NavP: Structured and Multithreaded
Distributed Parallel Programming

Lei Pan
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109, USA
1-818-393-0477
lei.pan@jpl.nasa.gov

Abstract—We present Navigational Programming (NavP) — a distributed parallel programming methodology based on the principles of migrating computations and multithreading. The four major steps of NavP are: (1) Distribute the data using the data communication pattern in a given algorithm; (2) Insert navigational commands for the computation to migrate and follow large-sized distributed data; (3) Cut the sequential migrating thread and construct a mobile pipeline; and (4) Loop back for refinement. NavP is significantly different from the current prevailing Message Passing (MP) approach. The advantages of NavP include: (1) NavP is structured distributed programming and it does not change the code structure of an original algorithm. This is in sharp contrast to MP as MP implementations in general do not resemble the original sequential code; (2) NavP implementations are always competitive with the best MPI implementations in terms of performance. Approaches such as DSM or HPF have failed to deliver satisfying performance as of today in contrast, even if they are relatively easy to use compared to MP; (3) NavP provides incremental parallelization, which is beyond the reach of MP; and (4) NavP is a unifying approach that allows us to exploit both fine- (multithreading on shared memory) and coarse- (pipelined tasks on distributed memory) grained parallelism. This is in contrast to the currently popular hybrid use of MP+OpenMP, which is known to be complex to use. We present experimental results that demonstrate the effectiveness of NavP.

Keywords—distributed parallel programming, navigational programming (NavP), migrating computations, distributed sequential computing (DSC), mobile pipelines

1. INTRODUCTION

Many aerospace software applications, such as ocean/atmosphere modeling, numerical weather prediction, data assimilation, structural/thermal analysis, optical system modeling, require tremendous processing power from modern supercomputers to enable accurate and in-time simulations and allow effective human-in-the-loop optimization. Although attempts are being made to transfer computationally intensive codes to parallel computing platforms, great difficulties have been encountered in parallelizing algorithms that are fundamental to simulations. Furthermore, once parallelized, the implementations usually have little resemblance to the original sequential algorithms, causing fear of losing control and hence hesitation to scientists and engineers in making investment in parallelization. However, the hope in waiting for new CPUs with endlessly growing clock speed and memory size to drive faster sequential executions is unrealistic because they are neither physically possible nor economically viable. In fact, today we are at a fundamental turning point toward concurrency and distribution in software development.

The state of the art for implementing large-scale high-performance scientific and engineering applications has been Message Passing (MP) or a hybrid use of MP and OpenMP, despite the fact that MP is widely considered to be hard to use. Studies reveal that the two statements at the heart of MP — Send() andRecv() — change the code like the goto statements, thus making structured programming problematic. This is reminiscent of the software crisis of the 1960s that led to Dijkstra’s recommendation that goto statements be abolished [9].

In this paper, we present a methodology called Navigational Programming (NavP) that is based on the use of self-migrating computations. NavP is philosophically different because it describes the quantity of interest – a distributed computation along with the small data it requires – following the movement of its locus. This changes the viewpoint of distributed computation from the conventional Eulerian view (or SPMD) to the Lagrangian view (or NavP) [38]. Program transformations from sequential algorithms in NavP involves the insertion of navigational statements (hop()), which, unlike Send() and Recv(), do not change code structure.

TABLE OF CONTENTS

1 INTRODUCTION 1
2 THE NAVP PROGRAMMING MODEL 2
3 THE NAVP METHODOLOGY 2
4 EXPERIMENTS 4
5 RELATED WORK 8
6 CONCLUSIONS 9
NavP enables scalable parallelization of the problems that are too hard or even impossible for MPI. It does so using incremental code transformations without changing the original sequential code structure, which is beyond the reach of MPI. It easily handles unstructured data layouts, which are necessary for good performance on distributed-memory machines but too hard for MPI. And it provides performance that is at least as good as that of MPI. We describe the steps provided by the NavP methodology, present performance studies of several important and non-trivial algorithms, and compare their implementations using both NavP and MP.

2. THE NAVP PROGRAMMING MODEL

Our approach is based on computation mobility enabled by self-migrating threads — threads capable of moving among different PEs (processing elements) in a network using special navigational commands, such as hop(). We call the resulting programming style Navigational Programming (NavP). The concept and mechanism of mobility have been pioneered by the research in mobile agents [25], [4], [31]. NavP threads use the same basic principles of strong mobility. Nevertheless, in NavP mobility is used as a programming model (i.e., a way of describing distributed computations). This is in contrast to mobile agent research work, where the emphasis is on actual code mobility. Our target application is high performance computing on a local area network and we make our parallel execution environment efficient accordingly. One feature worth noting is that we do not move code to a PE more than once irrespective of how many times the locus of computation revisits that PE [39], [40]. The overhead of our runtime system is negligible as borne out by our performance studies, some of which will be shown in this paper.

As a programming methodology, NavP uses the following execution and memory models.

Execution model:

- Logical network. The NavP runtime environment consists of three layers of network: the physical network, the daemon network (runtime system), and the logical network. At most one daemon process is running on any one physical node, but the daemon network topology can be different than that of the physical network (e.g., the daemons can be fully connected while the physical network is not). The logical network is an application-specific network. Any number of logical nodes can be mapped onto a daemon node. The topology of the logical network is arbitrary;
- Self-migration. A NavP program pauses its computation at a navigational statement, i.e., a hop() statement, migrates to the destination logical node as defined by the argument to hop(), and resumes its computation on the new logical node. The hop() statements are inserted to the code either by a programmer;
- User-level multithreading. A NavP program consists of multiple self-migrating logical units that are each a user-level thread;
- Synchronization. Synchronization among threads uses events. The statements signalEvent() and waitEvent() implement the classical operations of process blocking and wake-up. The synchronization events are all local to a logical node;
- Non-preemptive and FIFO scheduling. A thread’s execution is not preempted between any two navigational statements. That is, a thread must explicitly relinquish control to others using statements such as hop(). Also, threads hopping from the same source logical node to the same destination logical node preserve their ordering;
- Function calls. Function calls are implemented as thread spawning. A thread can spawn another thread using the inject() or dot threads statement. Injected threads communicate and synchronize with other threads on the same node using local events.

Memory model:

- A thread-carried variable is private to a migrating thread and is available to the thread wherever it navigates. It is primarily used to communicate small-sized data across memory boundaries;
- A node variable is stationary to a logical node and shared by all threads running on the logical node. It is used to store large-sized stationary data and to facilitate synchronizations within a logical node. It can move with its hosting logical node;
- A distributed shared variable (DSV) is logically one variable but physically a collection of node variables residing in disjoint address space. It is shared by multiple threads and it provides a partitioned global address space. This is in contrast to the localized view of data and computation provided by MP [6], [28]. Unlike in DSM, shared data stored in a DSV is not globally accessible; computations migrate to large-sized stationary data to access rather than pulling the data back. It is the DSVs and computation mobility that make NavP shared variable programming beyond shared memory [28];

3. THE NAVP METHODOLOGY

In this section, we provide the steps, schematically depicted in Fig. 1, that are followed in the NavP methodology. We
start from a sequential algorithm and Step 1 is to derive a data mapping that is efficient in data communication for that given algorithm. With the given data distribution from Step 1, we use Step 2 to transform the original sequential program to a Distributed Sequential Computing (DSC) program by inserting `hop()` statements. In Step 3, we cut the DSC thread into shorter DSC threads and build mobile pipelines using the shorter DSC threads. This will give us a Distributed Parallel Computing (DPC) program. Step 4 is a feedback loop, in which we adjust the granularity level and the degree of parallelism by fine tuning the data distribution or changing how the DSC is cut. These steps can be used by a programmer, or they can be used to build automatic tools to facilitate program transformations.

**Step 1: Data Distribution**

Data distribution for migrating computations is done by first building a Navigational Trace Graph (NTG) and then partitioning the NTG using a heuristical graph partitioning algorithm. An NTG is an undirected weighted graph where the nodes are individual entries of all distributed arrays and the weight associated with an edge represents the trace between the two incident array entries as the DSC thread navigates through them. An NTG is generated by instrumenting the sequential algorithm using a small problem as an input. The NTG is partitioned using heuristics [21] to minimize the overall edge cuts in the graph; this corresponds to a heuristical minimization of the overall communication cost for a distributed implementation. Data distribution itself is a research area and we report our results in a separate publication [30].

**Step 2: Distributed sequential computing**

In this step of DSC, Navigational statements (e.g., `hop()`) are inserted into the sequential code, and the resulting DSC thread navigates to large-sized data carrying small-sized data for computation to happen. Figure 2(a) and (b) schematically depicts the DSC code transformation. This step follows the principle of *pivot-computes*, which requires that a subcomputation should happen at the computer node that hosts the large-sized data. This code transformation preserves algorithmic integrity, which says that the codes before and after the transformation are structurally the same. This will be seen in our examples in this paper.

**Step 3: Distributed parallel computing**

In this DPC step, the DSC thread is transformed into multiple "shorter" migrating threads, and these threads are composed into a DPC program to form mobile pipelines — pipelined computations as a result of DSC threads following each other. Figure 2(c) shows such a mobile pipeline and Fig 2(d) depicts a phase-shifted mobile pipeline as the dependency relationship among the computations allows. Synchronization statements (e.g., `waitEvent()` and `signalEvent()`) are inserted into the code to ensure correct execution of producer-consumer computations. This step of code transformation exhibits composition orthogonality, which means the code intersection among the composing DSC threads is minimum.

**Step 4: Feedback loop**

This step is for feedback and refinement. The NavP transformations can be systematically applied repeatedly or hierarchically in different dimensions of a network of PEs. At each step, we have a fully functional implementation that is an improvement of the previous step. Further development can be stopped if satisfying performance is achieved or a resource limit is reached. In this sense, the NavP methodology supports incremental parallelization.

One possible feedback adjustment is data redistribution (e.g., increasing the number of blocks in a block cyclic data distribution). Figure 3 qualitatively depicts how the execution time.

---

**Figure 2.** The code transformations in Navigational Programming. (a) Sequential. (b) DSC. (c) DPC with pipelining. (d) DPC through phase-shifting.
4. Experiments

In this section, we present our experimental results. 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. The C compiler used was gcc 3.2.2, the MPI used was LAM MPI 7.0.6 [37], and the NavP compiler and runtime system used was MESSENGERS 1.2.05 [8].

A simple example

We use a simple, contrived example to illustrate how NavP programming works. Consider the simple algorithm listed in Fig. 5(a), in which the $j^{th}$ iteration of the outer loop, which computes $a[j]$, consumes the values of $a[i]$ produced by all the previous $j$ iterations. Since the computation of $a[j]$ is neither associative nor communicative, the consumption of $a[i]$ is in a sequential order that cannot be changed. This algorithm is an example of the class of so-called “left-looking” (or, consumer-driven) matrix algorithms [10], [27]. We assume a block data distribution pattern for simplicity, and use individual arrays on the PEs to host the data blocks. These arrays logically form a DSV. The auxiliary array $\text{node}_{\text{map}}[.]$ provides the logical node hosting a given array entry, and $1[.]$ contains the local array index of an entry with a given global index. A DSV thus provides a partitioned global address space. The computation of $a[j]$ should take place on the PEs where the $a[i]$’s reside, so that the cost of communication for the subcomputation of $a[j]$ is minimized. We therefore put $a[j]$ in a thread-carried variable, $x$, and insert $\text{hop()}$ statements in the sequential code so that the computation follows the data it accesses (i.e., the $a[i]$’s) through the network. The result is a DSC program: the computation uses distributed data but has a single locus of computation. Figure 5(b) shows the DSC code. Three $\text{hop()}$ and load/unload compound statements are inserted (at lines (1.1), (2.1), and (4.1)) without changing the code structure. In the pseudocode, $x$, $i$, and $j$ are thread-carried variables, and $a[.]$ is a DSV. If we cut the single long DSC thread into multiple shorter threads, we get a DPC program, listed in Fig. 5(c). Each computation of $j$ becomes a DSC thread that is spawned by a $\text{dothreads}$ at line (1). The NavP $\text{dothreads}$ construct generalizes the classical $\text{DOACROSS}$ and $\text{DOALL}$ parallelism constructs, but the spawned threads are DSC threads. The code for each thread, lines (1.1) through (5), remains almost the same as the DSC code listed in Fig. 5(a). The only difference is the insertion of two new lines 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 logical node that it has finished accessing $a[1]$. In this way, the threads organize themselves into a mobile pipeline when they access $a[1]$: the thread computing $a[j]$ runs immediately after the thread computing $a[j−1]$. Because of their FIFO scheduling, migrating threads do not pass each other in the mobile pipeline. Each computation migrates through the pipeline, progressively visiting the successive stages (the entries $a[i]$ that it successively incorporates into its computation). Notice that in NavP synchronizations are only local among the collocated threads. Figure 4 schematically depicts how a mobile pipeline works.

ADI integration

ADI integration is an example used by several papers on data distribution [23], [3], [22], [26]. The pseudocode for ADI is listed in Fig. 6 [23], [22]. There are three 2D arrays, namely c, a, and b, involved in the computation. This code is usually subdivided into two phases, namely a row sweep phase (lines (2)-(15)) and a column sweep phase (lines (16)-(29)). These two phases are surrounded by an outer loop of time iteration (line (1)). One possible solution, existed in previous work, is to find two different data mappings suited for their respective phases. We use our tool to find these two separate solutions and plot them in Figs. 7(a) and (b). Figure 7(c) depicts the data distributions for two phases combined together. The two sweeps are two $\text{DOALL}$ loops (i.e., full parallelism with no communication) if they use their own data distribution, but in between the sweeps a dynamic data redistribution is needed. If both phases are combined, pipeline parallelism can still be exploited. The advantage of this data distribution

![Figure 3](image-url) Performance as block cyclic data distribution is refined (assuming two PEs, $T$ is the sequential execution time.).

![Figure 4](image-url) Mobile pipeline of DSC threads.
for the entire program is that no dynamic data remapping is needed between the two phases. The cost of a dynamic data remapping can vary dramatically on different platforms.

We first turn the ADI code into a block implementation. That is, we introduce “distribution blocks” — submatrix blocks that are basic units for data distribution — in the matrices and convert the loops over the matrix entries into the loops over the entries within the distribution blocks surrounded by the loops over the distribution blocks. Next, we go through the NavP steps to parallelize ADI. In particular, we first make the sweeps two DSCs and turn the outer loop another DSC responsible for injecting the sweeper DSCs. We then cut the sweeper DSCs into shorter ones and pipeline them. These steps are illustrated using the simple example presented in earlier in this section, we therefore skip the details here.

Figure 8 depicts two different block cyclic patterns in 1D and 2D cases. Each box in this figure represents a submatrix block and the number in a box indicates the ID of the PE that this block is assigned to. It is assumed that in the 1D case we have two PEs and in the 2D case we have four PEs. As in Fig. 6, the three square matrices are each of order $N$. In Fig. 8(a), a matrix is cut into four vertical slices each of size $N/4$ and the blocks are assigned to the two PEs in a block fashion (that is, the first two blocks go to PE1 and the last two blocks go to PE2). Figure 8(b) depicts a 1D block cyclic pattern where the blocks are assigned to the PEs in order until the PEs are exhausted, used, at which time the block assignment cycles back. In HPF [34], a 2D block cyclic pattern is the cross prod-
uct of two 1D block cyclic patterns, shown in Fig. 8(c). For 2D, each submatrix block is \( \frac{N}{4} \times \frac{N}{4} \). A different block cyclic pattern is depicted in Fig. 8(d), in which the first row of blocks are assigned to all the PEs in order. (This is unlike the HPF pattern where the PEs are arranged as a 2 \( \times \)2 processor grid and the first row of blocks are assigned cyclically along the first row of processors.) The next rows are assigned to all the PEs in a similar way, except that they are shifted east-ward one position from their previous rows. This block distribution is effectively a “twisted pattern.” When the sweeper threads sweep through all the rows or columns, all PEs are busy simultaneously. That is, we achieve full parallelism, at the cost of \( O(N) \) as one layer of the matrix entries is carried over from block to block. In contrast, in the example shown in Fig. 8(c), only two PEs are busy at any time as the sweeper DSCs sweep through. The situation for the HPF pattern is worse when the PEs are arranged as a 1D grid when, e.g., the number of PEs is a prime number. As for the cost of communication, the DOALL approach mentioned earlier requires \( O(N^2) \) in data redistribution.

As presented in Fig. 9 (the numbers in the legend are matrix orders), the NavP program using the twisted block cyclic data distribution pattern performs the best. Using the HPF block cyclic pattern, the NavP program incurs the same communication cost of \( O(N) \) but has less degree of parallelism. Therefore, the performance is inferior, especially when the number of PEs is a prime number\(^1\). Finally, if we employ data redistribution in the DOALL approach, even though the two sweeps are fully parallel, the cost of data redistribution, \( O(N^2) \), is so large that the overall performance is poor. We used the MPI library call \( \text{MPI}_\text{All} \text{t} \text{o} \text{a} \text{l} \text{l} \) to obtain the cost for matrix redistribution.

With this example of ADI, we are able to demonstrate the following: (1) The data distribution for NavP is obtained from minimizing the cost of communication with load balancing as a constraint. Parallelism is exploited later using mobile pipelines. The HPF style block cyclic data distribution helps to improve parallelism by making the PEs busy earlier, and the twisted block cyclic data distribution enables the NavP program to achieve full parallelism; and (2) On loosely coupled systems such as clusters, data redistribution between the two phases, aimed at achieving full DOALL parallelism for both phases, is prohibitively expensive. As a result, choosing a data distribution that minimizes communication and further minimizing communication using DSCs that follow the principle of pivot-computes are of decisive importance to overall performance. Using pipelining may result in loss of some degree of parallelism, but this impact to performance is secondary. Furthermore, with careful adjustment in data distribution using the twisted cyclic pattern, it is still possible to achieve full parallelism using mobile pipelines at a cost of asymptotically less communication than what is required in the DOALL approach.

### Crout factorization

Crout factorization [20] has the data access pattern similar to the simple example presented earlier in this section, except that the problem is now 2D. We initially use a block data distribution (with blocks of columns) and program our DSC thread to compute following the large-sized data. The difference from the simple example is that the DSC now carries a column (entries on and above the diagonal line) of the 2D matrix rather than an entry of the 1D array. The DPC is obtained in the same way as described in the simple example and block cyclic data distribution (using a block of columns as a distribution unit) is used to adjust the performance of the code. The sequential, DSC, and DPC pseudocodes are listed in Fig. 10 and the performance data of DSC and DPC Crout factorization are presented in Fig. 11 and Fig. 12, respectively. DSC is useful by itself because an “all-memory” performance is obtained for large-sized problems as heavy paging, or disk thrashing, is replaced by a modest cost of network communication. Besides, code modification from sequential to DSC is small, as depicted by Fig. 10(a) and (b).

Parallel implementation of Crout factorization using MP is extremely difficult and is therefore left as an open problem. To our best knowledge, we are not aware of any such implementation in literature.

### Cholesky factorization

Cholesky factorization [14] is an algorithm for factorizing symmetric positive definite matrices. The DSM pseudocode for Cholesky factorization is adopted from a classical textbook for matrix computation [14] and listed in Fig. 13(a). Our NavP DPC code is listed in Fig. 13(b). Again, a cyclic data distribution scheme for updating the columns is used for load balancing. Performance data from three different implementations, namely MP, ScaLAPACK [5], and NavP are

---

\(^1\)We use a true 2D processor grid for the HPF block cyclic pattern whenever possible.
Figure 10. Pseudocode for Crout factorization. (a) Sequential; (b) DSC using NavP; (c) Sequential; (d) Pipelining using NavP

Figure 11. Performance of DSC Crout factorization.

Figure 12. Performance of DPC Crout factorization.

presented in Table 1.

Pseudocode for an MP solution of Cholesky factorization, adapted from the implementation by Golub and Van Loan [14], is presented in Fig. 14. It is obvious that the original code structure is dramatically changed. The Send() and
Recv() statements are the reason for this change because they behave just like the goto statement in sequential programming [15].

5. RELATED WORK

There is a large body of literature devoted to MP [32], [36], [16] is the de facto standard of high-performance parallel programming, and is considered a success [17]. Conceptually, MPI is very simple: it consists of a small number of send/receive primitives used directly by the programs. MPI-2 [16], [1], [13] has several new features, such as one-sided communication based on remote memory access, parallel I/O, dynamic process, and threads. The basic programming style, however, is unchanged. In most cases, the programmer must restructure a given sequential algorithm to make it work in a distributed memory environment because
of the SPMD view used. In contrast, NavP programs evolve incrementally from sequential algorithms, and hence are easier to derive. Furthermore, our preliminary results indicate that NavP programs perform as well as or faster than MP programs.

Because of the difficulty of hand-crafted parallelism in MPI and the difficulty of writing parallelizing compilers, several approaches have emerged that allow the programmer to suggest data distribution and parallel structures. Examples of such systems include HPF [24], [18], [2], OpenMP [7], [35], [19], and UPC [12], [11]. OpenMP assumes a shared memory model, and provides annotations for thread management, work distribution, data scope specification, and synchronization. HPF and UPC also have constructs to control the mapping of data. In general, these approaches are not able to deliver a performance that is competitive to that of handcrafted MPI programs.

6. CONCLUSIONS

NavP has a number of advantages. (1) As validated by our preliminary results, NavP implementations are always competitive with the best MPI implementations in terms of performance, and in some cases are considerably better. As a special use of NavP, DSC threads can speed up the execution of even a single sequential process (refer to the performance of DSC Crout factorization in Fig. 11). (2) NavP is structured distributed programming, as it directly captures the algorithm. MP, by comparison, requires significant restructuring of the program, obscuring its original purpose. It has been pointed out that send and receive are harmful today for much the same reason that unrestricted goto statements have been considered harmful since the “software crisis” of the 1960’s [15]. Because of its structured programming, NavP allows us to parallelize certain programs that are generally considered unparallelizable using other approaches [29]. (3) NavP provides incremental parallelization, in the sense that a sequential program can be converted into a fully parallel program through a sequence of small transformations, where each intermediate step is a fully functioning program. This is in sharp contrast to MP, where a parallel program usually requires a complete rewrite and major restructuring. (4) Today a hybrid programming model of MP+OpenMP is sometimes used [33], but this requires extensive programming efforts. NavP is a unifying approach that allows us to exploit both fine- (multithreading on shared memory) and coarse- (pipelined tasks on distributed memory) grained parallelism. These advantages make NavP a competitive alternative to MP as an intermediate representation for manual programming as well as automatic source-to-source code transformations by a compiler.

Our future work includes using real-world, non-trivial aerospace applications to further evaluate the effectiveness of the NavP methodology and porting the NavP underlying infrastructure to multi-core or cell clusters.

REFERENCES


[36] Marc Snir, Steve Otto, Steven Huss-Lederman, David


Lei Pan is a senior computer scientist at the Jet Propulsion Laboratory, California Institute of Technology. He holds an MS in Mechanical Engineering (Rensselaer Polytechnic Institute, NY) and a PhD in Information & Computer Science (University of California, Irvine). His research interests include distributed parallel computing, parallelizing compilers, and numerical methods.