic implementation for a cluster computer. (1) **Fine granularity level.** There may be several ways to write an MP program from a sequential algorithm, but here we will make a view change from NavP to SPMD to derive our fine-grained MP program, listed in Fig. 8(b). The idea is to "stay" at a stationary location (e.g., a PE that hosts the entry $a[i]$), "observe" the phenomenon (i.e., the $a[j]$'s "flying over" the location and computing using $a[i]$), and record the phenomenon in code. Since the entry $a[i]$ will see all the entries $a[j], j = i + 1$ *to* N, "stop by" and compute using its value, we can use a loop (lines (7)–(15)) to describe this. The *Recv*() at line (9) or (11) tells us where the $a[j]$ is from and the *Send*() at line (14) defines where the $a[j]$ should be going after computing using $a[i]$ at line (13). Of course, the entry $a[i]$ itself needs to be "mature" before it can "serve" the computation of the $a[j]$'s. This is done at lines (1)–(6), in which line (3) sends $a[i]$ to the PE that hosts $a[1]$, line (4) receives it back from this PE's left neighbor, and line (5) scales the value down using the value of i. When $i == 1$ (lines (1)–(2)), all these do not happen because $a[1]$ is "mature." Because this implementation is fine grained, the local array index is always 1. In the pseudocode in Fig. 8(b), *node*[.] is a map from a global array index to the hosting PE ID.

```
(1)  for j = 2 to N

(2)    for i = 1 to j − 1

(3)      a[j] ← j ∗ (a[j] + a[i])/(j + i)
(4)    end for

(5)    a[j] ← a[j]/j
(6)  end for
```

(a)

```
(1)  i ← rank
(2)  if (i! = 1)
(3)    Send(a[1], node[1])
(4)    Recv(x, node[i − 1])
(5)    a[1] ← x/i
(6)  end if

(7)  for j = i + 1 to N
(8)    if (rank == node[1])
(9)      Recv(x, node[j])
(10)   else
(11)     Recv(x, node[i − 1])
(12)   end if

(13)   x ← j ∗ (x + a[1])/(j + i)
(14)   Send(x, node[i + 1])
(15) end for
```

(b)

Fig. 8. Pseudocode for the simple algorithm. (a) Sequential; (b) Pipelining using MP (fine granularity level)

The derivation from NavP code to MP code described above is called a "card dropping" transformation, in which all the "cards" representing computations "drop" at a stationary observation point (i.e., a PE) as the migrating computations pass by and the cards are "popped out of the card stack" and used to write the corresponding MP code. An outgoing hop is converted to a *Send*() statement, and an incoming hop is made a *Recv*() statement.

```
for j = 2 to N by 3
   for i = 1 to j − 1 by 2
      a[j] ← j ∗ (a[j] + a[i])/(j + i)
   end for
   a[j] ← a[j]/j
end for
```

(a)

```
i ← rank
if ((i − 2)%3 == 0)
   Send(a[1], node[1])
   Recv(x, node[left_i(i)])
   a[1] ← x/i
end if

if ((i − 1)%2 == 0)
   for j = right_j(i) to N by 3
      if (rank == node[1])
         Recv(x, node[j])
      else
         Recv(x, node[left_i(i)])
      end if

      x ← j ∗ (x + a[1])/(j + i)
      Send(x, node[min(right_i(i), j)])
   end if
end for
```

```
// the 1st i value to this PE's left
int Function left_i(int i)
   i1 ← i − 1
   while ((i1 − 1)%2 != 0)
      i1 ← i1 − 1
   end while

   return i1
end Function

// the 1st i value to this PE's right
int Function right_i(int i)
   i1 ← i + 1
   while ((i1 − 1)%2 != 0)
      i1 ← i1 + 1
   end while

   return i1
end Function

// the 1st j value to this PE's right
int Function right_j(int i)
   i1 ← i + 1
   while ((i1 − 2)%3 != 0)
      i1 ← i1 + 1
   end while

   return i1
end Function
```

(b)

Fig. 9. Pseudocode for the simple algorithm modified. (a) Sequential; (b) Pipelining using MP (fine granularity level)

It is observed that the MP code is structurally different from the original sequential code. Indeed, if we compare the two codes in Fig. 8(a) and (b), we notice that the j loop has become the inner loop in the MP code from the outer loop in the sequential code. Another interesting observation is that if we change the original loops slightly into $for\ j = 2\ to\ N\ by\ 3$ and $for\ i = 1\ to\ j - 1\ by\ 2$ respectively, the NavP programs will continue to work without any change other than the new loop increments, but the MP code in Fig. 8(b) requires considerable modification. Specifically, a PE can host $a[k]$ with k being an "i value," or a "j value," or both, or neither. These four different "card stacks" will require more if statements to describe in the MP code. And a PE needs to compute the "roles" of its neighbors in order to send/recv to/from the right PEs. Figure 9(b) lists a possible MP solution. A small perturbation in the original code could result in a large deviation in the MP implementation. This indicates that the SPMD view is not ideal for manual as well as automatic code transformations.

The MP code listed in Fig. 8(b) or 9(b) is a parallel program. From the viewpoint of debugging, the DSC to the DPC steps in NavP separate the two concerns of distributed computing and concurrent computing. Therefore, debugging NavP programs is conceivably easier than debugging the equivalent MP programs.

(2) **Coarse granularity level.** The following are some of the difficulties that arise in developing a coarse-grained MP program (we do not consider a block fashion algorithm here):

Adapting to arbitrary data mapping: In the NavP computation, the threads initially organize themselves into a pipeline as follows: the thread computing $a[j]$ starts on the node where all threads are injected, hops to the node containing $a[j]$, and then hops to the node containing $a[1]$, where the threads then synchronize their references to $a[1]$ using events as discussed in Sect. 2.1. In an analogous MP computation, each process must order the requests for all the $a[j]$'s that it is responsible for in increasing order of j (a partial ordering) and send them to the node containing $a[1]$. The process responsible for $a[1]$ must then store these requests and process them in ascending order of j (a total ordering) to start the pipeline.

Termination: In an MP implementation, each stationary process must anticipate how many messages it will receive, so it knows when it has processed them all. In the NavP implementation, each thread can terminate when it has successfully computed and stored the value of $a[j]$ for which it is responsible.

Self-sending: In MP, when the source and the destination are the same processes, special handling is necessary to bypass TCP/IP for efficiency. In the environment we have, we saw huge overhead using self-sending. So we had to use a queue in local memory. In NavP, a "hop" to the same machine is made a no-op.

Deadlock: If the PE hosting $a[1]$ posts $Recv()$'s in an order that is different from the actual order in which the $a[j]$'s are sent, system buffer could overflow causing deadlock. One possible solution is to post all the $Recv()$'s (in the non-blocking form of $MPI\_Irecv()$ so the control will return immediately after the calls) before any $Send()$ is posted. This is not shown in the pseudocode in Fig. 10, and it requires some code modification. An alternative solution is to use multi-threading so that the three tasks of receiving incoming data, processing it, and sending it on can be intermixed. To do this, an MPI environment that fully supports MPI_THREAD_MULTIPLE [8] such as Open MPI [9] is needed, and additional modification to the pseudocode is required to separate the three tasks and assign them to different threads. MESSENGERS provides multi-threading and therefore deadlock is not a problem in NavP.

With some difficulty, the above issues can be addressed. Figure 10 lists a preliminary pseudocode of an MP implementation in order to show the amount of work involved. In the pseudocode, the array $oa[.]$ provides the aforementioned (ascending) partial ordering, and the array $g[.]$ maps a local index of array $a[.]$ into a logical global index. In the MP pseudocode, we have to introduce a queuing mechanism for total ordering and self-sending. Such a mechanism is also needed in the NavP implementation, but it is factored out from the application level code and put into the MESSENGERS daemon. Threads wait in a daemon queue for their turns to be executed. If a thread posts a $signalEvent(evt)$ call but the event evt is not signaled, the thread will be put back to the queue.

3 Crout Factorization

Crout factorization is a convenient variant of Gauss elimination [10]. Figure 11(a) lists the sequential Crout algorithm. This version factorizes symmetric matrices. For simplicity, we assume that the matrix being factorized, K, is a dense matrix. This particular algorithm is left-looking because the updating of the j^{th} column uses all the columns to its left from 1 to $j-1$.

3.1 DSC

Figure 11(b) lists the pseudocode of DSC Crout factorization. Three $hop()$ and load/unload compound statements are inserted at lines (1.1), (2.1) and (4.1). The columns of matrix K are distributed to the nodes in a block fashion. Each and every column is the basic unit of

```
// send to the node that hosts a[1]
recv_cnt $\leftarrow$ 0
for l1 = 1 to local_cnt
    l2 $\leftarrow$ oa[l1];  j $\leftarrow$ g[l2]
    recv_cnt $\leftarrow$ recv_cnt + N - j
    if (j != 1)
        recv_cnt $\leftarrow$ recv_cnt + 1
        if (rank != node_map[1])
            Send(a[l1], j, node_map[1])
        end if
    else
        recv_cnt $\leftarrow$ recv_cnt - 1
        enqueue(Q[rank], a[l1], j, 1)
    end if
end for

// compute recv_cnt
l2 $\leftarrow$ oa[1]
g2 $\leftarrow$ g[l2]
for l1 = 2 to local_cnt
    l3 $\leftarrow$ oa[l1];  g3 $\leftarrow$ g[l3]
    if (g2 + 1 == g3)
        recv_cnt $\leftarrow$ recv_cnt - (N - g3 + 1)
    end if
    g2 $\leftarrow$ g3
end for

// the node that hosts the 1st entry
if (rank == node_map[1])
    for l1 = 2 to N
// check if any queue has this entry
        l1_flg $\leftarrow$ 0
        for l2 = 1 to num_nodes
            if (Q[l2] not empty)
                if (l1 found)
                    dequeue(Q[l2], x, j, i)
                    Call request_proc(i, j, x)
                    l1_flg $\leftarrow$ 1
                    break
                end if
            end if
        end for
// not in the queues, so receive
        if (l1_flg == 0)
            while (recv_cnt > 0)
                recv(x, j, ANY_SRC)
                src_rank $\leftarrow$ the sender's rank
                recv_cnt $\leftarrow$ recv_cnt - 1
                i $\leftarrow$ 1
// if received not the one, enqueue it
                if (l1 != j)
                    enqueue(Q[src_rank], x, j, i)
                else
                    break
                end if
            end while
            Call request_proc(i, j, x)
        end if
    end for
end if

// receive until all msgs received
while (recv_cnt > 0)
    recv(x, j, i, ANY_SRC)
    recv_cnt $\leftarrow$ recv_cnt - 1
    Call request_proc(i, j, x)
end while


Procedure request_proc(i, j, x)
    for l1 = i to j - 1
        if (rank == node_map[l1])
            x $\leftarrow$ j * (x + a[l[l1]])/(j + i)
        else
            break
        end if
    end for

    if (l1 == j And rank == node_map[j])
        a[l[j]] $\leftarrow$ x/j
    else
        send(x, j, l1, node_map[l1])
    end if
end Procedure
```

Fig. 10. Pseudocode for MP implementation of the simple algorithm

data distribution and is hence indivisible. In the pseudocode, *node*[.] provides the node ID of a given column. In the real code, the matrix K is implemented as a 1-D array, and the map $l[.,.]$ from a global index pair $[i,\ j]$ to a local 1-D index is needed. The detail is omitted in the pseudocode. For performance of this code, please see Sect. 5.1.

3.2 Pipelined DPC

Figure 11(d) lists the pseudocode for a pipelined DPC implementation. This is compared side by side with the sequential code re-written with procedure calls, where the inner loop becomes a procedure, listed in Fig. 11(c). Each loop j is now assigned to a thread, and the outer loop becomes a "spawner" thread. In addition to the *hop*() compound statements, two event handling statements are inserted at lines (5.2) and (6.1). Similar to the simple example in Sect. 2, we utilize the FIFO scheduling of Messengers so that the event handling only happens on the node that hosts column 1 of K.

This pipelined NavP code works for "any data distribution" in the sense that the columns of K can be distributed in any fashion to the nodes. Similar to the simple example, we use block cyclic column distribution to exploit parallelism.

```
(1)   For $j = 1..N$
(2)     For $i = 1..j-1$
(3)       $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(4)     End For
(5)     For $i = 1..j-1$
(6)       $T \leftarrow K_{ij}$
(7)       $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(8)       $K_{jj} \leftarrow K_{jj} - T K_{ij}$
(9)     End For
(10)  End For
```

(a)

```
(1)    For $j = 1..N$
(1.1)    hop (node[j])
(1.2)    load (column j)
(2)      For $i = 1..j-1$
(2.1)      hop (node[i])
(2.2)      load ($K_{ii}$)
(3)        $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(4)      End For
(4.1)    hop (node[j])
(4.2)    unload (column j, $K_{ii}$[ ])
(5)      For $i = 1..j-1$
(6)        $T \leftarrow K_{ij}$
(7)        $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(8)        $K_{jj} \leftarrow K_{jj} - T K_{ij}$
(9)      End For
(10)   End For
```

(b)

```
(1)   For $j = 1..N$
(2)     call col_proc(j)
(3)   End For

(4)   Procedure col_proc(int j)
(5)     For $i = 1..j-1$
(6)       $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(7)     End For
(8)     For $i = 1..j-1$
(9)       $T \leftarrow K_{ij}$
(10)      $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(11)      $K_{jj} \leftarrow K_{jj} - T K_{ij}$
(12)    End For
(13)  End Procedure
```

(c)

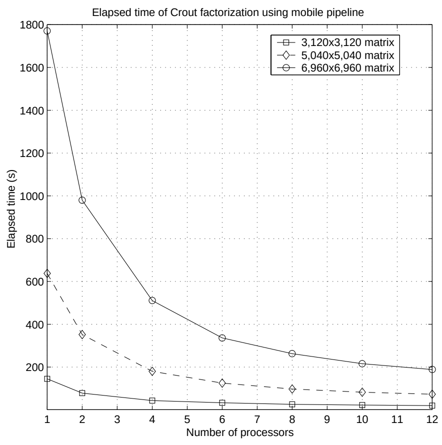
```
(1)     For $j = 1..N$
(1.1)     hop (node[j])
(2)       inject (col_proc(j))
(3)     End For

(4)     Agent col_proc(int j)
(4.1)     load (column j)
(5)       For $i = 1..j-1$
(5.1)       hop (node[i])
(5.2)       waitEvent(evt, i)
(5.3)       load ($K_{ii}$)
(6)         $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li} K_{lj}$
(7)       End For
(7.1)     hop (node[j])
(7.2)     unload (column j, $K_{ii}$[ ])
(8)       For $i = 1..j-1$
(9)         $T \leftarrow K_{ij}$
(10)        $K_{ij} \leftarrow \frac{T}{K_{ii}}$
(11)        $K_{jj} \leftarrow K_{jj} - T K_{ij}$
(12)      End For
(12.1)  signalEvent(evt, j)
(13)    End Agent
```

(d)

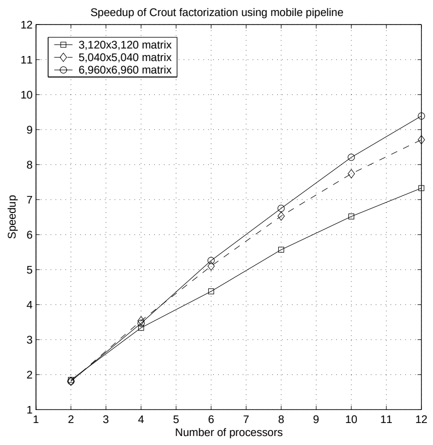
Fig. 11. Pseudocode for Crout factorization. (a) Sequential; (b) DSC using NavP; (c) Sequential re-written (with procedure call); (d) Pipelining using NavP

4 Performance

Performance data for the simple example is presented in Fig. 6, and for Crout factorization is in Fig. 12 and Table 1. This was obtained from SUNW Ultra-60's with 450 MHz UltraSPARC-II CPU, 256MB of main memory, 1GB of virtual memory, and 100Mbps of Ethernet connection with collision-free switch. These machines use NFS.



(a)



(b)

Fig. 12. Performance of Crout factorization. (a) Elapsed time; (b) Speedup

Table 1. Performance of Crout factorization

Matrix Order	3120		5040		6960	
Num Proc	Time (s)	Speedup	Time (s)	Speedup	Time (s)	Speedup
Sequential						
1	145.22	1.00	637.69	1.00	1770.74	1.00
MESSENGERS						
2	78.33	1.85	351.79	1.81	980.00	1.81
4	43.51	3.34	180.15	3.54	510.81	3.47
6	33.16	4.38	125.01	5.10	336.39	5.26
8	26.05	5.57	97.63	6.53	262.48	6.75
10	22.29	6.52	82.42	7.74	215.69	8.21
12	19.80	7.33	73.21	8.71	188.63	9.39

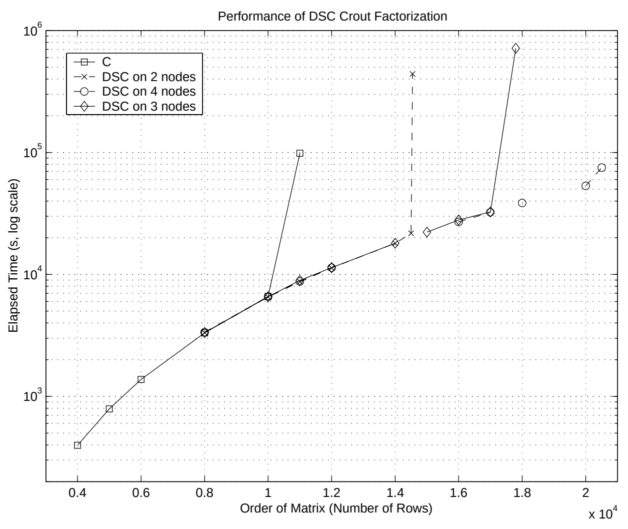


Fig. 13. Performance of DSC Crout factorization

In this paper, we used non-block implementations of Crout algorithm for the sake of simple presentation. Even though the sequential implementation is not the fastest possible (as block algorithms usually have better cache performance), our parallel implementation does not use block algorithm either. This makes the speedup numbers a fairly good indication of the efficiency and scalability of our approach. We are unable to find a parallel Crout implementation in literature with reasonable search efforts. For performance comparison we compare our speedup numbers with those of the Cholesky factorization implementation in ScaLAPACK [11] and find that they are comparable [6]. We are comparing the performances of two different algorithms based on the observation that they are two variants of LU decomposition and have asymptotically the same complexity.

5 Related Work

5.1 Distributed Sequential Computing

We originally introduced DSC as an end in itself, as a way of improving the performance of sequential programs by reducing paging and thrashing, trading this improvement against a modest cost in communication [2]. In this earlier work, we were able to solve very large problems using DSC on a network that had a total collective memory of roughly the size of the working set of the problem. The performance data from Fig. 13 represents a considerable improvement [2] – the communication overhead in DSC is no longer significant, and the DSC program performs as if it were a sequential run with all needed memory available on a single machine. The improvement is partially due to improved network speed (from 10Mbps to 100Mbps) and partially due to an improved version of Messengers. In this paper, we focus on DSC as a means to an end: we use DSC threads to compose parallel programs.

5.2 Pipelined Computations

Pipelining is a well-known technique for parallelizing sequential computations. It is achieved by dividing a task into a sequence of smaller tasks, each of which is executed on a piece of

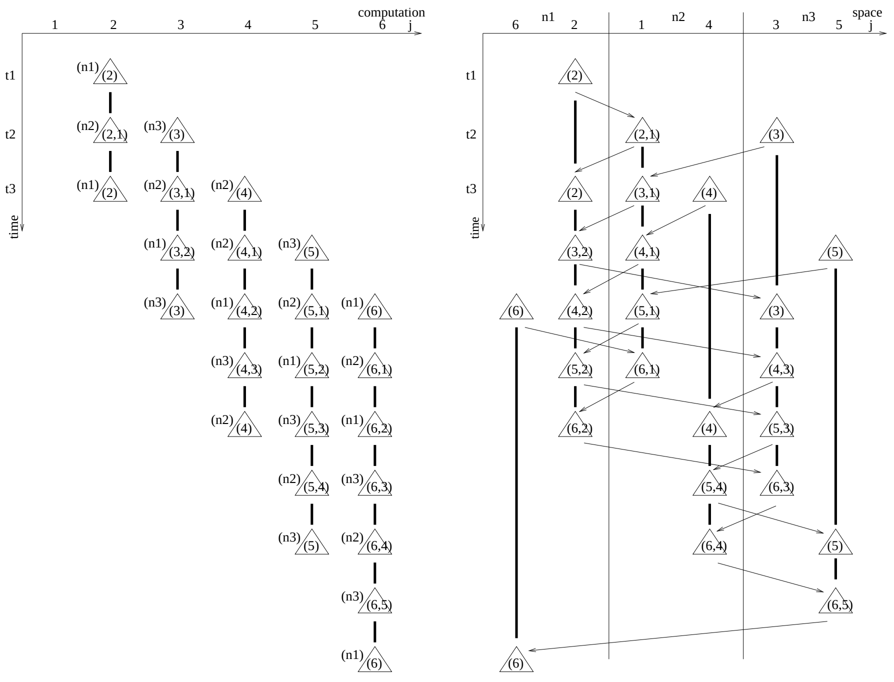


Fig. 14. The simple problem in two different views. (a) NavP; (b) SPMD

hardware that operates concurrently with the other stages of the pipeline. Successive tasks are streamed into the pipe and get executed in an overlapped fashion with the other subtasks [12]. A recent survey [13] describes three situations in which sequential computations can benefit from pipelining. The examples discussed in this paper fall into the situation when "a series of data items must be processed, each requiring multiple operations."

However, the method discussed [13] for achieving parallelism using pipelining in situations of this type is not directly applicable to these examples. This is because it only considers regular data distribution and assumes that all data items reside on the first node initially. (Note that this second assumption is non-scalable.) Our examples address the above two details; and once these details are considered, MP programming becomes significantly harder.

A classical pipeline is the segmentation of a functional unit into different parts, each of which is responsible for partial execution of an operation. The concept of pipelining is similar to that of an assembly line process in a factory. In contrast, a mobile pipeline operates like farmland work. That is, what is needed to be done (e.g., weeding, watering, or harvesting) is fully carried by a mobile pipeline of different equipment (e.g., tractors or harvesters) following each other. A mobile pipeline also carries small-sized data (e.g., seeds or fertilizers) that is needed when it carries out the operations to the large-sized data (i.e., the soil).

5.3 The Two Views in Distributed Computing

There are two views of distributed computation; the SPMD view describes distributed computations at stationary locations, whereas the NavP view describes a computation following the movement of its locus [3]. Our simple example from Sect. 2 looks very different in the two views. A pattern that may look quite irregular and mysterious in the SPMD view is quite regular in the NavP view. This is illustrated in Fig. 14. The array $a[.]$ is distributed as follows: $a[6]$ and $a[2]$ are on node $n1$, $a[1]$ and $a[4]$ are on node $n2$, and $a[3]$ and $a[5]$ are on node $n3$. This distribution is shown in Fig. 14(b), where the horizontal axis represents spatial locations (i.e., computer nodes) with the array entries assigned to them. Each triangle with a pair (j, i) indicates the computation to update entry $a[j]$ with $a[i]$. The arrows represent the messages being sent, and the solid lines describe execution flows. In the NavP view shown in Fig. 14(a), the horizontal axis describes the computations of entry $a[j]$, and the number in parentheses to the left of each triangle provides the node that the computation of $a[j]$ needs to hop to in order to access $a[i]$. We can easily see, for example, that computing $a[3]$ requires hopping to node $n2$ to access $a[1]$, then to node $n1$ to access $a[2]$, and finally to node $n3$ to store the value $a[3]$. Under the NavP view, our program effectively describes a skewed nested loop using the original sequential code structure. In contrast, the SPMD view shown on the right obscures this simple description.

An important step in distributed parallel programming is code transformation. In MP programming, the $Send()$ and $Recv()$ statements are inserted into the code. A $Send()$ statement behaves similarly to a *goto* statement; the execution jumps directly to the code line right after the corresponding $Recv()$ statement. The difference is that this jump is in the space dimension rather than the time dimension, i.e., across the network. If the space dimension is trivial (i.e., in the context of a uniprocessor), one needs to replace a $Send()/Recv()$ pair with a *goto* statement such that the *goto* replaces the $Send()$ and the label replaces the $Recv()$, in order to restore the execution flow. This suggests that the $Send()$ and $Recv()$ statements are as harmful to use as the *goto* statement. Gorlatch [14] made the same claim and proposed a solution of using collective operations (e.g., broadcast, reduction, scattering, and gathering) as substitute for $Send()$ and $Recv()$.

It is generally believed that with the use of *goto* it becomes very easy to produce inconsistent, incomplete and unmaintainable spaghetti code. Dijkstra called for abolishing the *goto* statement in programming languages in 1968 [15]. Since then the use of *goto* has been widely discouraged and replaced by the use of language constructs such as if, $while$, or the exceptional handling clauses, even though there are situations when structured use of *goto* is possible [16].

In contrast to the $Send()$ and $Recv()$ statements, a $hop()$ statement does not change code structure. Using Dijkstra's terminology, it does not change the successive (in time) action as described by the original successive (in text space) action descriptions (i.e., code lines), even though the spatial location of the successive (in time) action is changed. NavP puts an end to the use of "goto" in distributed programming.

6 Final Remarks

We have shown that NavP can be used to parallelize a class of sequential programs, namely left-looking programs, that are difficult to parallelize using conventional methods. There are several reasons why it is useful to be able to do this directly, as opposed to first turning the sequential algorithm into right-looking and then parallelizing. One reason is *algorithmic integrity* [2]: because the parallel algorithm is similar to the original sequential algorithm, it is easier to understand and hence to maintain or modify. A second reason is performance: a sequential program may have been carefully crafted to make effective use of the cache, for example. The closer the parallel algorithm is to the sequential algorithm the more likely it is to preserve such performance enhancements that were in the original sequential code. It was pointed out [1] that converting between a left-looking and a right-looking algorithm may have a significant effect on performance. Yet another reason is that making a minor modification to the original left-looking program may totally invalidate the parallel implementation if it relies on the corresponding right-looking algorithm. For example, suppose after we parallelized the right-looking version of the program in Fig. 2(a), we decide to insert the line $a[i] \leftarrow a[i] + a[j]/1000.0$ between line (3) and line (4) in Fig. 2(a) for numerical purposes (e.g., faster convergence for an iterative solver). This new line changes data dependency relationship, so the resulting program is neither left-looking nor right-looking, and it cannot be easily converted to a purely right-looking algorithm anymore. So the right-looking-based parallel program is no longer useful. However, our parallel implementation using mobile pipelines remains unchanged, other than adding this one line to the code of the composing DSC thread. More precisely, all we need to do is inserting this line $a[l[i]] \leftarrow a[l[i]] + x/1000.0$ below line (3) in Figs. 2(b) and 4(b).

In the NavP implementations of the left-looking algorithms, initially all migrating threads hop to one node. This situation itself may seem to be a source of low performance, but it is efficiently handled using multi-threading and non-preemptive scheduling. That is, if a thread arrives too early, it will be put to sleep in a queue (via a *waitEvent*() call) to avoid wasting CPU cycles; but when it is a thread's turn to compute, it will finish its computation and hop out to the next node without being interrupted. After hopping out of the first node, the migrating threads organize themselves into a mobile pipeline, effectively balancing the computational load among the processors.

Our approach can be used for a wide variety of data distributions and adapts automatically to changes in data distribution as long as we update the *node_map*[.] and l[.] arrays which are byproducts of data distribution. This is useful for situations where data distribution pattern is unknown at compile time (e.g. in Grid computing).

This NavP approach is highly mechanical: it requires insertion of *hop*()s and insertion of events. The former uses the knowledge of data distribution, and the latter is based on dependency analysis. Code transformations are incremental and code structures remain essentially the same throughout the process. Potentially, much or all of these transformations can be automated by a compiler. This is part of our future research.

References

1. Menon, V., Pingali, K.: Look left, look right, look left again: An application of fractal symbolic analysis to linear algebra code restructuring. International Journal of Parallel Programming **32** (2004) 501–523
2. Pan, L., Bic, L.F., Dillencourt, M.B.: Distributed sequential computing using mobile code: Moving computation to data. In Ni, L.M., Valero, M., eds.: Proceedings of the 2001 International Conference on Parallel Processing (ICPP 2001), Los Alamitos, Calif., IEEE Computer Society (2001) 77–84
3. Pan, L., Bic, L.F., Dillencourt, M.B., Lai, M.K.: NavP versus SPMD: Two views of distributed computation. In Gonzalez, T., ed.: Proceedings of the Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems. Volume 2, Algorithms., Anaheim, Calif., ACTA Press (2003) 666–673
4. Department of Computer Science, University of California, Irvine Irvine, Calif.: MESSENGERS User's Manual (Version 1.2.04). (2005) `http://www.ics.uci.edu/~messengr/messengersC/messman1_2_04.ps`.
5. Wicke, C., Bic, L.F., Dillencourt, M.B., Fukuda, M.: Automatic state capture of self-migrating computations in MESSENGERS. In Rothermel, K., Hohl, F., eds.: Proceedings, Second International Workshop on Mobile Agents, MA '98. Volume 1477 of Lecture Notes in Computer Science., Berlin, Germany, Springer-Verlag (1998) 68–79
6. Pan, L., Lai, M.K., Noguchi, K., Huseynov, J.J., Bic, L.F., Dillencourt, M.B.: Distributed parallel computing using navigational programming. International Journal of Parallel Programming **32** (2004) 1–37

7. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann, San Francisco, Calif. (2002)
8. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W., Snir, M.: MPI–The Complete Reference. Volume 2, The MPI-2 Extensions. The MIT Press, Cambridge, Mass. (1998)
9. Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary (2004) 97–104
10. Hughes, T.J.R.: The Finite Element Method : Linear Static and Dynamic Finite Element Analysis. Prentice Hall, Englewood Cliffs, N.J. (1987)
11. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. Society for Industrial and Applied Mathematics, Philadelphia, Pa. (1997)
12. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Solving Linear Systems on Vector and Shared Memory Computers. Society for Industrial and Applied Mathematics, Philadelphia, Pa. (1991)
13. Wilkinson, B., Allen, M.: Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. 2 edn. Pearson Prentice Hall, Upper Saddle River, N.J. (2005)
14. Gorlatch, S.: Send-receive considered harmful: Myths and realities of message passing. ACM Transactions on Programming Languages and Systems **26** (2004) 47–56
15. Dijkstra, E.W.: Go to statement considered harmful. Communications of the ACM **11** (1968) 147–148
16. Knuth, D.E.: Structured programming with go to statements. ACM Computing Surveys **6** (1974) 261–301