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 a sequential program to obtain programs that represent successive steps in exploiting parallelism in the original algorithm. Our Navigational Programming (NavP) methodology is based on the principle of self-migrating computations. Our methodology for achieving parallelism is truly incremental, in the sense 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 step is similar to the classical Gentleman’s Algorithm, but there are some important differences. We present a detailed performance comparison. The NavP methodology is conducive to new ways of thinking that lead to ease of programming and high performance. A closer look at our NavP algorithm suggests some ways in which message-passing implementations could be improved.

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). NavP gives us a different methodology toward how one goes about developing parallel programs on distributed memory computers.

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 the resulting parallel programs are 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 detailed comparison of the two implementations in Section 6. 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 [1]. 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,” while 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 of the underlying system of NavP, MESSENGERS, can be found in our other publications [2, 3].

NavP provides a different view of distributed computation from the classical SPMD (Single Program Multiple Data) view [4]. The SPMD view describes distributed computations at stationary locations, while the NavP view describes a computation following the movement of its locus. This different view changes the way distributed parallel programs are composed and provides new benefits that will be seen from the case study below.

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 (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 Figures 1(a) and (b). The resulting program performs “Distributed Sequential Computing,” or DSC. 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 has completely removed paging overhead by trading it against a modest amount of network communication [5]. DSC also serves as the starting point of parallel program development in NavP. (2) **Pipelining Transformation**: This transformation is depicted using Figures 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 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 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 Section 6.

3. **Incremental parallelization of matrix multiplication**

Matrix multiplication is a fundamental operation of many numerical algorithms. 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 multiplication is not as straightforward as one might think. Suppose we parallelize the two outer loops using the popular doall notation, as shown in Figure 3. We can get, for example, two concurrent state-
ments run by two PEs:

\begin{align*}
(1) & \text{do all } i=0,N-1 \\
(2) & \text{do all } j=0,N-1 \\
(3) & C(i,j) = 0.0 \\
(4) & \text{do } k=0,N-1 \\
(5) & C(i,j) += A(i,k) \ast B(k,j) \\
(6) & \text{end do} \\
(7) & \text{end do all} \\
(8) & \text{end do all}
\end{align*}

Figure 2. Pseudocode for sequential matrix multiplication.

\begin{align*}
(1) & \text{do all } i=0,N-1 \\
(2) & \text{do all } j=0,N-1 \\
(3) & C(i,j) = 0.0 \\
(4) & \text{do } k=0,N-1 \\
(5) & C(i,j) += A(i,k) \ast B(k,j) \\
(6) & \text{end do} \\
(7) & \text{end do all} \\
(8) & \text{end do all}
\end{align*}

Figure 3. Pseudocode for parallel matrix multiplication using doall.

We apply DSC Transformation to sequential matrix multiplication, as depicted in Figure 4. The essence of this DSC transformation is to distribute the computation in the loop over the PE 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

In this section, we provide a solution that does not trigger contention (i.e., the situation when 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).

In the following, we describe the problem and our solution at a fine granularity level for simplicity. That is, we assume \( N = 2 \), 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 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.

In the following, we describe the problem and our solution at a fine granularity level for simplicity. That is, we assume \( N = 2 \), 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 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

In this section, we describe the problem and our solution at a fine granularity level for simplicity. That is, we assume \( N = 2 \), 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 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.

In the following, we describe the problem and our solution at a fine granularity level for simplicity. That is, we assume \( N = 2 \), 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 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.
3.2. DSC pipelining

We apply Pipelining Transformation to the DSC code obtained from last step. This is depicted in Figure 6. Now 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.

```
(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 matrix multiplication.

```
(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 matrix multiplication.

3.3. From DSC to full DPC

We apply Phase-shifting Transformation to achieve 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.

```
HnodeID

0  1  2

A00 A01 A02

A10 A11 A12

A20 A21 A22

Figure 7. Pseudocode for pipelined DSC matrix multiplication.

Figure 8. Full DPC through phase shifting.
```

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

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

3.4. DSC in the second dimension

We 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, and apply DSC Transformation in the second dimension, as depicted in Figure 10. The essence of this DSC trans-
The matrices are initially distributed such that $A(N-1-1,*)$ and $B(*,1)$ are on node$(N-1-1,1)$, 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 $C$.

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

Figure 11. Pseudocode for matrix multiplication with DSC in the second dimension.

3.5. DSC with pipelining in both dimensions

We apply 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 $B$Carrier 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 $EC(i,)$. .

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 such that $A(N-1-1,*)$ and $B(*,1)$ are on node$(N-1-1,1)$, 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.
3.6. Full DPC in both dimensions

We apply 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 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 [8]. 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 $(HnodeID = 1, VnodeID = 1)$ in Figure 14 (assuming the entire PE network is the upper-left $2 \times 2$ processors), we can see algorithmic blocks as depicted by lowercase letters (e.g., a57 or c46) in Figure 17 of Section 6.1. As an example, the distribution block of C11 in Figure 14 is decomposed into algorithmic blocks contained in the thick box (which indicates a physical node) in Figure 17. Our sequential and MPI (Message Passing Interface) implementations described below use algorithmic blocks as well.

![Figure 12. DSC pipelining in both dimensions.](image)

![Figure 13. Pseudocode for matrix multiplication with DSC pipelining in both dimensions.](image)

4. Gentleman’s Algorithm

Gentleman’s Algorithm [6, 9] is a classical SPMD algorithm for parallel matrix multiplication. The pseudocode is listed in Figure 16, in which an arrow represents a receive from a remote PE, which needs to call a send in order to complete the communication. 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 network, and matrix staggering is accomplished directly (not shown in Figure 16) instead of stepwise (as shown in Figure 16). For example, an entry of matrix $A$ on node $(i, j)$ will be directly staggered to node $(i, (j - i) \% N)$. In other words, the transformation for direct staggering is $T_A(i, j) = (i, (j - i) \% N)$. 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., 
\text{MPI}_{\text{Irecv}}()) are used in conjunction with blocking sends to prevent deadlocking. \text{MPI}_{\text{Wait}}(), which blocks until the incoming matrix has been received, assists in providing synchronization between PEs.

A consequence of needing to receive matrices before sending matrices is that two temporary buffer matrices must be allocated on each PE in order to store the incoming A and B matrices. On each iteration, these temporary matrices are used to receive the incoming matrices, and these matrices are the ones multiplied together, with the results added to the C matrix. At the end of each iteration, the temporary A and B matrices swap pointers with the actual A and B matrices.

As a result of using algorithmic blocks, many blocks are shifted from a PE to itself during the computation. Instead of using MPI to send 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 [3]. The message passing system used was LAM 7.0.6 from Indiana University [10]. The ScaLAPACK used was version 1.7 from University of Tennessee, Knoxville and Oak Ridge National Laboratory [11]. 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 total memory use on a PE reaches or exceeds the available physical memory, performance becomes poor. This is because of a 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 ob-

\begin{verbatim}
(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
\end{verbatim}
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. ScALAPACK uses a logical LCM hybrid algorithmic blocking technique [8], 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 Section 6. 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. 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.

6.1. 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 17 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).

We use a scenario depicted in Figure 17 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 BCcarrier arrives, the north neighbor becomes slow for some reason and delays the BCcarriers of b75, b66, and b57 from hopping into the node; meanwhile, let us also imagine that the west neighbor...
Table 1. Performance of matrix multiplication on 3 PEs

<table>
<thead>
<tr>
<th>Matrix order</th>
<th>Block order</th>
<th>Time (s)</th>
<th>Speed up</th>
<th>Time (s)</th>
<th>Speed up</th>
<th>Time (s)</th>
<th>Speed up</th>
<th>Time (s)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>128</td>
<td>65.44</td>
<td>1.10</td>
<td>67.22</td>
<td>0.97</td>
<td>27.72</td>
<td>2.36</td>
<td>24.55</td>
<td>2.60</td>
</tr>
<tr>
<td>2304</td>
<td>128</td>
<td>219.71</td>
<td>1.00</td>
<td>229.45</td>
<td>0.96</td>
<td>91.03</td>
<td>2.41</td>
<td>81.23</td>
<td>2.70</td>
</tr>
<tr>
<td>3072</td>
<td>128</td>
<td>520.30</td>
<td>1.00</td>
<td>543.91</td>
<td>0.96</td>
<td>205.87</td>
<td>2.53</td>
<td>189.50</td>
<td>2.75</td>
</tr>
<tr>
<td>4608</td>
<td>128</td>
<td>1934.73</td>
<td>1.00</td>
<td>1809.73</td>
<td>0.96</td>
<td>688.18</td>
<td>2.54</td>
<td>653.64</td>
<td>2.67</td>
</tr>
<tr>
<td>5376</td>
<td>128</td>
<td>3033.92</td>
<td>1.00</td>
<td>2926.24</td>
<td>0.93</td>
<td>1151.07</td>
<td>2.38</td>
<td>990.05</td>
<td>2.76</td>
</tr>
<tr>
<td>6144</td>
<td>256</td>
<td>5055.93</td>
<td>1.00</td>
<td>4697.32</td>
<td>0.91</td>
<td>1811.77</td>
<td>2.36</td>
<td>1554.99</td>
<td>2.74</td>
</tr>
</tbody>
</table>

(*) Obtained from least squared curve fitting and used in calculating speedup.
(#) ScaLAPACK uses a logical LCM hybrid algorithmic blocking technique, not controlled by users [8].

Table 2. Performance of matrix multiplication on 8 PEs

<table>
<thead>
<tr>
<th>Matrix order</th>
<th>Block order</th>
<th>Time (s)</th>
<th>Speed up</th>
<th>Time (s)</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>9216</td>
<td>128</td>
<td>3653.49</td>
<td>1.00</td>
<td>14959.42</td>
<td>0.93</td>
</tr>
</tbody>
</table>

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

Figure 17. One scenario of matrix multiplication using algorithmic blocks on a physical node.

continues to run at a normal speed, allowing ACarrier 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 ACarriers hop in. At this time, assuming the BCarriers of b75, b66, and b57 all arrive, the ACarrier of a40 will be woken up and finish the computations that contribute to c45, c46, and c47 in a row respectively (in MESSENGERS, waitEvent(E) will fall through if 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 Figure 17. (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 sub-computations 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...
There are several possible 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 OpenMP, to assign independent computations to different threads. Hybrid use of MPI and OpenMP has been successfully applied [12, 13], but using multi-threading under MPI to increase performance on each and every MPI node in effect requires case-by-case manual handling of the artificial computation sequencing that did not even exist in the NavP program in the first place. 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.

### 6.2. 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 $A$, $B$, and $C$ 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 Figure 17 in which contributions to algorithmic blocks $c_{45}$, $c_{46}$, and $c_{47}$ are computed by the $ACarrier$ of $a_{40}$ 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 estimate how much savings a better cache performance can obtain. With matrix order of $N = 6144$, a block order of $256$, and a $3 \times 3$ network, on average, the MPI code spent $0.334$ seconds on each product of a pair of $256 \times 256$ blocks, while 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, one needs to hold an A block and look for all corresponding B blocks that are available to compute, and “context switch” to the next A block when no B blocks for this A block are ready or when the computations for this A block are all done. A queuing mechanism, as described in Section 6.1, is needed because we need to be able to come back to the A blocks that are unfinished.

6.3. Initial staggering

Figure 18. An example in which forward staggering takes more than two steps. \( N = 5 \) and position shift is 1.

In the final NavP program listed in Figure 14, “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 \((i,j)\) is directly staggered to node \(T_A(i,j)\), where \(T_A(i,j) = (i,(N - 1 - j - i) \% N)\). The staggering for B is defined similarly: \(T_B(i,j) = ((N - 1 - j - 1) \% N, j)\).

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 Figure 15 is exactly two communication steps. This can be seen by observing that \(T_A(T_A(i,j)) = (i,j)\). 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 \( T_A(i,j) = (i,j) \); 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 only shifts the positions of the entries without reversing the order. The staggering formula for the A matrix in Gentleman’s Algorithm is \(T'_A(i,j) = (i,(j-i) \% N)\). This forward staggering may require three communication steps, as illustrated in Figure 18. 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 in order to avoid wasting a communication step, as shown in Figure 19. In our implementation of Gentleman’s Algorithm, we do not have this mechanism in place.

Initial staggering in Cannon’s Algorithm [14, 15] 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.

7. Final remarks

In incremental parallelization, a programmer uses a sequential code as the starting point and exploits and introduces parallelism step by step incrementally, 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 by many to be more programmable and more amenable to incremental parallelization [16]. The reason is that data need not be distributed among the processors (in the case of DSM (Distributed Shared Memory) [17] or HPF (High Performance Fortran) [18], 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) [19]) provide special language constructs such as \texttt{doall} or \texttt{forall}, so ideally parallelization is as simple as changing \texttt{do} or \texttt{for}
Figure 19. An example in which forward staggering can take two or three steps. \( N = 8 \) and position shift is 3. (a) Three steps if communication in node pairs \((0, 5), (7, 4), (6, 3)\) takes place in the first step. (b) Two steps if communication in node pairs \((0, 5), (6, 3), (4, 1), (2, 7)\) takes place in the first step.
message passing is harder to use than NavP.

A hybrid use of MPI and OpenMP is another way of introducing efficient run-time scheduling. This is based on the use of multiple threads. 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 [23] 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 (e.g., `hop()`) and synchronization commands (e.g., `waitEvent()` and `signalEvent()`).

Similar to Gentleman’s Algorithm, our NavP implementation exploits parallelism using a skewed matrix and loop transformation. Two existing ways of loop skewing are: (1) Automatic loop skewing with affine transformations [24]; and (2) Parallel algorithm development under the SPMD view (e.g., Gentleman’s Algorithm). Our NavP methodology distinguishes itself from the above in that it uses highly mechanical and incremental steps to guide the programmers to achieve 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

The authors wish to thank Koji Noguchi for his great help with MESSENGERS and valuable discussions.

References


