Optimization of Out-of-Core Computations Using Chain Vectors

M. Kandemir[1] J. Ramanujam[2] A. Choudhary[3]

[1] EECS Dept., Syracuse University, Syracuse, NY 13244
[2] ECE Dept., Louisiana State University, Baton Rouge, LA 70803-5901
[3] ECE Dept., Northwestern University, Evanston, IL 60208-3118

Abstract. Over the last decade, processor speed has become significantly higher than memory and disk speeds. Therefore, exploiting the memory hierarchy has emerged as a key problem in parallel computing. An out-of-core computation is one which operates on disk-resident data. This paper uses the concept of chain vectors for tiling out-of-core codes. The theory of chain vectors is discussed and extended, and their relation to reuse vectors is established. Then, chain vectors are used to optimize the tile size, shape, allocation and scheduling for out-of-core codes.

1 Introduction

A computation which operates on disk-resident arrays is called *out-of-core*, and an optimizing compiler for out-of-core computations is termed as *out-of-core compiler.* In contrast a computation which operates on data sets in memory is called *in-core.* Chain vectors [7] are used in this paper primarily to guide to handling out-of-core programs; they can also be employed in a variety of compiler-based techniques such as automatic data distribution. We emphasize the necessity of a data space based approach for out-of-core computations in contrast to the iteration space based approaches employed by contemporary in-core parallelizing compilers.

Tiling [1, 3, 8, 10], which has received considerable attention lately, is a compiler technique which groups a number of iterations together in order to minimize communication costs and/or maximize locality. Irigoin and Triolet [3] introduced tiles which are atomic, identical and bounded. In [8, 1], tiling has been used to combine (vectorize) the communication associated with each group of iteration points in order to amortize the high communication startup cost. The other approaches [10] have generally concentrated on enhancing the cache performance by improving degree of data reuse. In out-of-core computations, however, primary data structures, such as arrays, 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 disk reads and writes, *data space tiling.* In this paper, we (1) use the concept of chain vectors to guide the compilation of out-of-core codes, (2) discuss how the loop and data transformations interact with the chain vectors, (3) show how to determine important tile parameters.

2 Mathematical Preliminaries

Reuse Vectors

We assume that the loop bounds and the array subscripts are affine functions of the enclosing loop indices, and all the statements are inside the deepest loop. Such a loop nest define a polyhedron in Z^K where K is the number of the loops in the nest. Each point of this polyhedron corresponds to an iteration of the loop nest, and can be denoted by a column vector $\mathbf{i} = [\iota_1, \iota_2, \cdots, \iota_K]^T$ where ι_j is the j^{th} outermost loop index. If $\mathbf{i}_1$ and $\mathbf{i}_2$ are two iterations that access the same data element, the *data reuse vector* for this access can be defined as $\mathbf{r} = \mathbf{i}_2 - \mathbf{i}_1$. Consider the following assignment statement inside such a loop nest.

$$X(A\mathbf{i} + \mathbf{a}) = \cdots X(B\mathbf{i} + \mathbf{b}) \cdots \tag{1}$$

We call A and B reference matrices, and $\mathbf{a}_1$ and $\mathbf{b}_2$ offset vectors. Data reuse between these two references can be found by solving the system

$$\left\{ A\mathbf{i}_1 + \mathbf{a}_1 = B\mathbf{i}_2 + \mathbf{b}_2 \qquad \mathbf{lb} \leq \mathbf{i}_1, \mathbf{i}_2 \leq \mathbf{ub} \qquad \mathbf{r} = \mathbf{i}_2 - \mathbf{i}_1 \right\}$$

for $\mathbf{r}$ where $\mathbf{lb}$ and $\mathbf{ub}$ denote the loop bounds. Each value of $\mathbf{r} = \mathbf{i}_2 - \mathbf{i}_1$ gives a data reuse vector, and all reuse vectors together constitute a *data reuse matrix* $\mathcal{R}$. We denote by $\mathcal{D}$ to a matrix, each column of which is a flow dependence vector; and by $\mathcal{A}$ to a matrix, each column of which is an anti-dependence vector. It is convenient to represent the reuse matrix as $\mathcal{R} = [\mathcal{D}; \mathcal{A}]$.

Chain Vectors

Although the reuse vectors capture the data reuse between iterations and the matrix $\mathcal{D} \subseteq \mathcal{R}$ contains all the necessary information to tile the iteration space in a deadlock-free manner; since we are interested in tiling the data (file) space, we need another abstraction which captures the precise relations between individual data points (array elements).

We begin by observing that an r-dimensional array defines an r-dimensional polyhedron. The vectors that define the relations between data points are called *chain vectors* [7]. Let us consider again the assignment statement given by (1). A chain vector $\mathbf{c}$ for this reference pair can be defined as $\mathbf{c} = (A\mathbf{i} + \mathbf{a}) - (B\mathbf{i} + \mathbf{b}) = (A - B)\mathbf{i} + (\mathbf{a} - \mathbf{b})$ [7]. It should be emphasized that the chain vectors, in general, can span different arrays, i.e. if $B[i]$ is needed in computing $A[i, j]$ then there is a chain vector from $B[i]$ to $A[i, j]$ for all i and j.

Relation Between Reuse Vectors and Chain Vectors

In statement (1), suppose that there is a reuse between two iterations $\mathbf{i}_1$ and $\mathbf{i}_2$. Let $\mathbf{d}_j$ denote a data point (an array element) in the data space (array X). Without loss of generality, we assume a flow dependence from $\mathbf{i}_1$ to $\mathbf{i}_2$ caused by the data point $\mathbf{d}_1$.

$$\mathbf{d}_1 = A\mathbf{i}_1 + \mathbf{a} = B\mathbf{i}_2 + \mathbf{b} \qquad \mathbf{d}_2 = B\mathbf{i}_1 + \mathbf{b} \qquad \mathbf{d}_3 = A\mathbf{i}_2 + \mathbf{a}$$

We can define the reuse vector $\mathbf{r} = \mathbf{i}_2 - \mathbf{i}_1$. On the other hand, there are two different chain vectors involved in this computation:

$$\mathbf{c}_1 = \mathbf{d}_1 - \mathbf{d}_2 = (A\mathbf{i}_1 + \mathbf{a}) - (B\mathbf{i}_1 + \mathbf{b}) \qquad \mathbf{c}_2 = \mathbf{d}_3 - \mathbf{d}_1 = (A\mathbf{i}_2 + \mathbf{a}) - (B\mathbf{i}_2 + \mathbf{b})$$

Taking the difference between $\mathbf{c}_2$ and $\mathbf{c}_1$, we get the following relation among $\mathbf{r}$, $\mathbf{c}_1$ and $\mathbf{c}_2$.

$$\mathbf{c}_2 - \mathbf{c}_1 = (A - B)(\mathbf{i}_2 - \mathbf{i}_1) \qquad \Rightarrow \qquad \mathbf{c}_2 - \mathbf{c}_1 = (A - B)\mathbf{r} \tag{2}$$

The rest of the paper concentrates on a special case where $A = B$. In this case $A\mathbf{i}_1 + \mathbf{a} = A\mathbf{i}_2 + \mathbf{b}$, and $A\mathbf{r} = \mathbf{a} - \mathbf{b}$. On the other hand, $\mathbf{c} = (A\mathbf{i}_1 + \mathbf{a}) - (A\mathbf{i}_1 + \mathbf{b}) = (A\mathbf{i}_2 + \mathbf{a}) - (A\mathbf{i}_2 + \mathbf{b}) = \mathbf{a} - \mathbf{b}$. We now have the following important relation:

$$\mathbf{c} = A\mathbf{r} \tag{3}$$

If $\mathbf{r} \in \mathcal{D}$, we call $\mathbf{c}$ an *effective chain vector*, and denote it by $\mathbf{u}$. In other words, an effective chain vector is a chain vector implied by a flow dependence. The *chain matrix* $\mathcal{C}$ is a matrix, each column of which is a chain vector. On the other hand, the *effective chain matrix* $\mathcal{U} \subseteq \mathcal{C}$ is a chain matrix, each column of which is an effective chain vector. It is easy to see that

$$\mathcal{U} = A\mathcal{D} \tag{4}$$

3 Transformations

In this section, we present some results in order to demonstrate the effect of loop and data transformations on chain vectors. The proofs of the theorems can be found in [4].

Theorem 1. *Any unimodular transformation [11] $\mathcal{T}$ of iteration space does not change $\mathcal{U}$.*

Some recent research [5] has concentrated on data space transformations. The work on this area can be divided into two categories as described below.

The approaches with fixed storage layout for all arrays In this approach, the data space is transformed using linear non-singular transformation matrices; but the transformed space of each array is stored on disk in a fixed storage order. [5] has applied this technique to in-core programs. Let $\mathcal{Y}$ be a linear non-singular data transformation matrix. Omitting the shift-type transformations, the data transformation denoted by $\mathcal{Y}$ is applied in two steps (1) The original reference matrix A is transformed to $\mathcal{Y}A$, and (2) The data layout on the disk is also transformed by using $\mathcal{Y}$, and the array declaration statements are changed accordingly.

Theorem 2. *If $\mathcal{C}$ is the original chain matrix, after the data transformation $\mathcal{Y}$ (as defined above), $\mathcal{YC}$ is the new chain matrix; but $\mathcal{D}$ remains unaffected by $\mathcal{Y}$.*

The approaches with different storage layouts for different arrays This technique assigns different disk layouts for different out-of-core arrays, if doing so promotes the spatial locality on disks. In this scheme, linear non-singular data transformation matrix $\mathcal{V}$ is applied only to the data space.

Theorem 3. *The linear non-singular data transformation matrix $\mathcal{V}$ (as defined above) does not change the chain matrix $\mathcal{C}$.*

4 An Application: Data Space Tiling for Out-of-Core Computations

Let N be the dimensionality of the data space, K be the dimensionality of the iteration space, and M be the number of columns of the effective chain matrix $\mathcal{U}$. A data tile can be succinctly described by one of two ways. Let $\mathcal{P}_d$ be an $N \times N$ matrix, each column of which corresponds to tile boundary in that dimension. It can be shown that the columns of $\mathcal{P}_d$ constitute an extreme vector set for the effective chain matrix $\mathcal{U}$. The other way to define a tile is an $N \times N$ matrix $\mathcal{H}_d$, each column of which is a vector perpendicular to the tile boundary along that dimension. The relation between $\mathcal{P}_d$ and $\mathcal{H}_d$ is $\mathcal{P}_d = \mathcal{H}_d^{-1}$ [3, 8]. In the rest of the paper, $\mathcal{P}_d$ and $\mathcal{H}_d$ are called *tiling matrices*. The factors that determine shape and size of a data tile are as follows:

Computation Constraint In order to reduce the extra overhead, the computation of data tiles should be atomic; that is, there should not be an effective chain vector cycle between any two neighboring data tiles. Tile shapes that cause effective chain vector cycles are called *illegal* tiles. As an example, for the data space shown in Figure 1:(A), the tiles (1) and (2) are legal, whereas the tile (3) is illegal. The grid intersections on the figure represent the data points (array elements), and the arrows represent the effective chain vectors. Two example effective chain vectors that lead to illegality for the tile (3) are shown as dashed lines.

For the mathematical interpretation, let $\mathbf{h}_1$,$\mathbf{h}_2$,....,$\mathbf{h}_N$ be the rows of the $\mathcal{H}_d$ matrix and $\mathbf{u}_1$,$\mathbf{u}_2$,....,$\mathbf{u}_M$ be the columns of the effective chain matrix $\mathcal{U}$. The computation constraint can be stated as follows:

$$\mathbf{h}_i.\mathbf{u}_j \geq 0 \tag{5}$$

for $i = 1, 2, ..., N$ and $j = 1, 2, ..., M$. This inequality has also been used by [3], [8] and [1] for iteration space tiling.

I/O Constraint The I/O cost of a tile is determined by the number of disk accesses (I/O calls) required to read it from disk. Without loss of generality, we assume *column-major* layout of out-of-core data on disk. Under this assumption, in order to minimize the I/O cost of a data tile, the number of (sub)-column accesses should be minimized. In Figure 1:(A), tile (1) and tile (2) have the same computation volume (6 data points); however I/O cost (number of sub-columns) of tile (1) is 3 while that of tile (2) is 2.

Communication Constraint The *communication volume* of a tile is the number of chain vectors (effective or not) going from one tile to the others. This volume may be reduced if the set of extreme vectors for chain vectors (i.e. columns of $\mathcal{P}_d$) is a subset of the chain vectors. Consider Figure 1:(B) for two legal tiles. The tile (1) leads to 5 communications whereas the tile (2) leads to 4 communications. However, the tile (1) needs only 2 I/O calls while the tile (2) needs 3 I/O calls.

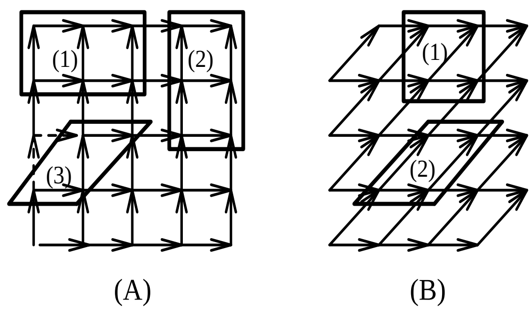

Fig. 1. Two different data spaces with different data tile shapes.

Minimal Neighborhood Constraint For any data 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. This constraint prevents us to choose column tiles which otherwise would be optimal from the I/O perspective.

Memory Constraint The *computation volume* of a data tile is the number of data points (array elements) it contains and is equal to $|det(\mathcal{P}_d)|$. The computation volume can not be larger than the size of the memory of a compute node. We can state this constraint as $M \geq V_w$, where M is the size of the memory of a single processor (compute node) and V_w is the computation volume.

4.1 Determining Tile Parameters

Table 1 presents some notation used in the rest of this paper. The overall cost of a data tile considering *send communication* and *disk read* costs only can be defined as follows:

$$\mathcal{T}_{cost} = \mathcal{T}_f + \mathcal{T}_m + \mathcal{T}_w \tag{6}$$

$$= \underbrace{\delta\mathcal{C}_f + V_w t_f}_{\mathcal{T}_f} + \underbrace{\lceil \frac{V_m}{\xi} \rceil \mathcal{C}_m + V_m t_m}_{\mathcal{T}_m} + \underbrace{V_w t_w}_{\mathcal{T}_w} \tag{7}$$

Table 1. Notation.

Symbol	Definition
$\mathcal{T}_{cost}$	overall cost of a data tile
$\mathcal{T}_f$	I/O cost of a data tile
$\mathcal{T}_m$	communication cost of a data tile
$\mathcal{T}_w$	computation cost of a data tile
V_w	computation volume of a data tile
V_m	communication volume of a data tile
$\mathcal{C}_f$	I/O startup cost
$\mathcal{C}_m$	communication startup cost
t_f	cost of accessing an element from disk
t_m	cost of communicating an element
t_w	cost of computing an element
δ	number of I/O calls to read a data tile
ξ	maximum message length of machine
p	number of processors (machine size)

Notice that we have assumed each data tile needs communication. The optimization problem is to find a $\mathcal{P}_d$ ($\mathcal{H}_d$) such that the $\mathcal{T}_{cost}$ will be minimized under the constraints $V_w \leq M$ and ${\mathcal{P}_d}^{-1}\mathcal{U} \geq 0$. Since this optimization problem is difficult to solve; we make a simple assumption in order to find a fast heuristic: We assume that $\forall \mathbf{u} \in \mathcal{U}$ is lexicographically positive.

Theorem 4. *In an N-dimensional data space, if $\forall \mathbf{u} \in \mathcal{U}$ is lexicographically positive, it is always possible to find a tiling matrix $\mathcal{P}_d$ of the form*

$$\mathcal{P}_d = \begin{bmatrix} 1 & 0 & 0 \cdots & & 0\ 0 \\ -e_1 & 1 & 0 \cdots & & 0\ 0 \\ 0 & -e_2 & 1 \cdots & & 0\ 0 \\ \cdots & \cdots & \cdots\cdots & & 1\ 0 \\ 0 & 0 & \cdots\cdots & -e_{N-1} & 1 \end{bmatrix} \tag{8}$$

where $e_i \geq 0$ so that ${\mathcal{P}_d}^{-1}\mathcal{U} \geq 0$.

Proof: Trivial. For sufficiently large e_i values, ${\mathcal{P}_d}^{-1}\mathcal{U} \geq 0$ can always be satisfied.□

This form of $\mathcal{P}_d$ defines specific data tile shapes on the data space with a computation volume (V_w) of 1. In order to increase the computation volume, these tiles should be re-scaled. Let $\mathbf{p}_1$ and $\mathbf{p}_2$ denote the columns of $\mathcal{P}_d$ for the two-dimensional case. Re-scaling $\mathbf{p}_1$ by ϕ and $\mathbf{p}_2$ by φ results in the following $\mathcal{P}_d$:

$$\mathcal{P}_d = \begin{bmatrix} \phi & 0 \\ -\phi e & \varphi \end{bmatrix}$$

Let us now impose our constraints on this specific type of re-scaled $\mathcal{P}_d$.

- **Computation Constraint:** $e \geq \max(\max(-u_{2i}/u_{1i}), 1)$, $(u_{1i} \neq 0)$ where $\mathbf{u}_i = \begin{pmatrix} u_{1i} \\ u_{2i} \end{pmatrix}$ is the i^{th} effective chain vector.

– **I/O Constraint:** ϕ should be minimized.
– **Communication Constraint:** $\sum_{i=1}^{2}\sum_{j=1}^{M}\sum_{k=1}^{2}(\mathcal{H}_d)_{i,k}\mathcal{C}_{k,j}$ should be minimized as this expression is a measure of communication volume of a data tile in two-dimensional data space [8].
– **Minimal Neighborhood Constraint:** $\phi \geq \max(|c_{1j}|)$ and $\varphi \geq \max(|c_{2j}|)$ where $\mathbf{c}_j = \begin{pmatrix} c_{1j} \\ c_{2j} \end{pmatrix}$ is the j^{th} chain vector.
– **Memory Constraint:** $\phi\varphi \leq M$ where M is the memory size of a compute node.

Our proposed heuristic is as follows:

Step 1: Using **I/O Constraint** and **Minimal Neighborhood Constraint** determine ϕ.
Step 2: Using **Memory Constraint** and **Minimal Neighborhood Constraint** determine φ.
Step 3: Using **Computation Constraint** and **Communication Constraint** determine e.

4.2 Scheduling of Data Tiles for Distributed-Memory Machines

The data tile shape determined in the previous section by using the chain vectors is I/O oriented. Since I/O cost ($\mathcal{T}_f$) is the dominating term in the overall cost expression given by equation (7); this tile is optimal from the I/O point of view and can be used effectively for uniprocessors. A large number of loop nests compiled by parallelizing compilers for massively parallel processors have data dependences requiring inter-processor communication within the loop nest. Since the data tiles defined by parameters (ϕ, φ) found in the previous section do not consider the issues like load balance and distribution of out-of-core data across processors, they are not necessarily optimal for distributed-memory multicomputers. In order to balance the I/O, communication and parallelism requirements, the tile parameters (ϕ, φ) should be chosen such that the total execution time of the nest will be minimized. The previous research on iteration space tiling considered the different types of dependence vectors, and for each type, the appropriate tile parameters were determined for *rectangular* tiles [6]. Instead, we take a different approach: We first fix the *generic* tile shape as in the previous section considering all possible chain vectors in the computation, and then tile the rectangular data (file) space on disk with parallelepiped data tiles. The desired scheduling scheme should minimize the inter-node communication and processor idle time. In two-dimensional case, the tiles along $\mathbf{p}_1$ or $\mathbf{p}_2$ direction can be allocated to a single processor. If the tiles are scheduled along the $\mathbf{p}_1$ direction, then the optimal tiling parameters are $\varphi = \frac{n}{p}$ and $\phi = \frac{pV_w}{n}$. On the other hand, if the tiles are scheduled along the $\mathbf{p}_2$ direction, the optimal values are $\varphi \approx \sqrt{\frac{n\mathcal{C}_f + p\mathcal{C}_m}{2pt_f}}$ and $\phi \approx \frac{V_w\sqrt{2pt_f}}{\sqrt{n\mathcal{C}_f + p\mathcal{C}_m}}$. We refer the interested reader to [4] for details.

5 Conclusions

The difficulty of efficiently handling out-of-core data limits the performance of supercomputers as well as the enormous potential of parallel machines. Unfortunately, most of the compiler techniques for optimizing locality are iteration space oriented and do not consider the efficient handling of out-of-core data on disks. In this paper, we discussed the necessity of a data space oriented approach for disk optimization, and used an extended version of the concept of *chain vectors* introduced in [7].

References

1. P. Boulet, A. Darte, T. Risset and Y. Robert. (Pen)-ultimate tiling? Technical Report 93-36, Ecola Normale Superieure de Lyon, France, November 1993.
2. 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.
3. F. Irigoin and R. Triolet. Supernode Partitioning. *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319-329, San Diego, CA, January 1988.
4. M. Kandemir, A. Choudhary, and J. Ramanujam. Optimization of out-of-core computations using chain vectors. CPDC Technical Report, Northwestern University, June 1997.
5. M. F. P. O'Boyle and P. M. W. Knijnenburg. Non-singular Data Transformations: Definition, Validity, Applications. In *Proc. 6th Workshop on Compilers for Parallel Computers*, Aachen, Germany, 1996.
6. H. Ohta, Y. Saito, M. Kainaga and H. Ono. Optimal Tile Size Adjustment in Compiling General DO-ACROSS Loop Nests. In *Proc. International Conference on Supercomputing*, Barcelona, July 1995.
7. J. Ramanujam and P. Sadayappan. A Methodology for Parallelizing Programs for Multicomputers and Complex Memory Multiprocessors. In *Proceedings of Supercomputing 89,* pp. 637–646, November 1989.
8. J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108-120, October 1992.
9. A. Schrijver. *Theory of Linear and Integer Programming*, Wiley-Interscience series in Discrete Mathematics and Optimization, John Wiley and Sons, 1986.
10. 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.
11. M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, CA, 1996.

This article was processed using the LaTeX macro package with LLNCS style