

Mobile Pipelines: Parallelizing Left-Looking Algorithms Using Navigational Programming

Lei Pan^{1,2}, Ming Kin Lai², Michael B. Dillencourt², and Lubomir F. Bic² *

¹ Jet Propulsion Laboratory,
California Institute of Technology, Pasadena, CA 91109-8099, USA

Lei.Pan@jpl.nasa.gov

² Donald Bren School of Information & Computer Sciences,
University of California, Irvine, CA 92697-3425, USA
{pan,ming1,dillenco,bic}@ics.uci.edu

Abstract. We consider the class of “left-looking” sequential matrix algorithms: consumer-driven algorithms that are characterized by “lazy” propagation of data. Left-looking algorithms are difficult to parallelize using the message-passing or distributed shared memory models because they only possess pipeline parallelism. We show that these algorithms can be directly parallelized using mobile pipelines provided by the Navigational Programming methodology. We present performance data demonstrating the effectiveness of our approach.

1 Introduction

In computational science, array-based algorithms (e.g., matrix factorization algorithms) are sometimes classified as “right-looking” or “left-looking” algorithms [1]. In both cases, the array is scanned from left to right. Right-looking algorithms are producer-driven: at each stage, the algorithm performs computations on the current element, and then immediately performs updates to the elements to the right of the current element. The fundamental data flow is eager propagation to the right, or scattering, as illustrated in Fig. 1(a). Left-looking algorithms, in contrast, are consumer-driven: at each stage, the algorithm updates the current element using previously computed values of elements to its left, after which the algorithm performs computations on the newly updated current element. Here the fundamental data flow is gathering previously computed data from the left, as illustrated in Fig. 1(b). Skeleton right-looking and left-looking algorithms are shown in Fig. 2(a) and Fig. 2(b), respectively. In a matrix application, each element $x[i]$ would be a matrix column.

Right-looking algorithms are easy to parallelize directly. In each iteration of the outer loop in Fig. 2(a), the iterations of the inner loop are independent of one another and hence can be parallelized using data-parallel constructs such as *doall* or *forall*. In contrast, the inner loop in Fig. 2(b) carries dependence and hence admits only pipelined parallelism.

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

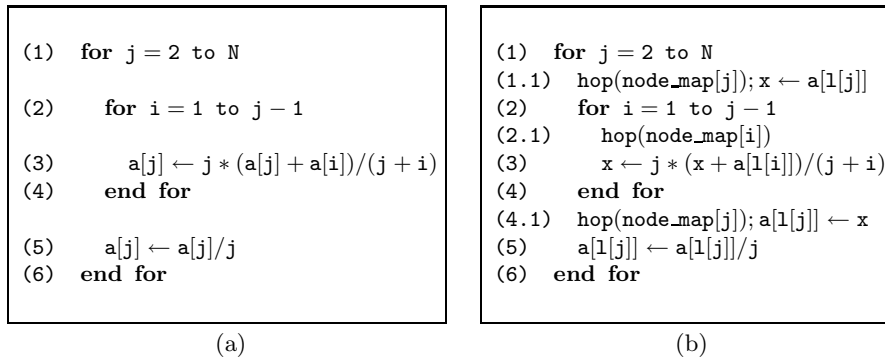


Fig. 3. Pseudocode for a simple algorithm. (a) Sequential; (b) DSC using NavP

A third, very important, reason for not transforming a left-looking algorithm to a right-looking algorithm is performance: a sequential program may have been carefully crafted to make effective use of the cache or a particular data layout. 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. Experimental evidence in [2] shows that converting between a left-looking and a right-looking algorithm can have a significant effect on performance.

In this paper we examine an alternative approach to 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 4 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) [3] threads to construct mobile pipelines to exploit pipeline parallelism 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 [5].

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. 3(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 describe our approach 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 our full length technical report [4], 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

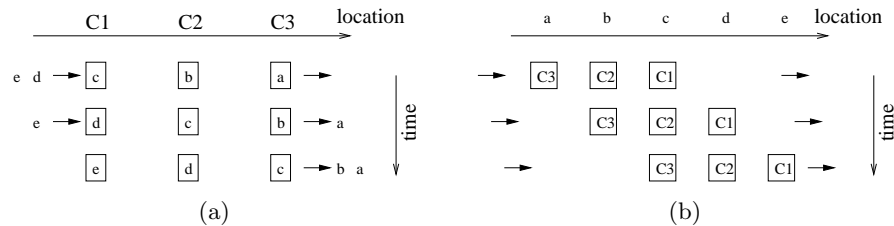


Fig. 4. The comparison of two pipelines. (a) Conventional; (b) Mobile

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 [5] from the classical SPMD (Single Program Multiple Data) view.

Our NavP approach uses the MESSENGERS [6–8] migrating thread environment. This parallel execution environment is efficient because, as pointed out in our technical report [4], we do not need to move code, we keep the cost of book keeping small, and we use user level multithreading to efficiently schedule the migrating threads.

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) [3]. In this step, we start with a data distribution pattern, and 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 3(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 distributions (e.g., block, cyclic, or block cyclic).

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. 5(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. 5(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

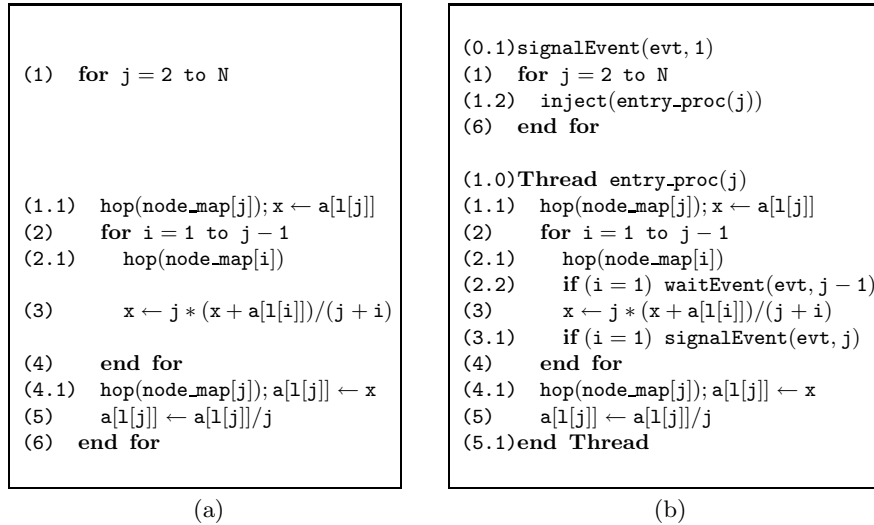


Fig. 5. The simple algorithm. (a) DSC using NavP; (b) Pipelining using NavP

ordering, the pipeline remains 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 $a[\cdot]$ 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 computation should occur on the node containing the largest amount of data to be used by the computation (the *pivot node*), so that a small amount of data is carried to meet with a large amount of data rather than the other way around. 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. Because threads are not allowed to access data remotely, all synchronization events are local to a node and hence efficient. (We note that when data parallelism is present, we can use multiple concurrent DSC threads to exploit this data parallelism [8], but this is not the focus of this paper.) (3) The NavP code as listed in Fig. 3(b) and Fig. 5(b) work for arbitrary data distribution. All that changes is the contents of the $node_map[\cdot]$ and $l[\cdot]$ arrays. This provides considerable 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. 6(a)) so the granularity is coarse. Figure 6(b) shows the details of

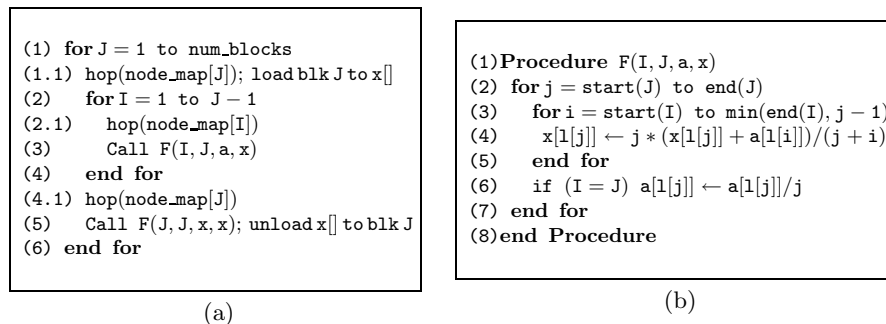


Fig. 6. Block pseudocode for the simple algorithm. (a) DSC; (b) The function $F()$

the block function $F()$ (called in lines (3) and (5) of Fig. 6(a)). The functions $start(I)$ and $end(I)$ return, respectively, the smallest and largest global indices of array entries stored in block I . The block pipeline code is similar to the code listed in Fig. 5(b) and therefore omitted. Transforming from the original sequential algorithm to the corresponding block algorithm can be automated using loop tiling techniques [9].

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 $k = 1, \dots, P$) 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 $N/P \cdot \sum_{k=1}^P k$, 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 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. 7, the block algorithm and block cyclic data distribution both help improve performance dramatically.

3 Crout Factorization

Crout factorization is a convenient variant of Gauss elimination [10]. Figure 8(a) lists the pseudocode for sequential Crout factorization of a symmetric matrix. For simplicity, we assume that the matrix being factorized, K , is a dense matrix. This algorithm is left-looking because the updating of the j^{th} column uses all the columns to its left from 1 to $j - 1$.

Figure 8(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 column is the basic unit of data distribution and is hence indivisible. In the pseudocode, *node[.]* provides the node ID of a given column. In the real code, the

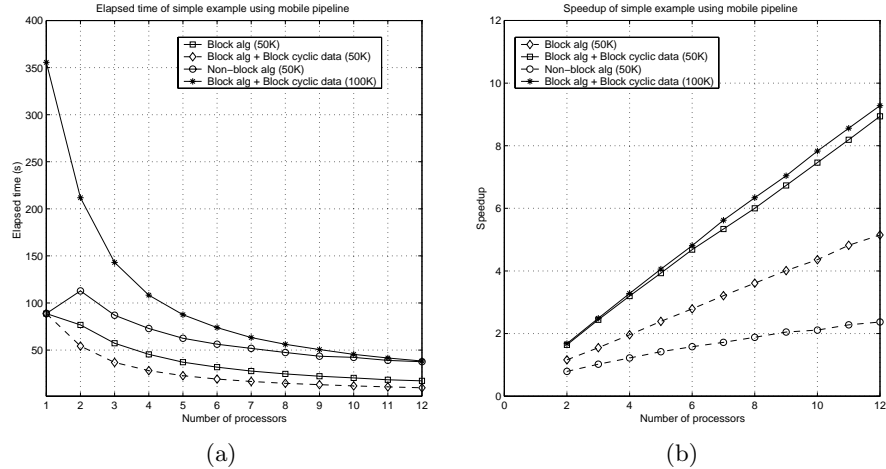


Fig. 7. Performance of the simple problem. (a) Elapsed time; (b) Speedup

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.

Figure 8(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. 8(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 MESSAGERS so that the event handling only happens on the node that hosts column 1 of K . This pipelined NavP code works correctly no matter how the columns are distributed. Similar to the simple example, we use block cyclic column distribution to exploit parallelism.

4 Performance

Performance data for the simple example is presented in Fig. 7, and for Crout factorization is in Fig. 9 and Table 1. The data was obtained using a network of SUNW Ultra-60’s with 450 MHz UltraSPARC-II CPU, 256MB of main memory, 1GB of virtual memory, 100Mbps of Ethernet connection with a collision-free switch, and using the NFS file-sharing system. To keep the presentation simple, we used non-block implementations of Crout algorithm in both the sequential and parallel versions of our algorithms. Thus, even though the sequential implementation is not the fastest possible, the speedup numbers relating our sequential and parallel implementations are based on a fair comparison, and they represent a good indication of the efficiency and scalability of our approach.

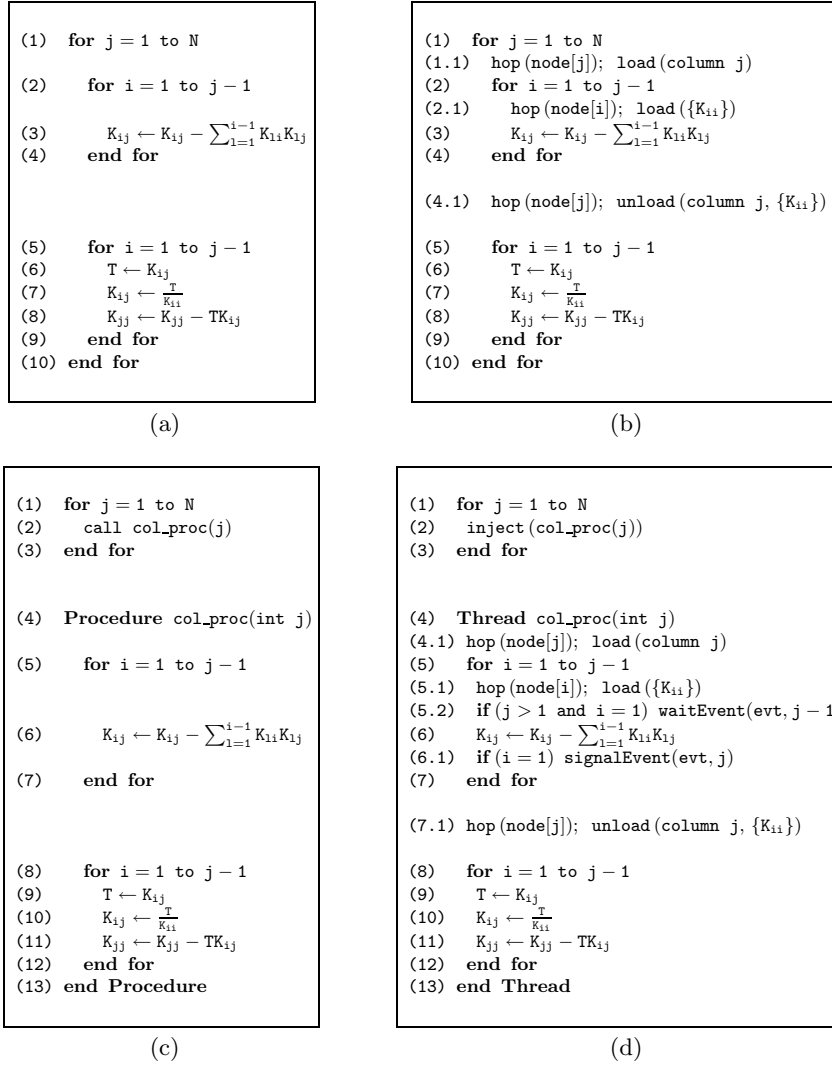


Fig. 8. Pseudocode for Crout factorization. (a) Sequential; (b) DSC using NavP; (c) Sequential re-written (with procedure call); (d) Pipelining using NavP

We were unable to find a parallel Crout implementation in literature, possibly because of the difficulty of parallelizing left-looking algorithms using conventional approaches. In [8], we compared our speedup numbers with those of the Cholesky factorization implementation in ScaLAPACK [11]. Crout factorization and Cholesky factorization are two variants of LU decomposition with the same asymptotic time complexity. Crout factorization on symmetric matrices is left-looking, and Cholesky factorization is right-looking. We found that the speedup numbers were very similar [8]. This indicates that the techniques presented in

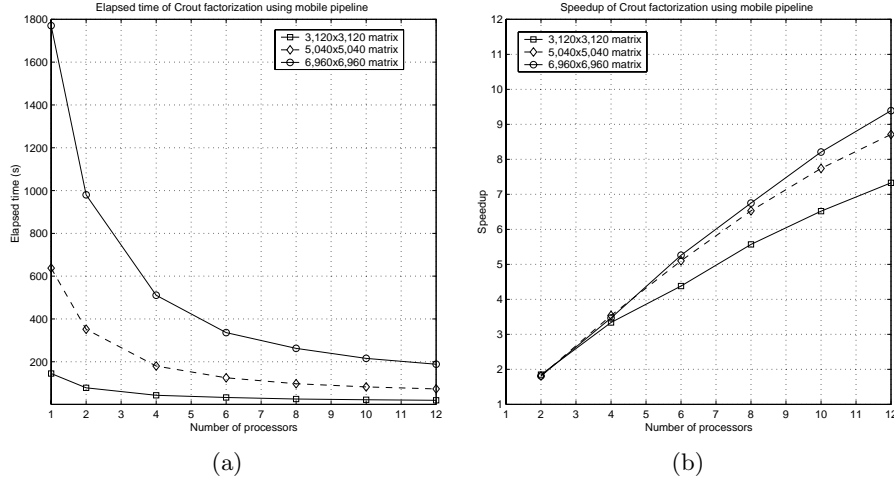


Fig. 9. Performance of Crout factorization. (a) Elapsed time; (b) Speedup

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

this paper for parallelizing left-looking algorithms are as effective as the classic (message-passing) approach to parallelizing right-looking algorithms.

5 Related Work

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 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 [1]. A recent survey [12] 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 in [12] for achieving parallelism using pipelining in situations of this type is not directly applicable to our examples. The reason is that this method assumes a regular data distribution with all data items initially residing on the first node initially. (Note that this second assumption is non-scalable.) In our examples, these assumptions do not hold, and once they are removed 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. It is similar to an assembly line process in a factory. In contrast, a mobile pipeline operates like farm work. The tasks (e.g., weeding, watering, or harvesting) are carried to the large data (the fields) by a mobile pipeline of equipment (e.g., tractors or harvesters) following each other. A mobile pipeline also carries small-sized data (e.g., seeds or fertilizer) that it needs when it carries out its operations.

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. 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 the data distribution pattern is unknown at compile time (e.g. in Grid computing). Our method can be easily extended to algorithms that are neither left-looking nor right-looking (for example Crout factorization on a non-symmetric matrix [1]). This is important, because most algorithms are neither purely left-looking or purely right-looking.

The reason for the effectiveness of our approach can be summarized as follows: supply-driven “pushing” is easier than demand-driven “pulling.” Right-looking (producer-driven) algorithms are easy to parallelize using conventional message-passing methods: when data is produced, it is propagated to the consumers, who consume it immediately. In contrast, left-looking (consumer-driven) algorithms based on movement of data require additional processing: once produced, any data that is not consumed immediately must either be replicated to multiple PEs and stored on each PE, or communicated multiple times. In our approach, even though the algorithm is consumer driven, the consumer process does not “pull” the data. Rather, it migrates (i.e., “pushes itself”) to the data. This additional flexibility is a fundamental advantage of migrating computations and Navigational Programming over more conventional methods of distributed programming.

The NavP approach is highly mechanical: it requires insertion of *hop*(s) and insertion of events. The former is based on data distribution, the latter on de-

pendency analysis. Code transformations are incremental and code structures remain essentially the same throughout the process. These transformations could potentially be semi-automated or perhaps fully automated by a compiler. Investigating this potential is part of our future research.

References

1. 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)
2. 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
3. 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
4. Pan, L., Lai, M.K., Dillencourt, M.B., Bic, L.F.: Mobile pipelines: Parallelizing left-looking algorithms using navigational programming. School of Information & Computer Sciences Technical Report TR# 05-12, University of California, Irvine, Irvine, Calif. (2005)
5. 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
6. 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.
7. 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
8. 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
9. Xue, J.: Loop Tiling for Parallelism. Kluwer Academic Publishers, Boston (2000)
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. 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)