PAPER Special Issue on Parallel/Distributed Computing and Networking

Toward Incremental Parallelization Using Navigational Programming*

Lei PAN[†,††], **Wenhui ZHANG**[††], **Arthur ASUNCION**[††], **Ming Kin LAI**[††], **Michael B. DILLENCOURT**[††], **Lubomir F. BIC**[††], *and* **Laurence T. YANG**[†††], ***Nonmembers***

SUMMARY The Navigational Programming (NavP) methodology is based on the principle of self-migrating computations. It is a truly incremental methodology for developing parallel programs: each step represents a functioning program, and each intermediate program is an improvement over its predecessor. The transformations are mechanical and straightforward to apply. We illustrate our methodology in the context of matrix multiplication, showing how the transformations lead from a sequential program to a fully parallel program. The NavP methodology is conducive to new ways of thinking that lead to ease of programming and high performance. Even though our parallel algorithm was derived using a sequence of mechanical transformations, it displays certain performance advantages over the classical handcrafted Gentleman's Algorithm.

key words: *programming methodologies, incremental parallelization, navigational programming (NavP), program transformation, matrix multiplication, Gentleman's Algorithm, Cannon's Algorithm*

1. Introduction

We show how our Navigational Programming (NavP) methodology can be applied to a sequential algorithm for matrix multiplication to obtain a series of programs that represent incremental steps in exploiting parallelism in the original algorithm. The final incremental program in the series— a fully parallel program for matrix multiplication—is similar to the classical Gentleman's algorithm, but has performance advantages over Gentleman's algorithm which we describe.

In NavP, migrating computations are the composing elements of a distributed parallel program. The code transformations in NavP – distributing the data and inserting corresponding navigational commands, pipelining, and phase shifting – can be used to incrementally turn a sequential program to a distributed sequential computing (DSC) program, and later to a distributed parallel computing (DPC) program. These transformations can be applied repeatedly, or in a hierarchical fashion, as illustrated below. The benefits of the NavP incremental parallelization include: (1) Every program is a result of applying the mechanics of one of the transformations and is a natural and incremental step from its predecessor. As a result, no abrupt change in code will happen between any consecutive steps; (2) Every intermediate program is an improvement from its predecessor. If program development is limited by time or resources, any one of the intermediate programs can be taken as production code; (3) The transformations are highly mechanical and straightforward to use, and yet the resulting parallel programs are elegant and efficient.

We briefly describe the NavP methodology in Sect. 2. Section 3 summarizes the application of NavP to the classical problem of matrix multiplication; for more details and the complete pseudocode at each intermediate step, the interested reader is referred to our conference paper [1]. Section 4 contains performance data. We present a detailed comparison of the parallel algorithm derived from our NavP solution with the classic Gentleman's algorithm in Sect. 5.

2. Navigational Programming

Navigational Programming (NavP) is a methodology for distributed parallel programming based on the use of self-migrating computations. In NavP code, a programmer inserts navigational commands, i.e., `hop()` statements, to migrate computation locus in order to access remotely distributed data and spread out computations. Small data that is "carried" by the moving computation is put in "agent variables," whereas large data that stays on a computer is held by "node variables." An agent variable is private to a computation thread, and is available to the thread wherever it migrates. The cost of a `hop()` is mainly spent in shipping the data stored in agent variables. The synchronization among different migrating computations is done through "events" (`signalEvent()` and `waitEvent()`). Details on how to use the MESSENGERS system can be found in the manual [2].

NavP provides a different view of distributed computation from the classical SPMD view. The SPMD view describes distributed computations at stationary

Manuscript received January 20, 2005.
Manuscript revised August 15, 2005.
Final manuscript received October 10, 2005.

[†]Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109-8099, USA, Lei.Pan@jpl.nasa.gov
[††]Donald Bren School of Information & Computer Sciences University of California, Irvine, CA 92697-3425, USA, {pan,wzhang,aasuncio,mingl,dillenco,bic}@ics.uci.edu
[†††]Department of Computer Science, St. Francis Xavier University, Antigonish, NS, B2G 2W5, Canada, lyang@stfx.ca

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

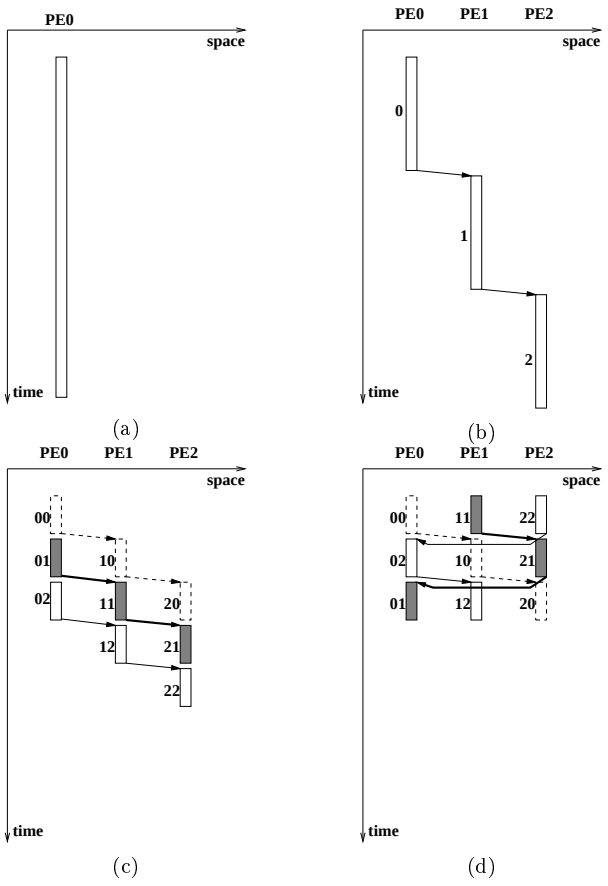

Fig. 1 The code transformations in NavP. (a) Sequential. (b) DSC. (c) Pipelining. (d) Phase shifting.

locations, whereas the NavP view describes a computation following the movement of its locus. The three transformations under the NavP view are depicted in Fig. 1. Here and throughout the paper, arrows indicate `hop()` operations. The basic idea behind the transformations is to spread out computations using self-migrating computation threads as soon as possible without violating any dependency conditions. **(1) DSC Transformation**: Large data is distributed among the PEs (processing elements), and `hop()` statements are inserted into the sequential code in order for the computation to "chase" large data while carrying small data. The DSC Transformation is schematically depicted using Figs. 1(a) and (b). The resulting program performs "Distributed Sequential Computing," which is more conveniently termed DSC. The immediate benefit of DSC is that, with a reasonable amount of work, a sequential program can be used to efficiently solve large problems that cannot fit in the main memory of one computer. By using a network of workstations, the DSC program has completely removed paging overhead by trading it against a modest amount of network communication [3]. DSC also serves as the starting point of parallel program development in NavP. **(2) Pipelining Transformation**: This transformation is depicted using Figs. 1(b) and (c). The basic idea is to pipeline multiple DSC computation threads. Synchronization may be necessary to keep the DSC threads ordered correctly in the pipeline. **(3) Phase-shifting Transformation**: Sometimes the dependency among different computations allows different DSC threads to enter the pipeline from different locations. In these situations, we can phase shift the DSC threads to achieve full parallelism, as depicted in Figs. 1(c) and (d).

The NavP transformations can be systematically applied repeatedly or hierarchically in different dimensions of a network of PEs, as will be shown with matrix multiplication later in this paper. At each step, we have a fully functional implementation of matrix multiplication that is an improvement of the previous step. The result of the final step has a resemblance to the classical Gentleman's Algorithm, but there are important differences as described in Sect. 5.

3. Incremental Parallelization of Matrix Multiplication

Matrix multiplication is a fundamental operation of many numerical algorithms. Pseudocode for sequential matrix multiplication is listed in Fig. 2. Throughout the paper, we assume `N` is the order of the square matrices involved. It is clear that the computation of each entry of the matrix `C` is independent of all other entries of `C`, and therefore there are N^2 updatings that can be done in parallel. Nevertheless, exploiting the abundant parallelism in matrix multiplication is not as straightforward as one might think. Suppose we parallelize the two outer loops using the popular `doall` notation, as shown in Fig. 3. We can get, for example, two concurrent statements run by two PEs: $C(1,1) += A(1,1) * B(1,1)$ and $C(1,2) += A(1,1) * B(1,2)$. These two parallel executions both need the entry $A(1,1)$. If the requests for $A(1,1)$ from the two PEs arrive at the same time at the PE that hosts $A(1,1)$, contention happens. On the other hand, if we cache multiple copies of $A(1,1)$ on the PEs that require it, this solution is not scalable. Gentleman conducted research into the data movement required for matrix multiplication, and his analysis confirmed that data movement – and not arithmetic operations – is often the limiting factor in the performance of algorithms [4], [5].

```
(1) do i=0,N-1
(2)   do j=0,N-1
(3)     t = 0.0
(4)     do k=0,N-1
(5)       t += A(i,k) * B(k,j)
(6)     end do
(7)     C(i,j) = t
(8)   end do
(9) end do
```

Fig. 2 Pseudocode for sequential matrix multiplication.

```
(1) doall i=0,N-1
(2)   doall j=0,N-1
(3)     C(i,j) = 0.0
(4)     do k=0,N-1
(5)       C(i,j) += A(i,k) * B(k,j)
(6)     end do
(7)   end doall
(8) end doall
```

Fig. 3 Pseudocode for parallel matrix multiplication using `doall`.

We provide a solution that does not trigger contention (i.e., we avoid the situation where multiple PEs get matrix entries from a single PE at the same time), and does not use data replication (i.e., at any given time, there is only one copy of any matrix entry). For simplicity, we describe the problem and our solution at a fine granularity level. That is, we assume $\texttt{N} == \texttt{P}$, where `P` is either the number of PEs in a 1D processor network or the order of a 2D processor network. To extend our solution to a coarser level, we would treat each element (e.g., `C01` or `A21`) as a sub-matrix block, instead of an entry of the matrix. Our solution is incremental and involves applying a series of transformations, obtaining a algorithm at each step. The pseudocode for each algorithm in the series is given in our conference paper [1]; here, because of page limitations, we simply give a summary.

We first apply the DSC Transformation to sequential matrix multiplication, as depicted in Fig. 4(a). The essence of this DSC transformation is to distribute the computation in the `j` dimension. The PE network is 1D in which each PE has a unique identifier $\texttt{HnodeID} = 0, 1, ..., \texttt{N} - 1$ from west to east. Thick boxes contain node variables on different machines, and thin boxes carry agent variables. Next, we apply the Pipelining Transformation to the DSC code obtained from the last step, as depicted in Fig. 4(b). Each row of matrix `A` is assigned to a computation thread, and these threads are "injected," or spawned, into the PE pipeline in turn, and follow each other in the network to compute the corresponding `C` entries. We then apply the Phase-shifting Transformation to achieve full DPC, as depicted in Fig. 4(c). This is possible because each row of `A`, though needed on all three PEs, can start its computation from any PE. At this point we have a matrix multiplication algorithm that is fully parallel, except that it only uses one dimension (the `j` dimension) rather than two.

To exploit the `i` dimension as well, we next introduce a 2D network in which each PE has a unique 2D identifier (`HnodeID`, `VnodeID`), where $\texttt{HnodeID} = 0, 1, ..., \texttt{N} - 1$ from west to east, and $\texttt{VnodeID} = 0, 1, ..., \texttt{N} - 1$ from north to south, and apply the DSC Transformation in the second dimension, as depicted in Fig. 4(d). We then apply the Pipelining Transformation in both dimensions, as depicted in Fig. 4(e). A pair of `A` and `B` entries can move on along their pipelines respectively as soon as they finish computing and contributing the corresponding `C` entry. A producer `BCarrier` needs to make sure that the `B` entry produced by its predecessor in the pipeline is consumed before it puts the `B` entry it carries in place. Finally, we apply the Phase-shifting Transformation in both dimensions to achieve full parallelization, as depicted Fig. 4(f).

In Fig. 4, each sub-matrix block, e.g., `A10` or `C11`, is called a "distribution block" in our implementation, as it is a basic unit of data distribution on a PE. To achieve better performance from a block algorithm, a further level of matrix decomposition is used [6]. A distribution block is decomposed into "algorithmic blocks," and each algorithmic block of `A` or `B` is carried by a migrating thread (i.e., `ACarrier` or `BCarrier`). If we "zoom in" to the physical node ($\texttt{HnodeID} = 1, \texttt{VnodeID} = 1$) in Fig. 4(f) (assuming the entire PE network is the upper-left 2×2 processors), we can see algorithmic blocks as depicted by lowercase letters (e.g., `a57` or `c46`) in Fig. 6 of Sect. 5.1. As an example, the distribution block of `C11` in Fig. 4(f) is decomposed into algorithmic blocks contained in the thick box (which indicates a physical node) in Fig. 6. Our sequential and MPI implementations described below use algorithmic blocks as well.

Pseudocode for DPC in both dimensions is listed in Fig. 5. The matrices are initially distributed such that $\texttt{A}(\texttt{i}, \texttt{j})$, $\texttt{B}(\texttt{i}, \texttt{j})$ and $\texttt{C}(\texttt{i}, \texttt{j})$ (initialized to `0`) are on $\texttt{node}(\texttt{i}, \texttt{j})$. In this pseudocode, `A` and `B` indicate node variables, whereas `mA` and `mB` represent agent variables. In our NavP programs, we adapt a naming convention of starting an agent variable's name with a lowercase `m`. Matrix `A` is loaded into agent variable `mA` and carried by the migrating thread. $\texttt{node}(\texttt{j})$ maps to the PE that hosts column `j` of matrices `B` and `C`. Every time the thread of computation hops back to $\texttt{node}(0)$, it will pick up a different row of matrix `A` for the computation of the loop over `j`. Detailed descriptions and the pseudocode for all individual incremental steps can be found in our conference paper [1].

4. Performance Data

We have implemented parallel matrix multiplication using both NavP and message passing. The NavP system used was MESSENGERS (Version 1.2.05 Beta) developed at the Donald Bren School of Information & Computer Sciences, University of California Irvine [2]. The message passing system used was LAM 7.0.6 from Indiana University [7]. The ScaLAPACK used was version 1.7 from the University of Tennessee, Knoxville and the Oak Ridge National Laboratory [8]. The C compiler used was GNU gcc-3.2.2, and the Fortran compiler used was GNU g77-3.2.2. The performance data was obtained from SUN workstations (SUN Blade 100, CPU: 502 MHz SUNW,UltraSPARC-IIe, OS: SunOS Release 5.8) with 256MB of main memory, 1GB of virtual mem-

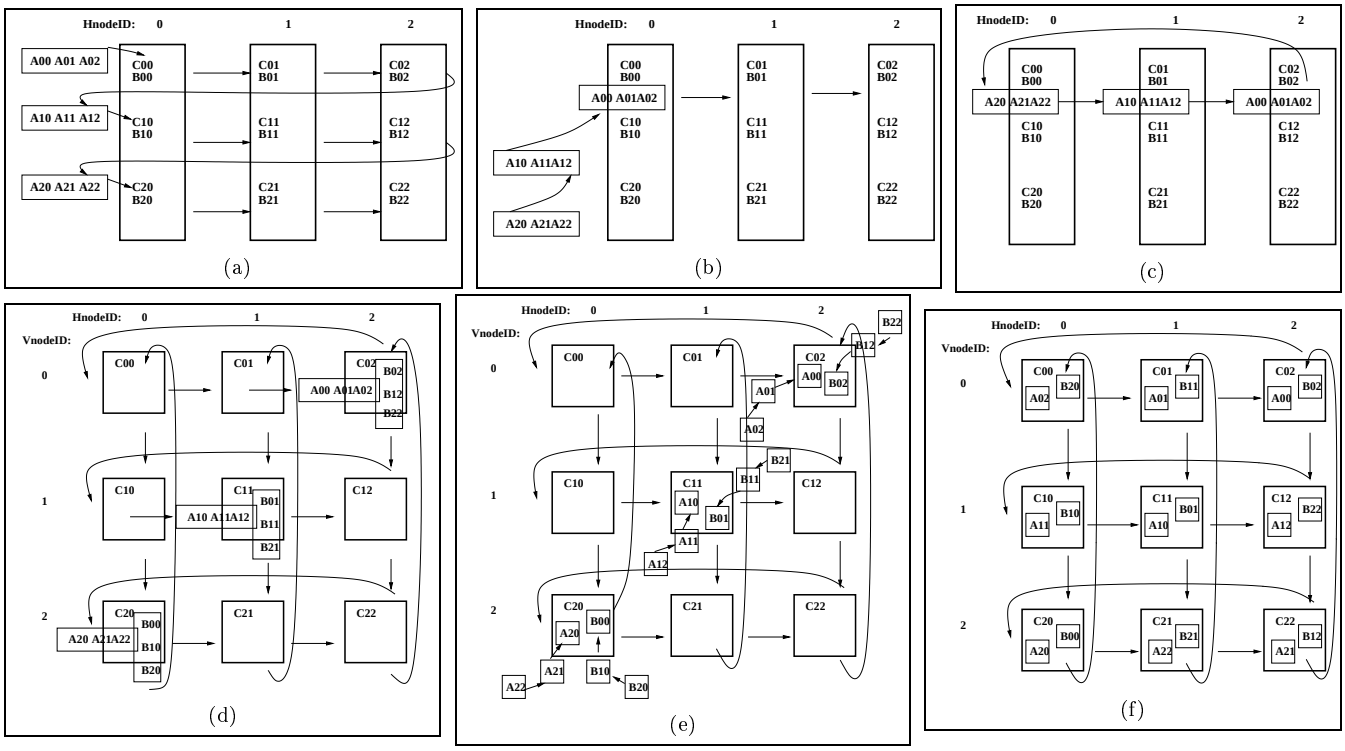

Fig. 4 From sequential to parallel matrix multiplication. (a) DSC. (b) 1D pipelining. (c) 1D phase shifting. (d) 2D DSC. (e) 2D pipelining. (f) 2D phase shifting.

```
(1)  do mj=0,N-1
(2)    hop(node(0,mj))
(3)    inject(spawner(mj))
(4)  end do

(1)  spawner(int mj)
(2)    do mi=0,N-1
(3)      hop(node(mi,mj))
(4)      signalEvent(EC(mi,mj))
(5)      inject(ACarrier(mi,mj))
(6)      inject(BCarrier(mi,mj))
(7)    end do
(8)  end

(1)  ACarrier(int mi, int mk)
(2)    mA = A
(3)    do mj=0,N-1
(4)      hop(node(mi,(N-1-mi-mk+mj)%N)
(5)      waitEvent(EP(mi,(N-1-mi-mk+mj)%N))
(6)      C += mA * B
(7)      signalEvent(EC(mi,(N-1-mi-mk+mj)%N))
(8)    end do
(9)  end

(1)  BCarrier(int mk, int mj)
(2)    mB = B
(3)    do mi=0,N-1
(4)      hop(node((N-1-mj-mk+mi)%N,mj))
(5)      waitEvent(EC((N-1-mj-mk+mi)%N,mj))
(6)      B = mB
(7)      signalEvent(EP((N-1-mj-mk+mi)%N,mj))
(8)    end do
(9)  end
```

Fig. 5 Pseudocode for matrix multiplication with full DPC in both dimensions.

ory, and 100Mbps of Ethernet connection. These workstations have a shared file system (NFS).

When the total memory use on a PE reaches or exceeds the available physical memory, performance becomes poor. This is because of paging overhead. For some algorithms, when the working set exceeds the physical memory, thrashing happens and the performance is completely unacceptable. In distributed computation, the data of a sub-problem may fit in the memory of a machine completely even if the entire problem is too large for one computer. In order to obtain fair speedup numbers, we calculate sequential timing for large problems using least squared curve fitting with a polynomial of order 3 using performance numbers collected with small problems.

In all tables, "Matrix order" means the order of matrices `A`, `B`, or `C`. "Block order" means the order of the algorithmic blocks. Table 1 lists the performance data for NavP and ScaLAPACK on a 1D PE network of three machines. It can be seen that the performance improves as we go from NavP DSC to NavP pipelining and then to NavP phase shifting. For small problems, NavP 1D DSC is only marginally slower than the corresponding sequential execution, but as the problem size grows it becomes faster. This can be seen by comparing the data in the "NAVP (1D DSC)" column with the unstarred data in the "Sequential" column (i.e., the actual data, as opposed to the data derived from curve fitting.) Table 2 indicates that with several networked computers DSC performs almost as fast as the sequential program running with enough main memory, and it is significantly faster than the sequential program pag-

Table 1 Performance of matrix multiplication on 3 PEs.

	Sequential			NavP (1D DSC)		NavP (1D pipeline)		NavP (1D phase)		ScaLAPACK[#]	
Matrix order	Block order	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
1536	128	65.44	1.00	67.22	0.97	27.72	2.36	24.55	2.67	26.80	2.44
2304	128	219.71	1.00	229.45	0.96	91.03	2.41	81.23	2.70	82.83	2.65
3072	128	520.30	1.00	543.91	0.96	205.87	2.53	189.50	2.75	211.45	2.46
4608	128	1934.73 (1745.94*)	1.00	1809.73	0.96	688.18	2.54	653.64	2.67	767.91	2.27
5376	128	3033.92 (2735.69*)	1.00	2926.24	0.93	1151.07	2.38	990.05	2.76	1173.46	2.33
6144	256	5055.93 (4268.16*)	1.00	4697.32	0.91	1811.77	2.36	1554.99	2.74	1984.18	2.15

(*) Obtained from least squared curve fitting and used in calculating speedup.
(#) ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique, not controlled by users [6].

Table 2 Performance of matrix multiplication on 8 PEs.

	Sequential			NavP (1D DSC)	
Matrix order	Block order	Time (s)	Speed up	Time (s)	Speed up
9216	128	36534.49 (13921.50*)	1.00	14959.42	0.93

(*) Obtained from least squared curve fitting and used in calculating speedup.

ing using virtual memory. With `N = 9216`, the total memory usage is about 1GB, but each of our machines has only 256MB of main memory.

Tables 3 and 4 list the performance data for MPI, NavP, and ScaLAPACK on a 2D PE network of nine machines. Again, performance improves as we hierarchically apply the three NavP transformations in the second dimension.

In both 1D and 2D cases, our DSC and pipelining programs achieve high performance. This can be attributed to the use of algorithmic blocks. The `RowCarriers` or `ACarriers`, each of which responsible for the computation of a row of algorithmic blocks or an algorithmic block, can spread out their computations to the entire network earlier than if a full distribution block on a PE has to be computed before these carriers can hop out.

The MPI implementation used for the comparison was Gentleman's Algorithm modified to use block partitioning of matrices, and with pointer swapping used to avoid unnecessary local data copying [1]. ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique [6], so the block orders in the tables do not apply to the ScaLAPACK numbers.

The performance data indicates that the NavP implementation achieves higher speedup than the MPI implementation. It would be possible to improve the performance of the MPI code by subtle fine-tuning at a cost of considerably more programming effort. Some ways that this could be done are described in Sect. 5. Nevertheless, the data makes it clear that the NavP program is faster than a straightforward implementation of Gentleman's Algorithm and competitive with a highly tuned version.

5. Comparison of Implementations

Not only does NavP bring in a new way of thinking, but the NavP implementation is also superior in performance. In the following, we compare our solution with message passing and try to explain why NavP is easier to use and faster than message passing.

5.1 Communication

In all of our sequential, NavP, and MPI implementations, we use block algorithms. The `C` matrix is partitioned into algorithmic blocks, and each physical node is assigned to a number of such blocks. The matrices `A` and `B` are partitioned in the same way as C. Figure 6 depicts an example in which the large thick box represents a physical node that hosts `C` algorithmic blocks (e.g., `c44`, `c45`, and etc.) and algorithmic blocks of `A` and `B` (e.g., `a40`, `a57`, or `b04`, `b75`, and etc.) come from west and north neighbors to participate in the computations that will contribute to the `C` algorithmic blocks. The benefit of this block algorithm is that by adjusting the order of algorithmic blocks, we can obtain the best cache and communication performance for our sequential, NavP, and MPI implementations. (For the sequential program, the block algorithm improves cache performance only.)

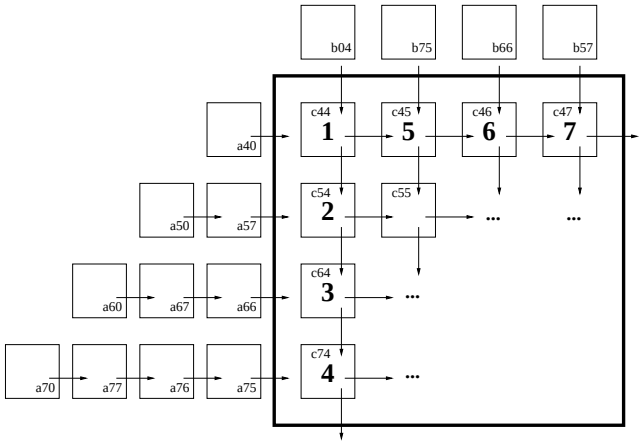

Fig. 6 One scenario of matrix multiplication using algorithmic blocks on a physical node.

We use a scenario depicted in Fig. 6 to explain how the NavP code can efficiently utilize CPU cycles and hide some of the communications. Let us suppose that, after the algorithmic block `b04` carried by a `BCarrier`

Table 3 Performance of matrix multiplication on 2×2 PEs.

		Sequential		MPI (Gentleman)		NavP (2D DSC)		NavP (2D pipeline)		NavP (2D phase)		ScaLAPACK[#]	
Matrix order	Block order	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
1024	128	19.49	1.00	6.02	3.24	7.63	2.55	5.88	3.31	5.54	3.52	5.23	3.73
2048	128	158.51	1.00	50.99	3.11	50.59	3.13	42.61	3.72	41.54	3.82	45.53	3.48
3072	128	520.30	1.00	157.53	3.30	158.06	3.29	144.09	3.61	137.39	3.79	156.27	3.33
4096	128	1281.58 (1238.21*)	1.00	367.04	3.37	362.73	3.41	328.98	3.76	321.70	3.85	417.83	2.96
5120	128	2727.86 (2373.32*)	1.00	733.91	3.23	792.23	3.00	757.67	3.13	624.87	3.80	907.16	2.62

(*) Obtained from least squared curve fitting and used in calculating speedup.

(#) ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique, not controlled by users [6].

Table 4 Performance of matrix multiplication on 3×3 PEs.

		Sequential		MPI (Gentleman)		NavP (2D DSC)		NavP (2D pipeline)		NavP (2D phase)		ScaLAPACK[#]	
Matrix order	Block order	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up	Time (s)	Speed up
1536	128	65.44	1.00	10.97	5.97	13.66	4.79	9.18	7.13	8.21	7.97	8.08	8.10
2304	128	219.71	1.00	29.95	7.34	39.53	5.56	29.93	7.34	26.74	8.22	29.39	7.48
3072	128	520.30	1.00	82.25	6.33	86.52	6.01	66.94	7.77	62.36	8.34	70.92	7.34
4608	128	1934.73 (1745.94*)	1.00	241.92	7.22	268.41	6.50	220.28	7.93	205.68	8.49	255.87	6.82
5376	128	3033.92 (2735.69*)	1.00	437.27	6.26	421.78	6.49	360.77	7.58	323.67	8.45	398.50	6.86
6144	256	5055.93 (4268.16*)	1.00	637.79	6.69	745.18	5.73	584.85	7.30	510.29	8.36	635.36	6.72

(*) Obtained from least squared curve fitting and used in calculating speedup.

(#) ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique, not controlled by users [6].

arrives, the north neighbor becomes slow for some reason and delays the `BCarrier`s of `b75`, `b66`, and `b57` from hopping into the node; meanwhile, let us also imagine that the west neighbor continues to run at a normal speed, allowing `ACarrier`s carrying `a40`, `a57`, `a66`, and `a75` and their followers `a50`, `a67`, `a76`, `a60`, `a77` and `a70` hop in as usual. The `ACarrier` of `a40` will be put to sleep after it computes with `b04` to contribute to `c44`, because the event to be signaled by the `BCarrier` of `b75` is not posted yet. Now the CPU cycles will be used for computations that contribute to `c54`, `c64`, and `c74`, as the corresponding `ACarrier`s hop in. At this time, assuming the `BCarrier`s of `b75`, `b66`, and `b57` all arrive, the `ACarrier` of `a40` will be signaled and finish the computations that contribute to `c45`, `c46`, and `c47`. (In MESSENGERS, `waitEvent(E)` falls through when the event `E` is signaled before the `waitEvent(E)` is posted.) So the computations actually happen in the order that is marked by numbers in bold font in Fig. 6. (This does not include the computations involving those algorithmic blocks of `B` that are already on this PE at the beginning of this scenario.) Since the CPU is mostly busy doing computations as the data they need (i.e., the corresponding algorithmic block pairs of `A` and `B`) become available, the communication overhead of the algorithmic blocks is mostly hidden from being seen in the overall elapsed time.

The run-time task scheduling described above is handled by the queuing mechanisms built into the MESSENGERS daemon. Thus it is handled at the system level, invisible to the application programmers. It is the NavP view that allows us to focus on describing the application level computations following their movement and to factor out the functionality associated with scheduling – code that describes behaviors at fixed locations – and put it into the MESSENGERS daemon code.

In MPI, the situation is quite different. The straightforward way to program the block implementation is to have a loop over all the algorithmic blocks of `C` that are hosted on a particular physical node. The loop introduces an artificial sequential order to the communications and computations, even though they are actually independent of each other. This artificial sequential order may result in slower performance in some scenarios. For example, if the load in the network is dynamically changing due to other users sharing some of the PEs or subnet and if the change is distributed randomly, the MPI implementation may be unable to adapt to the change efficiently because CPU cycles are wasted while waiting for a particular sub-matrix pair to arrive. In contrast, the NavP solution is able to "absorb" the impacts, as described above. Even when the load in the network is perfectly homogeneous and balanced, the best order in which to perform the subcomputations depends on the application. In matrix multiplication, it is likely to be a skewed order that is neither row-major nor column-major and may be difficult to describe with nested loops. Any predefined order that is not carefully chosen may cause unnecessary "synchronization cuts" in the network that slow down the execution. In contrast, the NavP solution performs computations as the data they need becomes available, rather than using a predefined order.

There are several ways to remove the artificial sequencing of computation in the MPI implementation. One way is to mimic the functionality of the MESSENGERS daemon by adding task-scheduling logic to the MPI application code. Because there is not a uniform way of combining task scheduling code and application code, this would need to be done separately for each application. So this shifts the burden of task scheduling from the system to the application programmer and

makes the programming task much more complicated. Another approach would use a compiler that performs dependency analysis for the code segments that are executed on a local node and assigns independent computations to different threads. This solution could be made to work, and it would be general enough to handle future applications. However, this solution involves writing a parallelizing compiler to achieve what is generally considered to be a manual programming method. And the compiler needs to be able to understand the behaviors of blocking and non-blocking `send` and `receive` and their use of buffers in MPI.

Yet another approach is to use parallel directives, such as those in HPF [9], UPC [10], or OpenMP [11], to assign independent computations to different threads. Hybrid use of MPI and OpenMP has been applied [12], and a thread-compliant implementation of MPI supporting `MPI_THREAD_MULTIPLE` in LAM/Open MPI has been developed [13]. Nevertheless, using multithreading under MPI to increase performance on every MPI node in effect requires case-by-case manual handling of an artificial computation sequencing problem that does not even exist in the NavP program. NavP does not have this problem because what a NavP programmer sees is a virtual multi-threading environment on top of networked distributed memory machines.

5.2 Cache Performance

During the execution of a block fashion sequential matrix multiplication program, an algorithmic block of `C` is updated using the products of several pairs of algorithmic blocks of `A` and `B`. This algorithmic block of `C` stays in cache for different pairs of `A` and `B` algorithmic blocks until it is fully updated.

By contrast, in our MPI implementation, since the loop over all algorithmic blocks of `C` that a physical node hosts updates all these blocks using the block pairs of `A` and `B` arrived during the last phase of communication, every triplet of `A B C` blocks are potentially fresh in cache. This may lead to less efficient cache use.

In the NavP implementation, an `ACarrier` continuously computes and contributes to the `C` algorithmic blocks as long as the corresponding algorithmic blocks of `B` are ready for use. One scenario of this is depicted in Fig. 6 in which contributions to algorithmic blocks `c45`, `c46`, and `c47` are computed by the `ACarrier` of `a40` without stop. This results in similar cache performance as the sequential execution because the `A` block stays in cache during the process.

The following numbers provide an estimate of the savings that can be achieved by better cache performance. With matrix order of $\mathtt{N} = 6144$, a block order of 256, and a 3×3 network, on average, the MPI code spent 0.334 seconds on each product of a pair of 256×256 blocks, whereas the NavP code spent 0.322 seconds. Applied to a total of $1,536$ blocks on each PE, the overall savings from a better cache performance of NavP is 18.43 seconds. This is roughly a 4% improvement from a total elapsed time of 510.29 seconds (refer to Table 4).

To achieve better cache performance in the MPI code, the program would need to hold an `A` block and look for all corresponding `B` blocks that are ready to compute. If no such `B` blocks are ready, there would have to be a "context switch" to the next `A` block. This would require a queuing mechanism, as described in Sect. 5.1, to allow the program to later return to the unfinished `A` blocks.

5.3 Initial Staggering

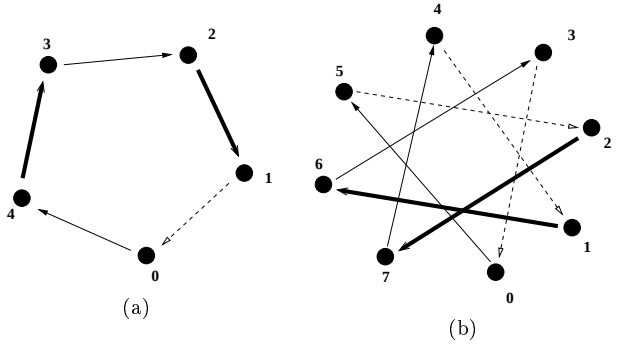

Fig. 7 (a) An example in which forward staggering takes more than two steps. $\mathtt{N} = 5$ and position shift is 1. (b) An example in which forward staggering can take three steps even if N is a power of 2 ($\mathtt{N} = 8$). The position shift is 3. Three steps if communication in node pairs $(0,5), (7,4), (6,3)$ takes place in the first step.

In the final NavP program listed in Fig. 5, "reverse staggering" is used for both matrices `A` and `B`. That is, the "chain" of a row or a column is reverse-ordered and shifted. An entry of matrix `A` on node $(\mathtt{i},\mathtt{j})$ is directly staggered to node $\mathtt{T_A}(\mathtt{i},\mathtt{j})$, where $\mathtt{T_A}(\mathtt{i},\mathtt{j}) = (\mathtt{i}, (\mathtt{N}-1-\mathtt{j}-\mathtt{i})\%\mathtt{N})$. The staggering for `B` is defined similarly: $\mathtt{T_B}(\mathtt{i},\mathtt{j}) = ((\mathtt{N}-1-\mathtt{j}-\mathtt{i})\%\mathtt{N}, \mathtt{j})$.

Assuming that a fully connected network and a collision-free switch are available, the cost of initial staggering for the `A` matrix in the NavP algorithm listed in Fig. 5 is exactly two communication steps. This can be seen by observing that $\mathtt{T_A}(\mathtt{T_A}(\mathtt{i},\mathtt{j})) = (\mathtt{i},\mathtt{j})$. Hence the staggering consists of a collection of independent swaps of `A` values between pairs of nodes, which can clearly be performed in two communication steps. (Note that for odd `N` there are nodes for which $\mathtt{T_A}(\mathtt{i},\mathtt{j}) = (\mathtt{i},\mathtt{j})$; such nodes stagger their values for free.) Similarly, the staggering for `B` can be performed in two communication steps.

The staggering of Gentleman's Algorithm is different from that of the NavP code. Gentleman's Algorithm uses "forward staggering," which shifts the positions of the entries without reversing the order. The

staggering formula for the A matrix in Gentleman's Algorithm is $\mathtt{T'_A(i,j)} = (\mathtt{i}, (\mathtt{j}-\mathtt{i})\%\mathtt{N})$. This forward staggering may require three communication steps, as illustrated in Fig. 7(a). In general, unless N is a power of two, there will be some row that requires three communication steps. (Proof: if i is the highest power of 2 that divides N, then the directed graph representing the forward staggering of row i will have an odd cycle, and hence the staggering of this row will require three communication steps.) Even when N is a power of two, special care must be taken for forward staggering to avoid wasting a communication step, as shown in Fig. 7(b). In our implementation of Gentleman's Algorithm, we do not have this mechanism in place.

Initial staggering in Cannon's Algorithm [14] moves the A entries east and the B entries south. While the staggering may look the same as NavP, it is different because the sequence of matrix entries is not reversed. The cost of initial staggering in Cannon's Algorithm is exactly the same as that of Gentleman's Algorithm.

The reverse staggering of our NavP algorithm, which is always as good as that of Gentleman's Algorithm and usually better, was not arrived at by accident. It is a direct result of our NavP methodology and our strict systematical application of the three code transformations that incrementally develop a parallel program from the sequential program. Of course, modifying the MPI algorithm to use reverse staggering is quite easy, unlike the fine tuning for improving communication overhead and cache performance discussed earlier in this section.

6. Final Remarks

We have shown that systematic application of NavP transformations yield a series of programs, each an improvement on the previous one, that constitute an incremental path from sequential matrix multiplication to a completely parallel version. The transformations are mechanical and straightforward to apply.

Our NavP matrix multiplication implementation is faster than our MPI code, as seen in Sect. 4. This is mainly because the NavP code successfully hides some of the communication overhead using an efficient but transparent run-time scheduling. This task scheduling functionality is factored out from the application code under the NavP view and put into the MESSENGERS daemon. Although it is entirely possible to achieve better task scheduling in the MPI code, with the MPI environment available today, the code that implements this would have to be developed separately for each application and interleaved with the application code. In this sense, message passing is harder to use than NavP.

Our performance numbers indicate that NavP is a promising approach, not only in terms of its simplicity but also in terms of the efficiency of the final program. Nevertheless, many questions and open research problems remain. The applicability of our method to other numerical problems, and its scalability on larger networks need to be assessed. Another key question is how well our method can handle irregular computations and problems on sparse matrices. Finally, the mechanical nature of our NavP transformations suggests that they are at least partially automatable. Building tools to perform this automation is part of our future work.

References

[1] L. Pan, W. Zhang, A. Asuncion, M.K. Lai, M.B. Dillencourt, and L. Bic, "Incremental parallelization using navigational programming: A case study," Proceedings of the 2005 International Conference on Parallel Processing (ICPP 2005), Oslo, Norway, pp.611–620, June 2005.

[2] Department of Computer Science, University of California, Irvine, Irvine, Calif., MESSENGERS User's Manual (Version 1.2.05 Beta), May 2005.

[3] L. Pan, L.F. Bic, and M.B. Dillencourt, "Distributed sequential computing using mobile code: Moving computation to data," Proceedings of the 2001 International Conference on Parallel Processing (ICPP 2001), ed. L.M. Ni and M. Valero, Los Alamitos, Calif., pp.77–84, IEEE Computer Society, Sept. 2001.

[4] W.M. Gentleman, "Some complexity results for matrix computations on parallel computers," Journal of the ACM, vol.25, no.1, pp.112–115, Jan. 1978.

[5] J.J. Modi, Parallel algorithms and matrix computation, Clarendon Press, Oxford, 1988.

[6] A.P. Petitet and J.J. Dongarra, "Algorithmic redistribution methods for block-cyclic decompositions," IEEE Transactions on Parallel and Distributed Systems, vol.10, no.12, pp.1201–1216, 1999.

[7] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," Proceedings of Supercomputing Symposium, pp.379–386, 1994.

[8] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, ScaLAPACK Users' Guide, Society for Industrial and Applied Mathematics, Philadelphia, Pa., 1997.

[9] R.S. Schreiber, "An introduction to HPF," Lecture Notes in Computer Science, vol.1132, pp.27–44, 1996.

[10] T. El-Ghazawi and S. Chauvin, "UPC benchmarking issues," Proceedings of the 2001 International Conference on Parallel Processing (ICPP 2001), ed. L.M. Ni and M. Valero, Los Alamitos, Calif., pp.365–372, IEEE Computer Society, Sept. 2001.

[11] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, Parallel Programming in OpenMP, Morgan Kaufmann Publishers, San Francisco, Calif., 2001.

[12] L. Smith and M. Bull, "Development of mixed mode MPI/OpenMP applications," Scientific Programming, vol.9, no.2–3, pp.83–98, Spring–Summer 2001.

[13] E. Gabriel, G.E. Fagg, G. Bosilca, T. Angskun, J.J. Dongarra, J.M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R.H. Castain, D.J. Daniel, R.L. Graham, and T.S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," Proceedings, 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary, pp.97–104, Sept. 2004.

[14] N. Petkov, Systolic Parallel Processing, Elsevier Science Publishers, Amsterdam, Holland, 1993.

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.

Wenhui Zhang is a PhD student in the Department of Computer Science, University of California, Irvine. Her research interests include parallelizing compilers and network and distributed systems.

Arthur Asuncion is an undergraduate student in the Department of Computer Science, University of California, Irvine. His research interests include distributed computing and artificial intelligence.

Ming Kin Lai attended California State University, Los Angeles. He received his MS degree and is a PhD Candidate at the Department of Computer Science, University of California, Irvine. He has been working on distributed and parallel computing.

Michael B. Dillencourt is an Associate Professor of Computer Science at the University of California, Irvine. He holds an MA degree in Mathematics (University of Wisconsin, 1975), an MS degree in Computer Sciences (University of Wisconsin, 1976), and a PhD in Computer Science (University of Maryland, 1988). His primary research interests are distributed computing and algorithm design and analysis.

Lubomir F. Bic is a Professor of Computer Science at the University of California, Irvine. He holds an MS degree in Informatics (Technical University of Darmstadt, Germany, 1976) and a PhD in Computer Science (University of California, Irvine, 1979). His primary research interests are distributed computing and mobile agent systems.

Laurence T. Yang is a professor in the Department of Computer Science, St. Francis Xavier University, Canada. His research interests include high performance computing and networking with applications, design and testing of embedded systems, ubiquitous/pervasive computing and intelligence, autonomic and trusted computing. He has published around 200 technical papers in referred journals, conference proceedings and book chapters in these areas.