A New Approach to Increase Parallelism for Dependent Loops

Yeong-Sheng Chen, Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan, *Tsang-Ming Jiang*, Arctic Region Supercomputing Center, University of Alaska, Fairbanks, AK 99775, and *Sheng-De Wangj*, Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan

ABSTRACT: *Loops are the primary source of parallelism in parallel processing. Two iterations in a loop are flow dependent if the results computed at one iteration are used by the other. Otherwise, they are independent. Independent iterations in loops can be scheduled in any orders or partitioned to any processors without explicit synchronizations. For dependent iterations, they are partitioned into sets (one or many) such that iterations in different sets are independent (e.g., the minimum distance method). These sets can be executed in parallel without explicit synchronizations. The degree of parallelism is the number of sets. This paper presents a new approach which can significantly increase the parallelism by adding appropriate synchronizations. The implementation feasibility and performance benefits of this approach are demonstrated on the CRAY MPP which has very fast synchronization mechanisms.*

1 Introduction

Loops provide the most fruitful source of parallelism for parallel processing systems. So, it is well recognized that one of the most crucial problems for parallel execution of programs on multiprocessor systems is to partition loops into independent groups so as to exploit parallelism within loops [2-6], [8-14]. Since synchronization overhead among processors will deteriorate the performance, many researches are focused on the development of techniques for partitioning nested loops so that when the loops are executed no synchronization among processors is incurred [3], [8], [12]. Minimum distance method is a typical example. It partitions loops into totally independent sets of computations. Each set of computations is then assigned to a processor. All the sets can be executed in parallel without explicit synchronization.

However, in practical implementation on distributed memory multiprocessor systems, no synchronization among processors does not imply no data communication among processors. If the computations that can be executed in parallel are not aggregated in a regular region of fixed size and shape, then it is very difficult to partition the data so that there is no data communication among processors. Mostly, data distribution can be done only in block or cyclic manner [7]. So, for an arbitrary independent set of computations that are executed in a particular processor, it is very difficult to distribute all the corresponding data to the same processor so that there is no need to access data located in other processors. In other words, although all the computations are partitioned into independent sets, in practice, there will exist data communications among processors.

Observing the above facts, in this paper, we pay our attentions to another important factor which can enhance performance— exploiting more parallelism. We add barrier synchronizations into loops to increase the degree of parallelism. Barrier synchronizations are very efficient in the distributed memory multiprocessor systems that have hardware support for them. For example, Cray T3D MPP has such a mechanism [1]. We propose an method to aggregate as many independent computations as possible into a block for maximizing parallelism. At first, *all* the computations that are independent when the loops just begin their execution are identified. They are called initially independent computations. We show that *all* the initially independent computations can be aggregated into rectangular blocks. So, based on them, the original loops can be partitioned into rectangular blocks to maximize parallelism. The corresponding computations of a block is assigned to different processors for parallel execution. A barrier

synchronization is added in the end of each block, and all blocks are executed sequentially.

In the proposed method, since each partitioned region is block-shaped, the parallel code generation is straightforward and easy. And, it is very likely that when the loops are partitioned into blocks, there is better chance to distribute data into processors so that data communication among different processors is minimized. (Mostly, data distribution can be done only in block or cyclic manner.) Some experiments are conducted on CRAY T3D MPP. As will be discussed in Section Four, it demonstrates the implementation feasibility and performance benefits of our proposed method.

2 Partitioning Dependent Loops

2.1 Basic terminology

The program model considered in this paper is the loops with uniform inter-iteration dependences [2-6, 8-14]. That is, the dependence pattern is the same for each iteration of loops. Such loops are called uniform dependence loops. Like most other related work, a loop iteration is considered as the basic scheduling unit. A loop nest with n levels forms an n-dimensional iteration space, which is a subspace of Z^n bounded by the loop bounds. This space is composed of points each of which represents an iteration of the nested loops. Each point in the iteration space is identified by the index vector $(p_1, p_2, \cdots, p_n)^T$ if its corresponding loop iteration is of loop index $(p_1, p_2, \cdots, p_n)$.

Definition 1. *Uniform dependence loop nests*: A uniform dependence loop nest $L_n(V, D)$ is perfectly nested loops of depth n where

1. V is the set of indexed points each of which represents one loop iteration and is called a *computation*;
2. D is the dependence matrix in which each column is a dependence vector [32] that describes the inter-iteration dependence of the loops;
3. all the components of the dependence vectors are constants;
4. $\forall \boldsymbol{v} \in V$, $\boldsymbol{v}$ takes one unit of time to execute; and
5. $\forall \boldsymbol{v} \in V$, $\boldsymbol{v}$ depends on $\{\boldsymbol{w} | \boldsymbol{w} \in V, \boldsymbol{v}\text{-}\boldsymbol{w}$ is a dependence vector in $D\}$

Without loss of generality, we assume that the loop nest has been normalized. That is, for each loop level i, $1 \le i \le n$, the lower bound of loop index is 0 and the loop index increment is 1.

2.2 Minimum Distance Method

The minimum distance method [3, 8] partitions the iteration space of a loop nest L into independent sets of computations. Let these sets are P_i, $1 \le i \le k$, $k \in N$. The basic idea behind this method is that $\forall p \in P_i$, p must be a linear combination of the dependence vectors of L from an initial origin-point $p_0 \in P_i$. That is, with matrix notation, it can be obtained as

$$P_i = P_{i,0} + A \times D,$$

where A is an integer matrix and D is the dependence matrix. To generate the parallel codes that correspond to the partitioned loops, the dependence matrix D is transformed into an upper triangular matrix D^s so that the "shifting" distance in each dimension can be easily derived. However, the transformation from D to D^s is formulated as a NP problem. D'Hollander [3] improves it by using a direct and inexpensive partitioning algorithm based on unimodular transformations and overrides the drawbacks of it when the dependence vectors are not linearly independent.

Although it is shown that the minimum-distance method can generate the maximal number of independent sets in the unbounded iteration space, the independent computations are not aggregated into regular regions of fixed size and shape. So, the transformed parallel codes cannot be efficiently executed due to data distribution and processor load balance problems. As discussed in the previous section, for an arbitrary set of computations, it is very difficult to partition the corresponding data into the same processor. Thus, although, theoretically, minimum distance method can partition loops into independent sets, in practice, data communications among processors are necessary.

Many researches have shown that the above problems can be relieved by partitioning loops into blocks (tiles) [2, 6, 10, 11, 14]. Tiles become natural units for scheduling parallel tasks. The compiler can easily generate parallel codes for tiles so that processors are load balanced when executing the codes. Moreover, most the existing parallelizing compilers support block or cyclic (block size is one) data distribution [7]. So, in addition to parallelism exploitation, partitioning loops into blocks is also a very important optimization issue for efficiently executing programs in distributed memory parallel systems. In this paper, we will partition loops into rectangular blocks while maximizing parallelism.

3 The Proposed method

3.1 Some definitions

Intuitively, to exploit the optimal (maximal) parallelism within a loop nest is to identify *all* the computations that can be executed in the first time unit, and then *all* those can be executed in the second time unit, ..., and so on. So, identifying all the independent computations at the first time unit provides a good start point for exploiting maximal parallelism.

Definition 2. Initially independent computations, initially independent computation sets: A computation is *initially independent* iff all its real dependences are satisfied when the loops just begin their execution. An *initially independent computation set* (IICS) is an independent computation set in which every computation is initially independent.

An initially independent computation set is *maximal* iff it aggregates *all* the initially independent computations.

Definition 3. *Boundary block computation sets*: A *boundary block computation set* $R=[r_1, r_2, \cdots, r_n]$ of a loop nest L_n is a set of computations where $(1 \le i \le n)$

(a) $r_i \in Z \cup \{\infty\}$; ($\infty$ denotes the unknown loop upper bound.)

(b) a computation $(p_1, p_2, \cdots, p_n)^T$ is in R iff it is *initially independent* and meets the following conditions:

(1) if $r_i<0$ then $u_i+r_i+1 \le p_i \le u_i$, ($u_i$ denotes the upper bound of the ith loop)

(2) if $r_i>0$ then $0 \le p_i \le r_i-1$,

(3) if $r_i=\infty$ then $0 \le p_i \le u_i$; and

(c) if $\exists\ r_i=0$, then R is an empty set;

Note that a boundary block computation set gathers *initially independent* computations *around the boundaries* of the iteration space and is a block-shaped IICS.

Example 3.1. The iteration space of a loop nest L_2 is shown in Fig. 1. Assume that R_1 and R_2 are boundary block computation sets. They are denoted as [∞, 3] and [2, -3], respectively. And, the computation sets corresponding to the regions S_1 and S_2 must not be boundary block computation sets.

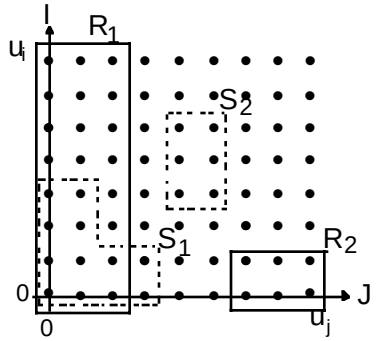

Figure 1: Boundary block computation sets.

Definition 4. *Maximal boundary block computation sets*: Let expand(r_i,1)=r_i+1 if $r_i>0$, and expand(r_i,1)=r_i-1 if $r_i<0$. A boundary block computation set $R=[r_1, r_2, \cdots, r_n]$ is *maximal* iff $\forall\ i, 1 \le i \le n$, $[r_1, \cdots, \text{expand}(r_i, 1), \cdots, r_n]$, which is a proper superset of R, is not an IICS (i.e., it will include a computation that is not initially independent). (Note that $\forall\ i$, $1 \le i \le n$, $r_i \ne 0$, since R is empty if $r_i=0$.)

In other words, Definition 4 says that R is the block-shaped IICS that has been "expanded" as much as possible.

3.2 Formulation of the basic steps

Lemma 3.1. For the loop nest $L_n(V, D)$ which has a dependence vector $\boldsymbol{d_i}=(d_{1i}, d_{2i}, \cdots, d_{ni})^T$,

$R_{d_i}=[d_{1i}, \infty, \infty, \cdots] \cup [\infty, d_{2i}, \infty, \cdots] \cup \cdots \cup [\infty, \infty, \cdots, d_{ni}]$

is an initially independent computation set with respect to $\boldsymbol{d_i}$. ($[d_{1i}, \infty, \infty, \cdots], \cdots, [\infty, \infty, \cdots, d_{ni}]$ are boundary block computation sets defined above.)

Lemma 3.2. A computation $\boldsymbol{v}$ of a loop nest L_n is an initially independent computation with respect to the dependence vector $\boldsymbol{d_i}$ iff $\boldsymbol{v}$ is in R_{d_i}.

From Lemmas 3.1 and 3.2, it is clear that R_{d_i} is the maximal initially independent computation set of L_n with respect to the dependence vector $\boldsymbol{d_i}$.

Theorem 3.1. For a loop nest L_n with m dependence vectors $\boldsymbol{d_i}$, $1 \le i \le m$,

$$R_0 = \bigcap_{i=1}^{m} R_{d_i}$$

is the maximal initially independent computation set of L_n.

The following algorithm derives the maximal initially independent computation set for a given loop nest.

Procedure IICS

Input: L_n with dependence matrix $D=[\boldsymbol{d_1}, \boldsymbol{d_2}, \cdots, \boldsymbol{d_m}]$ ($\boldsymbol{d_i}=(d_{1i}, d_{2i}, \cdots, d_{ni})^T$, $1 \le i \le m$)

Output: R_0

Begin

$R_{d_1} = [d_{11}, \infty, \infty, \cdots] \cup [\infty, d_{21}, \infty, \cdots] \cup \cdots \cup [\infty, \infty, \cdots, d_{n1}]$;

$R_{d_2} = [d_{12}, \infty, \infty, \cdots] \cup [\infty, d_{22}, \infty, \cdots] \cup \cdots \cup [\infty, \infty, \cdots, d_{n2}]$;

...

$R_{d_m} = [d_{1m}, \infty, \infty, \cdots] \cup [\infty, d_{2m}, \infty, \cdots] \cup \cdots \cup [\infty, \infty, \cdots, d_{nm}]$;

$R_0 = R_{d_1} \cap R_{d_2} \cap \cdots \cap R_{d_m}$;

Return(R_0);

End.

Theorem 3.2. The output of Procedure IICS, R_0, is the union of *all* the maximal boundary block computation sets.

Now, we should show that R_0 always contains a maximal boundary block computation set with size greater than one if a proper wavefront transformation is applied to the original loops.

Lemma 3.3. Any uniform dependence loop nest can be transformed (by skewing) into a canonical form— fully permutable loop nest. (A fully permutable loop nest is a loop nest whose dependence vectors have no negative elements.)

Theorem 3.3. Consider a fully permutable loop nest L_n with m dependence vectors $\boldsymbol{d_i}=(d_{1i}, d_{2i}, \cdots, d_{ni})^T$, $1 \le i \le m$. If a proper wavefront transformation is applied to the original loop nest L_n, then $R_t=[\max_{i=1}^{m}(d_{1i}+d_{2i}+\cdots+d_{ni}), \infty, \infty, \cdots] \subset R_0$, where R_0 is the maximal IICS derived from Procedure IICS.

3.3 An illustrative example

We want to exploit parallelism within loops by partitioning them into blocks so that all the blocks are identical by translation and each of them aggregates as many independent computations as possible. To facilitate the presentation, we define the following term.

Definition 5. *Valid independent computation sets*: For a loop nest L, an independent computation set (ICS) R is *valid* iff L can be partitioned into blocks so that each of them is disjunctive, atomic and identical by translation to R [4]. We call this situation "*tiling L with R*." Being disjunctive, each computation is executed exactly only once; and, being atomic, no two blocks are

cyclically dependent on each other and therefore can be scheduled without violating dependences [8]. (*R* is said to be *invalid* if it is not valid.)

For a loop nest, if it can be tiled with an ICS *R*, clearly, *R* must be a subset of the maximal initially independent computation set R_0. From Theorem 3.2, we see that R_0 is the union of all the maximal boundary block computation sets. So, among them, we can choose the maximal boundary block computation set that is valid and has the largest size to tile the original loops for maximizing parallelism.

We describe how to tile the loops with a valid ICS by going through the following example.

Example 3.2. Consider the nested loops $L_3(V, D)$ with $V=\{(i, j, k)^T | 0\le i, j, k\le\infty\}$ and $D=\begin{bmatrix} 1 & 2 & 0 \\ -2 & 4 & 4 \\ 4 & -1 & 2 \end{bmatrix}$. With Procedure IICS, R_0 $=([1, \infty, \infty]\cup[\infty, -2, \infty]\cup[\infty, \infty, 4])\cap([2, \infty, \infty]\cup[\infty, 4, \infty]\cup[\infty, \infty, -1])\cap([\infty, 4, \infty]\cup[\infty, \infty, 2]) = [1, 4, \infty]\cup[\infty, 4, 4]\cup[2, \infty, 2]$. We intend to tile the loops with a valid independent computation set R_t. Since [1, 4, ∞], [∞, 4, 4] and [2, ∞, 2] are all the maximal boundary block computation sets, R_t can be [1, 4, ∞], [∞, 4, 4] or [2, ∞, 2]. Each case is discussed as follows:

1. R_t = [1, 4, ∞]: Let the loop nest be tiled with [1, 4, ∞]. As shown in Fig. 2(a), this is to strip-mine dimensions 1 and 2 with strip lengths 1 and 4, respectively. (Dimension 3 is not strip-mined.) After tiling the loops, since all blocks are identical by translation, each of them can be represented by an arbitrary point within it (the so-called tile origin) [1]. Thus, the tile origins define a lattice (see Fig. 2(b)). It is easy to see that, in this lattice, the "new" dependence vectors are $\boldsymbol{e}_1^1=(1, 0, 0)^T$, $\boldsymbol{e}_1^2=(1, -1, 0)^T$, $\boldsymbol{e}_2=(2, 1, 0)^T$, and $\boldsymbol{e}_3=(0, 1, 0)^T$. In the sense that tiling is a transformation of the original loops, we consider $\boldsymbol{e}_1^1$, $\boldsymbol{e}_1^2$, $\boldsymbol{e}_2$, and $\boldsymbol{e}_3$ the transformed dependence vectors. Since all the transformed dependence vectors are lexicographically positive [13], such a tiling is valid.

   We should explain the notation $\boldsymbol{e}$ that we are using: When the original loops are tiled with $R_t=[r_1, r_2, \cdots, r_n]$, $\boldsymbol{e}_i=(e_{1i}, e_{2i}, \cdots, e_{ni})^T$ represents the dependence vector in the lattice of tile origins that corresponds to the original dependence vector $\boldsymbol{d}_i=(d_{1i}, d_{2i}, \cdots, d_{ni})^T$ ($1\le i\le m$, *m* is the number of dependence vectors). For convenience, we may call $\boldsymbol{e}_i$ the corresponding *block dependence vector* of $\boldsymbol{d}_i$.

2. (2) R_t = [∞, 4, 4]: Tiling the loops with [∞, 4, 4] is strip-mining dimensions 2 and 3 both with strip length 4. (Dimension 1 is not strip-mined.) After tiling, the block dependence vectors are $\boldsymbol{e}_1^1=(0, -1, 1)^T$, $\boldsymbol{e}_1^2=(0, 0, 1)^T$, $\boldsymbol{e}_2^1=(0, 1, -1)^T$, $\boldsymbol{e}_2^2=(0, 1, 0)^T$ and $\boldsymbol{e}_3=(0, 1, 1)^T$. Since $\boldsymbol{e}_1^1=-\boldsymbol{e}_2^1$, two blocks are cyclically dependent on each other. This is the so-called dead-locked condition [3]. So, the loops cannot be tiled with [∞, 4, 4]. In fact, such a tiling is invalid because not all $\boldsymbol{e}$'s are lexicographically positive. (Lexicographically positive dependence vectors will never cause dead-lock.)

3. (3) R_t = [2, ∞, 2]: As the same reasoning in (1), the loop nest can be tiled with [2, ∞, 2].

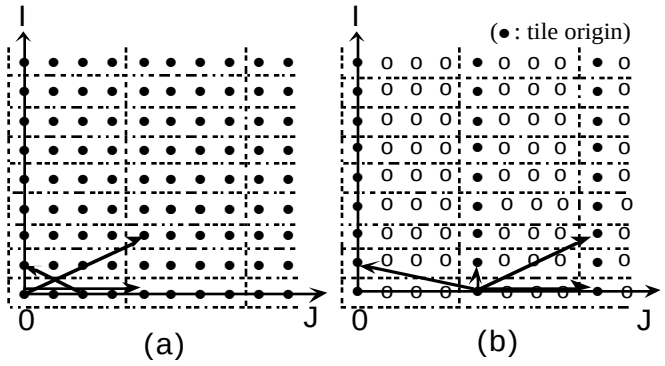

Figure 2: Illustration of Examples 3.2

So, we have R_t = [1, 4, ∞] or R_t = [2, ∞, 2]. (They have the same number of computations, 4×∞. ∞ denotes the unknown loop upper bound). With R_t =[1, 4, ∞], the original loops can be partitioned into the following parallel form. (On the other hand, one may let R_t =[2, ∞, 2]. Since it has the same degree of parallelism, we omit the discussion of it.)

```
Do 200 SI=0, Ui-1
 Do 200 SJ=0, ⌈Uj/4⌉-1
  DoAll 100 I= SI,SI
   DoAll 100 J= SJ×4, min(SJ×4+3,Uj)
    DoAll 100 K= 1, Uk
      Loop Body;
100  Continue
      Barrier();
200  Continue
```

In contrast, with the minimum distance method, the "shifting distance" in each dimension is 1, 4, and 13, respectively [3], [8]. Hence, the degree of parallelism is limited to 52. The original loops are partitioned into the following parallel form. (Note that to conform to the original papers, in the following loops, the lower bound of every loop index is 1.)

```
DoAll 100 I0=1,1
 DoAll 100 J0=1,4
  DoAll 100 K0=1,13
   Do 100 I=I0, Ui,1
    Y1=(I-I0)/1
    Jmin=1+(J0-1-2*Y1) mod 4
    Do 100 J= Jmin, Uj, 4
```

```
        Y2=(J-J0+2*Y1)/4
        Kmin=1+(K0-1+4*Y1+2*Y2) mod 13
       Do 100 K=Kmin,U_k,13
        Loop Body;
100   Continue
```

Obviously, in the above codes, there are index calculation overheads. And, each independent set is not in a region of regular shape and size. So, in practical implementation, it is very difficult to optimize the data distribution among processors.

4 Experimental Results

We conduct some experiments on CRAY T3D. The test loops are modified from the example given in [8] whose dependence matrix is the same as that of Example 3.2. As discussed in Example 3.2, the loops are partitioned into blocks whose size is 4×∞ (∞ denotes the loop upper bound). That is, at most, the loops can be executed in parallel on 4×∞ processors. However, with minimum distance method, the "shifting distances" [8] in the first, the second and the third dimensions are 1, 4, and 13, respectively. That is, there are at most 52 processors that can execute in parallel. So, the method proposed in this paper has much more parallelism, but it requires a global synchronization in each block of loops. Although the minimum distance method produces complete independent sets of computations (no synchronization is required), its parallelism is limited.

Table 1 shows the time results of some experiments on the CRAY T3D. All time are shown in clocks, which is about 6.7 nsec. In Table 1, "Private" means that data is not shared among processors, and "(:, :block(4), :block(1))" stands for the data distribution pattern: the data array is not divided in the first dimension, is divided with block size 4 in the second dimension, and is cyclically divided in the third dimension.

From the results, we see that the proposed method can run faster than minimum distance method when the processor number is small (1, 2, 4, 8 and 16). This is due to the code simplicity and low barrier overhead. As the number of processors increases, the barrier overhead increases, and so the proposed method loses its advantage. However, with minimum distance method, the degree of parallelism is limited (52 in this example). That is to say, it cannot take advantage of massively parallel processors, which have hundreds or thousands of processors. In contrast, with the proposed method, the degree of parallelism is 4×N (N is the loop bound), which is usually very large. So, its execution time should be bounded by the size of the machine as long as the barrier overhead is low and the problem size is large enough. As shown in Table 1, the proposed method runs faster again at 128 processors due the degree of parallelism.

5 Conclusions

For efficiently executing loops on distributed memory multiprocessor system, some existing methods partition loops into totally independent sets of computations [3, 8, 12]. However, for an arbitrary independent set of computations that are executed in a particular processor, it is very difficult to distribute all the corresponding data to the same processor so that there is no need to access data located in other processors. So, although all the computations are partitioned into independent sets, in practice, there will exist data communications among processors.

Table 1. Some experimental results on T3D

No. of PEs	private		(:, :block(4), :bloc	
	MD	II	MD	II
1	24236361	27671122	30559652	22119819
2	12166284	9150964	5731319	3110879065
4	6080896	5427779	2841515	1388120162
8	3274001	3212839	1562074	636534041
16	1871366	1874381	1063544	722313588
32	939187	698507	791766	389817765
64	468233	260243	578960	333147239
128	468272	843385	078874	469 84045

(II: the proposed method, MD: minimum distance method)

In this paper, we focus to exploiting more parallelism, which is also an important factor for enhancing performance. The proposed method can increase parallelism within loops by adding appropriate barrier synchronizations. The original loops are partitioned into rectangular blocks. Each block aggregates as many independent computations as possible for maximizing parallelism. Since each partitioned region is block-shaped, the parallel code generation is straightforward and easy. With the proposed method, the degree of parallelism usually is a function of loop bounds, and therefore can be very large. Thus, it can take advantage of massively parallel processor systems, which have hundreds or thousands of processors. And, its execution time should be only bounded by the size of the machine as long as the barrier overhead is low and the problem size is large enough. The experiments conducted on CRAY T3D MPP, which has very fast barrier synchronization, shows the implementation feasibility and performance benefits of our proposed approach.

6 Acknowledgments

The authors would like to thank the Arctic Region Supercomputing Center for access its CRAY T3D. The project was funded in part by the University of Alaska and the National Science Council in Taiwan.

7 References

[1] "Cray T3D System Architecture Overview Manual," Cray Research , Inc., HR-04033, 1993

[2] Y. S. Chen, S. D. Wang, and C. M. Wang, "Compiler Techniques for Maximizing Fine-grain and Coarse-grain Parallelism in Loops with Uniform Dependences," in *Proc. 8th ACM Conf. Supercomputing*, Jul. 1994, pp. 204-213.

[3] E. H. D'Hollander, "Partitioning and labeling of loops by unimodular transformations," *IEEE Trans. Parallel Distributed Syst.* Vol. 3, No. 4, pp. 465-476, July 1992.

[4] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proc. 15th annual ACM SIGACT-SIGPLAN Symposium on Principles Programming Languages*, Jan. 1988, pp. 319-329.

[5] C. King and L. M. Ni, "Grouping in nested loops for parallel execution on multicomputers," in *Proc. Int. Conf. Parallel Processing*, Vol. II, 1989, pp. 31-38.

[6] A. Nicolau, "Loop quantization: A generalized loop unwinding technique," *J. Parallel Distributed Comput.*, Vol. 5, No. 10, pp. 568-586, 1988.

[7] D. M. Pase, T. MacDonald, A. Meltzer, "MPP Fortran Programming Model," Cray Research, Inc., SN-2513, Feb. 1994.

[8] J.-K. Peir and R. Cytron, "Minimum Distance: A method for partitioning recurrences for multiprocessors," *IEEE Trans. Comput.*, Vol. 38, No. 8, pp. 1203-1211, Aug. 1989.

[9] C. D. Polychronopoulos, "Compiler optimizations for enhancing parallelism and their impact on architecture design," *IEEE Trans. Comput.* Vol. 27, No. 8, pp. 991-1004, Aug. 1988.

[10] J. Ramanujam and P. Sadayappan, "Tiling multidimension-al iteration space for multicomputers," *J. Parallel Distributed Comput.*, Vol. 16, No. 2, pp. 108-120, Oct. 1992.

[11] R. Schreiber and J. Dongarra, "Automatic blocking of nested loops," Technical Report CS-90-108, Univ. of Tennessee, Knoxville, TN, 1990.

[12] W. Shang and J. A. B. Fortes, "Independent partitioning of algorithms with uniform dependencies," *IEEE Trans. Comput.*, Vol. 41, No. 2, pp. 190-206, Feb. 1992.

[13] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Trans. Parallel Distributed Syst.*, Vol. 2, No. 4, pp. 452-471, Oct. 1991.

[14] M. Wolfe, "More iteration space tiling," in *Proc. Supercomputing '89*, Nov. 1989.