Incremental Parallelization Using Navigational Programming: A Case Study

Lei Pan, Wenhui Zhang, Arthur Asuncion, Ming Kin Lai, Michael B. Dillencourt, and Lubomir F. Bic *

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

Abstract

We show how a series of transformations can be applied to incrementally parallelize sequential programs. Our Navigational Programming (NavP) methodology is based on the principle of self-migrating computations and is truly incremental, in that each step represents a functioning program and every 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. Our final stage is similar to the classical Gentleman's Algorithm. The NavP methodology is conducive to new ways of thinking that lead to ease of programming and high performance.

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

1. Introduction

In this paper, we show how a series of transformations can be applied to a sequential algorithm to obtain programs that represent incremental steps in exploiting parallelism in the original algorithm. The transformations are provided in Navigational Programming (NavP).

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. 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, as illustrated here, the resulting parallel programs can be elegant and efficient. The NavP methodology is conducive to different ways of thinking that lead to ease of programming and high performance.

We will briefly describe the NavP methodology in Section 2 and apply NavP to the classical problem of matrix multiplication in Section 3. The well-known message-passing solution to the same problem, i.e., Gentleman's Algorithm, is presented in Section 4. Section 5 contains performance data, followed by a brief comparison of the two implementations. Our final section includes a brief survey and comparison of some competing approaches.

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 the computation locus in order to access remotely distributed data and spread out computations. Small data is carried by the moving computation in "agent variables," which are private to a computation thread and available to the thread wherever it migrates. Large data that stays on a computer is held in "node variables" that are resident on a particular PE (processing element) and are shared by all computation threads currently on that PE. The cost of a hop() is essentially the cost of moving the data stored in agent variables plus a small amount of state data. Although the state of the computation is moved on each hop, the code is not moved. The synchronization among different migrating computations is done through "events"
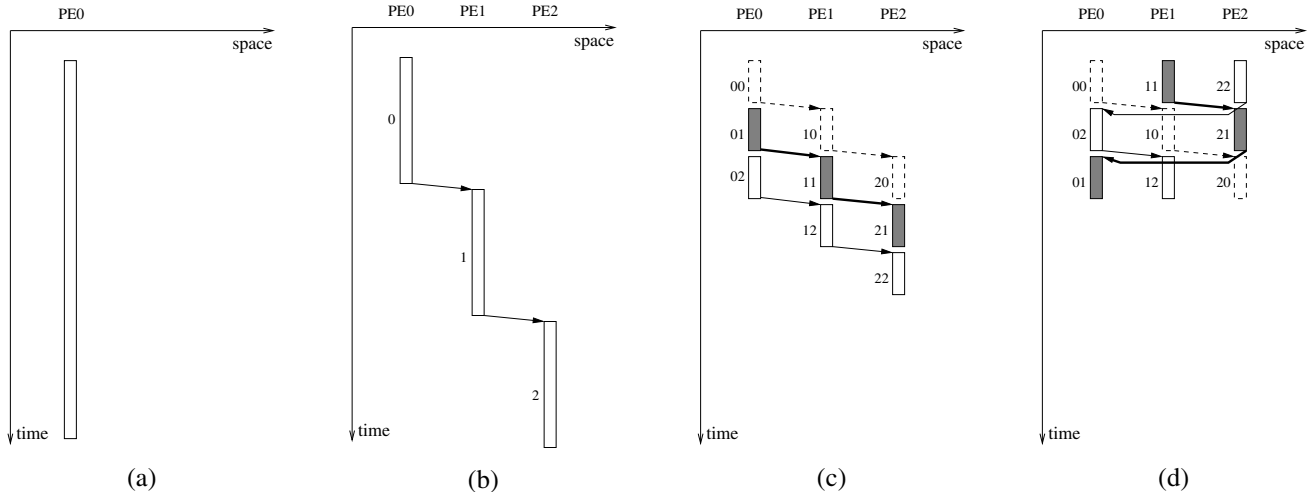
Figure 1. The code transformations in NavP. (a) Sequential. (b) DSC. (c) Pipelining. (d) Phase shifting.

(`signalEvent()` and `waitEvent()`). A programmer can "inject," or spawn, a migrating thread at command line. The injection of a thread can also be done by another thread, called a "spawner." All injections happen locally (i.e., a thread can spawn another thread only on the node on which it currently resides). Details of the underlying system of NavP, MESSENGERS, can be found online at http://www.ics.uci.edu/~bic/messengers.

NavP provides a different view of distributed computation from the classical SPMD (Single Program Multiple Data) view [14]. The SPMD view describes distributed computations at stationary locations, while the NavP view describes a computation following the movement of its locus. The NavP view changes the way distributed parallel programs are composed and provides some new benefits.

The three transformations under the NavP view are depicted in Figure 1. The 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, 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 by Figures 1(a) and (b). The resulting program performs distributed sequential computing. The immediate benefit of DSC is that, with a small amount of work, a sequential program can efficiently solve large problems that cannot fit in the main memory of one computer. By using a network of workstations, the DSC program removes paging overhead by trading it against a modest amount of network communication [13]. DSC also serves as the starting point of parallel program development

in NavP. **(2) Pipelining Transformation**: This transformation is depicted by Figures 1(b) and (c). The basic idea is to overlap the execution of multiple DSC threads by staggering their starting times. Synchronization may be necessary to ensure that the data dependencies among the DSC threads are not violated. **(3) Phase-shifting Transformation**: Sometimes the dependency among different computations allows different DSC threads to enter the pipeline from different PEs. In these situations, we can phase shift the DSC threads to achieve full parallelism, as depicted in Figures 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 that is an improvement over 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 briefly in Section 5.

3. Incremental parallelization of matrix multiplication

Matrix multiplication is a fundamental operation of many numerical algorithms. We show how the transformations of Section 2 can be repeatedly applied to incrementally parallelize matrix multiplication. Pseudocode for sequential matrix multiplication is listed in Figure 2. Throughout the paper, we assume N is the order of the square matrices.

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 mul-

```
(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
```

Figure 2. Sequential pseudocode.

```
(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
```

Figure 3. Parallel pseudocode using `doall`.

tiplication is not as straightforward as one might think. If we parallelize the two outer loops using the popular `doall` notation, as shown in Figure 3, contention could happen as multiple PEs request the same entries at the same time. On the other hand, if we cache multiple copies of the same entry on the PEs that require it, we have a non-scalable solution. 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 [9, 12].

Throughout this paper, we describe the problem and our solution at a fine granularity level for simplicity. That is, we assume $N == P$, where P is either the number of PEs in a one-dimensional (1D) processor network or the order of a two-dimensional (2D) processor network. To extend our solution to a coarser level, we simply need to take each and every element (e.g., `C01` or `A21`) as a sub-matrix block, instead of an entry of the matrix.

3.1. From sequential to DSC

We first apply DSC Transformation to sequential matrix multiplication, as depicted in Figure 4 where we assume $N = 3$. This DSC transformation essentially distributes the computation in the j dimension. The PE network is 1D in which each PE has a unique identifier $HnodeID = 0, 1, ..., N - 1$ from west to east. Again, the arrows represent hop() operations. Thick boxes contain node variables on different machines, and thin boxes carry agent

variables. All PEs are assumed to be fully connected via a collision-free switch, rather than being connected as a ring. This assumption is true for most modern hardware environments, and it makes the initial staggering (i.e., moving the entries of the three matrices to the right places before any computation begins) faster, because each matrix entry can be shipped to any destination directly instead of having to go stepwise through a number of intermediate PEs.
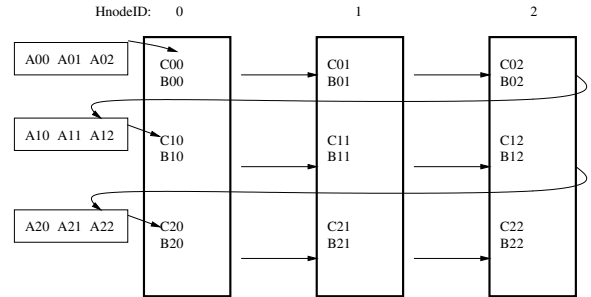

Figure 4. DSC.

Pseudocode for DSC matrix multiplication is listed in Figure 5. In the pseudocode hereafter, A and B indicate node variables, whereas `mA` and `mB` represent agent variables. [1] Matrix A is loaded into agent variable `mA` and carried by the migrating thread.

In Figure 4, matrix A is initially put on the PE with $HnodeID = 0$, and the columns of matrices B and C are distributed such that $B(*, j)$ and $C(*, j)$ are on the PE with $HnodeID = j$. In Figure 5, $node(j)$ maps to the PE that hosts column j of matrices B and C. Every time the computation thread hops back to $node(0)$, it will pick up a different row of matrix A for the computation of the loop over j.

```
(1)  hop(node(0))
(2)  inject(RowCarrier)

(1)  RowCarrier
(2)    do mi=0,N-1
(3)      do mj=0,N-1
(4)        hop(node(mj))
(5)        if(mj=0) mA(*) = A(mi,*)
(6)        t = 0.0
(7)        do k=0,N-1
(8)          t += mA(k) * B(k)
(9)        end do
(10)       C(mi) = t
(11)     end do
(12)   end do
(13) end
```

Figure 5. Pseudocode for DSC.

---

[1] In our NavP programs, we adapt a naming convention of starting an agent variable's name with a lowercase m.

3.2. DSC pipelining

We apply our Pipelining Transformation to the DSC code obtained from the previous step. The result of this transformation is depicted in Figure 6. Each row of matrix A is assigned to a different computation thread. Injected into the PE pipeline in order, these threads follow each other in the network to compute the corresponding C entries.
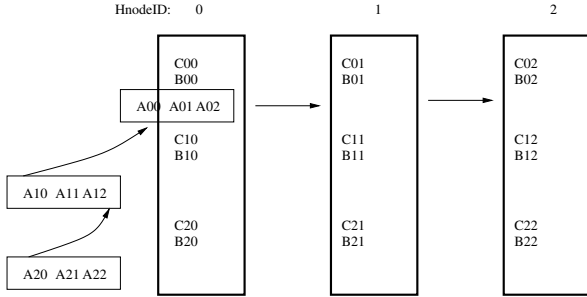

Figure 6. DSC with pipelining.

Pseudocode for pipelined DSC matrix multiplication is listed in Figure 7. The matrix A is initially put on the PE with HnodeID = 0, and the columns of matrices B and C are distributed such that $B(*, j)$ and $C(*, j)$ are on the PE with HnodeID = j.

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

(1)   RowCarrier(int mi)
(2)     mA(*) = A(mi,*)
(3)     do mj=0,N-1
(4)       hop(node(mj))
(5)       t = 0.0
(6)       do k=0,N-1
(7)         t += mA(k) * B(k)
(8)       end do
(9)       C(mi) = t
(10)    end do
(11) end
```

Figure 7. Pseudocode for pipelined DSC.

3.3. From DSC to full DPC

We apply our Phase-shifting Transformation to achieve a full DPC, as depicted in Figure 8. This is possible because each row of A, though needed on all three PEs, can start its computation from any PE.

Pseudocode for phase-shifted DPC matrix multiplication is listed in Figure 9. Rows of matrix A are carried by the corresponding agent variables mA.
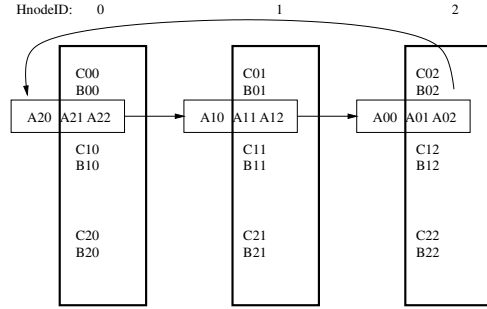

Figure 8. Full DPC through phase shifting.

In this full DPC implementation, matrix A is initially distributed such that $A(i, *)$ is on the PE with HnodeID = i, and the columns of matrices B and C are distributed such that $B(*, j)$ and $C(*, j)$ are on the PE with HnodeID = j.

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

(1)   RowCarrier(int mi)
(2)     mA(*) = A(*)
(3)     do mj=0,N-1
(4)       hop(node((N-1-mi+mj)%N))
(5)       t = 0.0
(6)       do k=0,N-1
(7)         t += mA(k) * B(k)
(8)       end do
(9)       C(mi) = t
(10)    end do
(11) end
```

Figure 9. Pseudocode for phase-shifted DPC.

3.4. DSC in the second dimension

What we have achieved so far (Figure 8) is a 1D DPC consisting of phase-shifted pipelined computations in which the rows move through the pipeline. What we will do in the next three steps is achieve further parallelization by introducing a second dimension, effectively letting each entry of each row move through a pipeline.

The first step is to introduce a 2D network in which each PE has a unique 2D identifier (HnodeID, VnodeID), where HnodeID = 0, 1, ..., N − 1 from west to east, and VnodeID = 0, 1, ..., N − 1 from north to south. Then the DSC Transformation is applied in the second dimension, as depicted in Figure 10. Essentially, this DSC transformation further distributes the computations in the i dimension.

Pseudocode for DSC in the second dimension is listed in Figure 11. The rows of matrix A and columns of ma-
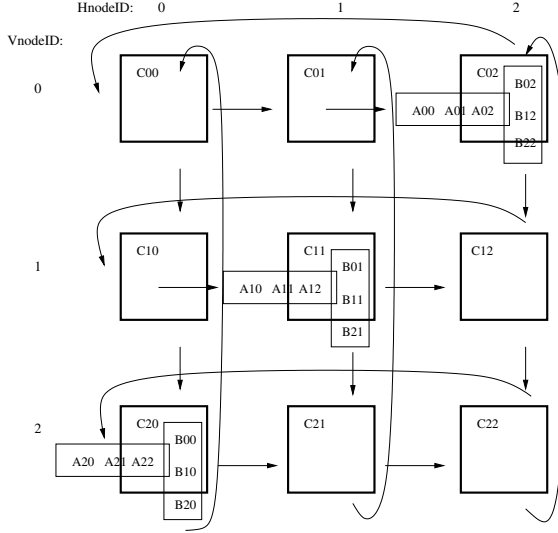
Figure 10. DSC in the second dimension.

```
(1)   do ml=0,N-1
(2)     hop(node(N-1-ml,ml))
(3)     inject(RowCarrier(N-1-ml))
(4)     inject(ColCarrier(ml))
(5)   end do

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

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

Figure 11. Pseudocode for DSC in the 2nd dimension.

trix B are carried in their corresponding agent variables mA and mB, respectively. The ColCarriers ship the B columns, and the RowCarriers use these B columns to compute with the A rows that they carry. The events are necessary because the consumers, i.e., the RowCarriers, need to hold on their computations until the producers, i.e., the ColCarriers, finish putting the columns of B in place.

The matrices are initially distributed, as shown in Figure 10, such that $A(N-1-l,*)$ and $B(*,l)$ are on $node(N-1-l,l)$, and $C(i,j)$ (initialized to 0) is on $node(i,j)$, where $node(i,j)$ maps to the PE that hosts entry (i,j) of matrix C.

3.5. DSC with pipelining in both dimensions

We apply our Pipelining Transformation in both dimensions, as depicted in Figure 12. Basically, 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. This is the reason for a second event $EC(.,.)$.

Pseudocode for DSC with pipelining in both dimensions is listed in Figure 13. The entries of matrices A and B are carried in their corresponding agent variables mA and mB, respectively.

The matrices are initially distributed, as shown in Figure 12, such that $A(N-1-l,*)$ and $B(*,l)$ are on $node(N-1-l,l)$, and $C(i,j)$ (initialized to 0) is on $node(i,j)$. An event $EC(i,j)$ is signaled on $node(i,j)$ for all values of i,j initially.
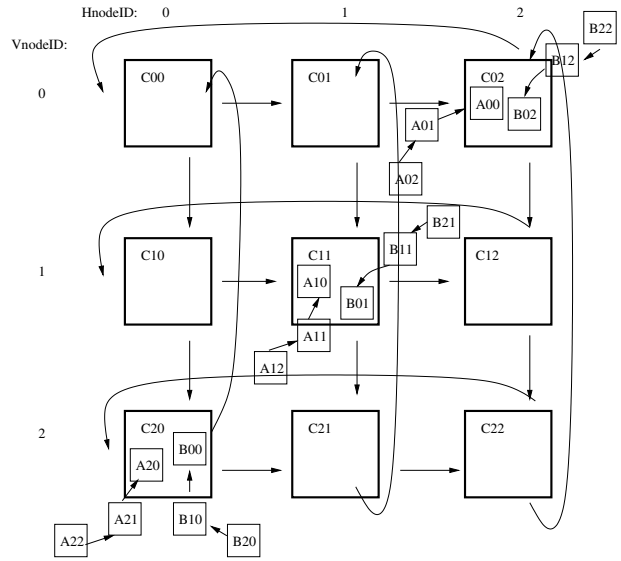

Figure 12. DSC pipelining in both dimensions.

3.6. Full DPC in both dimensions

We apply our Phase-shifting Transformation in both dimensions to achieve full parallelization, as depicted Figure 14.

Pseudocode for DPC in both dimensions is listed in Figure 15. The entries of matrices A and B are carried in their

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

(1)  spawner(int ml)
(2)    do mk=0,N-1
(3)      inject(ACarrier(N-1-ml,mk))
(4)      inject(BCarrier(mk,ml))
(5)    end do
(6)  end

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

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

Figure 13. Pseudocode for DSC pipelining in both dimensions.

```
(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
```
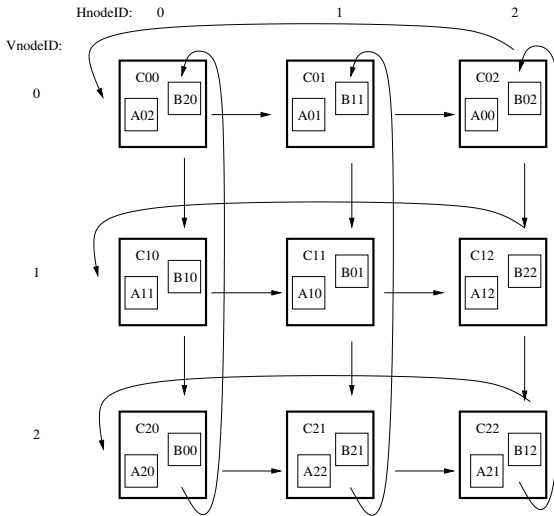
Figure 15. Pseudocode for full DPC in both dimensions.

Figure 14. Phase shifting in both dimensions.

corresponding agent variables mA and mB, respectively.

The matrices are initially distributed such that $A(i,j)$, $B(i,j)$ and $C(i,j)$ (initialized to 0) are on $node(i,j)$.

In the above figures such as Figure 14, 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 [16]. 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). Our sequential and MPI (Message Passing Interface) implementations described below use algorithmic blocks as well.

4. Gentleman's Algorithm

Gentleman's Algorithm [9, 19] is a classical SPMD algorithm for parallel matrix multiplication. The pseudocode is listed in Figure 16, in which an arrow represents a combination of receive and send among remote PEs. During initial staggering, each entry of matrix A will stagger i times to the west, where i is the entry's row number, and each entry in matrix B will stagger j times to the north, where j is the entry's column number. An entry can be either a single value or a sub-matrix. Thus, a skewed transformation of matrices A and B results. Like the NavP pseudocode, our MPI implementation assumes a fully connected

```
(1)  do k=0,N-2
(2)    doall node(i,j) where 0<=i,j<=N-1
(3)      if i>k then
(4)        A ← east(A)
(5)      end if
(6)      if j>k then
(7)        B ← south(B)
(8)      end if
(9)    end do
(10) end do

(11) doall node(i,j) where 0<=i,j<=N-1
(12)   C = A * B
(13) end do
(14) do k=0,N-2
(15)   doall node(i,j) where 0<=i,j<=N-1
(16)     A ← east(A)
(17)     B ← south(B)
(18)     C += A * B
(19)   end do
(20) end do
```

Figure 16. Pseudocode for Gentleman's Alg.

network, and matrix staggering is accomplished in a single step (not shown in Figure 16) rather than in a series of steps. Throughout the entirety of Gentleman's Algorithm, matrix C remains stationary.

Once the initial staggering completes, matrices A and B are multiplied and the results are placed in matrix C. For $N - 1$ iterations, matrix A shifts its columns one step to the west and matrix B shifts its rows one step to the north, and A and B are multiplied with the results added to the C matrix.

In our implementation, non-blocking receives (i.e., MPI_Irecv()) are used in conjunction with blocking sends to prevent deadlocking. MPI_Wait(), which blocks until the incoming matrix has been received, assists in providing synchronization between PEs.

As a result of using algorithmic blocks, many blocks are shifted from a PE to itself during the computation. Instead of sending an algorithmic block to a PE itself, or copying an algorithmic block from a local memory, we use pointer swapping to shift an algorithmic block locally.

5. 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 in Donald Bren School of Information & Computer Sciences, University of California Irvine. The message passing system used was LAM 7.0.6 from Indiana University [3]. The ScaLAPACK used was version 1.7 from University of Tennessee, Knoxville and Oak Ridge National Laboratory [2]. 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 memory, and 100Mbps of Ethernet connection. These workstations have a shared file system (NFS).

When the working set of a sequential program exceeds the physical memory on a PE, thrashing happens and the performance degrades. In a distributed program, however, the data of a sub-problem may fit in the memory of a machine completely even if the entire problem is too large. 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.

Table 1 lists the performance data for NavP and ScaLA-PACK 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; however, as the problem size grows, NavP 1D DSC becomes faster, as indicated by the data from actual runs (not curve-fitted data). 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 paging using virtual memory. With $N = 9216$, the total memory usage is about 1GB, but our machines each have 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; moreover, pointer swapping was used in order to avoid unnecessary local data copying. ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique [16], so the block orders in the tables do not apply to the ScaLA-PACK numbers.

The performance data indicates that the NavP implementation achieves a higher speedup than the MPI implementation. Some differences between these two implementations are discussed briefly below. More details can be found in our full-length technical report [15].

1. **Communication.** We use block algorithms for better cache and communication performance. The algo-

Table 1. Performance on 3 PEs

Matrix order	Block order	Sequential Time (s)	Speed up	NavP (1D DSC) Time (s)	Speed up	NavP (1D pipeline) Time (s)	Speed up	NavP (1D phase) Time (s)	Speed up	ScaLAPACK(#) 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 [16].

Table 2. Performance on 8 PEs

Matrix order	Block order	Sequential Time (s)	Speed up	NavP (1D DSC) 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.

rithmic blocks of C on a PE can be updated in different orders. In the case of NavP, the order is not pre-defined and the CPU cycles are thus efficiently utilized in computations as the data they need arrives. An efficient run-time task scheduling, handled by the queuing mechanisms built into the MESSENGERS daemon, is provided to the NavP programmers. As a result, NavP programmers only need to concern themselves with the two event handling commands as the interface to the queuing mechanisms that are otherwise hidden at the system level. 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. 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 on a PE. The loop introduces an artificial sequential order to the communications and computations even though they are actually independent of each other and hence may result in slower performance. Possible ways to remove the artificial sequencing are proposed [15], but they all require significantly more programming work.

2. **Cache performance.** The NavP and the sequential programs have a similar cache performance because in both cases during the execution there is an algorithmic block (of C for the sequential program and of A for the NavP program, respectively) that would stay in the cache for the duration of computation using other two algorithmic blocks. In contrast, in the block-oriented MPI program, triplets of A B C blocks are frequently fresh in the cache, which leads to less efficient cache use. A simple analysis shows that this cache performance of NavP can account for as much as a 4% improvement over MPI [15].

3. **Initial staggering.** The NavP program uses "reverse staggering" for matrices A and B. That is, the "chain" of a row or a column is both shifted and reverse-ordered. In contrast, both Gentleman's Algorithm and Cannon's Algorithm [4, 17] use "forward staggering," which only shifts the positions of the entries without reversing the order. It is shown [15] that reverse staggering never requires more than two communication phases, while forward staggering often requires three communication phases.

It would be possible to improve the performance of the MPI code by subtle fine-tuning at a cost of considerably more programming effort. 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.

6. Final remarks

In incremental parallelization, a programmer uses sequential code as the starting point and introduces parallelism in a step by step fashion, until satisfactory performance is achieved or a time/resource constraint is reached. Oftentimes, programmers begin with the performance critical "hot spots" in a program and gradually parallelize other parts of the program.

Shared-memory programming is believed to be more programmable and more amenable to incremental parallelization [11]. The reason is that data need not be distributed among the processors (in the case of DSM (Distributed Shared Memory) [18] or HPF (High Performance Fortran) [20], data is distributed but a logical single address space is provided). Shared-memory programs are similar to the familiar sequential original codes, and therefore the transition is easier for programmers. Some programming languages (e.g., HPF or UPC(Unified Parallel C) [7]) provide special language constructs such as `doall` or `!HPF$ INDEPENDENT`, so ideally parallelization is as simple as changing `do` loops to `doall` loops. In OpenMP [5], parallel directives (e.g., `!$ OMP PARALLEL`)

Table 3. Performance on 2×2 **PEs**

Matrix order	Block order	Sequential Time (s)	Speed up	MPI (Gentleman) Time (s)	Speed up	NavP (2D DSC) Time (s)	Speed up	NavP (2D pipeline) Time (s)	Speed up	NavP (2D phase) Time (s)	Speed up	ScaLAPACK(#) 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 [16].

Table 4. Performance on 3×3 **PEs**

Matrix order	Block order	Sequential Time (s)	Speed up	MPI (Gentleman) Time (s)	Speed up	NavP (2D DSC) Time (s)	Speed up	NavP (2D pipeline) Time (s)	Speed up	NavP (2D phase) Time (s)	Speed up	ScaLAPACK(#) 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 [16].

can be used to parallelize any program segment (called "parallel region") that the programmer chooses. Unfortunately, although changing do's to doall's or using OpenMP parallel regions in matrix multiplication does exploit parallelism in the algorithm, both of these methods will cause communication contention for a "zero-inventory" implementation, as pointed out in Section 3.

For better performance, programmers must take care of data distribution explicitly (e.g., HPF or UPC provides such mechanism), and hence the advantage of not needing explicit data distribution on shared-memory is weakened. OpenMP is targeted mainly at SMP (Symmetric MultiProcessor) architectures, and therefore does not provide the opportunity for its programmers to specify data distribution. Consequently, the OpenMP implementations on distributed memory (with an underlying DSM system such as TreadMarks [1]) have seen less satisfying performance. The reported speedups on SMP clusters for OpenMP are within 7-30% of those of MPI implementations [10].

Message passing programming is less amenable to incremental parallelization. Transforming a sequential program into a message-passing one is an abrupt break, since data must be distributed and code structure is often dramatically changed. This is seen in the matrix multiplication example – one either gets no parallelism at all with the sequential code, or one gets all parallelism with Gentleman's Algorithm. Going directly from the sequential code to a parallel algorithm such as Gentleman's Algorithm requires considerable ingenuity. Nevertheless, message passing programming usually leads to good performance. This phenomenon can be attributed to the message passing programmers' explicit control of data distribution and careful avoidance of communication contention and extra data movement.

In NavP, the DSC Transformation involves data distribution and insertion of migration statements (i.e., hop()). The other two code transformations exploit parallelism by decomposing the long DSC threads and properly managing the synchronization among the shorter threads. The programmability of NavP is similar to that of HPF in that they both require explicit control of data distribution and explicit synchronization (through the use of barriers, events, critical regions, etc. in HPF, and events in NavP). Similar to HPF, synchronization errors are more likely to happen in NavP than in message passing. Unlike HPF, NavP requires its programmers to handle details in communication by using agent variables to carry data around. As a result, the NavP programmers know exactly how much is communicated to where at what time. NavP composes parallel code from shorter DSC threads, and the parallel code is structurally the same as the original sequential code. This property of NavP is referred to as Algorithmic Integrity [13].

Our NavP matrix multiplication implementation is faster than our MPI code. 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 will have to be developed for each and every application and will be interleaved with the application code. In this sense, message passing is harder to use than NavP.

A hybrid use of MPI and OpenMP [6, 21], with

OpenMP's multi-threading capability used for the computation on a computer node, is another way of introducing efficient run-time scheduling. Traditionally, multi-threading and message passing are significantly different methods rooted in two different architectures – shared-memory and message-passing architectures. Recent years have seen a trend of merging these two different styles of parallel programming in order to efficiently program the next generation supercomputers: cluster of multi-processor systems. Some examples include a thread-compliant implementation of MPI supporting `MPI_THREAD_MULTIPLE` in LAM/Open MPI [8] and a hybrid use of MPI and OpenMP. NavP is a uniform methodology that conveniently provides the combined functionalities of message passing and multi-threading, using navigational commands and synchronization commands.

Our NavP methodology uses highly mechanical and incremental steps to guide programmers in achieving elegant implementations with superior performance. The NavP transformations are at least partially automatable. Building tools to automate them is part of our future work.

Acknowledgements

References

[1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[2] 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.

[3] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[4] L. E. Cannon. *A cellular computer to implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

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

[6] E. Chow and D. Hysom. Assessing performance of hybrid MPI/OpenMP programs on SMP clusters. Technical Report UCRL-JC-143957, Lawrence Livermore National Laboratory, Livermore, Ca., 2001.

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

[8] 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. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 2004.

[9] W. M. Gentleman. Some complexity results for matrix computations on parallel computers. *Journal of the ACM*, 25(1):112–115, Jan. 1978.

[10] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. In *Proceedings IPPS/SPDP*, pages 302–310. IEEE Computer Society Press, 1999.

[11] C. Leopold. *Parallel and Distributed Computing: A Survey of Models, Paradigms, and Approaches*. John Wiley & Sons, New York, 2001.

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

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

[14] L. Pan, L. F. Bic, M. B. Dillencourt, and M. K. Lai. NavP versus SPMD: Two views of distributed computation. In T. Gonzalez, editor, *Proceedings of the Fifteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, volume 2, Algorithms, pages 666–673, Anaheim, Calif., Nov. 2003. ACTA Press.

[15] L. Pan, W. Zhang, A. Asuncion, M. K. Lai, M. B. Dillencourt, and L. F. Bic. Incremental parallelization using navigational programming: A case study. School of Information & Computer Sciences Technical Report TR# 05-04, University of California, Irvine, Irvine, CA, Mar. 2005.

[16] A. P. Petitet and J. J. Dongarra. Algorithmic redistribution methods for block-cyclic decompositions. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1201–1216, 1999.

[17] N. Petkov. *Systolic Parallel Processing*. Elsevier Science Publishers, Amsterdam, North-Holland, 1993.

[18] J. Protic, M. Tomasevic, and V. Milutinovic, editors. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society, Los Alamitos, CA, 1998.

[19] M. J. Quinn. *Parallel computing theory and practice*. McGraw-Hill, 1994.

[20] R. S. Schreiber. An introduction to HPF. *Lecture Notes in Computer Science*, 1132:27–44, 1996.

[21] L. Smith and M. Bull. Development of mixed mode MPI/OpenMP applications. *Scientific Programming*, 9(2–3):83–98, Spring–Summer 2001.