Automatic Optimization of Communication in Compiling Out-of-core Stencil Codes

Rajesh Bordawekar **Alok Choudhary**
ECE Dept., 121, Link Hall, Syracuse University, Syracuse, NY 13244
{rajesh, choudhar}@cat.syr.edu

J. Ramanujam
ECE. Dept., Louisiana State University, Baton Rouge, LA 70803
jxr@gate.ee.lsu.edu

Abstract

In this paper, we describe a technique for optimizing communication for out-of-core distributed memory stencil problems. In these problems, communication may require both inter-processor communication and file I/O. We show that in certain cases, extra file I/O incurred in communication can be completely eliminated by reordering in-core computations. The in-core computation pattern is decided by: (1) how the out-of-core data distributed into in-core slabs (tiling) and (2) how the slabs are accessed. We show that a compiler using the stencil and processor information can choose the tiling parameters and schedule the tile accesses so that the extra file I/O is eliminated and overall performance is improved.

1 Introduction

The use of parallel computers to solve large scale scientific problems has increased considerably in recent times. Majority of these problems exhibit large memory requirements (of order of GBytes). Since main memories are not large enough to hold such large amount of data, data needs to be stored in files and accessed repeatedly during execution of the program. Consequently, performance of these *Out-of-core* problems depends on how fast the input-output (I/O) is performed.

Many scientific problems exhibit *regular* computation patterns. This class of computation arises in cellular automata and numerical solutions of partial differential equations, e.g., explicit CFD problems. A regular problem can be characterized by the corresponding *stencil*. Figure 1 illustrates some of the commonly used stencils: (1) 5-point stencil, (2) 9-point star stencil and (3) 9-point cross stencil. More complex stencils are found in problems in cellular automata and seismic modeling [BHMJ91, BCR95].

In this paper, we analyze two-dimensional out-of-core stencil problems. Specifically, we focus on compilation issues for such problems when developed using data-parallel languages like High Performance Fortran (HPF)[Hig93]. We

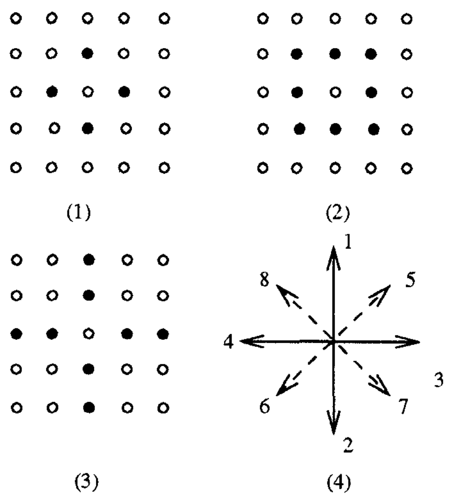

Figure 1: Commonly Observed Stencils.

show that communication for out-of-core problems involves file I/O as well as inter-processor communication. We describe compiler techniques for minimizing file I/O incurred during communication.

The paper is organized as follows. In Section 2, we describe a concise way of representing two-dimensional stencils. Section 3 describes compilation of out-of-core stencil problems. Section 4 analyzes the tiling problem and presents a simple tiling heuristic based on the I/O cost and the communication pattern of the stencil. Section 5 describes techniques for computing schedules which eliminate file I/O from communication. Section 6 presents experimental performance results. Section 7 outlines future extensions of this work. Related work is described in Section 8. Finally we conclude in Section 9.

2 Algebraic Representation of Stencils

We define a *stencil* $\mathfrak{S}$ as a set of elements defined over a two-dimensional lattice $\mathbb{L}^2$, $\mathfrak{S}=\{(i,j) \mid (i,j) \in \mathbb{L}^2\}$. Each stencil is associated with a *seed*, s and a *neighborhood set*, $\mathfrak{N}$ [Lee90]. Usually the seed of a stencil is an element in the computational domain which is updated during the computation. The neighborhood set, $\mathfrak{N}$ is computed as $\{(i,j) \mid (i,j) \neq \text{s}, (i,j) \in \mathbb{L}^2\}$. The elements of $\mathfrak{N}$ are called neighbors of the seed s.

Each element of a stencil can be indexed using a 2-element vector (i, j). For simplicity, we assume that the seed is always situated at the origin $\vec{0} = (0,0)$. Using the seed as the origin, each neighbor can be appropriately indexed. For example, the 5-point cross stencil, $\mathfrak{S}_5{}^{\times}$, can be described by the set $\mathfrak{S}_5{}^{\times} = \{(0,0),(1,0),(-1,0),(0,-1),(0,1)\}$. The corresponding neighborhood set, $\mathfrak{N}_5{}^{\times}$, is given by $\{(1,0),(-1,0),(0,-1),(0,1)\}$.

The stencil matrix $\mathcal{S}$ provides a concise way of representing a given stencil. For a given stencil $\mathfrak{S}$, the stencil matrix stores the relative indices of the neighbors. The generalized stencil matrix $\mathcal{S}$ is a $2 \times n$ matrix, where n is the number of *directions* in which neighbors of the seed may lie. Figure 1:4 illustrates the order in which the directions are indexed. If there are more than one points spanned in a given direction (e.g., (1,0) and(2,0)), only the largest point in that direction is chosen ((2,0) in this case). For example, the stencil matrix for the first-order stencil $\mathfrak{S}_9{}^{\star}$ (Figure 1:2), $\mathcal{S}_9^{\star}$ is given by

$$\begin{pmatrix} 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 \end{pmatrix}$$

A given stencil can also be represented using the stencil-direction matrix $\mathcal{SD}$. The stencil-direction matrix can be computed from the corresponding stencil matrix using the direction operator $\|\ \ \|_D$. Specifically, $\mathcal{SD}\ (i,j) = \|\mathcal{S}(i,j)\|_D$ and $\|x\|_D$ is 1 if $x > 0$, $\|x\|_D$ is -1 if $x < 0$ and $\|x\|_D$ is 0 if $x = 0$. Hence forth, a given stencil $\mathfrak{S}$ will be analyzed using its stencil matrices $\mathcal{S}$ and $\mathcal{SD}$.

A stencil $\mathfrak{S}$ is called an *irregular* stencil if the seed s has no neighbors in at least one direction, otherwise, it called *regular stencil*[BCR95]. In this paper, we focus only on regular stencils.

3 Compiling Out-of-core Stencil Computations

In this section, we focus on stencil computations involving out-of-core arrays. Figure 2 presents an HPF program performing 9-point stencil computation on array A. The global array A is marked OUT_OF_CORE using a special compiler directive. In this example, array A is distributed in BLOCK fashion in both dimensions. The underlying processor grid consists of nine processors arranged as a 3×3 grid (Figure 4:B). Each processor has an *out-of-core local array* (OCLA) corresponding to the global array A. Computation on the OCLA is carried out in several stages. Each stage reads parts of OCLA into memory, performs computations on the in-core local array (ICLA) and stores the ICLA back in the file (if necessary).

```
    REAL A(1023,1023), B(1023,1023)
    !HPF$ PROCESSORS P(3,3)
    !HPF$ DISTRIBUTE A(BLOCK,BLOCK) ONTO P
    !HPF$ OUT_OF_CORE :: A
    ...........
    FORALL (I=2:1022, J=2:1022)
         A(I,J) = (A(I,J-1)+A(I,J+1)+A(I+1,J)+A(I-1,J)
         +A(I+1,J+1)+A(I+1,J-1)+A(I-1,J-1)+A(I-1,J+1))/8
```

Figure 2: HPF Program Fragment performing a 9-point relaxation over an out-of-core array A.

We focus on the compilation of a FORALL statement which uses the out-of-core array A. For compiling such FORALL statements, we assume the underlying execution model to be the *Local Placement Model* (LPM) [Bor96]. In the Local Placement Model, each processor stores its OCLA into a separate logical file called the *Local Array File* (LAF). LPM can be considered as a straight-forward extension of the distributed memory model where the secondary memory is extended to include the file system.

3.1 In-core Communication Method

For the example illustrated in Figure 2, what is the amount of data communicated among processors? Figure 4:A illustrates the data which a processor may be required to fetch from its neighbors and which is required for computation of the entire OCLA (called *overlap* or *ghost* data).

In in-core problems (i.e., when OCLA can fit into memory), the stencil computation illustrated in the Figure 4 requires each processor to communicate its ghost data with its neighbors. This communication pattern is known as *nearest-neighbor communication*[FJL+88] and can be easily implemented by *exchanging* data between processor pairs. All regular stencils exhibit the exchange communication pattern.

In out-of-core problems, computation on the OCLA is carried out in steps. Therefore, one can fetch the ghost data for the entire OCLA before the OCLA computation starts or in each step, one can fetch the ghost data for the ICLA. The first method is called Generalized Collective communication method while the latter is called In-core communication method [Bor96]. It has been observed that for stencil problems, In-core communication method performs better [Bor96], therefore, we will focus on the In-core communication method. Figure 3 illustrates the sequential code generated for the Local Placement Model using the In-core communication method.

```
DO 10 l=1, k
Call I/O routine to read the ICLA.
Call communication routine for ghost data
DO j = lower_bound, upper_bound
    DO i = lower_bound, upper_bound
    TEMP_A(i,j)=(A(i,j-1)+A(i,j+1)+A(i-1,j)+A(i+1,j)
    +A(i-1,j-1)+A(i+1,j-1)+A(i+1,j+1)+A(i-1,j+1))/8
    ENDDO
ENDDO
Call I/O routine to store the results.
10 CONTINUE
```

Figure 3: Sequential Output Program for Local Placement Model using the In-core communication method.

Figures 4:B shows ghost areas for two processors for column version of the ICLA. The ghost area is colored using two shades. The ghost area denoted by black represents that data which needs to be fetched from the LAF of the same processor (*local* ghost area). The lighter shade denotes the data which needs to be fetched from the LAFs of neighboring processors.

In Figure 4:B, for the given ICLA, processor 3 requires data from its top and bottom neighbor (processors 0 and 6). However, for the similar ICLA, processor 4 requires data from 5 processors; namely, its top, bottom, top-left,top-bottom and right neighbor (processors 0,1,3,6 and 7).

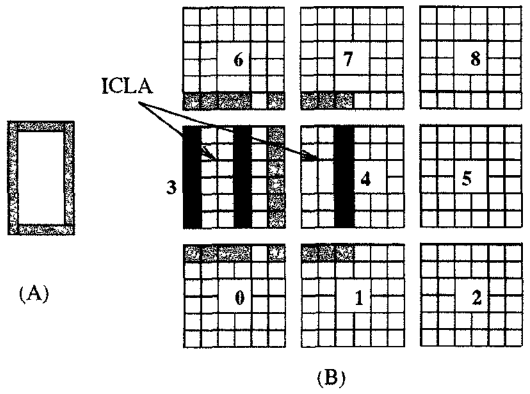

Figure 4: In-core communication Patterns for the stencil $\mathfrak{S}_9^*$

Next question to ask is, when a processor requests data, where does this data come from? From other processors memory or from its LAF or from the LAF of the requesting processor? To answer these questions, let us make the following assumptions: (1) Each processor has same amount of available memory, (2) Each processor has same ICLA shape and size and (3) All processors execute computations in SPMD fashion, i.e, all processors fetch their ICLAs in the same order.

In Figure 4 processors 3 and 4 are both using the first ICLA. For this ICLA, processor 3 requires data from its top and bottom neighbor (processor 0 and 6). Since these processors will also be working on the first ICLA, they have the necessary boundary data in their memory and there is no need of fetching it from external file.

However, for the first ICLA, processor 4 needs to communicate with processors 0,1,3,6 and 7. Processor 4 needs to fetch the last column from processor 3 and the corner elements from processors 0 and 6. Since processor 3 is also working on the first ICLA, it does not have this data in its memory. Hence processor 3 needs to read column from its LAF and send it to processor 4. This results in processor 3 performing *extra* file accesses. Similarly, processors 0 and 6 perform extra file accesses. Moreover, processor 4 has to wait until processors 3, 0 and 6 have read the data and sent it to processor 4. This also results in extra overhead. We can conclude the following:

1. Amount of data communicated for a processor depends on the logical mapping of the processors. For a given ICLA, amount of data communicated depends on the size, shape and relative location of the ICLA.

3. In-core communication method modifies the *exchange* communication pattern of the stencil computation. In other words, for a given ICLA, it may not be necessary for a processor to *exchange* the data with its neighbor (processors 3 and 4, Figure 4:B).

One may then ask a natural question- is it possible to eliminate the extra file accesses? To answer this question, let us analyze the corresponding scheduling problem.

3.2 The Scheduling Problem

The extra file accesses are generated because we assume that each processor reads and performs computations on the ICLAs in the same order (for example, in positive x direction). Therefore, the *exchange* communication pattern is no longer applicable. The extra file accesses can be eliminated *iff* ghost data for an ICLA of any processor always lies in some other processor's memory. If somehow, we are able to alter the computational ordering such that there are no extra file accesses then we can considerably improve the overall performance.

From Figure 2, we can observe that the basic stencil computation is governed by the semantics of the FORALL statement. By definition, a FORALL statement exhibits *copy-in-copy-out* semantics [Hig93]. As a result, there are no dependencies in the FORALL execution and the FORALL iterations can be executed in any order [Hig93].

In out-of-core problems, the iteration execution order is decided by the order in which the ICLAs are fetched from LAFs. Taking advantage of the *copy-in-copy-out* semantics of the FORALL, each processor can access its ICLAs in any arbitrary order. Using the stencil and processor information, we can schedule the ICLA accesses so that data to be communicated is always in-core. Since in In-core communication method, communication depends on the ICLA parameters (shape and size), it is important to compute appropriate ICLA parameters before scheduling the ICLA accesses. Formally, we can frame the scheduling problem as follows: Given an out-of-core regular problem with the associated stencil $\mathfrak{S}$ and available memory $\mathcal{M}$, find an access schedule for ICLAs such that extra file accesses are eliminated.

The scheduling problem can be split into two subproblems:

Tiling Problem: Given the amount of available memory $\mathcal{M}$ and stencil $\mathfrak{S}$, compute the ICLA parameters.

Scheduling Problem: For a given ICLA (tile), generate a ICLA schedule such that extra file accesses are eliminated.

4 Tiling Strategies

This section addresses the first part of the ICLA Scheduling Problem, i.e., computing ICLA parameters for a given processor grid, given available memory $\mathcal{M}$ and the computational stencil $\mathfrak{S}$ (and its stencil matrices $\mathcal{S}$ and $\mathcal{SD}$).

4.1 Logical Processor Mapping

The logical processor mapping is provided by the source HPF program using the PROCESSORS directive. Since we assume 2-dimensional processor map, we consider P processors arranged as two-dimensional grid of m rows and n columns. Viewing the processor grid as a collection of points in first quadrant of the two dimensional integer lattice $\mathbb{L}^2$, we can assign address (i,j) to each processor. Each processor is also assigned an index (Figure 4:B). There is a one-to-one mapping between the processor address and the corresponding index. Using the processor indices and the addresses, each processor can find its neighborhood set. The neighborhood set N of a processor p_k is defined as $N(p_k) = \{(i',j') \mid (i,j)_{p_l} - (i,j)_{p_k}\}$, where p_l and p_k are neighbors. The neighborhood set N can be represented using a matrix called, the Neighbor Matrix $\mathcal{N}$. In a 2-dimensional processor grid, each processor can have upto 8 neighbors. Therefore, the neighbor matrix has 2 rows and 8 columns. Each column represents the *normalized* address

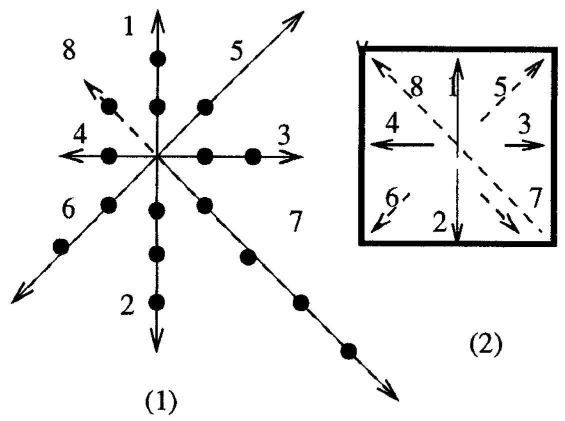

Figure 5: Choosing the Tile Shape. Figure shows the effect of computational stencil on tile shape.

of a neighbor. The neighbors are arranged using the order used to store neighbor information in stencils. Note that unlike the stencil matrices $\mathcal{S}$ and $\mathcal{SD}$, each processor will have a different $\mathcal{N}$ matrix. Any processor using its own address, the index mapping information and the Neighbor matrix $\mathcal{N}$, can find the indices of its neighbors.

4.2 Computing Tiling Parameters

An ICLA (or tile) is a section of the out-of-core array which can fit in the in-core memory. Physically, an ICLA is a sequence of data stored in memory using a predefined storage order (e.g., Fortran column-major). The ICLA parameters determine the logical address space of a given tile. For out-of-core problems, this address space forms the *local strip-mined* address space[1]. A tile can be characterized using the following parameters: Size, Dimension, Shape and Position. For a given application, the tile size is decided by the amount of allocated memory and the tiles dimensions are decided by the out-of-core array. Hence the tile size and the dimensions are fixed. However, the tile shape can be adjusted according to the given application. The tile position depends on how the tile accesses are scheduled.

4.3 Choosing the Tile Shape

For simplicity, we allow only rectangular tiles in 2-dimension. *Shape* of a given tile will be decided by the extent in each dimension. The shape of a tile is dependent on:

1. Associated I/O Cost: I/O cost of a tile is determined by the number of file accesses required to access it. Number of file accesses required for a tile are decided by the amount of contiguous data in the tile. Assuming Fortran storage order, the I/O cost of a rectangular tile can be approximated by the number of *contiguous* columns in that tile. For example, among row and column tiles, row tile will have the largest I/O cost. Ideally the (rectangular) tile shape should chosen such that it requires minimum I/O cost (in other words, the tile should have minimum number of non-contiguous columns). This is called the **I/O Constraint**.

2. Communication Pattern:

   The communication pattern in a stencil application is determined by the type of stencil and the logical processor mapping. Consider an *asymmetric regular* stencil [2] shown in Figure 5. Assume the processors are arranged in a two-dimensional grid. For the given stencil, each processor will have to exchange data with its eight neighbors.

   Since each processor has the same tile shape[3], *each tile should be large enough to satisfy the communication requirements of the tiles from the neighboring processors.*[4] This constraint is called the **Communication Constraint**. According to the stencil pattern, each processor receives 4 elements per seed from its bottom-right neighbor (and sends 4 elements per seed to it's top-left neighbor). Therefore, the tile should contain at least 4 elements on its diagonal. Similar constraints should be also satisfied for the remaining stencil directions. Figure 5:2 illustrates the corresponding tile.

   Note that the stencil matrix $\mathcal{S}$ denotes the amount of data to be received and the stencil-direction matrix $\mathcal{SD}$ denotes the direction of communication. In order to find an appropriate tile shape, therefore, requires analyzing the stencil matrices $\mathcal{S}$ and $\mathcal{SD}$, and the neighborhood matrix $\mathcal{N}$. Since the order of storage in both these matrices is the same, it is straightforward to find out whether communication is required and to whom a processor should send/receive data.

4.3.1 Conforming Directions

The application stencil $\mathfrak{S}$ determines the *conforming directions* for the given application. Conforming directions serve two purposes: (1) Using $\mathcal{SD}$ and $\mathcal{N}$, determine the processor pairs which should exchange data and (2) Using $\mathcal{S}$ and $\mathcal{N}$, compute data to be sent/received per seed.

Assume that processors are arranged in a rectangular grid. Any processor's top neighbor can be found by checking the first column of the Neighborhood matrix $\mathcal{N}$. The first column of $\mathcal{SD}$ will show *whether* the processor should communicate with its top neighbor and the amount of data the processor should receive from its top neighbor will be found by checking the first column of stencil matrix $\mathcal{S}$. Similarly, any processor's bottom neighbor can be found by checking the second column of the neighborhood matrix and the data to be received can be found by checking the second column of the stencil matrix $\mathcal{S}$. For the stencil shown in Figure 5:1, the stencil matrix $\mathcal{S}$ is

$$\begin{pmatrix} 0 & 0 & 2 & -1 & 1 & -2 & 4 & -1 \\ 2 & -3 & 0 & 0 & 1 & -2 & -4 & 1 \end{pmatrix}$$

and the stencil-direction matrix $\mathcal{SD}$ is

$$\begin{pmatrix} 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -1 & 0 & 0 & 1 & -1 & -1 & 1 \end{pmatrix}$$

For this stencil, a processor receives 2 elements per seed from its top neighbor and sends 3 elements per seed to its top neighbor. Consider a processor pair (i, j), where i is a top-neighbor of j. For this stencil, processor i receives 3 elements per seed from processor j and processor j receives

[1]Not to be confused with the local address space in in-core distributed memory problems

[2]Note that the stencil seed has four neighbors in direction 7 but only one in direction 8. Such stencils are called asymmetric regular stencils.

[3]A requirement because of SPMD nature of the program

[4]Since computation requires non-local data, communication requirements define the computation requirements and vice versa.

2 elements per seed from processor i. Hence, for any processor pair which exhibits top-bottom connectivity, amount of data exchanged can be found by checking columns 1 and 2 of the stencil matrix. The directions 1 and 2 are termed as *conforming directions* and the corresponding columns in the stencil matrix $\mathcal{SD}$ are called *conforming columns*. If a processor pair (i,j) exhibits left-right connectivity, then the conforming directions will be (3,4). In case of two dimensional stencils, possible pairs of conforming directions include (1,2), (3,4), (5,6) and (7,8). The conforming directions also provide some extra information. Recall that the first column of $\mathcal{S}$ corresponds to a processor *receiving* two elements per seed from its top neighbor (i.e., communication in direction 1). If the processor has a bottom neighbor, this column also corresponds to the given processor *sending* two elements per seed to its bottom neighbor (communication in direction 2).

How does information about conforming directions help in finding a suitable tile shape? The communication pattern depends on the position of a tile. Since a tile can exist in different positions, the tile size should be large enough to store the data that needs be *sent* in any position. For example, for the stencil in Figure 5:1, a processor needs to send 4 elements per seed in direction 8 (top-left) but only 1 element per seed in direction 3 (right). In this case, each tile needs to be large enough to store at least 4 elements along the diagonal but only 1 element along x direction (Figure 5:2).

4.3.2 Tiling Heuristic

For a given stencil matrix $\mathcal{S}$, memory $\mathcal{M}$ and the neighborhood matrix $\mathcal{N}$, the tiling heuristic computes the appropriate shape of the rectangular tile. The tiling heuristic consists of two main phases:

1. The tiling heuristic first uses the stencil matrix to compute the amount of communication along the conforming directions. Using this information, the heuristic computes the *approximate* size of a rectangular tile such that the tile can fit in memory and the Communication constraint is satisfied.

2. In the next phase, the tiling heuristic analyzes the I/O cost and tries to choose a tile shape which has the minimum I/O cost. The heuristic will try to find a rectangular tile with the minimum number of contiguous columns (e.g., column tiles) so the the I/O constraint is satisfied.

The tiling heuristic represents a tile using a two dimensional diagonal matrix called the tiling matrix $\mathbb{T}$. Diagonal elements of the tiling matrix denote the extent of the tile in the corresponding dimension. The tiling matrix $\mathbb{T}$ can be also used to compute the *degree of freedom* ($\mathfrak{F}$) of the corresponding tile. $\mathfrak{F}$ denotes the possible direction(s) in which the next tile can be fetched. For example, the column and row tiles have $\mathfrak{F}$=1 whereas generalized rectangular tiles (e.g., square) have $\mathfrak{F}$=2. Let $\mathbb{O}$ be a diagonal matrix representing the out-of-core local array (OCLA), where each diagonal element denotes the size of the array in the corresponding dimension. Then the degree of freedom $\mathfrak{F}$ of a given tile is computed as the number of the non-unit diagonal elements in the diagonal matrix $\mathbb{F}=\lceil \mathbb{T}^{-1}\mathbb{O} \rceil$. Note that ceiling function takes into account the cases in which extents of the tiles are not divisors of the OCLA dimensions.

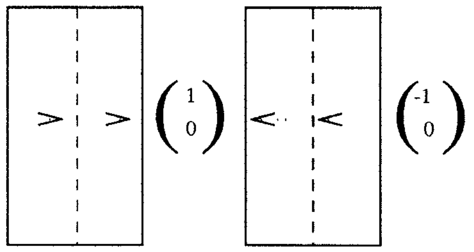

Figure 6: Access Patterns for column tiles and the Schedule Matrices.

The position(s) of the non-unit element(s) denote the *direction of freedom*. The determinant of the matrix $\mathbb{F}$ denotes the total number of tiles.

5 Scheduling Techniques for eliminating out-of-core communication

This section discusses the second part of the ICLA scheduling problem, i.e., for a given stencil, compute the access patterns of the tiles (ICLAs) so that the extra file accesses are eliminated.

5.1 Schedule Matrix

The schedule matrix $\mathcal{H}$ determines in which order the data tiles are fetched from the files. The dimensions of the $\mathcal{H}$ are decided by the degree of freedom $\mathfrak{F}$ of the corresponding tile. Since we are concerned only with 2-dimensional arrays, 1 and 2 dimensional schedule matrices are possible.

Figure 6 represents the column tile ($\mathfrak{F}$=1) and the corresponding schedule matrices. This tile can be fetched in either positive/negative x or y dimension, i.e., the access pattern is monotonic. Figure 6 represents the corresponding schedule matrices. Matrix $\mathcal{H} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ denotes access in positive x direction whereas matrix $\mathcal{H} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$ denotes access in negative x direction. A rectangular tile with $\mathfrak{F}$=2 will have two dimensional schedule matrices [BCR95].

5.2 Scheduling Algorithm

This section describes how to use the schedule matrices for eliminating extra file accesses generated in the In-core communication method.

As noted in the previous section, extra file accesses can be eliminated *iff* the data to be communicated in always in-core. This condition can be satisfied by: (1) Finding processor groups which can *exchange* data and (2) For every such group, computing schedules such that communication is performed only on the in-core data. Such schedules are called *conforming schedules*. Formally, we can write

Theorem 5.1 *Tile Scheduling*

Schedules $\mathcal{H}_i$ and $\mathcal{H}_j$ eliminate extra file I/O iff they satisfy the following equality

$\mathcal{H}_i^T\mathcal{SD}_k = \mathcal{H}_j^T\mathcal{SD}_l$.

where $\mathcal{H}$ is the schedule matrix for processors i and j, $\mathcal{SD}_k$, $\mathcal{SD}_l$ denote the k^{th} and l^{th} column of the stencil-direction matrix $\mathcal{SD}$ (i.e., conforming columns for processors i and j).

Proof:

***if* part:** If schedules $\mathcal{H}_i$ and $\mathcal{H}_j$ eliminate extra file I/O then they satisfy the equality, is a natural consequence of the definition of the conforming schedules. In order to regain the exchange pattern, two neighboring processors must simultaneously fetch the ICLAs which are on the processor boundary. Since only monotonic access patterns are allowed, the processors i and j, should exhibit schedules having different *directions*. In other words, their schedule directions must correspond with the conforming directions of the $\mathcal{SD}$ matrix. Therefore, these schedules must satisfy Theorem 5.1.

***only if* part:** Any schedule pair that satisfies the equality must eliminate extra file I/O, can be proved by contradiction. Let us assume that there exists a schedule pair which satisfies the equality but not eliminate extra file I/O. In other words, these schedules do not differ in directions (i.e., both may correspond to increasing or decreasing access pattern). Recalling the definition of the conforming directions, if $\mathcal{SD}_k$ and $\mathcal{SD}_l$ denote conforming directions then, $\mathcal{SD}_k$=-$\mathcal{SD}_l$ or vice versa. Substituting this property in the equality, we get $\mathcal{H}_i^T$=-$\mathcal{H}_j^T$. Therefore, the schedules that satisfy the equality must differ in sign (or direction). This contradicts our assumption. Hence, any schedule pair that satisfies the equality must eliminate extra I/O.

Theorem 5.1 is not valid for irregular stencils. In other words, there may exist schedules $\mathcal{H}_i'$ and $\mathcal{H}_j'$ which will eliminate extra file I/O but not satisfy the equality.

We now sketch an argument that for a regular stencil, any schedule computed using Theorem 5.1 does not result in a deadlock. Formal proof is omitted due to lack of space. Let us assume that for a stencil $\mathfrak{S}'$ and processor map P', the scheduling algorithm generates a schedule which results in deadlock. Deadlock occurs when data required by the ICLA of a processor is not present in its neighbor's memory. Since extra file accesses are not allowed, the processor will suspend its computation. As a result, for the next ICLA, more processors will suspend their computations because the data required by them will not be in memory of the already suspended processor. This effect will snowball; resulting in a deadlock.

Any legal schedule (i.e., a schedule that satisfies Theorem 5.1) satisfies constraints generated by conforming directions. Conforming directions clearly define groups of processors which should be scheduled together so that In-core communication will involve only the *exchange* of in-core data. Since data exchange involves communication between processor pairs, deadlock is avoided.

The scheduling algorithm consists of three steps:

- Assign a symbolic schedule matrix to each processor. The dimension of the schedule matrix will be equal to the degree of freedom of the tiles.
- Compute the schedule equations for each processor using the stencil matrix and the conforming directions. The schedule equations represent the constraints imposed by the conforming directions.
- Initialize the schedule matrix of any one processor by a legal schedule and compute the corresponding schedules of the remaining processors by reducing the schedule equations. These schedules satisfy the constraints imposed by conforming directions and regain the *exchange* communication pattern.

We demonstrate the scheduling algorithm using the running example (Figure 2) for the column tiles. Initially the algorithm is explained for 2×2 processor grid and then extended for a generalized $m \times n$ processor grid.

5.2.1 Tile Scheduling for 2×2 Processor Grid

Consider 9-point star stencil ($\mathfrak{S}_9^\star$) application running on 4 processors arranged as 2×2 processor grid (Figure 7). Each processor is associated with an unique two-element address and a corresponding processor index. For example, processor with index 3 has an address (1,1). Figure 7 illustrates the ghost area generated for processor 3 by $\mathfrak{S}_9^\star$. Processor 3 requires data from processors 0,1,2. Processors 0,1 and 2 also exhibit similar communication patterns. Figures 7:C illustrates the column tiles.

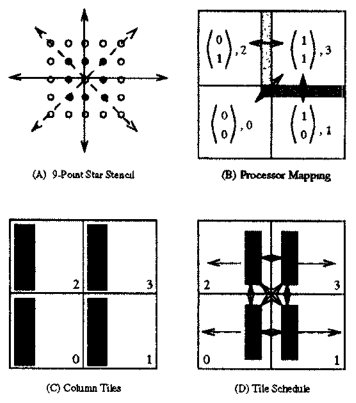
(A) 9-Point Star Stencil
(B) Processor Mapping
$\begin{pmatrix}0\\1\end{pmatrix}$,2
$\begin{pmatrix}1\\1\end{pmatrix}$,3
$\begin{pmatrix}0\\0\end{pmatrix}$,0
$\begin{pmatrix}1\\0\end{pmatrix}$,1
2
3
0
1
(C) Column Tiles
2
3
0
1
(D) Tile Schedule

Figure 7: Tiling a $\mathfrak{S}_9^\star$ application.

We now outline the computation of schedules using the column tiles. A detailed description of the algorithm using different tile shapes is given in [BCR95].

Step 1: Assigning Symbolic Schedule Vectors

Since column tiles have $\mathfrak{F}$=1, we use symbolic schedule vectors. Let $\begin{pmatrix} a_1^i \\ b_1^i \end{pmatrix}$ denote a symbolic schedule vector for processor i.

Step 2: Computing Schedule Equations. For example, equations for Processor 3 are

1. $\begin{pmatrix} a_1^3 & b_1^3 \end{pmatrix} \begin{pmatrix} -1 \\ -1 \end{pmatrix} = \begin{pmatrix} a_1^0 & b_1^0 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$

2. $\begin{pmatrix} a_1^3 & b_1^3 \end{pmatrix} \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} a_1^1 & b_1^1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

3. $\begin{pmatrix} a_1^3 & b_1^3 \end{pmatrix} \begin{pmatrix} -1 \\ 0 \end{pmatrix} = \begin{pmatrix} a_1^2 & b_1^2 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

Step 3: Computing Individual Schedule Matrices

In order to generate individual schedule matrices, initially assign a random schedule to a processor and compute the remaining schedules using the schedule equalities. In our example, let the schedule matrix for processor 3, $\mathcal{H}_3$ be

$\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Using $\mathcal{H}_3$ we can compute the following schedules $\mathcal{H}_0 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$, $\mathcal{H}_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $\mathcal{H}_2 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$.

Figure 7:D presents the corresponding tile access pattern. In this access pattern, for each ICLA, every processor *exchanges* data with its neighbors. Thus communication involves only inter-processor communication and extra file I/O is eliminated.

5.2.2 Tile Scheduling for $m \times n$ Processor Grid

In the previous section, we described the scheduling algorithm for a simple 2×2 processor grid. In this section, we extend the algorithm to any arbitrary grid of size $m \times n$.

Generalized Algorithm

1. Consider the processor grid as a collection of points in the integer lattice $\mathbb{L}_2$. Each processor can be addressed using a 2-element vector (i, j). Assign indices to the processors, p_i, according to their positions in the lattice (Figure 8).
2. Choose a 2×2 grid of processors. For simplicity, we always choose a grid containing processor 0. For example, in Figure 8 we choose the grid of processors 0,1,3 and 4. We call this grid as a *reference grid.*
3. From the chosen reference grid, choose an interior processor (not situated on a grid boundary) as a *reference processor.* It should be noted that using our index ordering, the reference processor will always have odd index (or address (i,j), where i and j are odd).
4. Generate schedule matrices and the corresponding equations for the reference grid. Assign a random schedule to the reference processor and compute the schedules of the remaining processors in the reference grid.
5. We can now extrapolate the schedule to all the remaining processors in the grid using the following lemma

Lemma 5.1 *Generalized Scheduling*
Consider a processor from the reference grid having address (i, j). Let $\mathcal{H}_{ref}$ denote its schedule matrix. Then all the processors having addresses $(i + l \times 2, j + k \times 2), 0 \leq k, l$ have the same schedule matrix.

Figure 8 presents a schedule for the running example using the column tiles. In all the cases, processor 3 is chosen as the reference processor. Processors 0 (address (0,0)), 2 (address (0.2)), 6 (address (0,2)) and 8 (address (2,2)) share the schedule matrix $\mathcal{H}_1$. It should be noted that this *cookie-cutter* approach allows schedule reuse and prevents computation of schedule for every processor.

6 Experimental Results

This section summaries experimental performance results of out-of-core stencil computations using the 9-point stencil on the Intel Paragon. The reader is referred to [BCR95] for more detailed results. These experiments were performed on out-of-core square arrays distributed in BLOCK-BLOCK fashion. The main goal of the experiments was to analyze the effects of allocated memory, tile shape and ICLA schedule on the communication cost. Each experiment was run for three tile shapes, column, row and square and in each case, the cost of local I/O, LIO and inter-processor communication, COMM was measured. Table 1 presents the COMM times for the 9-point star stencil using 8K×8K array distributed on 64 processors. The table shows COMM for unscheduled and scheduled access patterns for two memory sizes ($\frac{1}{4}$th and $\frac{1}{16}$th of the OCLA size) and computes the resultant performance gain. Note that we didnot run experiments using the row tiles for unscheduled access patterns since the excessive I/O cost made them impractical.

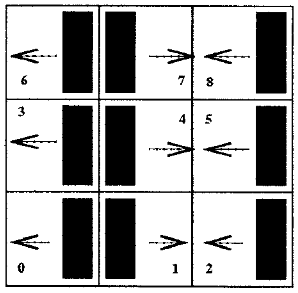

Figure 8: Tile Scheduling for $\mathfrak{S}_9^\star$ for a 3×3 processor grid.

Table 1: COMM gain for column, row and square tiles for $\mathfrak{S}_9^\star$. Time in seconds

Tile	Memory	Unscheduled	Scheduled	Gain
Column	1/16	1.88	0.05	31.86
	1/4	2.06	0.05	39.84
Row	1/16	1.88	0.06	28.96
	1/4	2.06	0.05	42.4
Square	1/16	213.18	0.015	14025
	1/4	192.2	0.0053	35992.51

From Table 1, we observe that

- In absence of scheduling, for all three tiles, COMM does not change significantly as the amount of allocated memory is varied. This is because, as the allocated memory is decreased, the cost of interprocessor communication decreases (since the amount of data to be communicated decreases) but the number of file accesses increase, thus offsetting any gain obtained from reduction in inter-processor communication. However, as the amount of allocated memory is decreased, the number of file accesses increase, thus increasing the local I/O cost, LIO. When the allocated memory is less, LIO cost dominates the overall cost. As the amount of allocated memory increases, COMM becomes dominant.
- LIO depends on the shape of the tile. Column tiles, incur least LIO, followed by square tiles and row tiles. For the application illustrated in Table 1, when the allocated memory is $\frac{1}{16}$th the OCLA size, LIO for column tiles is 13.77 seconds, for square tiles is 417.77 seconds and for row tiles is 1540.95 seconds.
- In all cases, the in-core communication cost is significantly reduced by scheduling the tiles. The effect of tile scheduling is more prominent in the cases where the

communication cost, COMM, matches the local I/O cost, LIO. For column and row tiles, improvement in COMM is relatively small (upto 260), but for square tiles the improvement in COMM is significant (upto 36000).

7 Discussion

Though the paper only focussed on 5- and 9-point stencils, the ICLA scheduling algorithm is also applicable to generalized two dimensional stencils like the Hex stencil [Lee90, BCR95]. Note that communication for the Hex stencil would require more data to be communicated but the communication pattern would be same as the 9-point stencil. Also, the scheduling algorithm could be easily extended for three dimensional stencils like 9- and 13-point stencils. For 3 dimensional stencils, the stencil, processor and schedule matrices ($\mathcal{S}, \mathcal{SD}, \mathcal{N}$ and $\mathcal{H}$) would be modified to accomodate the extra dimension. Currently, we are incorporating the techniques presented in this paper in a prototype out-of-core HPF compiler [Bor96].

8 Related Work

Fortes and Moldovan studied stencil-based computations in the framework of VLSI design [MF86]. Lee [Lee90]analyzed communication patterns of different commonly observed stencils and described different partitioning strategies to minimize the communication-to-computation ratio. A lot of work has been done for developing compiler techniques for optimizing stencil based codes for data-parallel languages like CM-Fortran[BHMJ91, BHTJ94].

9 Conclusions

We showed that communication in out-of-core stencil computations require file I/O as well as inter-processor communication. We used the In-core communication method for performing communication. In this method, the communication pattern and the amount of data communicated is a function of the in-core data slab (called ICLA). We showed that by taking advantage of **FORALL** semantics, ICLA accesses can be scheduled such that any extra file I/O are eliminated. Furthermore, we illustrated how the compiler, using the stencil and processor information, can *automatically* choose the ICLA parameters and decide a suitable scheduling pattern. Finally, we demonstrated, through, experimental results, that: (1) In-core communication cost depends on the ICLA parameters and (2) ICLA access scheduling is an effective method for eliminating extra file I/O incurred in the In-core communication method.

Acknowledgments

The work of R. Bordawekar and A. Choudhary was supported in part by NSF Young Investigator Award CCR-9357840, grants from Intel SSD and in part by the Scalable I/O Initiative, contract number DABT63-94-C-0049 from Advanced Research Projects Agency(ARPA) administered by US Army at Fort Huachuca. R. Bordawekar is also supported by a Syracuse University Graduate Fellowship. The work of J. Ramanujam was supported in part by an NSF Young Investigator Award CCR-9457768, an NSF grant CCR-9210422 and by the Louisiana Board of Regents through contract LEQSF(1991-94)-RD-A-09. This work was performed in part using the Intel Paragon System operated by Caltech on behalf of the Center for Advanced Computing Research (CACR). Access to this facility was provided by CRPC. Authors thank the anonymous reviewers for their suggestions.

References

[BCR95] Rajesh Bordawekar, Alok Choudhary, and J. Ramanujam. Automatic Optimization of Communication in Out-of-core Stencil Codes. Technical Report SIO 114, Scalable I/O Initiative, CACR, California Insititute of Technology, November 1995.

[BHMJ91] Mark Bromley, Steven Heller, Tim McNerney, and Guy Steele Jr. Fortran at Ten Gigaflops: The Connection Machine Convolution Compiler. In *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 145–156, June 1991.

[BHTJ94] Ralph Brickner, Kathy Holian, Balaji Thiagarajan, and S. Lennart Johnsson. Designing a Stencil Compiler for the Connection Machine Model CM-5. Technical Report LA-UR-94-3152, Los Alamos National Laboratory, 1994.

[Bor96] Rajesh Bordawekar. *Techniques for Compiling I/O Intensive Parallel Programs*. PhD thesis, Electrical and Computer Engineering Dept., Syracuse University, 1996. In preparation.

[FJL+88] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors, Volume 1: General Techniques and Regular Problems*. Prentice Hall, 1988.

[Hig93] High Performance Fortran Forum. High Performance Fortran Language Specification. *Scientific Programming*, 2(1-2):1–170, 1993.

[Lee90] Fung F. Lee. Partitioning of Regular Computation on Multiprocessor Systems. *Journal of Parallel and Distributed Computing*, 9:312–317, 1990.

[MF86] Dan Moldovan and Jose A. B. Fortes. Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays. *IEEE Transactions on Computers*, C-35(1), January 1986.