A Geometric Approach for Partitioning N-Dimensional Non-Rectangular Iteration Spaces

Arun Kejariwal¹, Paolo D’Alberto¹, Alexandru Nicolau¹
Constantine D. Polychronopoulos²

¹ Center for Embedded Computer Systems
University of California at Irvine
Irvine, CA 92697, USA

arun_kejariwal@computer.org, paolo@cecs.uci.edu, nicolau@cecs.uci.edu
<http://www.cecs.uci.edu/>

² Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA

cdp@csrd.uiuc.edu
<http://www.csrd.uiuc.edu/>

Abstract. Parallel loops account for the greatest percentage of program parallelism. The degree to which parallelism can be exploited and the amount of overhead involved during parallel execution of a nested loop directly depend on partitioning, i.e., the way the different iterations of a parallel loop are distributed across different processors. Thus, partitioning of parallel loops is of key importance for high performance and efficient use of multiprocessor systems. Although a significant amount of work has been done in partitioning and scheduling of rectangular iteration spaces, the problem of partitioning of non-rectangular iteration spaces - e.g. triangular, trapezoidal iteration spaces - has not been given enough attention so far. In this paper, we present a geometric approach for partitioning N-dimensional non-rectangular iteration spaces for optimizing performance on parallel processor systems. Speedup measurements for kernels (loop nests) of linear algebra packages are presented.

1 Introduction

High-level parallelization approaches target the exploitation of regular parallelism at loop-nest level for high performance and efficient use of multiprocessor systems [1,2,3]. For example, one approach is to distribute the iterations of a nested loop across multiple processors, also known as *loop spreading*. The efficiency (and usefulness) of loop spreading depends on the ability to detect and partition the independent iterations in a manner that distributes the load uniformly across different processors. Though partitioning loop nests with rectangular iteration spaces has received a lot of attention [4,5], the problem of partitioning nested loops with non-rectangular iteration spaces has not been

given enough attention so far, except in [6,7]. However, the approaches proposed in [6,7] do not partition the iteration space uniformly across different processors and have several limitations, as discussed in Section 7.

In this paper, we focus on the problem of partitioning parallel loop nests with N-dimensional non-rectangular iteration spaces.¹ We present a geometric approach to partition an iteration space for optimizing performance i.e. achieving minimum execution time on a minimum number of processors and achieving balanced work load among processors. We partition an iteration space along the axis corresponding to the outermost loop and achieve a near-optimal partition. The partition thus obtained consists of *contiguous* sets, unlike the approach discussed in [7], which facilitate exploitation of data locality. Furthermore, our approach obviates the need to remap the index expressions and loop bounds of inner loops, in contrast to linearization-based approaches [8,9]. Thus, our approach provides a simple, practical and intuitive solution to the problem of iteration space partitioning. In this paper, we only consider loop nests with no loop-carried dependences. Discussion about partitioning of loops with loop-carried dependences is beyond the scope of this paper.

The rest of the paper is organized as follows - in Section 2, we present an intuitive idea behind iteration space partitioning with an illustrative example. Section 3 presents the terminology used in the rest of the paper. In Section 4 we present a formal description of the problem. Next, we present our partitioning scheme. Section 6 presents our experimental results. Next, we present previous work. Finally we conclude with directions for future work.

2 Iteration Space Partitioning

Consider the loop nest shown in Figure 1. The corresponding iteration space is shown in Figure 1(b). Note that the loop nest has a *triangular* geometry in the (i_1, i_2) plane and a rectangular geometry in the (i_1, i_3) plane. Assuming 3 processors are available, one can partition the outermost loop into three sets along the i_1 -axis, as shown in Figure 2(a).² The cardinality of the sets thus obtained is as follows (from left to right in Figure 2(a)): $|S(1, 3)| = 18$, $|S(3, 5)| = 42$ and $|S(5, 7)| = 66$, where $S(x, y) = \{\mathbf{i} | x \leq i_1 < y\}$.

However, one can interchange the loops corresponding to the loop indices i_1 and i_3 before partitioning. The sets thus obtained have equal cardinality, i.e., $|S(1, 3)| = |S(3, 5)| = |S(5, 7)|$, as shown in Figure 2(b). Unfortunately, in general, such partitioning may be impossible or impractical due to the following reasons: firstly, an equi-partitionable axis may not exist in a given N-dimensional convex polytope; secondly, determining functions for the loop bounds of the permuted loop is non-trivial.

¹ Naturally, our technique can also handle rectangular loop nests too.

² Though an iteration space can be partitioned along an axis corresponding to an inner loop, however, it requires remapping of the index expressions (due to affine loop bounds) which introduces significant overhead. Therefore, we always partition an iteration space along the axis corresponding to the outermost loop.

```

do  $i_1 = 1, N$ 
  do  $i_2 = 1, i$ 
    do  $i_3 = 1, N$ 
      LOOP BODY
    end do
  end do
end do

```

(a) An example loop nest

(b) Iteration space of loop nest shown in Figure 1(a), $N = 6$

Fig. 1. Partitioning non-rectangular iteration space

Fig. 2. Effect of geometry of an iteration space on partitioning

In practice, it is imperative to take into account the iteration space geometry to achieve a balanced work load distribution amongst the different sets of a partition. Also, the number of processors plays an important role in the efficient partitioning of a parallel loop nest. For example, we may partition the iteration space of Figure 1(b) across 4 and 5 processors. The corresponding partitions are shown in Figures 3(a) and 3(b) respectively. We observe that the size of the largest set (represented by boxes in Figures 3(a) and 3(b)) is the same in both the cases, and so is the execution time - if we exclude the scheduling overhead. However, unlike the partition in Figure 3(a), the partition of Figure 3(b) may incur additional scheduling overhead (e.g., communication due to an ex-

Fig. 3. Determining the number of processors ($\leq \mathcal{P}$) for optimal performance based on the minimum size of the largest partition

tra processor and partition) with no performance gain. Thus, efficient processor allocation may also be important during iteration space partitioning.

3 Terminology

Let $\mathcal{L}$ denote a perfect nest of DOALL loops as shown in Figure 4. Without loss of any generality, we assume that the outermost loop is normalized from 1 to N with $s_1 = 1$.³ The **index variables** of the individual loops are $i_1, i_2, \dots, i_n$ and they compose an **index vector** $\mathbf{i} = (i_1, i_2, \dots, i_n)$. An instance of the index vector $\mathbf{i}$ is an **iteration**. The set of iterations of a loop nest $\mathcal{L}$ is an **iteration space** $\Gamma = \{\mathbf{i}\}$. We model an iteration space as a convex polytope in $\mathbb{Z}^n$.

The lower and upper bounds (e.g., f_3 and g_3 , respectively) of an index variable (e.g., i_3) are assumed to be affine functions of the outer loop indices (e.g., i_1 and i_2). We assume that $f_j \leq g_j$ for $1 \leq j \leq n-1$.

The set of hyperplanes, defined by $i_1 = 1, i_1 = N, i_r = f_{r-1}$ and $i_r = g_{r-1}$ for $2 \leq r \leq n$, determine the *geometry* of an iteration space. For example, if $f_r = l_r$ and $g_r = u_r$ where $l_r, u_r \in \mathbb{Z}$ (i.e., constant bounds) for $1 \leq r \leq n-1$, then the geometry of Γ is a rectangular parallelepiped. In this paper, we consider iteration spaces with complex geometries (e.g. tetrahedral, prismoidal) and with different, but constant, strides along each dimension. Of course, our technique can handle simpler, more regular loops as well.

Definition 1. For each $x, y \in \mathbb{R}$ such that $x < y$, an **elementary set** $S(x, y)$ in Γ is defined as follows : $S(x, y) = \{\mathbf{i} | \mathbf{i} \in \Gamma, x \leq i_1 < y\}$.

³ Note that the lower and upper bound of the outermost loop can be any two real numbers $x, y \in \mathbb{R}$. Loops with real bounds are common in numerical applications [10] such as the Newton-Raphson method and Runge-Kutta method.

```

doall  $i_1 = 1, N, s_1$ 
  doall  $i_2 = f_1(i_1), g_1(i_1), s_2$ 
    .
    .
    . doall  $i_n = f_{n-1}(i_1, i_2, \dots, i_{n-1}), g_{n-1}(i_1, i_2, \dots, i_{n-1}), s_n$ 
       $H(i)$ 
    . end doall
  . end doall
end doall

```

Fig. 4. A perfectly nested DOALL loop

Let $\beta(\Gamma) = \{\gamma_k, k = 1, 2, \dots, m\}$ denote a set of **breakpoints** of the outermost loop of $\mathcal{L}$ such that $\gamma_1 < \gamma_2 < \dots < \gamma_m$, where $\gamma_k \in \mathbb{R}$ and $m \leq \mathcal{P} \leq N$, $\mathcal{P}$ is a set of identical processors. A **boundary set** B_{γ_k} in Γ is a set of iterations $B_{\gamma_k} = \{\mathbf{i} | \mathbf{i} \in \Gamma, i_1 = \gamma_k, \text{ and } \gamma_k \in \beta(\Gamma), \gamma_k \in \mathbb{Z}\}$, i.e. B_{γ_k} is a set of all iterations that lie on a hyperplane corresponding to an integer breakpoint.

Definition 2. A contiguous outermost partition of Γ is defined as follows:
 $Q_{\beta(\Gamma)} = \{\bigcup_{\forall k} S(\gamma_k, \gamma_{k+1}) | \gamma_k \in \beta(\Gamma)\}.$

4 Problem Statement

The execution time of an elementary set $S(\gamma_i, \gamma_{i+1})$ is proportional to the number of iterations, $|S(\gamma_i, \gamma_{i+1})|$, in the set. Therefore, the **execution time of a contiguous outermost partition** is

$$\mathcal{T}(Q_{\beta(\Gamma)}) = \max_{\gamma_i \in \beta(\Gamma)} |S(\gamma_i, \gamma_{i+1})| \times t_{LB} \quad (1)$$

where, t_{LB} is the execution time of the innermost loop body (from hereon, we omit t_{LB} , being a constant,⁴ in future discussions). Now, we present a rigorous formulation of the problems we want to solve.

Problem 1 (Minimum execution time). Given an iteration space Γ and $\mathcal{P}$ processors, find a contiguous outermost partition $Q_{\beta(\Gamma)}$ so as to minimize the execution time:

$$\mathcal{T}_{\min}(\Gamma, \mathcal{P}) = \min_{|\beta(\Gamma)|=\mathcal{P}-1} \mathcal{T}(Q_{\beta(\Gamma)}) \quad (2)$$

⁴ We assume that the loop body $H(\mathbf{i})$ has the same execution time in each iteration.

However, as shown in Figure 3(b), there may exist two (or more) contiguous outermost partitions (e.g., $\beta_1(I)$ and $\beta_2(I)$) having a different number of processors (e.g. $\mathcal{P}_1 < \mathcal{P}_2$) but with the same execution time (e.g., $\mathcal{T}(Q_{\beta_1(I)}) = \mathcal{T}(Q_{\beta_2(I)})$). Therefore, we need to find an optimal number of processors so as to minimize the scheduling overhead and maximize efficiency. More precisely, the **processor partition** problem can be stated as follows:

Problem 2 (Minimum processor number). Given a minimum execution time $\mathcal{T}_{\min}(I, \mathcal{P})$ for $\mathcal{P}$ processors, find the minimum number of processors $\mathcal{P}_{\min} \leq \mathcal{P}$ such that :

$$\mathcal{T}_{\min}(I, \mathcal{P}_{\min}) \leq \mathcal{T}_{\min}(I, \mathcal{P})$$

5 The Approach

In this section, we present our algorithm for partitioning an iteration space I across $\mathcal{P}$ processors, with an illustrative example.

First, we compute a partial volume, denoted by $\mathcal{V}(x)$ in Algorithm 1, of the convex polytope corresponding to the loop nest $\mathcal{L}$ as a function of the outermost index variable.⁵ In presence of affine loop bounds, a closed form of the partial volume can be obtained using mathematical packages like Matlab [30]. Note that $\mathcal{V}(x)$ is a non-linear polynomial. Then we compute the total volume of the convex polytope using Equation (3). Next, we determine the breakpoints, using Equation (4), along the i_1 -axis for partitioning I across the given processors. Note that Equation (4) is a non-linear equation. In case of a quadratic equation, closed form of the roots may be used to determine the breakpoints. However, it is important to note that the same approach is not viable for higher (> 2) degree polynomials as it is non-trivial to determine the closed form of their roots and is impossible for polynomial equations higher than fourth degree as stated by Abel's Impossibility Theorem [31]. To alleviate this problem, we use binary search to determine the breakpoints. The solution of Equation (4) corresponds to the k^{th} breakpoint γ_k .⁶ In contrast, algebraic approaches [6,7,11] solve a system of $\mathcal{P}$ simultaneous equations to determine the breakpoints, which incurs significant overhead. The breakpoints thus obtained define the boundaries of the elementary sets in I .

Note that unlike the approach presented in [7], the sets obtained by our approach are contiguous which eliminates the need for multiple loops, required in case of non-contiguous sets. Furthermore, contiguous sets facilitate exploitation of data locality. In contrast, previous approaches [6,7] balance load across different processors at the expense of data locality, as the sets obtained by applying these approaches are non-contiguous.

⁵ Assuming constant loop strides, partial volume is proportional to the number of index points in it.

⁶ Note that $\mathcal{V}(x)$ is a monotonically increasing function of x . Therefore, there exists only one real solution of Equation (4).

Algorithm 1 Near-Optimal Partitioning of N-dimensional Non-Rectangular Iteration Spaces

Input : An N-dimensional non-rectangular iteration space Γ and $\mathcal{P}$ processors.

Output : A near-optimal (w.r.t. load balance amongst the different processors) partition of the iteration space.

/* Compute the partial volume $\mathcal{V}(x)$ of Γ */

$$\mathcal{V}(x) = \int_1^x \int_{f_1}^{g_1} \dots \int_{f_{n-1}}^{g_{n-1}} di_1 di_2 \dots di_n$$

/* Compute the total volume $\mathcal{V}$ of Γ */

$$\mathcal{V} = \int_1^N \int_{f_1}^{g_1} \dots \int_{f_{n-1}}^{g_{n-1}} di_1 di_2 \dots di_n \quad (3)$$

/* Determine the k^{th} breakpoint γ_k . For $1 < k < (\mathcal{P} - 1)$, determine the *
* solution of Equation (4) using binary search */

$$\mathcal{V}(\gamma_k) = \frac{k}{\mathcal{P}} \times \mathcal{V} \quad (4)$$

$\beta(\Gamma) = \{\gamma_1, \gamma_2, \dots, \gamma_{\mathcal{P}-1}\}$

/* Eliminate void sets */

$i \leftarrow 0, E \leftarrow 0$

while $i \neq |\beta(\Gamma)|$ **do**

if $\lfloor \gamma_i \rfloor = \lfloor \gamma_{i+1} \rfloor$ **then**

$\beta(\Gamma) \leftarrow \beta(\Gamma) - \{\gamma_i\}$

$i \leftarrow i - 1, E \leftarrow E + 1$

end if

$i \leftarrow i + 1$

end while

/* Determine the loop bounds for the remaining $\mathcal{P} - E$ sets */

$$lb_k = \begin{cases} 1, & \text{if } k = 1 \\ ub_{k-1} + 1, & \text{otherwise} \end{cases} \quad (5) \quad ub_k = \begin{cases} \gamma_k - 1, & \text{if } \gamma_k = \lfloor \gamma_k \rfloor \\ \lfloor \gamma_k \rfloor, & \text{if } \gamma_k \neq \lfloor \gamma_k \rfloor \\ N, & \text{if } k = \mathcal{P} - E \end{cases} \quad (6)$$

where, lb and ub are the integer lower and upper bounds of an elementary set.

Next, we eliminate the void sets i.e. the sets which do not contain any index points and the corresponding breakpoints. The remaining sets are allocated to each processor. Given a minimum size of the largest set, the elimination process reduces the total number of sets which in turn minimizes the scheduling overhead. The (integer) lower and upper bounds of the loops corresponding to each set are determined using the equations 5 and 6 respectively. The algorithm is formally presented as Algorithm 1 on page 7.

Note that, enumeration-based approaches may be used alternatively for computing the volume of a convex polytope. However, such techniques are not applicable in the presence of *ceiling* and *floor* functions in the loop bounds.⁷ Furthermore, enumeration-based techniques such as [6,7] are not viable in the presence of non-unit strides. In fact, when the loop is normalized so that all index variables have unit stride, a non-linear constraint may appear in the loop bound. A similar problem is encountered during the data cache interference analysis for scientific computations. The non-linear constraint can be reduced to a linear one but with the introduction of further constraints. The authors of Omega Test and Polylib ([13,14]) formalize this transformation and propose a valid solution for the enumeration of rather complex iteration spaces. Nonetheless, the enumeration may be impractical for non-unit strides. In any case, our volumetric approach provides a simple, integrated and intuitive solution to the partitioning and processor allocation problem.

Let us now examine the behavior of our algorithm on the example shown in Figure 1(a). First, we compute a partial volume of the convex polytope as a function of the outermost index variable. Next, we compute the volume of the convex polytope corresponding to the iteration space I . The total and partial volumes are given by :

$$\begin{aligned}\mathcal{V} &= \int_1^N \int_1^{i_1} \int_1^N di_3 di_2 di_1 = 62.5 \\ \mathcal{V}(x) &= \int_1^x \int_1^{i_1} \int_1^N di_3 di_2 di_1 = (N-1) \frac{(x-1)^2}{2}\end{aligned}$$

Notice that alternatively, enumeration-based approaches may apply for the computation of volumes as follows:

$$\Xi = \sum_{i_1=1}^N \sum_{i_2=1}^{i_1} \sum_{i_3=1}^N 1 = N \cdot \sum_{i_1=1}^N \sum_{i_2=1}^{i_1} 1 = N \cdot \sum_{i_1=1}^N i_1 = \frac{N^2(N+1)}{2}$$

Next, assuming 5 processors, we determine the breakpoints for the iteration space of Figure 1(b) are :

⁷ Floors and ceilings in loop bounds may arise after applying unimodular transformations [12]. Similarly, in many real applications, rational loop bounds can be found. In such cases, it is essential to convert the loop bounds to integers using the functions *ceiling* and *floor*.

Fig. 5. Elimination of void sets

$$\begin{aligned}
\frac{(\gamma_1 - 1)^2}{2}(N - 1) &= \frac{1}{5}\gamma \Rightarrow \gamma_1 = 3.2, & \frac{(\gamma_2 - 1)^2}{2}(N - 1) &= \frac{2}{5}\gamma \Rightarrow \gamma_2 = 4.1 \\
\frac{(\gamma_3 - 1)^2}{2}(N - 1) &= \frac{3}{5}\gamma \Rightarrow \gamma_3 = 4.8, & \frac{(\gamma_4 - 1)^2}{2}(N - 1) &= \frac{4}{5}\gamma \Rightarrow \gamma_4 = 5.4
\end{aligned}$$

For simplicity, in this example we use the closed form of the roots of a quadratic equation to determine the breakpoints. Next, we eliminate the void sets i.e. the sets which do not contain any index points and the corresponding breakpoints, as shown in Figure 5. Next, we determine the lower and upper bounds of the remaining sets using the equations 5 and 6 respectively. The new loop bounds for the partition shown in Figure 5 are given by:

$$(lb_1, ub_1) = (1, 4), \quad (lb_2, ub_2) = (4, 4), \quad (lb_3, ub_3) = (5, 5), \quad (lb_4, ub_4) = (6, 6)$$

In the previous example, we assume that sets with equal volumes contain same number of iterations. It is important to note that the assumption is valid only if $\gamma_i \in \mathbb{R} \forall i$. In presence of integer breakpoints, the iterations belonging to the corresponding boundary set must be allocated to either one adjoining elementary sets. This introduces load imbalance in the resulting partition. In Algorithm 1, we allocate such iterations to the elementary set on the right. For example, consider the iteration space shown in Figure 6. Assuming 3 processors, from Figure 6 we observe that $\gamma_1, \gamma_2 \in \mathbb{Z}$. The cardinality of the contiguous sets obtained from Algorithm 1 is 6, 8 and 10.

Ideally, in this case the iterations in each boundary set must be distributed equally amongst the adjoining elementary sets in order to achieve perfect load balance. The boundaries corresponding to such a (optimal) partition is shown by dotted lines in Figure 6. However, such a distribution is not possible as the iteration space is partitioned only along the axis corresponding to the outermost loop. In the worst case, one might have to distribute the iterations in a boundary set

Fig. 6. Iteration space partitioning in presence of integer breakpoints

equally amongst different processors. We now present a bound on performance deviation of the partition obtained by Algorithm 1 from the optimal partition.

Remark 1. Given any contiguous outermost partition $Q_{\beta(\Gamma)}$ determined by Algorithm 1, we have

$$0 \leq \mathcal{T}(Q_{\beta(\Gamma)}) - \mathcal{T}_{\text{opt}}(\Gamma, \mathcal{P}) \leq \max_{\gamma_i \in \beta(\Gamma) \cap \mathbb{Z}} \frac{|B_{\gamma_i}| + |B_{\gamma_{i+1}}|}{\mathcal{P}} \quad (7)$$

where, $\mathcal{T}_{\text{opt}}(\Gamma, \mathcal{P})$ corresponds to the execution time of the optimal partition.

Any elementary set $S_i \in Q_{\beta(\Gamma)}$ is bounded by at most two hyperplanes $i_1 = \gamma_i$ and $i_1 = \gamma_{i+1}$. In Remark 1, if $\gamma_{i+1} \notin \mathbb{Z}$, then $|B_{\gamma_{i+1}}| = 0$.

So far, we have only considered perfect loop nests. However, our algorithm can be applied to *multiway* loop nests⁸ in a similar fashion. Likewise, our approach can be easily extended to handle non-constant strides. Detailed discussion of the above is outside the scope of this paper.

6 Experiments

To illustrate our algorithm's behavior, we experimented with different loop nests taken from linear algebra packages and the literature. Table 1 lists the loop nests - the geometry of their iteration spaces and their source. The loops were partitioned using Algorithm 1. Subsequently, we refer to our approach as *VOL* and the canonical loop partitioning technique as *CAN*.

First, we illustrate the effect of contiguous sets on performance. For the same, we partitioned loop L_1 uniformly across P processors. Since L_1 has rectangular geometry, both our approach and *CAN* result in the same number of iterations per processor. Thus, since both approaches have identical workload on each processor, the difference in performance, refer to Figure 7, can be attributed to

⁸ A loop is *multiway nested* if there are two or more loops at the same level. Note that the loops may be nested themselves[15].

#	Geometry	Source
L_1	Rectangular	SSYRK routine (BLAS3)
L_2	Tetrahedral	[7]
L_3	Trapezoidal	Appendix A [16]
L_4	Rhomboidal	Appendix A [16]
L_5	Parallelepiped	pp. 58 [17]

Table 1. Geometries of the loop kernels. For each loop, $N=1000$.

Fig. 7. Impact of data locality on performance of loop nest L_1 (see Table 1)

data locality effects associated with contiguous sets, as explained in Section 5. Note that, our approach generates contiguous sets even with non-rectangular iteration spaces, thus facilitating exploitation of data locality in such iteration spaces as well.

Next, we illustrate the performance of our algorithm for non-rectangular iteration spaces. For this, we determine the size (cardinality) of the largest set of partitions corresponding to each technique. Note that the size of the largest set in a partition correlates to the performance of that partitioning scheme. Table 2 presents the size of the largest set corresponding to our approach and (for comparison purposes) *CAN*. From Table 2 we observe that our approach results in smaller sizes of the largest set i.e. more uniform distribution of iterations across different processors. The difference in performance can be attributed to the fact that our approach accounts for the geometry of an iteration space during partitioning. We observe that the difference in performance decreases (expectedly) with increasing number of processors, as the volume of each set in a partition also decreases.

As explained earlier, the applicability of *CAN* is restricted to canonical loops only, which explains the presence of dashes for loop nests L_4 and L_5 (non-canonical loop nests) for *CAN*. Furthermore, in case of large number of pro-

Loop Nest	# of Processors							
	2		4		8		16	
	VOL	CAN	VOL	CAN	VOL	CAN	VOL	CAN
L_2	83368284	83596878	41810044	43807393	21003180	22397760	10738024	11538395
L_3	476000	516000	219900	225037	109500	112056	55000	57232
L_4	200000	-	100000	-	50000	-	25000	-
L_5	25000	-	12500	-	62500	-	31500	-

Table 2. Performance (size of the largest set of a partition) of our geometric approach and canonical loop partitioning.

processors, we also address processor allocation issues which eliminates the communication overhead due to extra processors and ensures efficient use of the processors.

7 Previous Work

It has been shown that loops without dependences among their iterations (traditionally known as **DOALLs** [18]) are a rich source of parallelism [19]. In addition, several compiler techniques [20] have been proposed to convert loops with inter-iteration dependences to **DOALL** loops for parallel execution. However, once this is done, the problem is how to partition the iteration space of the **DOALLs** across a given number of processors so as to minimize execution time and optimize processor utilization.

In [21], Anik et al. discuss several models for parallel execution of nested loops. The simplest model is to execute the outermost loop in parallel and all the inner parallel loops sequentially. Another model involves *collapsing* [22] the nested loops into a single loop using compiler transformations. In another model, the inner loops are executed in parallel and a blocking barrier is used at the end of each parallel loop, which prevents the overlapping between execution of inner loops.

Techniques such as loop concurrentization [23] partition the set of iterations of a loop and assigns a different subset to each processor. Irigoin and Triolet’s *supernode partitioning* approach [24] divides an iteration space of a loop nest into nodes with several goals : vector computation within a node, loop tiling for data reuse and parallelism between tiles. In [7], Sakellariou discusses the necessary condition for partitioning a loop nest across different processors with *equal* workload. Based on whether the iterations are distributed among processors before or during run-time, loop partitioning can be classified as *static* or *dynamic*. In *static* partitioning, each processor is assigned a fixed number of iterations such that the distribution among processors is as even as possible. The most common approaches for *static* partitioning are:

- *Cyclic partitioning (CP)* : It distributes the iterations in a round robin fashion; thus given n iterations and p processors, processor i executes iterations $i + kp$, $k = 0, 1, \dots, n/p$. However, this approach may deteriorate performance due to *false sharing*.⁹
- *Block partitioning (BP)* [25] : This approach maps contiguous iterations onto processors in a consecutive manner; thus a processor i executes iterations $in/p + 1$ through $(i + 1)n/p$. The efficiency of BP is governed by the block size. Assuming zero scheduling overhead, the optimal block size is $k = \lceil n/p \rceil$ number of iterations.
- *Balanced chunk scheduling (BCS)* [26] : BCS attempts to distribute the total number of iterations of the loop body among processors as evenly as possible as opposed to cyclic and block partitioning which distribute only the iterations of the outer loop. An example of the latter is shown in Appendix B of [7]. However, Haghighat and Polychronopolous restrict their discussion to double loops.
- *Canonical loop partitioning (CLP)* [7] : Sakellariou introduce a notion of *canonical loop nest* for loop partitioning. CLP assumes that the outermost loop can be equi-partitioned into $2p^{m-1}$ parts, where p is the number of processors and m is the depth of a loop nest. However, this may generate empty sets which leads to load imbalance. Moreover, CLP generates a fragmented partition i.e. each individual set is a collection of non-contiguous subsets. CLP employs an enumeration-based approach to determine the total number of index points in an iteration space. It relies on loop normalization in the presence of non-unit strides. However the introduction of floors and ceilings renders this approach nonviable in practice (see Section 5). Furthermore, determination of the set boundaries in CLP is very cumbersome.

Similarly, Boyle et al. [11] present an approach for load balancing of parallel affine loops using unimodular transformations. Their approach transforms a loop into a load balanced form, i.e. it identifies an index variable i_k , referred to as an invariant iterator, that neither makes any reference to any other index variable in its loop bounds, nor is referenced by any other index variable and reorders the iterations to move i_k as far out as possible. However, this approach relies on the existence of an invariant iterator which restricts its applicability to rectangular iteration spaces only. Similarly, several techniques have been proposed in [27, 28, 29] for mapping affine loops on to multiple processors. However, these techniques focus primarily on communication minimization between the processors.

8 Conclusions

In this paper we presented an algorithm for partitioning N-dimensional iteration spaces. Unlike previous approaches [6, 7], we follow a geometric approach for par-

⁹ *False sharing* occurs when multiple processors access data in the same cache line and one of the accesses is a ‘write’, thus causing the cache line to be exchanged between processors even though the processors access different parts of it.

tioning an iteration space. The partition thus obtained consists of contiguous sets, which facilitates exploitation of data locality. Our approach provides an integrated solution to the partitioning and processor allocation problems. Our experiments show that our approach has better performance than the CAN partitioning technique. As future work, we would like to extend our approach to partition iteration spaces at run-time.

9 Acknowledgments

The first author is highly grateful to Dr. Utpal Banerjee for the discussions, comments and suggestions which helped a lot in preparing the final version of the paper. He would like to thank Rafael Lopez, Carmen Badea and Radu Cornea for their help and feedback.

References

1. D. A. Padua. Multiprocessors: Discussion of theoretical and practical problems. Technical Report 79-990, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1979.
2. R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, August 1986.
3. J. Davies. Parallel loop constructs for multiprocessors. Technical Report 81-1070, Dept. of Computer Science, University of Illinois at Urbana-Champaign, May 1981.
4. C. Polychronopoulos, D. J. Kuck, and D. A. Padua. Execution of parallel loops on parallel processor systems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 519–527, August 1986.
5. E. H. D'Hollander. Partitioning and labeling of loops by unimodular transformations. *IEEE Trans. Parallel Distrib. Syst.*, 3(4):465–476, 1992.
6. Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996.
7. R. Sakellariou. *On the Quest for Perfect Load Balance in Loop-Based Parallel Computations*. PhD thesis, Department of Computer Science, University of Manchester, October 1996.
8. U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976. Report No. 76-837.
9. M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.
10. M.K. Iyengar, S.R.K. Jain, and R.K. Jain. *Numerical Methods for Scientific and Engineering Computation*. John Wiley and Sons, 1985.
11. M. O'Boyle and G. A. Hedayat. Load balancing of parallel affine loops by unimodular transformations. Technical Report UMCS-92-1-1, Department of Computer Science, University of Manchester, January 1992.
12. U. Banerjee. *Loop Transformation for Restructuring Compilers*. Kluwer Academic Publishers, Boston, MA, 1993.

13. W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.
14. P. Clauss and V. Loechner. Parametric analysis of polyhedral iteration spaces. In IEEE, editor, *Int. Conf. on Application Specific Array Processors*, Chicago, Illinois, August 1996.
15. C. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. In S. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*. Pennsylvania State University Press, August 1987.
16. S. Carr. *Memory-Hierarchy Management*. PhD thesis, Dept. of Computer Science, Rice University, September 1992.
17. U. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers, Boston, MA, 1994.
18. S. Lundstrom and G. Barnes. A controllable MIMD architectures. In *Proceedings of the 1980 International Conference on Parallel Processing*, St. Charles, IL, August 1980.
19. D. Kuck et al. The effects of program restructuring, algorithm change and architecture choice on program performance. In *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984.
20. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.
21. S. Anik and W. W. Hwu. Executing nested parallel loops on shared-memory multiprocessors. In *Proceedings of the 1992 International Conference on Parallel Processing*, pages III:241–244, Boca Raton, Florida, August 1992.
22. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, CA, 1996.
23. D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.
24. F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on the Principles of Programming Languages*, San Diego, CA, January 1988.
25. C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, 1985.
26. M. Haghighat and C. Polychronopoulos. Symbolic program analysis and optimization for parallelizing compilers. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.
27. N. Koziris, G. Papakonstantinou, and P. Tsanakas. Mapping nested loops onto distributed memory multiprocessors. In *International Conference on Parallel and Distributed Systems*, pages 35–43, December 1997.
28. M. Dion and Y. Robert. Mapping affine loop nests: new results. In *HPCN Europe 1995*, pages 184–189, 1995.
29. M. Dion and Y. Robert. Mapping affine loop nests. *Parallel Computing*, 22(10):1373–1397, 1996.
30. Matlab. <http://www.mathworks.com>.
31. Eric W. Weisstein. "Abel's Impossibility Theorem." From MathWorld—A Wolfram Web Resource. <http://mathworld.wolfram.com/AbelsImpossibilityTheorem.html>.