Toward Automatic Data Distribution for Migrating Computations

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

Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
jingling@cse.unsw.edu.au

Ming Kin Lai, Michael B. Dillencourt,
and Lubomir F. Bic
Department of Computer Science
University of California, Irvine
Irvine, CA 92697, USA
{mingl,dillenco,bic}@ics.uci.edu

Abstract

Program parallelization requires mapping computation and data to processing elements. Navigational Programming (NavP), based on the principle of migrating computations, offers a different approach than the conventional solutions that use a SPMD model. This paper focuses on data distribution for NavP. We introduce the Navigational Trace Graph (NTG), a mathematical structure that captures the alignment and distribution preferences of a sequential program. Graph partitioning is applied to NTGs to obtain data distribution solutions. The major advantage is that our methodology can focus exclusively on reducing communication overhead first and later determine the actual computation partition and parallelization, because NavP computations migrate freely across partitions. This is in stark contrast to SPMD, where the data partitioning imposes hard constraints on the threads because they are stationary. We present experimental results to demonstrate the effectiveness of our approach.

Keywords: data distribution, navigational trace graph (NTG), graph partitioning, navigational programming

1. Introduction

Distributed parallel programming is traditionally done in the Single Program Multiple Data (SPMD) style, in which a process or thread is stationary to a local partition of the data and thus is the owner of both the data and the computation associated with the data. Remote data that is required by a process is communicated by a `recv()` and a `send()` posted by the requester and the owner processes, respectively. Navigational Programming (NavP) [12], which is the programming of self-migrating computations, is another means to distributed programming. The characteristics of NavP are: **(1)** Migration is made possible by explicitly inserting `hop(dest)` statements into the code. Remote communication is achieved by threads carrying data from one location to another; **(2)** Migrating computations are lightweight, user-level threads. They are non-preemptive and the synchronizations among them are through local events using the `signalEvent(evt)` and `waitEvent(evt)` statements; and **(3)** There are three kinds of variables: small data that follows a migrating computation is loaded to a *thread-carried variable*, while large data that is stationary to a PE is stored in a *node variable*. Multiple disjoint node variables can be used to construct a logical array spanning several PEs, called a *Distributed Shared Variable (DSV)*. A DSV provides a partitioned global address space.

The NavP methodology provides four steps of code transformations. **Step 1. Data Distribution.** The input to this step is a sequential program to be parallelized. The objective is to find a data distribution that minimizes the cost of communication for the given sequential program, with a balanced (data) load as the constraint. What is being distributed is the large-sized data usually stored in a DSV. **Step 2. Sequential → DSC**. Using the data distribution obtained from Step 1, the sequential code is augmented with `hop()` statements to obtain a *distributed sequential computing (DSC)* program, following the *principle of pivot-computes*. That is, the computation should take place on the PE, called a *pivot node*, that owns the largest portion of the distributed data. **Step 3. DSC → DPC.** The DSC thread from Step 2 can be cut into several shorter DSC threads to build *mobile pipelines* for *distributed parallel computing (DPC)*. The objective is to spread out computations as early as possible, while respecting dependency requirements. `signalEvent()` and `waitEvent()` are inserted to synchronize the DSCs constituting the mobile pipeline. **Step 4. Feedback loop.** This step estimates the tradeoffs between communication/parallelism and adjusts other steps for a minimum overall wall clock time.

In our earlier papers, we introduced our NavP methodology [12] and described how to exploit parallelism using mobile pipelines [11]. Data layouts used in our earlier ex-

periments are standard block or block cyclic distributions, which are assumed to be given by the programmer. Achieving good data distribution is the topic of this paper. We present a data decomposition approach and intend to use it as part of a data layout assistant tool for regular applications. The application programs that we are trying to help out thus are assumed to exhibit repeatable data accessing patterns – patterns that are seen in small-sized input data are going to show in very large problems. This assumption holds also for existing automatic data decomposition techniques for regular applications [1, 2, 3, 5, 8, 13, 10], since they need static or dynamic performance analysis to find out problem size parameters such as loop bounds and array sizes. Irregular applications that are attacked using run-time solutions (e.g., the Adaptive Mesh Refinement technique) are outside the scope of this paper.

The rest of this paper is organized as follows. Section 2 reviews the related work. In Section 3, we present our data decomposition technique in the context of turning sequential into DSC programs. Section 4 discusses how to find data distributions for DPC programs. Section 5 presents experimental results. Section 6 concludes the paper.

2. Related Work

In the SPMD model, data decomposition is performed first while computation decomposition is inferred from the data decomposition using the owner-computes rule. In NavP, computation decomposition is done in the DSC step. So we will review only some automatic data decomposition techniques below.

Given a code region, two different approaches are distinguished: *static decompositions* [3, 8] (under which the data distribution for an array is fixed in the entire region) and *dynamic decompositions* [1, 2, 5, 13, 10] (under which different data distributions for an array may be used in different segments of the region). In the latter case, the region under consideration is divided into code segments, called *phases*, such that data remapping is only allowed between phases [2, 5]. Given a phase, some techniques [5, 8, 13] decompose the mapping problem within the phase into two sequential steps: alignment and distribution. The alignment step identifies the dimensions of all arrays that should be mapped to the same dimension of a PE network. The distribution step decides which aligned dimensions should be distributed in a BLOCK, CYCLIC or BLOCK-CYCLIC manner. The underlying mathematical representation is mainly CAGs (*component affinity graph*) [8] or their variants – CPGs *communication-parallelism graph* [2]. Either only CYCLIC distributions are considered [2] (for triangular loop nests only) or BLOCK-CYCLIC distributions are found by an exhaustive search. In this paper, both distribution and alignment are solved using navigational trace graphs. Both standard data layouts and other regular ones are supported.

The problem of finding optimal data decomposition is known to be NP-complete. Previously, different techniques use different heuristics to estimate the benefits of parallelism and the cost of communication in their formulations. They find approximately optimal solutions analytically or by integer programming. Our approach is "numerical" in the sense that we find optimal solutions by using a graph partitioning tool (e.g., Metis [4]). Our approach is approximate since such a partitioning tool is.

3. Finding Data Layouts for DSC

When turning a sequential program into a DSC program, we must first find a data distribution for the DSC program. In this section, we present an intra-procedural technique for achieving this task. Presently, our technique works on individual phases, which are well-defined basic algorithms that are usually in the form of functions. In what follows, by a program we mean a phase (e.g., a code region) for which data layouts are to be found. How to find data layouts for multiple phases is our future work.

There are three key steps: **(1)** Build a so-called *navigational trace graph* (NTG) by program instrumentation; **(2)** Find a data layout by partitioning the NTG using a graph partitioning tool; and **(3)** Express the data layout found using the data distribution mechanisms that NavP supports.

Let there be `K` PEs. To find a data distribution for a DSC program, we will find a `K`-way partition of the corresponding NTG. The objective is to find such a data distribution by minimizing the cost of communication, with a balanced (data) load as the constraint. In Section 4, we will discuss how to find a cyclic data distribution for a DPC program with a balanced computation load. By using cyclic distributions, we can also make the tradeoffs between communication cost and exploitable parallelism.

3.1. Building an NTG

Definition 1 An NTG for a program is a weighted undirected graph (without self-loops), where the vertices are the entries of DSVs (one for every entry of every DSV) and the edges (with positive weights) represent the affinity relations among the vertices as the locus of computation finds it way through them.

The NTG for a program is generated by running the program against a small problem. It captures the trace of a DSC program as the migrating computation follows the data that it accesses. The larger the weight of an edge is, the stronger the two incident vertices want to stay together on the same PE. In addition, the data entries of the arrays that will be distributed, regardless of which array they belong to, become the vertices of the same graph. In this way the problems of

1 **Algorithm** BUILD_NTG
2 **INPUT:** a program
3 **OUTPUT:** a (weighted undirected graph) NTG $G = (V, E)$
4 Let *ListOfStmt* be a list of all statements executed in that order for a given program with respect to a small problem size
5 ***// Step 1: Edge Creation (with G being a multi-graph)***
6 Let V be the set of all DSV entries accessed in the program
7 Let $E = \emptyset$
8 *// Add L edges*
9 **for** every entry v in every DSV array
10 Add to E an L edge between v and each of its neighboring entries
11 *// Add PC edges*
12 **for** every statement s in *ListOfStmt* whose LHS is a DSV entry
13 Repeatedly replace every non-DSV data entry v in the RHS of s, where v is defined by the statement of the form v = "...", with the "..."
14 Let RHS_s be the set of all DSV entries in the RHS of s
15 Add to E a PC edge between the LHS and every entry in RHS_s
16 *// Add C edges*
17 **for** every two statements s and t in *ListOfStmt* such that no statement in between in *ListOfStmt* has access to DSV data entries
18 Let V_s be the set of all DSV entries accessed in s
Let V_t be the set of all DSV entries accessed in t
19 Add to E a C edge between every entry in V_s and every entry in V_t
20 Remove all self-loops in G
21 ***// Step 2: Edge Weight Selection***
22 #define L_SCALING = a nonnegative value (typically within $[0, 1]$)
23 Let *num_Cedges* be the total of C edges
24 Set $c = 1$
25 Set $p = num\_Cedges + 1$
26 Set $\ell =$ L_SCALING $* p$
27 Merge the multiple edges linking the same two vertices into one single edge by accumulating their edge weights

Figure 1. An algorithm for building an NTG.

```
(1) for i = 1 to M − 1
(2)   for j = 0 to N − 1
(3)     a[i][j] ← a[i − 1][j] + 1
(4)   end for
(5) end for
```

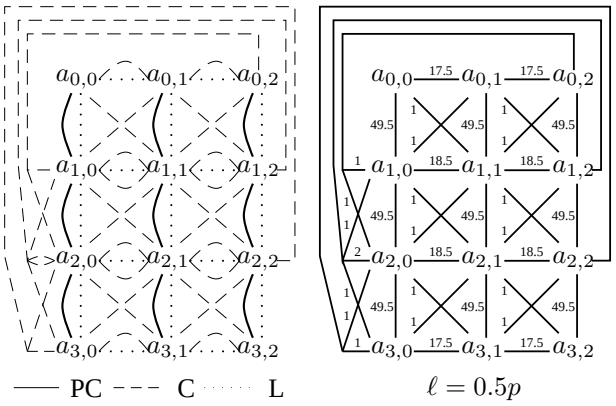
Figure 2. A program for illustrating construction of NTGs.

alignment and distribution are addressed in a unified manner.

An NTG is constructed in two steps: **(1)** edge creation, and **(2)** edge weight selection.

3.1.1. Edge Creation

The construction of an NTG is based on three kinds of edges. First, locality (or L) edges are introduced between the neighboring entries of a DSV and they are assigned with the weight ℓ. These edges represent the locality of data access exhibited in many algorithms and they aim at obtaining regular data layouts for each array. Second, a producer-consumer (or PC) edge with the weight p is introduced between an LHS DSV array entry and a RHS DSV array entry. These edges indicate the occurrence of communication



(a) NTG (after Step 1) (b) NTG (after Step 2)

Figure 3. NTGs for Fig. 2 (M=4, N=3).

if the two linked entries do not reside on the same PE. Finally, every array entry of a DSV array in one statement is connected with every DSV entry in its successive (in time) statement with a continuity (or C) edge with the weight c. These edges represent the change of locus of computation (i.e., trace of hops) if the two linked entries do not reside on the same PE and their purpose is to help improve the granularity of computation.

Our algorithm given in Fig. 1 creates these edges in lines 5 – 20. In line 4, *ListOfStmt* is the list of all execution instances of the assignment statements obtained by running the sequential program for a relatively small problem size. In lines 8 – 10, we introduce locality edges. In lines 11 – 15, we introduce PC edges, which represent data dependences among DSV entries. Note that a PC edge exists between two DSV entries if one depends on the other directly or indirectly via a chain of non-DSV data entries. Hence, line 13 is needed to detect these PC edges. Consider the following sequence of dynamically executed statements in a program:

```
...
t1 = b[3] + 1
t2  = a[2] + t1
a[5] = t2 + a[4]
...
```

where `a[]` and `b[]` are DSVs, and `t1` and `t2` are non-DSV entries. After line 13, `a[5] = t2 + a[4]` becomes:

```
a[5] = a[2] + b[3] + 1 + a[4].
```

Thus, in lines 14 – 15, a PC edge is added between the DSV entry `a[5]` and each of the three DSV entries, `a[2], b[3]` and `a[4]`. After line 13, all the statements that define the non-DSV entries are ignored. It is possible to

have multiple PC edges between the same two entries since the RHS entry may be fetched from its hosting PE multiple times. This can happen since the RHS entry is written multiple times and must be fetched each time before it is used.

In lines 16 – 19, we add C edges to the NTG. Again, there may be multiple C edges between the same two entries representing multiple hops required if they do not reside on the same PE. In line 20, we remove all edges linking a vertex to itself.

At the end of this step, the NTG obtained is a multi-graph with possibly one L edge, multiple PC edges and multiple C edges between any two vertices. As an example, applying this part of our algorithm to the program in Fig. 2 yields the NTG shown in Fig. 3(a).

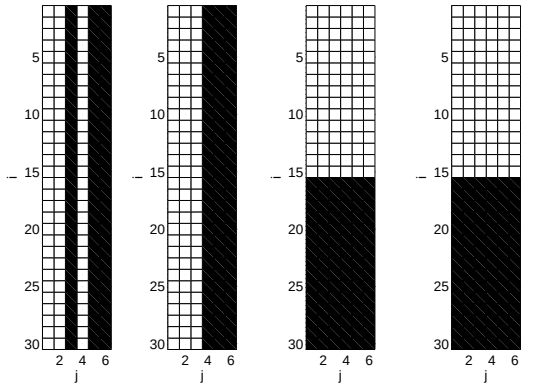
3.1.2. Edge Weight Selection

Given the roles that L, PC and C edges play, the relative magnitudes of their weights will be chosen such that if the weight of PC edges is $p = 1$, then the C edges will be assigned the weight of infinitesimal $c = \epsilon > 0$, and the L edges a nonnegative value $\ell \geqslant 0$.

The motivations for this weight assignment are as follows. As we shall see in Section 3.2, we obtain a data distribution for a program by partitioning its NTG such that the weights of the total cut edges are minimized. Since C edges have infinitesimal weights compared to PC edges, they cannot (and should not) collectively affect the producer-consumer affinity relationship of the data entries. Thus, C edge cuts are encouraged and so is parallelism because the C edges are not true dependences but artificial sequencing relations. As a result, the entries linked with PC edges tend to stay on the same PE. As for L edges, choosing different weights makes it possible to tradeoff between data locality and parallelism. If ℓ is close to p or larger, we will obtain a more regular partition, which usually results in better data locality. If ℓ is close to 0, the resulting data partition will reflect more accurately the actual cost of communication of the program. Such a partition tends to be less regular but may allow more parallelism to be exploited.

There can be more than one way of assigning edge weights. Our solution is given in lines 22 – 27 in Fig. 1, where `L_SCALING` is a program-dependent parameter, which can be tuned in the feedback loop of NavP based on performance profiling and evaluation. Figure 3(b) depicts the final NTG obtained for the program given in Fig. 2, under the assumption that $\ell = 0.5p$.

To understand the roles that L, PC and C edges play and our solution for their weight assignment (lines 22 – 27), let us consider the four partitions given in Fig. 4 for the example given in Fig. 2. In these (and all other) partition diagrams, all data entries sharing the same grey scale are assigned to the same partition. The NTGs for the example (with and without final edge weights) can be found in Fig. 3. Let us consider Fig. 4(a). When only PC edges are used, all array columns are not linked by any edges. A 2-way partition thus can contain any half of the columns. Such a partition exhibits full parallelism at the expense of some thread hops (i.e., fine grained computation). If we now include C edges and choose the weights of PC and C edges according to line 25, the C edges will play the role of tie-breakers and bring us a coarser grained data distribution shown in Fig. 4(b). Through edge weight selection, we prefer to cut all C edges rather than even a single PC edge when the NTG is partitioned. As a result, the data distribution obtained in Fig. 4(b) admits full parallelism with also a minimal number of hops. If we did not set edge weights using line 25, or in other words, if we set the C edges to be larger than infinitesimal compared to the PC edges, we might get the partition shown in Fig. 4(c) if the matrix is shaped long and thin. By introducing L edges, we will obtain more regular layouts, or precisely, block distributions if the weights of the L edges are chosen to be relatively large, as shown in Fig. 4(d). Compared to the first solution, the third and fourth solutions reduces the number of hops. Compared to the second solution, they lose some degree of parallelism; pipeline parallelism is exploitable but full parallelism is not since the computations on the two partitions cannot start simultaneously due to dependences within columns.



(a) PC edges (b) PC + C (c) PC + large C (d) PC + C + L ($\ell = 0.5p$)

Figure 4. Two-way data distributions for the program given in Fig. 2 (M=50, N=4).

3.2. Partitioning the NTG

An NTG will be fed to a graph partitioning tool, Metis [4], to find a `K`-way partition with the overall objective of minimizing communication cost incurred by the partition under the constraint of a balanced (data) load. Metis uses

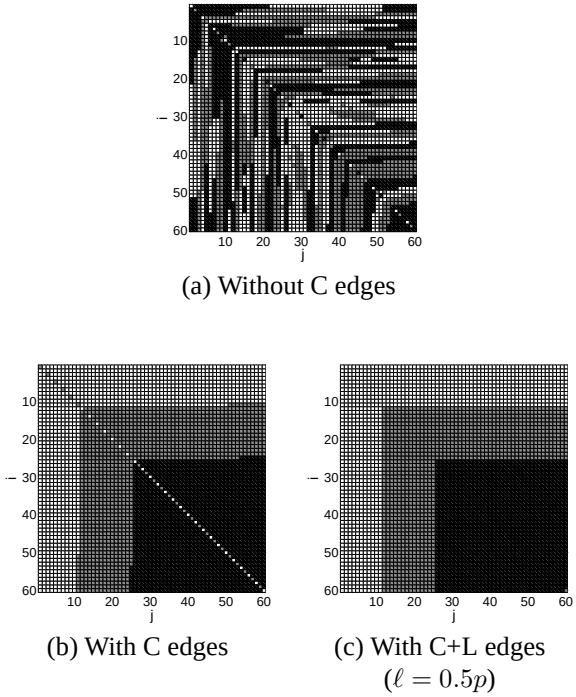
a parameter called UBfactor to specify the imbalance allowed between the partitions during recursive bisection [4]. If there are n vertices in the NTG, the number of vertices in each partition during each bisection step is between $(50 - \mathtt{b})\mathtt{n}/100$ and $(50 + \mathtt{b})\mathtt{n}/100$. In all the applications considered in this paper, UBfactor=1. In finding a K-way partition, Metis will minimize the sum of the weights of the cut edges spanning all K partitions. According to the Metis' web site, graphs with over 1M vertices can be partitioned in 256 parts (i.e., $\mathtt{K} = 256$) in a few seconds on current generation workstations and PCs. The number of vertices of an NTG is the total number of entries of all the DSVs involved (DEFINITION 1) and the examples presented in this paper have at most $3,600$ vertices.

By finding a minimum cut to partition an NTG, we are able to minimize the total data movement among the PEs. We also maintain a data load balance in terms of data amount on the PEs because a balanced partition is used as an optimization constraint. However, balanced data load does not imply balanced computation load. This will not affect DSC since it runs in one thread. As a matter of fact, a balanced data load leads to a scalable DSC program. For DPC, we use block cyclic data distribution to achieve computation load balancing and better parallelism (more in Section 4).

Due to the presence of C edges in the NTG, which represent change of locus of computation, we minimize the number of hops. In other words, the C edges keep a coarse level granularity, which is important for performance. We introduce C edges to capture the artificial sequential dependency introduced in sequential algorithms. Our NTGs are generated such that cuts are more likely to be placed on the C edges to exploit parallelism, other things being equal. If cuts are on PC edges, they are more likely placed in the "direction" that is "parallel" to the PC edge chains because this results in less PC edge cuts. For this reason, we claim that our approach does not hinder parallelism.

3.3. Expressing the Partitions

We build a visualization tool to present the recommended data layouts. Our preliminary results are shown in Figs. 4, 5, and 7. These results are from regular algorithms that access structured data structures (e.g., dense square matrices), which means that unstructured data distribution is desirable even for seemingly simple applications. Therefore, NavP needs to support not only the classic distribution mechanisms such as BLOCK and BLOCK-CYCLIC as in HPF, GEN_BLOCK and INDIRECT mappings as in HPF-2 but also others that can describe the unstructured data layouts. How to describe unstructured data layouts in NavP will be part of our future work.



(a) Without C edges

(b) With C edges

(c) With C+L edges ($\ell = 0.5p$)

Figure 5. Transpose of a matrix (3-way).

3.4. Applications

This section discusses how our data distribution tool can be used to find data distributions for two applications, matrix transpose, and ADI (Alternating Direction Implicit) Integration. For matrix transpose, our approach is able to find L-shaped communication-free data distributions that cannot be found by previously existing approaches [2, 3, 5, 8, 9]. We are able to find data distributions for ADI [6, 5, 7] but do so by solving both alignment and distribution at the same time using NTGs. In addition, by using a twisted data layout we are able to achieve full parallelism at a low cost of communication in our NavP implementation. We compare the performance of this implementation with the one that uses data redistribution between two different phases.

3.4.1. Matrix transpose

Matrix transpose swaps the anti-diagonal entries of a matrix. The pseudocode is omitted. The data distribution found as shown in Fig. 5 consists of L-shaped partitions; it is optimal in the sense that it is communication-free.

If we did not have C edges in the NTG, each anti-diagonal pair will still be distributed in the same partitions, but pairs will be distributed in a dispersed fashion, as shown in Fig. 5(a), unlike what is shown in Figs. 5(b) and (c), where contiguous partitions are seen.

With L edges (weight $\ell = 0.5p$), the resulting partition is regular (except that the bottom-right entry is included in the

top-left partition), as shown in Fig. 5(c). In the absence of L edges ($\ell = 0$), the partition becomes less regular, especially along the main diagonal of the matrix, as shown in Fig. 5(b).

Our solution cannot be found by prior approaches since they are limited to BLOCK and BLOCK-CYCLIC [2, 3, 5, 8, 9]. This optimal solution enables the programmers to explore full parallelism with zero communication at a coarse granularity level.

3.4.2. ADI integration

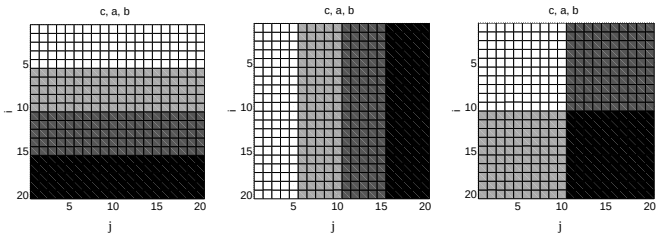
```
// time iteration
(1) for iter = 1 to niter
     // Phase I : row sweep
(2)  for j = 2 to N
(3)   for i = 1 to N
(4)    c[i][j] = c[i][j] − c[i][j − 1] ∗ a[i][j]/b[i][j − 1]
(5)    b[i][j] = b[i][j] − a[i][j] ∗ a[i][j]/b[i][j − 1]
(6)   end for
(7)  end for

(8)  for i = 1 to N
(9)   c[i][N] = c[i][N]/b[i][N]
(10) end for

(11) for j = N − 1 to 1 by − 1
(12)  for i = 1 to N
(13)   c[i][j] = (c[i][j] − a[i][j + 1] ∗ c[i][j + 1])/b[i][j]
(14)  end for
(15) end for
     // Phase II : column sweep (omitted)
(16)end for
```

Figure 6. Pseudocode of ADI

ADI integration is an example used by several papers on data distribution [6, 5, 7]. The pseudocode for ADI integration is listed in Fig. 6 [6, 5]. There are three 2D arrays, namely `c`, `a`, and `b`, involved in the computation. This code is usually subdivided into two phases, namely a row sweep phase (lines (2)-(15)) and a column sweep phase (not shown). These two phases are surrounded by an outer loop of time iteration (line (1)). One possible solution, existed in previous work, is to find two different data mappings suited for their respective phases. We use our tool to find these two separate solutions and plot them in Figs. 7(a) and (b). Figure 7(c) depicts the data distributions for two phases combined together. The two sweeps are two `DOALL` loops (i.e., full parallelism with no communication) if they use their own data distribution, but in between the sweeps a dynamic data redistribution is needed. If both phases are combined, pipeline parallelism can still be exploited. The advantage of this data distribution for the entire program is that no dynamic data remapping is needed between the two phases. The cost of a dynamic data remapping can vary dramatically on different platforms.



(a) Phase I (b) Phase II (c) Phases I + II

Figure 7. ADI integration on a 20x20 matrix (4-way).

4. Finding Data Layouts for DPC

In NavP, we parallelize a program by first transforming it to DSC and then turning DSC into DPC (Section 1). In Section 3, we presented our methodology for finding data layouts for DSC. The data partitions found do not hinder parallelism. Furthermore, they will also serve as the starting point to exploit more parallelism.

We propose to use a block cyclic data distribution evolved from the solution suggested by our tool and apply pipelining code transformation to further improve performance. By "block cyclic distribution" we mean an `n`-round cyclic distribution of an $(\mathtt{n}K)$-way partition to a `K`-processor machine, where the partitions can be rectangular or other shaped (e.g., L-shaped) blocks. For example, to obtain a block cyclic distribution for the problem depicted in Fig. 5(c), we get a 6-way partition first and then cyclically assign the L-shaped partitions onto 3 PEs. So our block cyclic distribution is a more general form of BLOCK-CYCLIC distribution.

Our data distribution tool provides a partition with the minimum communication cost as our starting point. As we increase the number of cyclic data blocks, we obtain more and more parallelism (hence less and less time) at the cost of increased communication. Note that we follow the data distribution pattern suggested by our tool when we increase the number of data blocks (when the number of data blocks exceeds the number of PEs we call the data blocks "virtual blocks") – this will make sure that the communication cost remains the minimum for each and every new partition we come up with. At some point, the total execution time will reach the minimum and then start growing if we further increase the communication cost. Our proposed approach thus provides a systematic way of achieving the best performance for a particular application.

As mentioned earlier, our data distribution for DSC guarantees data load balancing but not necessarily computation load balancing. Block cyclic data distribution helps to achieve computation load balancing because computations

will migrate to all the PEs more evenly.

5. Experimental Results

In this section, we present our experimental results. The data was obtained using a network of SUNW Ultra-60's with 450 MHz UltraSPARC-II CPU, 256MB of main memory, 1GB of virtual memory, 100Mbps of Ethernet with a collision-free switch, and using the NFS file-sharing system. The C compiler used was `gcc3.2.2` (`-O3` optimization option turned on), the MPI used (for matrix redistribution in ADI integration) was LAM MPI 7.0.6, and the NavP compiler and runtime system used was MESSENGERS 1.2.05.

5.1. Matrix Transpose

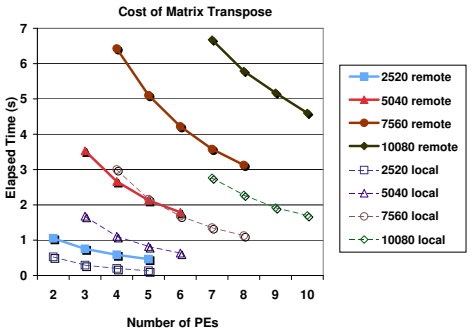


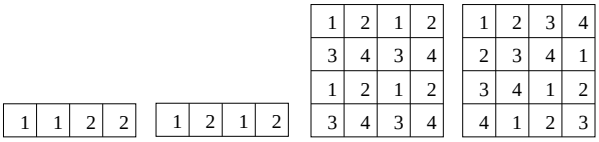
Figure 8. The cost of matrix transpose.

We have compared the costs of transposing a matrix in parallel under two circumstances: **(1)** Each PE gets a vertical slice of the matrix, as depicted in Fig. 7(b). This data distribution requires remote data communication; and **(2)** Each PE gets an L-shaped slice of the matrix, as depicted in Fig. 5(c). Only local data movement is needed for this data distribution. Our experiment, as presented in Fig. 8, shows that matrix transposing involving remote communication is more than twice as expensive as done locally.

5.2. ADI integration

We first turn the ADI integration code into a block implementation. That is, we introduce "distribution blocks" — submatrix blocks that are basic units for data distribution — in the matrices and convert the loops over the matrix entries into the loops over the entries within the distribution blocks surrounded by the loops over the distribution blocks. Next, we go through the NavP steps to parallelize ADI integration. In particular, we first make the sweeps two DSCs and turn the outer loop another DSC responsible for injecting the sweeper DSCs. We then cut the sweeper DSCs into shorter ones and pipeline them.

Figure 9 depicts data distribution patterns in 1D and 2D cases. Each box in this figure represents a submatrix block



(a) 1D block (b) 1D blk cyclic (c) 2D blk cyclic (d) 2D twisted

Figure 9. Block cyclic distribution patterns.

and the number in a box indicates the ID of the PE that this block is assigned to. It is assumed that in the 1D case we have two PEs and in the 2D case we have four PEs. As in Fig. 6, the three square matrices are each of order `N`. In Fig. 9(a), a matrix is cut into four vertical slices each of $N \times N/4$ and the blocks are assigned to the two PEs in a block fashion (that is, the first two blocks go to PE1 and the last two blocks go to PE2). Figure 9(b) depicts a 1D block cyclic pattern where the blocks are assigned to the PEs in order until the PEs are exhaustively used, at which time the block assignment cycles back. In HPF, a 2D block cyclic pattern is the cross product of two 1D block cyclic patterns, shown in Fig. 9(c). For 2D, each submatrix block is $N/4 \times N/4$. We also use a twisted pattern, depicted in Fig. 9(d), in which the first row of blocks are assigned to all the PEs in order. (This is unlike the block cyclic pattern where the PEs are arranged as a 2×2 grid and the first row of blocks are assigned cyclically along the first row of PEs.) The next rows are assigned to all the PEs in a similar way, except that they are shifted east-ward one position from their previous rows. This distribution pattern is effectively a "skewed pattern." When the sweeper threads sweep through all the rows or columns, all PEs are busy simultaneously. That is, we achieve full parallelism, at the cost of $O(N)$ as one layer of the matrix entries is carried over from block to block. In contrast, in the example shown in Fig. 9(c), only two PEs are busy at any time as the sweeper DSCs sweep through. The situation for the block cyclic pattern is worse if the PEs are arranged as a 1D grid when, e.g., the number of PEs is a prime number. As for the cost of communication, the `DOALL` approach mentioned in Section 3.4.2 requires $O(N^2)$ in data redistribution.

As presented in Fig. 10 (the numbers in the legend are matrix orders), the NavP program using the twisted data distribution pattern performs the best. Using the HPF block cyclic pattern, the NavP program incurs the same communication cost of $O(N)$ but has less degree of parallelism. Therefore, the performance is inferior, especially when the number of PEs is a prime number[1]. Finally, if we employ data redistribution in the `DOALL` approach, even though the two sweeps are fully parallel, the cost of data redistribution,

[1] A true 2D PE grid for the HPF block cyclic pattern is only possible when the number of PEs is not a prime number.

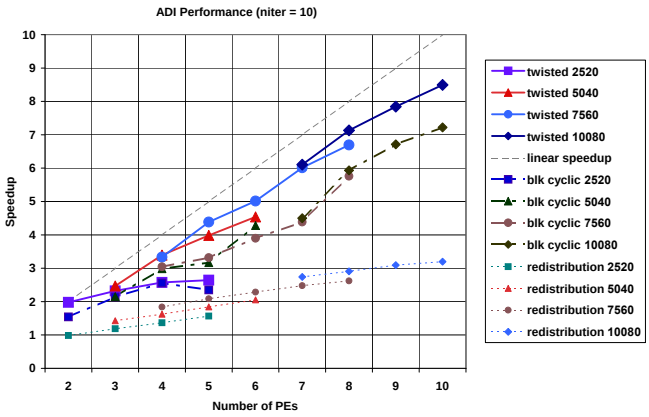


Figure 10. The performance of ADI integration.

$O(N^2)$, is so large that the overall performance is poor. We used the MPI library call `MPI_Alltoall()` to obtain the cost for matrix redistribution.

With ADI integration, we demonstrate the following: **(1)** We can solve both alignment and distribution in a unified manner; **(2)** The data distribution for DSC is obtained from minimizing the cost of communication with load balancing as a constraint. Parallelism is exploited later using mobile pipelines. Block cyclic data distribution helps to improve parallelism by making the PEs busy earlier, and twisted data distribution allows to achieve full parallelism; and **(3)** On loosely coupled systems such as clusters, data redistribution between the two phases, aimed at achieving full `DOALL` parallelism for both phases, is prohibitively expensive. As a result, choosing a data distribution that minimizes communication and further minimizing communication using DSCs that follow the principle of pivot-computes are of decisive importance to overall performance. Using pipelining may result in loss of some degree of parallelism, but this impact to performance is secondary. Furthermore, with some adjustment in data distribution using the twisted pattern, it is still possible to achieve full parallelism using mobile pipelines at a cost of asymptotically less communication than what is required in data redistribution.

6. Conclusions

In this paper, we present a new mathematical representation, called *navigational trace graph* (NTG), for representing the alignment and distribution preferences in a unified manner. The NTGs aim at a minimum communication cost, but they do not hinder parallelism because of the weights of the edges chosen. More parallelism and load balancing are achieved by using block cyclic data distribution and mobile pipelining. We apply graph partitioning to a NTG to obtain a data distribution. Our partitioning tool can find unstructured data layouts such as the L-shaped blocks. It is able to do this because it aligns entries rather than dimensions of the arrays and thus captures more accurately the cost of data communication. Our approach is independent of array storage schemes used. We can hence help the programs that use sparse storage schemes. We use a twisted distribution pattern, effectively a skewed block data distribution, that allows mobile pipelines to exploit full parallelism without redistributing large amount of data. We present experimental results to show the effectiveness of our technique.

References

[1] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *PLDI '93: Proc. of the ACM SIGPLAN 1993 conf. on Prog. language design and imple.*, pp. 112–125, NY, 1993. ACM Press.

[2] Jordi Garcia, Eduard Ayguadé, and Jesús Labarta. A framework for integrating data alignment, distribution, and redistribution in distributed memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):416–431, 2001.

[3] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Trans. Parallel Distrib. Syst.*, 3(2):179–193, 1992.

[4] George Karypis and Vipin Kumar. *h*METIS *A hypergraph partitioning package (version 1.5.3)*. Dept. of Computer Science & Engineering, Univ. of Minnesota, 1998.

[5] Ken Kennedy and Ulrich Kremer. Automatic data layout for distributed-memory machines. *ACM Trans. on Prog. Languages and Systems*, 20(4):869–916, July 1998.

[6] J. Knoop and E. Mehofer. Distribution assignment placement: Effective optimization of redistribution costs. *IEEE Trans. on Para. and Dist. Sys.*, 13(6):628 – 647, June 2002.

[7] Peizong Lee and Zvi Meir Kedem. Automatic data and computation decomposition on distributed memory parallel computers. *ACM Trans. on Prog. Languages and Sys.*, 24(1):1–50, Jan 2002.

[8] Jingke Li and Marina Chen. The data alignment phase in compiling programs for distributed-memory machines. *J. Parallel Distrib. Comput.*, 13(2):213–221, 1991.

[9] Angeles Navarro, Emilio Zapata, and David Padua. Compiler techniques for the distribution of data and computation. *IEEE Trans. Parallel Distrib. Syst.*, 14(6):545–562, 2003.

[10] Daniel J. Palermo and Prithviraj Banerjee. Automatic selection of dynamic data partitioning schemes for distributed-memory multicomputers. In *LCPC '95: Proc. of the 8th Int'l Workshop on Languages and Compilers for Parallel Computing*, pp. 392–406, London, 1996.

[11] Lei Pan, Ming Kin Lai, Michael B. Dillencourt, and Lubomir F. Bic. Mobile pipelines: Parallelizing left-looking algorithms using navigational programming. In *Proc., 12th Int'l Conf. on High Perf. Comp. - HiPC 2005*, vol. 3769 of *LNCS*, pp. 201–212, Berlin, Dec 2005.

[12] Lei Pan, Ming Kin Lai, Koji Noguchi, Javid J. Huseynov, Lubomir Bic, and Michael B. Dillencourt. Distributed parallel computing using navigational programming. *Int'l J. of Parallel Programming*, 32(1):1–37, Feb 2004.

[13] Thomas J. Sheffler, Robert Schreiber, William Pugh, John R. Gilbert, and Siddhartha Chatterjee. Efficient distribution analysis via graph contraction. In *LCPC '95: Proc. of the 8th Int'l Workshop on Languages and Compilers for Para. Comp.*, pp. 377–391, London, 1996.