A Unified Tiling Approach for Out-Of-Core Computations*

M. Kandemir
CIS Dept., Syracuse University, Syracuse, NY 13244
mtk@top.cis.syr.edu

R. Bordawekar
CACR, Caltech, Pasadena, CA 91125
rajesh@cacr.caltech.edu

A. Choudhary†
ECE Dept., Northwestern University, Evanston, IL 60208-3118
choudhar@ece.nwu.edu

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

Abstract

This paper describes a framework by which an out-of-core stencil program written in a data-parallel language can be translated into node programs in a distributed-memory message-passing machine with explicit I/O and communication. We focus on a technique called *Data Space Tiling* to group data elements into slabs that can fit into memories of processors. Methods to choose *legal* tile shapes under several constraints and deadlock-free scheduling of tiles are investigated. Our approach is *unified* in the sense that it can be applied to both FORALL loops and the loops that involve flow-dependences.

1 Introduction and Related Work

Since, today, almost every processor has some kind of memory hiearachy organized into layers with different costs, compiler optimizations to reduce memory access costs are important. Tiling, one such optimization, was first used by Abu-Sufah et al.[Abu81] in order to optimize loop nests in a paging-memory system. The later applications were generally on cache memories and registers [Wol89, WL91]. In [RS92] a number of loop iterations were aggregated into tiles that execute atomically without any synchronization in a distributed-memory message-passing machine. Irigoin and Triolet [IT88] introduce tiles which are atomic, identical and bounded. Within this context different (and sometimes contradictory) optimization criteria have been offered to choose the best tile shape and size [RS92, SD90, BDRR93].

The orientation of this paper is different from those of the previous works in one important aspect: We tile the data that reside on disks; that is, we address so called *out-of-core problem.* The primary data structures for the programs reside on disks and the programs explicitly read from and write into disks. We call the unit of transfer between disk and memory a *Data Tile* and the technique to schedule read and writes *Data*

*This work was supported in part by NSF Young Investigator Award CCR-9357840, NSF CCR-9509143 and in part by the Scalable I/O Initiative, contract number DABT63-94-C-0049 from Defense Advanced Research Projects Agency(DARPA) administered by US Army at Fort Huachuca. The work of J. Ramanujam was supported in part by an NSF Young Investigator Award CCR-9457768 and by the Louisiana Board of Regents through contract LEQSF(1991-94)-RD-A-09.

†A longer version of this paper may be obtained from `http://web.ece.nwu.edu/~choudhar`.

Space Tiling. We demonstrate the tradeoffs in choosing good tile shapes for FORALL loops[KLS94] and the loops that contain flow-dependences. *Extra File I/O* is introduced and scheduling techniques to eliminate it are presented.
The rest of the paper is organized as follows: Section 2 describes the problem and the underlying model. In Section 3, reuse vectors and chain vectors are discussed. How tiling parameters are determined is discussed in Section 4. Section 5 studies scheduling of data tiles and we conclude in Section 6.

2 Problem Description and Our Model

The problem addressed in this paper is to compile applications that use very large amount of data on distributed memory message-passing architectures. A computation is called *Out-Of-Core*(OCC) if the data used by it cannot fit in the memory; that is, parts of data reside in files.

In the rest of the paper, a message passing distributed memory machine is assumed. Out-of-core arrays are divided by the programmer into local out-of-core arrays each of which is stored on a logical disk attached to processor. We call this model *Local Placement Model* (LPM). Each processor has its local out-of-core files stored on the logical disk attached to it, and data sharing is performed by explicit message passing[Bor96]. During the course of program, parts of the local out-of-core file, called *tiles* or *slabs*, are fetched into memory, the new values are computed and the tile is stored back (if necessary) into appropriate locations in the local file. *Computation Volume* of a tile is the number of data points it contains.

Our programming model is based on the data parallel programming paradigm. In this model, parallelism is achieved by decomposing data among processors. We assume a fixed set of distribution patterns e.g., row-block, column-block, block-block. Array partitioning results in each processor storing in memory a local array associated with each array. In an out-of-core program local arrays are also out-of-core.

3 Preliminaries

3.1 Reuse Vectors and Reuse Matrices

We assume that loop bounds and array subscripts are affine functions of the enclosing loop indices, and all the statements are inside the deepest loop. Data reuse in a program can be expressed by an integer vector called *Data Reuse Vector* [WL91, Li94, Wol96]. If $\vec{i}_1$ and $\vec{i}_2$ are two iterations that access the same data element, the reuse vector for this access can be defined as $\vec{r} = \vec{i}_2 - \vec{i}_1$. *Reuse Matrix*[Li94] is a matrix every column of which is a reuse vector.
Suppose that $X(\vec{f}(\vec{i}))$ is a reference for an array X on the LHS of an assignment and $X(\vec{g}(\vec{i}))$ is a reference for the same array on the RHS of assignment where $\vec{f}(\vec{i}) = A\vec{i} + \vec{c}_1$ and $\vec{g}(\vec{i}) = A\vec{i} + \vec{c}_2$. Here A is the access (reference) matrix and $\vec{c}_1$ and $\vec{c}_2$ are constant vectors. Data reuse betweeen these two references can be found by solving $A\vec{i}_1 + \vec{c}_1 = A\vec{i}_2 + \vec{c}_2$. Then, the temporal reuse vector, $\vec{r} = \vec{i}_2 - \vec{i}_1$ can be found from $A\vec{r} = \vec{c}_1 - \vec{c}_2$. Note that the reuse matrix captures both flow-dependences and anti-dependences. Since in a FORALL statement all dependences are resolved as anti-dependences, reuse matrix contains only anti-dependences. It is convenient to represent reuse matrix R as $R = [D; S]$ where D and S denote the matrices that contain flow-dependence and anti-dependence vectors respectively. D is frequently called *Data Dependence Matrix*[ZC90, Wol96].

3.2 Chain Vectors and Chain Matrices

An r-dimensional array defines an r-dimensional polyhedron. The vectors that define the relation between data points (array elements) are called *Chain Vectors* [RS89]. For example, the relation between data points for $X(i) = X(i-1) + X(i+1)$ can be represented by two chain vectors for each value of i. One of them is in direction 1 whereas the other one is in direction -1. This corresponds informally to the statement *in order to compute the new value of $X(i)$ both $X(i-1)$ and $X(i+1)$ are needed.* It should be emphasized that the chain vectors, in general, can span different arrays[1]. In this paper we only consider stencil applications, and therefore we assume that there is only one array referenced in the nest and all references to it have the same access matrix.[2] In that case chain vectors can be represented in a graph called *Data Space Graph (DSG).* Suppose that $X(\vec{f}(\vec{i}))$ and $X(\vec{g}(\vec{i}))$ are two references for the same array X and the latter occurs on RHS whereas the former occurs on LHS. Let $\vec{f}(\vec{i}) = A\vec{i} + \vec{a}_1$ and $\vec{g}(\vec{i}) = A\vec{i} + \vec{a}_2$, where A is the access matrix and $\vec{a}_1$ and $\vec{a}_2$ are constant vectors.
A data reuse exists between two iterations $\vec{i}_1$ and $\vec{i}_2$ *iff* $A\vec{i}_1 + \vec{a}_1 = A\vec{i}_2 + \vec{a}_2 \Rightarrow A(\vec{i}_2 - \vec{i}_1) = \vec{a}_1 - \vec{a}_2 \Rightarrow A\vec{r} = \vec{a}_1 - \vec{a}_2$ where $\vec{r}$ is the reuse vector (it may be dependence or anti-dependence vector). On the other hand, we can define a chain vector $\vec{t}$ for this reference pair as follows: $\vec{t} = (A\vec{i} + \vec{a}_1) - (A\vec{i} + \vec{a}_2) = \vec{a}_1 - \vec{a}_2$. From these two equations we obtain an important relation between $\vec{t}$ and $\vec{r}$:

$$\vec{t} = A\vec{r}$$

where A is the access matrix, $\vec{r}$ is the reuse vector and $\vec{t}$ is the chain vector. If $\vec{r} \in D$ we call $\vec{t}$ an *Effective Chain Vector.* In other words, an effective chain vector is a chain vector implied by a flow-dependence. A *Chain Matrix T* is a matrix every column of which is a chain vector. On the other hand an *Effective Chain Matrix U* is a chain matrix every column of which is an effective chain vector. We have now the following relation between U and D:

$$U = AD \tag{1}$$

And for every $\vec{u} \in U$ and $\vec{d} \in D$ we have

$$\vec{u} = A\vec{d}$$

We assume that access matrix A is square and invertible. That is, the dimensionality of data space is equal to that of iteration space. As an example suppose that a 2-deep nest has the statement X(i,i+j)=X(i-1,i+j)+X(i-1,i+j+2). Then $A = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$, $D = \begin{pmatrix} 1 & 1 \\ -1 & -3 \end{pmatrix}$, and $U = \begin{pmatrix} 1 & 1 \\ 0 & -2 \end{pmatrix}$.

4 Constraints on the Tile Shape and Size

Let n be the dimension of the iteration (and data) space and m be the number of columns of the effective chain matrix U. Every tile shape can be succintly described by one of two ways. Let P_d be an nxn matrix every column of which corresponds to tile boundary in that dimension. It is known that the columns of P_d constitute an extreme vector set for the effective chain matrix U [RS92]. The second way to define a tile is a matrix H_d every column of which is a vector perpendicular to the tile boundary along that dimension. The relation between P_d and H_d is $P_d = H_d^{-1}$ [IT88, BDRR93]. In the rest of the paper P_d and H_d are called *tiling matrices.*

In general, there are four factors that determine shape and size of a tile on data space:

- Computation Constraint. Tile shape must be legal in the sense that all effective chain vector traffic between two tiles must be in one direction (i.e. from one tile to the other.)

[1] The chain vectors that span different arrays are called *Cross Chain Vectors* whereas the chain vectors defined entirely on a single array are called *Regular Chain Vectors*. In this paper we concentrate only on regular chain vectors.
[2] The references defined that way form a Uniformly Generated Reference Set [GJK88, WL91].

- I/O Constraint. Tile shape must be compatible with the file layout.
- Communication Constraint. The number of chain vectors (effective or not) going from one tile to the others and the number of tiles that a tile communicates with should be minimized.
- Memory Constraint. Tile size cannot be larger than the size of the memory of a node.

Of course, for a loop that contains only anti-dependences, computation constraint does not exist.

4.1 Computation Constraint (Legality)

Arbitrary clustering of data space points into tiles might result in effective chain vector cycles between tiles. Tile shapes that produce effective chain vector cycles are called *illegal* tiles. The reason for illegality is that processing of tiles must be *atomic* in the sense that a tile must take all the data it requires from outside before computation begins, and all the data required by other tiles should be available after computation terminates. Allowing effective chain vector cycles among tiles violates this requirement. As an example, for the data space graph shown in Figure 1:(A), tiles (1) and (2) are legal, while tile (3) is illegal. The arrows on the figure represent effective chain vectors.

Let h_1,h_2,....,h_n be the rows of the H_d matrix and u_1,u_2,....,u_m be the columns of the effective chain matrix U. The legality condition can now be stated as follows [IT88, RS92, BDRR93]:

$$h_i.u_j \geq 0 \tag{2}$$

for $i = 1, 2, ..., n$ and $j = 1, 2, ..., m$. We note that H_d is not necessarily unique.

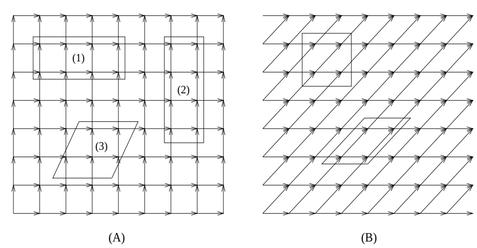

Figure 1: Different Tile Shapes

4.2 I/O Constraint

I/O cost of a tile is determined by the number of file accesses (I/O calls) required to read it from disk. Under *column-major* layout assumption, in order to minimize the number of file accesses, number of (sub)-column reads should be minimized. This may not always possible in practice as minimization of I/O calls can lead to illegal tiles or some communication requirements may impose lower bounds on some edges of tile. In Figure 1:(A), tile (1) and tile (2) have the same computation volume (8 data points); however I/O cost (number of sub-columns) of tile (1) is 4 while that of tile (2) is 2. Everything else being equal, tile (2) is a better choice than tile (1). Note that this constraint is unique to data space tiling.

4.3 Communication Constraint

Communication Volume [BDRR93] of a tile is the number of chain vectors (effective or not) going from one tile to the others. Communication volume may be reduced if the set of extreme vectors for chain vectors

(i.e. columns of P_d) is a subset of the chain vectors. An important observation is that minimizing the communication volume (cost), in some cases, leads to an increase in the I/O cost. This situation is shown in Figure 1:(B) (taken from [BDRR93]). The tile on the left leads to 5 communications whereas the right one leads to 4 communications. On the other hand, the left one needs 2 I/O calls while the one on the right needs 3 I/O calls. This example clearly demonstrates the tradeoff between I/O and communication constraints.
There is another aspect of the communication as well. For any tile, the tile size along each dimension must be larger than the magnitude of the maximum of the corresponding components of chain vectors. This will ensure that all the communicating tiles will be neighbors.

4.4 Memory Constraint

Tile size cannot be larger than the size of node memory. We can state this constraint as $M \geq V_{comp}(T)$ where M is the size of the memory of a single processor and $V_{comp}(T)$ is the computation volume of data tile T.

Let H_i and P_i denote iteration space (IS) tiling matrices while H_d and P_d denote data space (DS) tiling matrices. The proofs of the following theorems can be found elsewhere[KBCR96].

Theorem: It is always possible to find an H_d and a P_d in order to tile the data space in a deadlock-free manner provided that A invertible.

Theorem: For any legal H_d(or P_d) on DSG, there is a corresponding legal H_i(or P_i) on the iteration space provided that A is invertible.

Theorem: If access matrix A is unimodular, then the number of integer points in the data tile and that of the corresponding iteration tile are equal; overmore there is one-to-one correspondence between the points of these two tiles.

4.5 Choosing Tile Shape and Size

Overall cost of a data tile (T_{cost}) has two components: I/O cost ($T_{I/O}$) and communication cost (T_{comm}). Consider now the tile and the effective chain vector shown in Figure 2:A. The computation volume of this tile $V_{comp} = |\vec{p_1}\vec{p_2}|$ and the communication volume can be approximated $V_{comm} \approx |\vec{p_1}\vec{u}| + |\vec{u}\vec{p_2}|$ [BDRR93]. First we need a few definitions.

$C_{I/O}$ = startup cost for an I/O read (write), $t_{I/O}$ = cost of reading (writing) an element from (into) a file, C_{comm} = startup cost for communication, t_{comm} = cost of communicating an element, and K=maximum message length of the machine.

The overall cost (file read and send communication) is

$$T_{cost} = p_{11}C_{I/O} + V_{comp}t_{I/O} + \lceil \frac{V_{comm}}{K} \rceil C_{comm} + V_{comm}t_{comm}$$

then the *optimization problem* is to minimize T_{cost} under the constraints $V_{comp} \leq M$ and ${P_d}^{-1}U \geq 0$. Moreover, all entries of P_d are restricted to be integers. This problem in general is difficult to solve. In the following we present a heuristic that works for restricted cases of $\vec{u}$ (or U in general).
Since $C_{I/O}$ is the most costly term in overall cost expression, we believe that the communication cost of a tile is of secondary importance as compared with its I/O cost.

Recall that the condition for legality of a data tile represented by tiling matrix H_d is

$$H_d.U \geq 0$$

where H_d is the tiling matrix for the data space and U is the effective chain matrix. We now consider a restricted version of U: We assume that $\forall \vec{u} \in U$ is lexicographically positive.

Theorem: In two-dimensional case if columns of U are lexicographically positive, it is always possible to choose a tiling matrix P_d of the form

$$P_d = \begin{pmatrix} 1 & 0 \\ -e & 1 \end{pmatrix} \tag{3}$$

where $e > 0$ so that $H_d U \geq 0$ (see [RS92] for the proof).

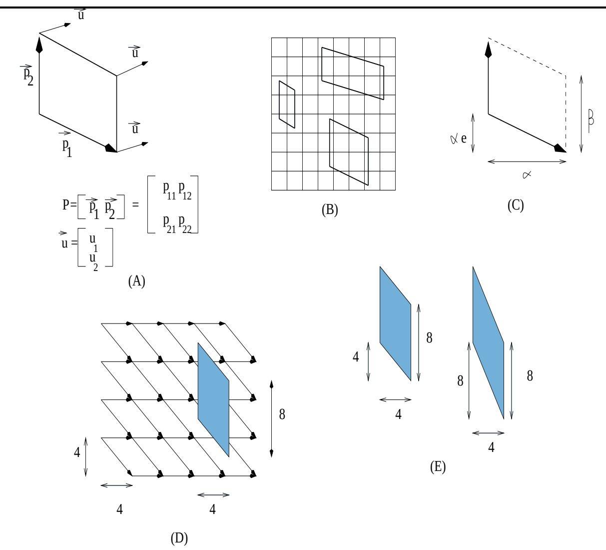

Figure 2: (A) A Data Tile defined by $\vec{p}_1$ and $\vec{p}_2$ and an Effective Chain Vector $\vec{u}$. (B) Legal Tile Shapes. (C) Entries of P_d on the Data Tile. (D) Legal Tile for $e = 1$ on the Data Space. (E) Legal Tile Shapes for $e = 1$ and $e = 2$.

A few legal tile shapes that are specified by such a P_d matrix are shown in Figure 2:B. Re-scaling $\vec{p}_1$ by α and $\vec{p}_2$ by β gives the following re-scaled tiling matrix:

$$P_d = \begin{pmatrix} \alpha & 0 \\ -\alpha e & \beta \end{pmatrix}$$

We now impose our constraints on this specific type of P_d (see Figure 2:C).

- Legality Constraint. $e \geq \max(\max(\frac{-u_{2i}}{u_{1i}}), 1)$, $(u_{1i} \neq 0)$ where $\vec{u}_i = \begin{pmatrix} u_{1i} \\ u_{2i} \end{pmatrix}$ is the ith effective chain vector.
- I/O Constraint. α should be minimized.
- Communication Constraint. $\alpha \geq \max(|t_{1j}|)$, $\beta \geq \max(|t_{2j}|)$ where $\vec{t}_j = \begin{pmatrix} t_{1j} \\ t_{2j} \end{pmatrix}$ is the jth chain vector.
- Memory Constraint. $\alpha\beta \leq M$ where M is the memory size of a node.

Considering I/O and communication constraints together it is clear that $\alpha = \max(|t_{1j}|)$. Then considering communication and memory constraints together we have

$$\max(|t_{2j}|) \leq \beta \leq \frac{M}{\alpha}$$

From this last expression β and from the legality constraint e can be set to appropriate values.

As an example suppose that $U = T = \begin{pmatrix} 4 & 4 \\ 0 & -4 \end{pmatrix}$ and $M = 32$. Using the constraints given above $\alpha = 4$, $e \geq 1$ and $4 \leq \beta \leq 8$. In order to utilize available memory as much as possible, we should set $\beta = 8$. Now different values for e give different solutions all of which have the minimum I/O cost. For example if $e = 1$ we have $P_d = \begin{pmatrix} 4 & 0 \\ -4 & 8 \end{pmatrix}$, if $e = 2$ on the other hand we obtain $P_d = \begin{pmatrix} 4 & 0 \\ -8 & 8 \end{pmatrix}$. Figure 2:E shows I/O optimized legal tiles for $e = 1$ and $e = 2$; and Figure 2:D shows the former one on the data space graph of the example.

5 Optimizing I/O in Stencil Codes

5.1 Translated Code

Translated node program of processor k for a 2-deep nest is as follows[3]:

```
DO IT = LB_IT,UB_IT,S_IT
   DO JT = LB_JT,UB_JT,S_JT
      read data tile DT defined by f_k(IT,JT) from local file
      compute the corresponding iteration tile CT for data tile DT
      handle communication and storage of boundary data
         DO IE = LB_IE,UB_IE,S_IE
            DO JE = LB_JE,UB_JE,S_JE
               perform computation on DT to compute the new data values
            ENDDO JE
         ENDDO IE
      handle communication and storage of boundary data
      write data tile DT defined by f_k(IT,JT) into local file
   ENDDO JT
ENDDO IT
```

In the translated loop nest, loops IT and JT are called *tiling loops* and loops IE and JE are called *element loops*. Thoroughout this section we try to find a suitable *scheduling function* f_k for a given **computation** such that overall I/O cost is minimized and potential parallelism is exploited. Note that LB_{IE}, UB_{IE}, LB_{JE}, and UB_{JE} depend on the iteration tile CT. In other words, out-of-core compiler first fetches the data tile, then computes the corresponding iteration tile, and after that executes the iterations in the iteration tile using the elements of the data tile. Note also that scheduling function f_k may be different for each processor, and this fact is used to find schedules that eliminate all unnecessary I/O and maximize parallelism.

5.2 Extra File I/O Problem

In this subsection we concentrate on how an out-of-core compiler should automatically schedule tile reads/writes to access data in an efficient manner in a distributed-memory environment. First we consider the case with cross-processor anti-dependences only.

Definition: Any file I/O performed for any purpose except local computation is termed as *Extra File I/O* [Bor96].

[3] Translated code for the FORALL statement is very similar, so it is omitted (see [Bor96]).

Since extra file I/O is pure overhead, it should be eliminated whenever possible. In out-of-core computations, communication cost can, in genaral, be negligible for most of the cases when compared with the I/O cost. But inappropriate communication methods can cause extra file I/O; in other words communication may necessiate extra I/O. The proposed technique tries to eliminate the types of communication that cause extra file I/O. Within this context, the efficacy of a scheduling method will be tested by 1) the amount of overhead it incurs, and 2) the amount of extra memory needed to hold the data to be transferred between tiles. The overhead involved can be divided into two groups: extra file I/O cost and communication cost. Let us now elaborate on extra file I/O cost. During computation when a non-local data is requested by a processor, there are two possibilities: 1) the data has been brought into memory by the owner processor prior to the request, or 2) the data has not been brought into memory yet. In the first case if a processor finds out that a data will later be requested by another processor,it can keep the data in memory till it is requested. This approach does not cause extra file I/O (if there is enough memory), but it requires extra memory allocation for the data to be transferred. In the second case however, the owner processor should read the data requested by issuing I/O call(s) and send it to the requesting processor. Our scheduling strategy must prevent the occurence of this second case whenever possible.

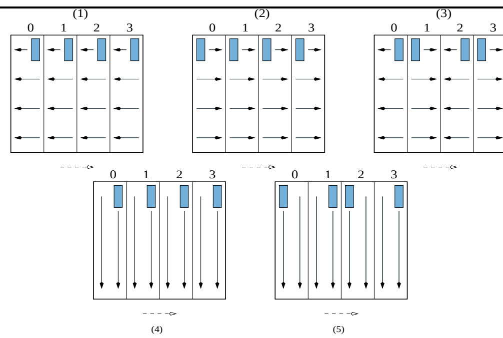

Figure 3: Different Scheduling Strategies

Consider now Figure 3 to see the overheads of different scheduling strategies for a square tile of size SxS (dashed arrows represent the direction of anti-dependence whereas solid arrows indicate execution order). Schedule (1) and (4) do not cause extra file I/O, but require extra memories. Schedule (2) leads to extra file I/O. This is because when, for example, processor 1 needs boundary data from processor 0, processor 0 should issue file read requests in order to read the data. The schedules (3) and (5) do not involve extra file I/O and do not require any extra storage for the data to be transferred. Note also that if there were two anti-dependences in the opposite directions instead of one, then the schedules (1) and (4) would involve extra file I/O as schedule (2). The next section discusses methods to eliminate extra file I/O.

5.3 Scheduling Tiles

We assume two-dimensional data sets and two-dimensional processor grid for the illustrative purposes. In such a grid, every processor has a *direction vector* with respect to each of its neighbors. As an example, consider Figure 5:A. The direction vector for processor 2 with respect to processor 1 is $\begin{pmatrix} 1 \\ -1 \end{pmatrix}$. In general, direction vector for processor i with respect to processor j is denoted by v_{ij}.

Each tile can be represented by a *size matrix* T_s. It is a square matrix diagonal elements of which are the sizes of the tile in the corresponding dimensions and all other elements are zero. Let O be a similar diagonal matrix representing the local out-of-core array in terms of its size. These two matrices can be used to compute an important parameter f called *Degree of Freedom* and a matrix F called *Freedom Matrix*. Given a T_s and an O, the freedom matrix can be computed as $F = \lceil T_s^{-1} O \rceil$. The number of the non-unit diagonal elements of F gives the degree of freedom f. *Scheduling matrix* G_i, for processor i, determines the order in which the data tiles are brought into memory for processing. For a nested loop that contains only anti-dependences (e.g. a FORALL statement[4]), tile access patterns and their corresponding scheduling matrices for f=1 and f=2 are given in Figure 4:A and Figure 4:B respectively. At the heart of the scheduling algorithm is the

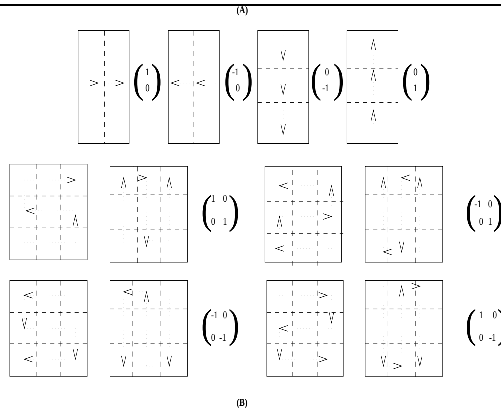

Figure 4: Tile Access Patterns and Scheduling Matrices for a loop that contains only anti-dependences.

following theorem the proof which as well as a generalization to arbitrary grid sizes may be found in [BCR95].

Theorem: In a two by two processor grid, schedules G_i and G_j eliminate extra file I/O between processor i and j *iff* they satisfy the following equality :

$$G_i^T v_{ij} = G_j^T v_{ji}$$

Given the theorem above, the scheduling algorithm for a nested loop that contains only anti-dependences 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 every processor pair using G_i,G_j,v_{ij} and v_{ji}.
- initialize the schedule matrix of a processor with an arbitrary schedule from Figure 4 and compute the corresponding schedules of the remaining processors by solving the schedule equations.

Consider the following elliptic solver example that uses five-point relaxation.

```
FORALL(i=2:n-1,j=2:n-1)
   X(i,j)=(X(i,j)+X(i+1,j)+X(i-1,j)+X(i,j+1)+X(i,j-1))/5.0
ENDFORALL
```

Suppose f=2 and $F = \left(\begin{array}{cc} 2 & 0 \\ 0 & 2 \end{array}\right)$ as shown in Figure 5:B.

Let $\left(\begin{array}{cc} a_i & b_i \\ c_i & d_i \end{array}\right)$ denote a symbolic schedule matrix for processor i. After obtaining the necessary equations using the preceding theorem, if we let the scheduling matrix for processor 0 to be $\left(\begin{array}{cc} a_0 & b_0 \\ c_0 & d_0 \end{array}\right) =$

[4]It should be noted that if a FORALL loop contains multiple statements, there may be flow-dependences among different statements, and they should be counted for.

$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$, then the remaining schedules are $G_1 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$, $G_2 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$, and $G_3 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. As shown in Figure 5:C, these values define a scheduling which does not involve any extra file I/O.

Theorem If a loop contains only anti-dependencies, then using LPM, it is always possible to schedule tiles such that all extra file I/O is eliminated [BCR95, Bor96].

Let us now consider the computations that involve flow-dependences, considering only rectangular tiles. A *Tiling Space Graph* (TSG) is defined to indicate the chain vectors among the tiles. We are dealing with the two-dimensional case where each component of the effective chain vectors between two tiles is 0 or 1. As a first step, the tile access patterns that are associated with scheduling matrices are re-defined as shown in Figure 6. The numbers indicate the order of execution. We discuss a technique called *minimal perturbation*. Suppose that an effective chain vector u_i in the TSG is a member of set $span\{\begin{pmatrix} 1 \\ 0 \end{pmatrix}\}$. The practical significance of this is that any scheduling in the opposite direction $\begin{pmatrix} -1 \\ 0 \end{pmatrix}$ is not acceptable. In order to avoid that, after all schedule matrices G_i are obtained (as in the previous case), if the first column of any of them is $\begin{pmatrix} -1 \\ 0 \end{pmatrix}$, it is converted to $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and the other column is left as it is. So, in order to reach legitimate schedules we apply minimal changes to schedule matrices, hence the name minimal perturbation.

The scheduling algorithm for a nested loop that may contain flow-dependences is the same as that of a loop that contains only anti-dependences, except that the initial schedule should be *legal* (observe the effective chain vectors) and maximize (pipeline-) parallelism. As a last step of algorithm, we apply minimal perturbation. Consider the following example that illustrates the technique (see Figure 5:D)

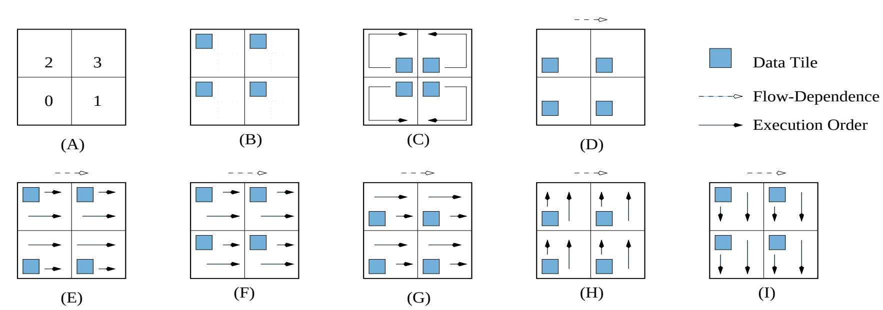

Figure 5: (A) A 2x2 Processor Grid. (B) Data Tiles with f=2. (C) A scheduling that eliminates extra file I/O. (D) Data Tiles with flow-dependence. (E) A scheduling that eliminates extra file I/O for flow-dependence case. (F) A scheduling that leads to extra file I/O. (G) A scheduling that requires extra storage. (H-I) Schedulings that sequentialize computation.

```
DO i = 2,n-1
   DO j = 1,n-1
      X(i,j)=(X(i+1,j)+X(i,j+1)+X(i-1,j))/3.0
   ENDDO j
ENDDO i
```

This nest contains an effective chain vector in $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ direction. If we write down the schedule equations as in the previous example and initialize $G_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ (to maximize pipeline parallelism), after the fourth step of the algorithm we obtain $G_1 = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$, $G_2 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$, and $G_3 = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$.

The schedules for processor 1 and processor 3 are violating the effective chain vector and are not acceptable. So, we apply minimal perturbation to correct them at the last step of the algorithm and we obtain $G_0 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $G_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $G_2 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$, and $G_3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$. These schedules are shown in Figure 5:E. It is easy to see that there is no extra file I/O involved. Also notice that the locality between processors 0 and 2 (similarly between 1 and 3) is exploited. The reason is that the minimal perturbation preserves the localities originating from anti-dependence relations as much as possible. Scheduling in Figure 5:F, on the other hand, leads to extra file I/O (because of the reference $X(i, j+1)$) and scheduling in Figure 5:G requires extra storage of size $n/2$ per processor. From the discussion above we can conclude the following, proof of which is omitted due to lack of space, but follows closely the preceding discussion.

Theorem: By scheduling data tiles appropriatelly, it is always possible using LPM to eliminate extra file I/O.

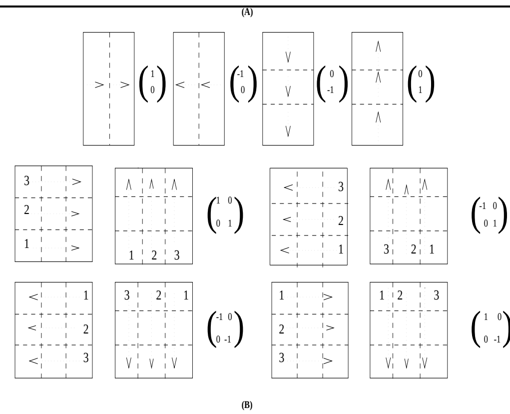

Figure 6: Tile Access Patterns and Scheduling Matrices for loop nests containing flow-dependences.

In a nested loop that contains only anti-dependences, it is not important what the initial schedule for a processor is. But in a nest that contains flow-dependences, an initial schedule must be legal (not violate any effective chain vector). In addition to that, among the candidate schedules some of them might be preferable over the others, especially when parallelism is considered. To see this, consider Figure 5:H and Figure 5:I. Although both schedules are perfectly legal, neither of them exploits the parallelism available.

6 Conclusion and Future Work

In this paper, we have presented a technique to decompose an out-of-core array stored on disk(s) into partitions called *data tiles*. We have discussed the extra file I/O problem and proven that it is always possible to find tile schedules so that extra file I/O is eliminated completely in *stencil computations*.
We are currently working on the feasibility of data space tiling approach for loop nests that contain arbitrary computations on out-of-core arrays in multicomputers.

References

[Abu81] W.Abu-Sufah. On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations, *IEEE Transactions on Computers*, C-30(5), pages 341-355, May 1981.

[Bar94] R.K.Barua, Global Partitioning of Parallel Loops and Data Arrays for Caches and Distributed Memory in Multiprocessors, Masters Thesis, Dept. of Electrical Engineering and Computer Science, MIT, May 1994.

[BC95] R.Bordawekar and A.Choudhary. Communication Strategies for Out-Of-Core Programs on Distributed Memory Machines. In *Proc.International Conference on Supercomputing*, pages 395-403,Barcelona, July 1995.

[BCR95] R.Bordawekar, A.Choudhary, and J.Ramanujam. Automatic Optimization of Communication in Out-Of-Core Stencil Codes. Technical Report, Scalable I/O Initiative, Center of Advanced Computing Research, CALTECH, November 1995.

[BDRR93] P.Boulet, A.Darte, T.Risset, and Y. Robert, (Pen)-ultimate tiling? Technical Report 93-36,Ecola Normale Superieure de Lyon, 46, Alle'e d'Italie, 69364 Lyon Cedex 07, France, November 1993.

[Bor96] R.Bordawekar. Techniques for Compiling I/O Intensive Parallel Programs, Ph.D. Thesis, Dept. of Electrical and Computer Eng., Syracuse University, April 1996.

[GJK88] D.Gannon, W. Jalby, and K.Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformations, *Journal of Parallel and Distributed Computing*,5:587-616, 1988.

[IT88] Francois Irigoin and Remi Triolet. Supernode Partitioning. *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319-329, San Diego, CA, January 1988.

[KBCR96] M.Kandemir, R.Bordawekar, A.Choudhary, and J.Ramanujam. Data Space Tiling for Out-of-Core Computations. Technical Report, ECE Dept., Northwestern University, Evanston, IL, September 1996.

[KLS94] C.Koebel, D.Lovemen, R.Schreiber, G.Steele, and M.Zosel. *High Performance Fortran Handbook*. MIT Press, 1994.

[Li94] W.Li. Compiler Optimizations for Cache Locality and Coherence, Technical Report 504, Dept. of Computer Science, University of Rochester, April 1994.

[RS89] J. Ramanujam and P. Sadayappan. A Methodology for Parallelizing Programs for Multicomputers and Complex Memory Multiprocessors, In *Proc. Supercomputing'89*, Reno, NV, pages 637-646, November 1989.

[RS92] J.Ramanujam and P.Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108-120, October 1992.

[SD90] R.Schriber and J.Dongarra. Automatic Blocking of Nested Loops. Technical Report, RIACS, May 1990.

[WL91] M.Wolf and M.Lam. A data Locality Optimizing Algorithm. in *Proc. ACM SIGPLAN 91 Conf.Programming Language Design and Implementation*, pages 30-44, June 1991.

[Wol89] M.Wolfe, More Iteration Space Tiling. in *Proc. Supercomputing' 89*, pages 655-664, Reno NV, November 1989.

[Wol96] M.Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, CA, 1996.

[ZC90] H.Zima and B.Chapman. *Supercompilers for Parallel and Vector Supercomputers*, ACM Press Frontier Series, 1990.