Chain-Based Scheduling: Part I – Loop Transformations and Code Generation

Peiyi Tang

July 21, 1992

Chain-Based Scheduling: Part I – Loop Transformations and Code Generation[*]

Peiyi Tang

Department of Computer Science
The Australian National University
Canberra ACT 2601 Australia

July 21, 1992

Abstract

Chain-based scheduling [1] is an efficient partitioning and scheduling scheme for nested loops on distributed-memory multicomputers. The idea is to take advantage of the regular data dependence structure of a nested loop to overlap and pipeline the communication and computation.

Most partitioning and scheduling algorithms proposed for nested loops on multicomputers [1,2,3] are graph algorithms on the iteration space of the nested loop. The graph algorithms for partitioning and scheduling are too expensive (at least $O(N)$, where N is the total number of iterations) to be implemented in parallelizing compilers. Graph algorithms also need large data structures to store the result of the partitioning and scheduling.

In this paper, we propose compiler loop transformations and the code generation to generate chain-based parallel codes for nested loops on multicomputers. The cost of the loop transformations is $O(nd)$, where n is the number of nesting loops and d is the number of data dependences. Both n and d are very small in real programs. The loop transformations and code generation for chain-based partitioning and scheduling enable parallelizing compilers to generate parallel codes which contain all partitioning and scheduling information that the parallel processors need at run time.

[*]This work is supported in part by **the Australian Research Council** under Grant No. S6600132 and by the donation of Fujitsu Laboratories LTD.

Keywords: nested loops, partitioning, scheduling, multicomputers, loop transformations, chain-based scheduling.

1 Introduction

The chain-based partitioning and scheduling [1] is an efficient scheme to execute nested loops with constant data dependences on distributed address space multicomputers. The idea is to take advantage of the regular data dependence structure to partition and schedule loop iterations in such a way that communication and computation can be overlapped and pipelined.

Due to the large context switch overhead of operating systems, partitioning and scheduling at the loop iteration level of a program need to be managed by compilers. Like most other partitioning and scheduling algorithms proposed for nested loops on multicomputers[2,3], the chain-based scheduling algorithm [1] is originally a graph algorithm on the iteration space of the nested loop. The cost of the graph-based algorithms is at least $O(N)$, where N is the total number of loop iterations. Note that the cost of the nested loop itself is also $O(N)$. In other words, the time for the compiler to run a graph-based partitioning and scheduling algorithm is at least as much as the sequential execution time of the nested loop itself. More importantly, graph-based partitioning and scheduling algorithms need a large data structure to store the result of the scheduling for each processor. The time cost of retrieving these large data structures from the disk at run time will be prohibitively high.

The solution to the problems of graph-based partitioning and scheduling lies in compiler program transformations. In order to be practically useful in massively parallel systems, any partitioning and scheduling scheme has to be incorporated in program transformations. For nested loops, the most time-consuming parts of numerical scientific programs, the partitioning and scheduling as well as the communication primitives generation should be done through loop transformations.

In this report, we present a series of loop transformations and code generation for chain-based partitioning and scheduling. A parallelizing compiler can use these transformations to generate SPMD (Single Program Multiple Data) style parallel programs, in which all the partitioning and scheduling information are incorporated. When the generated programs are executed by the parallel processors, the computation and the communication are overlapped and pipelined.

In section 2, the program model of nested loop with constant data dependences is introduced. Section 3 presents the program transformations for chain-based partitioning and scheduling. They include loop skewing, loop tiling, loop normalization and chain-based code generation. Section 4 concludes the paper with a brief discussion of performance of the generated chain-based parallel codes.

2 Nested Loops and Constant data dependences

Nested loops consume most of the CPU time in scientific and engineering supercomputing. To speed up the execution of nested loops on massively parallel computers like Intel Touchstone Delta System and Fujitsu AP1000, the computation needs to be parallelized, partitioned and scheduled to parallel processors.

```
DO $i_1 = L_1, U_1$
    ⋮
      DO $i_n = L_n, U_n$
          $B(i_1, \cdots, i_n)$
      ENDDO
    ⋮
ENDDO
```

Figure 1. Program model of uniform recurrence

In this paper, we concentrate on a class of nested loops known as *uniform recurrences.* It covers many important classes of scientific computations including iterative methods for linear systems to solve partial differential equations.

A uniform recurrence is a perfectly-nested loop with constant data dependences between the loop iterations. Figure 1 shows the program model of a uniform recurrence. Each loop bound, L_k or U_k, can be a linear function of the indices of the surrounding loops, i.e.

$$\begin{aligned} L_k &= a_{k,0} + a_{k,1}i_1 + \cdots + a_{k,k-1}i_{k-1} \\ U_k &= b_{k,0} + b_{k,1}i_1 + \cdots + b_{k,k-1}i_{k-1} \end{aligned}$$

The loop body of the nested loop, denoted as $B(i_1, \cdots, i_n)$ in Figure 1, is a sequence of assignments without IF or EXIT statements. The nested loop in Figure 1 defines an iteration space $I \subset Z^n$:

$$I = \{(i_1, \cdots, i_n) \in Z^n : L_k \leq i_k \leq U_k, k = 1, \cdots, n\}$$

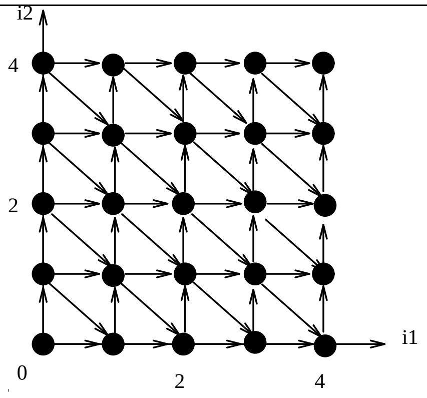
A data dependence exists from iteration $\vec{i} = (i_1, \cdots, i_n)$ and to iteration $\vec{i\prime} = (i\prime_1, \cdots, i\prime_n)$ if

1. $\vec{i} \neq \vec{i\prime}$ and $\vec{i}$ executes before $\vec{i\prime}$ in the temporal order of the nested loop, and
2. both iterations $\vec{i}$ and $\vec{i\prime}$ access the same scalar or array element and at least one of the accesses is write, and
3. there is no other iteration between $\vec{i}$ and $\vec{i\prime}$ that writes to the same data element.

If $\vec{i}$ writes and $\vec{i\prime}$ reads the data element, the data dependence is *true* dependence, because it reflects a data flow between the two iterations. If $\vec{i}$ reads and $\vec{i\prime}$ writes the data element, the data dependence is called *anti* dependence. If both $\vec{i}$ and $\vec{i\prime}$ writes the data element, it is called *output* dependence. Anti and output dependences are caused by reuse of memory

```
do $i_1 = 0, 4$
    do $i_2 = 0, 4$
        $a(i_1, i_2) = f_1(c(i_1, i_2 - 1), b(i_1, i_2));$
        $b(i_1, i_2) = f_2(a(i_1 - 1, i_2 + 1), c(i_1, i_2));$
        $c(i_1, i_2) = f_3(b(i_1 - 1, i_2), a(i_1, i_2));$
    enddo;
enddo;
```

(a) Program



(b) Iteration space

Figure 2. An Example of Nested Loop

storage and, therefore, are not essential. They can be eliminated by scalar renaming and scalar or array expansion. For this reason, we consider only flow data dependences in this paper, assuming that all the anti and output data dependences in original programs, if any, have been eliminated by renaming and expansion.

If there is a data dependence from $\vec{i}$ to $\vec{i\prime}$, vector $\vec{d} = \vec{i\prime} - \vec{i}$ is called *distance vector* of the dependence. A distance vector is always lexicographically positive, i.e., the first non-zero element is positive and not all elements are zeros due to the fact that $\vec{i\prime}$ is later than $\vec{i}$. The dependence is *constant* if the distance of the dependence is independent of the source, i.e., it exists between every pair of $\vec{i}, \vec{i\prime} \in I$ such that $\vec{i} + \vec{d} = \vec{i\prime}$. A *uniform recurrence* is a perfectly nested loop whose data dependences are all constant. Figure 2 shows an example of a uniform recurrence and its iteration space with data dependences. There are three constant data dependences with distance vectors $\vec{d_1} = (1, -1)$, $\vec{d_2} = (1, 0)$ and $\vec{d_3} = (0, 1)$ associated with arrays a, b and c, respectively, all of which are flow dependences. The data dependences of a uniform recurrence can be represented by the set of distance vectors, $D = \{\vec{d_1}, \cdots, \vec{d_m}\}$. If the rank of D, r, is less than n, the number of loops in the nested loop, we can transform the nested loop to a new nested loop of n loops in which the $n-r$ outermost loops are *DOALL* loops [4]. A DOALL loop is a parallel loop without data dependences across its iterations. Outer DOALL loops can be efficiently executed on multicomputers without interprocessor communication [5]. If the rank of D is equal to n, the outmost loop cannot be transformed to a DOALL loop. In this paper, we concentrate on this kind of nested loops. Using the hyperplan method [4,6,7], such a nested loop can be transformed to a nested loop with n loops in which the outmost loop is a sequential loop and all the inner $n-1$ loops are all DOALL loops. However, the overhead of the fork-join synchronization for the outmost sequential loop could be very large. In this paper, a different parallelization approach is used. We use interprocessor communication to enforce the data dependences if the related iterations are allocated to difference processors. A parallel loop with dependences between iterations enforced by synchronization (as in shared address space multiprocessors) or communication (as in

Algorithm 1 (Skewing Matrix)

```
inputs:
    $D$: set of dependence vectors;
outputs:
    $E$: set of dependence vectors;
    $T$: unimodular matrix;

T = identity matrix;
for i = 2 to n do
  A = identity matrix;
  for each $\vec{d} \in D$ do
    if $\sum_{j=1}^{i-1} a_{ij} d_j + d_i < 0$ then
      let $k$ be the smallest in $\{1, \cdots, i-1\}$
        such that $d_k > 0$;
      $a_{ik} = \max(a_{ik}, \lceil (-d_i - \sum_{j=k+1}^{i-1} a_{ij} d_j)/d_k \rceil)$;
    endif;
  endfor;
  for each $\vec{d} \in D$ do
      $d_i = \sum_{j=1}^{i} a_{ij} d_j$;
  endfor;
  T = AT;
endfor;
return(D, T);
```

distributed address space multicomputers) is called *DOACROSS* loop. The chain-based partitioning and scheduling spread iterations of DOACROSS loops across parallel processors and use communication to enforce data dependences between the loop iterations.

3 Program Transformations for Chain-Based Scheduling

In this section, we present program transformations to generate chain-based parallel programs for nested loops on multicomputers. The transformations are: (1) loop skewing, (2) loop tiling, (3) loop normalization and (4) chain-based partitioning and scheduling.

3.1 Loop Skewing

The purpose of the loop skewing, as the first step of the series of transformations, is twofold:

1. to facilitate the transformation for loop tiling;

2. to guarantee deadlock-free execution of tiles.

The time to pass a message of n data (integer or floating-point number) from one processor to another in a multicomputer can be expressed as

$$T_n = W + n\tau \tag{1}$$

where W is the startup delay of the message and τ is the data transfer rate. In current multicomputers, W is usually two orders of magnitude longer than τ. It would be extremely inefficient to send messages with a single datum. Therefore, the computation of iterations of the nested loop need to be grouped so that the data to be passed between processors can be aggregated to form larger messages. Each group is called a *tile* and it usually is a set of neighbouring iterations in the iteration space. The computation of a tile is atomic with respect to message passing. In other words, the processor receives all the data it needs before it starts the computation of the tile. It does not sends data to other processors until the computation of the tile is finished.

Irigoin and Triolet suggested to use hyperplanes to tile the iteration space [7]. Given an iteration space $I \subset Z^n$, a set of hyperplane families

$$H = \{(\vec{h_1}, s_1), \cdots, (\vec{h_n}, s_n)\}$$

can be used to partition the iteration space into tiles. In particular, each $(\vec{h_k}, s_k) \in H$ is a familay of parallel hyperplanes with the norm $\vec{h_k}$ and s_k is the distance between them. A tile is the set of iterations between the parallel hyperplanes defined by H, i.e. a tile with index $(q_1, \cdots, q_n)$ is

$$\{\vec{i} \in I : \lfloor \frac{\vec{h_k} \cdot \vec{i}}{s_k} \rfloor = q_k, k = 1, \cdots, n\}$$

Since tiles are atomic with respect to message passings, deadlocks are possible [7]. To prevent deadlocks, the following condition is sufficient:

$$\forall i \in \{1, \cdots, n\}, \forall j \in \{1, \cdots, m\} : \vec{h_i} \cdot \vec{d_j} \geq 0$$

Tiling with hyperplanes with arbitrary norms is too general to be used by a parallelizing compiler to generate simple and efficient parallel codes. After loop skewing transformation, we can use orthogonal hyperplane families to tile the transformed iteration space to rectangular tiles. Compilers can generate simple and deadlock-fee parallel codes for the rectangular tiles as will be seen shortly

```
DO $j_1 = j_1^{\min}, j_1^{\max}$
⋮
DO $j_n = G_n, H_n$
    $B'(j_1, \cdots, j_n)$
ENDDO
⋮
ENDDO
```

Figure 3. Nested loop after skewing

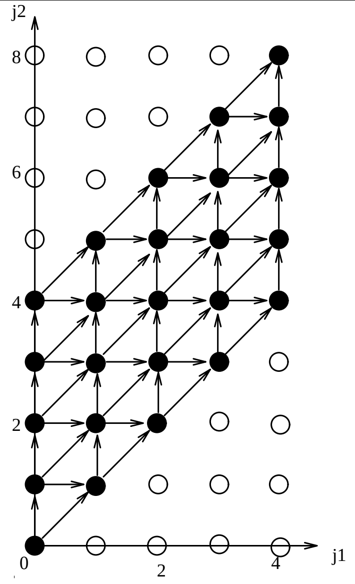
Loop skewing is a loop transformation originally used to improve vectorization [8]. Loop skewing and other two loop transformation, loop interchange and loop reversing, have been recently unified into a single loop transformation called *unimodular* transformation [4,9]. An $n \times n$ integer matrix T is unimodular if its determinant is ± 1, i.e. $|\det(T)| = 1$. A unimodular matrix T transforms each index vector $\vec{i} \in I$ to an index vector in a new iteration space $J = \{\vec{j} \in Z^n : \exists \vec{i} \in I \text{ such that } T\vec{i} = \vec{j}\}$[1]. Under a unimodular transformation, the new iteration space J is the set of index vectors $\vec{j} = (j_1, \cdots, j_n)$ confined in a convex hull determined by the new loop bounds. The methods of calculating these new loop bounds can be found in [4,10]. Since $|\det(T)| = 1$ and the inverse of T is still an integer matrix. Therefore, for each $\vec{j}$ within the loop bounds, there is an $\vec{i} \in I$ such that $\vec{i} = T^{-1}\vec{j}$. In other words, there is a one-to-one mapping between the two iteration spaces and there is no "holes" in iteration space J. The unimodular matrix T also transforms each dependence distance vector $\vec{d} \in D$ to a new distance vector $T\vec{d}$. A unimodular transformation T is legal if and only if for all $\vec{d} \in D$, $T\vec{d}$ is lexicographically positive. Under a legal unimodular unimodular transformation, all of the data dependences will be satisfied when the iterations in iteration space J are traversed in the lexicographic order.

In this paper, loop skewing is used to transform a nested loop to be *fully-permutable*. A nested loop is fully-permutable if all elements of every dependence vector are non-negative. Given a nested loop with the arbitrary data dependences represented by D, we want to find a unimodular matrix T such that each element of $(e_1, \cdots, e_n) = \vec{e} = T\vec{d}$ for all $\vec{d} \in D$ is non-negative. Algorithm 1 shows the algorithm to construct such unimodular matrix T. In Algorithm 1, T is constructed as a product of sequence of $n-1$ unimodular matrices $A_j (i = n, \cdots, 2)$, $T = A_n A_{n-1} \cdots A_2$. Define T_i $(2 \le j \le n)$ as $A_i T_{i-1}$ and T_1 is the identity matrix. It can be seen from Algorithm 1 that, for all $\vec{d} \in D$, the first

[1]Here, both $\vec{i}$ and $\vec{j}$ are column vectors. In this paper, we use comma-separated tuples to denote both column and row vectors and their usage can be figured out from the context of the formulae.

```
do $j_1 = 0, 4$
do $j_2 = j_1, j_1 + 4$
  $a(j_1, j_2 - j_1) =$
        $f_1(c(j_1, j_2 - j_1 - 1), b(j_1, j_2 - j_1));$
  $b(j_1, j_2 - j_1) =$
        $f_2(a(j_1 - 1, j_2 - j_1 + 1), c(j_1, j_2 - j_1));$
  $c(j_1, j_2 - j_1) =$
        $f_3(b(j_1 - 1, j_2 - j_1), a(j_1, j_2 - j_1));$
enddo;
enddo;
```

(a) Program



(b) Skewed iteration space

Figure 4. The example after loop skewing

i elements of $T_i\vec{d}$ are non-negative. T_n gives the matrix T required. In Algorithm 1, the elements on the diagonal of the matrix A_i are all 1. All the other elements are 0, except the first $i-1$ elements of the i-th row, $a_{ij}, 1 \leq j \leq i-1$. Assume that the first $i-1$ elements of each $\vec{d^{i-1}} = T_{i-1}\vec{d} : \vec{d} \in D$ are non-negative, after the $(i-1)$-th iteration. In iteration i, the elements, $a_{ij}, 1 \leq j \leq i-1$, of A_i are determined to make $\sum_{j=1}^{i-1} a_{ij}d_j^{i-1} + d_i^{i-1} \geq 0$ for every $\vec{d^{i-1}} = T_{i-1}\vec{d} : \vec{d} \in D$. Note that $a_{ik}, 1 \leq k \leq i-1$ never decrease and all $d_1^{i-1}, \cdots, d_{i-1}^{i-1}$ are non-negative. It is clear that when $\vec{d^{i-1}} = T_{i-1}\vec{d}$ are processed one at a time in the inner loop of the algorithm, the work done for previous vectors can never be undone. Since The first element of every $\vec{d^1} = T_1\vec{d} = \vec{d} : \vec{d} \in D$ is always non-negative, the above inductive reasoning leads to the correctness of the algorithm.

Once the skewing matrix T is obtained, the nested loop in Figure 1 can be transformed to the nested loop shown in Figure 3. The transformation consists of two parts:

1. transformation of the loop body;

2. transformation of the loop bounds;

The transformation of the loop body is simple. Since $\vec{i} = T^{-1}\vec{j}$, replacing each i_k, $k = 1, \cdots, n$ with the corresponding expression of $j_1, \cdots, j_n$ in the loop body produces the now loop body $B\prime(j_1, \cdots, j_n)$. The new loop bounds can be obtained by the method described in [4]. The low and upper bounds of the first loop is the minimum and maximum possible values of j_1, $j_1^{\min}$ and $j_1^{\max}$, respectively. The loop bounds of the other loops are

```
DO $k_1 = j_1^{\min}, j_1^{\max}, s_1$
⋮
DO $k_n = j_n^{\min}, j_n^{\max}, s_n$
    DO $j_1 = k_1, \min(k_1 + s_1 - 1, j_1^{\max})$
    ⋮
    DO $j_n = \max(k_n, G_n^1, G_n^2, \cdots)$,
            $\min(k_n + s_n - 1, H_n^1, H_n^2, \cdots)$
        $B\prime(j_1, \cdots, j_n)$
    ENDDO
    ⋮
    ENDDO
ENDDO
⋮
ENDDO
```

Figure 5. Nested loop after tiling

```
DO $k_1 = 0, \lceil (j_1^{\max} - j_1^{\min} + 1)/s_1 \rceil - 1$
⋮
DO $k_n = 0, \lceil (j_n^{\max} - j_n^{\min} + 1)/s_n \rceil - 1$
    DO $j_1 = j_1^{\min} + k_1 s_1$,
            $\min(j_1^{\min} + k_1 s_1 + s_1 - 1, j_1^{\max})$
    ⋮
    DO $j_n = \max(j_n^{\min} + k_n s_n, G_n^1, G_n^2, \cdots)$,
            $\min(j_n^{\min} + k_n s_n + s_n - 1, H_n^1, H_n^2, \cdots)$
        $B\prime(j_1, \cdots, j_n)$
    ENDDO
    ⋮
    ENDDO
ENDDO
⋮
ENDDO
```

Figure 6. Nested loop after normalization

of the form $(k = 2, \cdots, n)$:

$$G_k = \max(G_k^1, G_k^2, \cdots)$$
$$H_k = \max(H_k^1, H_k^2, \cdots)$$

where each G_k^p or H_k^p is a linear functions of the outer loop index variables, $j_1, \cdots, j_{k-1}$ as follows:

$$G_k^p = \lceil (g_{k,0}^p + \sum_{i=1}^{k-1} g_{k,i}^p j_i)/g_{k,k}^p \rceil$$
$$H_k^p = \lfloor (h_{k,0}^p + \sum_{i=1}^{k-1} h_{k,i}^p j_i)/h_{k,k}^p \rfloor$$

The unimodular matrix for skewing the nested loop in Figure 2 is

$$T = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$$

The dependence vectors become $\vec{e_1} = (1,0)$, $\vec{e_2} = (1,1)$ and $\vec{e_1} = (0,1)$. The skewed nested loop and its iteration space with data dependences are shown in Figure 4.

3.2 Loop Tiling

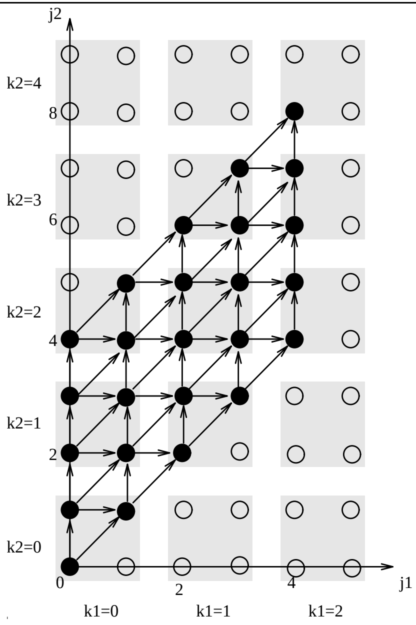
Let E the set of transformed dependence vectors, $E = \{T\vec{d} : \vec{d} \in D\}$ and we have $\vec{e} \geq 0$[2] for all $\vec{e} \in E$. Now we can tile iteration space J with the set of orthogonal hyperplane families:

$$H = \{(\vec{h_1}, s_1), \cdots, (\vec{h_n}, s_n)\}$$

[2] $\vec{e} \geq 0$ if each element of $\vec{e}$ is greater than or equal to 0.

```
do $k_1 = 0, 2$
do $k_2 = 0, 4$
  do $j_1 = 2k_1, \min(2k_1 + 1, 4)$
  do $j_2 = \max(2k_2, j_1), \min(2k_2 + 1, j_1 + 4)$
    $a(j_1, j_2 - j_1) =$
      $f_1(c(j_1, j_2 - j_1 - 1), b(j_1, j_2 - j_1));$
    $b(j_1, j_2 - j_1) =$
      $f_2(a(j_1 - 1, j_2 - j_1 + 1), c(j_1, j_2 - j_1));$
    $c(j_1, j_2 - j_1) =$
      $f_3(b(j_1 - 1, j_2 - j_1), a(j_1, j_2 - j_1));$
  enddo;
  enddo;
enddo;
enddo;
```

(a) Program



(b) Tiled iteration space

Figure 7. The example after tiling and normalization

where

$$\vec{h_1} = (1, 0, \cdots, 0)$$
$$\vec{h_2} = (0, 1, \cdots, 0)$$
$$\vdots$$
$$\vec{h_n} = (0, 0, \cdots, 1)$$

and $\vec{s} = (s_1, \cdots, s_n)$ is called *size vector* of tiling. For every $\vec{e} = (e_1, \cdots, e_n) \in E$, $\vec{h}_k \cdot \vec{e} = e_k \geq 0$, $k = 1, \cdots, n$. Therefore, the tiling is legal and deadlock-free.

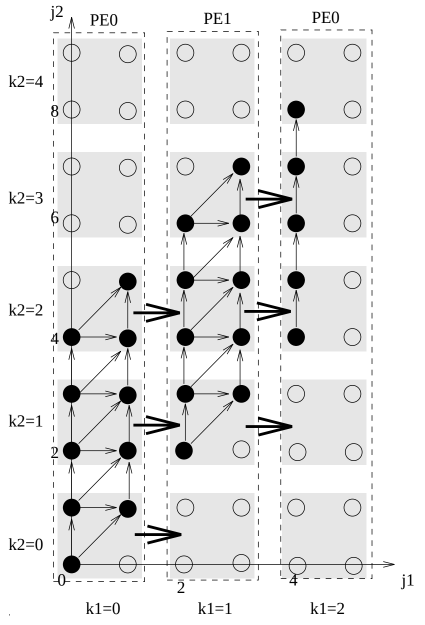
The loop transformation of the orthogonal tiling above is straightforward. We only need to introduce n additional loops to control the execution of the tiles as shown in Figure 5. These new k-loops for tiles are called *controlling loops*. The execution of iterations within a tile is still controlled by the j-loops. In Figure 5, the lower and upper bounds of look k_l, $(l = 1, \cdots, n)$ are $j_l^{\min}$ and $j_l^{\max}$, the minimum and the maximum possible values of j_l in iteration space J, respectively. The method to determine the lower and upper bounds of controlling k-loops as well as inner j-loops can be found in [4].

To facilitate the chain-based partitioning and scheduling, the controlling k-loops need to be normalized [11]. The normalized controlling loops are shown in Figure 6. Notice that normalizing the controlling loops does not affect the loop body $B\prime(j_1, \cdots, j_n)$, because the index variables of the controlling loops are only used in the loop bounds of the inner j-loops.

After tiling and normalization, the nested loop of the example and its tile space are

```
do $k_1 = pid, 2, 2$
do $k_2 = 0, 4$
  receive message from $PE_{(pid-1 \bmod 2)}$;
  do $j_1 = 2k_1, \min(2k_1+1, 4)$
  do $j_2 = \max(2k_2, j_1), \min(2k_2+1, j_1+4)$
    $a(j_1, j_2-j_1) =$
      $f_1(c(j_1, j_2-j_1-1), b(j_1, j_2-j_1))$;
    $b(j_1, j_2-j_1) =$
      $f_2(a(j_1-1, j_2-j_1+1), c(j_1, j_2-j_1))$;
    $c(j_1, j_2-j_1) =$
      $f_3(b(j_1-1, j_2-j_1), a(j_1, j_2-j_1))$;
  enddo;
  enddo;
  send message to $PE_{(pid+1 \bmod 2)}$;
enddo;
enddo;
```

(a) Chain-based parallel code



(b) Communication between chains

Figure 8. Chain-based Scheduling of the Example

shown in Figure 7. Note that some tiles are incomplete or empty. The loop bounds of the inner j-loops will guarantee that only the relevant iterations will be executed [4].

3.3 Chain-Based Partitioning and Scheduling

After loop skewing and loop tiling and normalization of the controlling loops, we can form chains of tiles and schedule them to the parallel processors. Let us use the transformed nested loop in Figure 7 to illustrate the idea of chain-based partitioning and scheduling first.

Suppose that there are $p = 2$ processors, PE_0 and PE_1, in the multicomputer system. To execute the tiles shown in Figure 7(b), we can allocate all the tiles with $k_1 = 0$ to PE_0 and all the tiles with $k_1 = 1$ to PE_1. After PE_0 finishes the tiles with $k_1 = 0$, it can proceed to execute all the tiles with $k_1 = 2$. The tiles with the same k_1 index form a *chain*. A chain is a sequence of tiles connected by data dependences.

The partitioning and scheduling of the example are shown in Figure 8. Figure 8(a) shows the code for PE_{pid}. The processor identity is denoted by *pid*. In this example, $pid \in \{0, 1\}$. After a processor finishes the computation of a tile, it packs a message and sends it to the next processor. It receives a message from the previous processor before the computation of the tile starts. The thick arrows on Figure 8(b) represent the messages passed between processors.

The rationale of chain-based partitioning and scheduling is as follows:

- Allocating all the tiles of a chain to the same processor can save the message-passing for data dependences between the tiles. For example, the data dependences from the tile $(k_1, k_2) = (0, 0)$ to the tile $(k_1, k_2) = (0, 1)$ can be satisfied by executing the former first and using the local memory for data passing.

- Due to the regular dependence structure of the computation, the interprocessor communication between chains can be aligned with each other and overlapped with the computation of chains in a pipeline fashion.
- The code generation for such partitioning and scheduling is simple as shown in Figure 8(a).

In this example, we use the inner controlling loop to form chains and spread the iterations of the outer controlling loop across the parallel processors. That is, the outer controlling loop is executed as a DOACROSS loop.

In general, partitioning and scheduling tiles is a mapping M from tile space K to processor space P, i.e., $M : K \rightarrow P$. A multicomputer has a network with fixed topology such as hypercube or mesh to connect processors. However, we can use an appropriate logical structure for processor space P for partitioning and scheduling and then embed the logical processor space into the physical parallel processors of the system. For a tile space with n controlling loops, we need at least one innermost loop to form chains. Assume that we are going to use $q \geq 1$ inner controlling loops to form chains. This means that we are going to allocate all iterations of the q inner controlling loops to a single processor. The $n-q$ outer controlling loops are DOACROSS loops and represent a $(n-q)$-dimensional *chain space*, C, which contains all the chains. The data dependences between chains are regular and can be satisfied by communication between processors. The best way to pipeline the communication and computation among the processors is to fully interleave chains across the processors. Therefore, the logical processor space we need is a $(n-q)$-dimensional mesh defined as follows:

$$P = \{(p_1, \cdots, p_{n-q}) : 0 \leq q_k \leq P_k - 1, k = 1, \cdots, n-q\}$$

where P_k is the size of the processor space in the $k-th$ dimension and $P_1 \cdots P_{n-q}$ is the total number processors used for the computation. Here we assume that the $(n-q)$-dimensional mesh can be embedded into the physical topology of the network of the system.

The chain-based partitioning now can be represented by the following function $M : K \rightarrow P$:

$$M(k_1, \cdots, k_{n-q}, \cdots, k_n) = (k_1 \bmod P_1, \cdots, k_{n-q} \bmod P_{n-q})$$

According to the above partitioning function, the chain with $k_1 = w_1, \cdots, k_{n-q} = w_{n-q}$ will be executed by the processor $(w_1 \bmod P_1, \cdots, w_{n-q} \bmod P_{n-q})$. In other words, the set of chains allocated to processor $(p_1, \cdots, p_{n-q})$ is

$$\{(k_1, \cdots, k_{n-q}) \in C : k_l \equiv p_l \pmod{P_l}, l = 1, \cdots, n-q\}$$

The processor will execute these chains in the increasing lexicographic order of

```
DO $k_1 = p_1, \lceil (j_1^{\max} - j_1^{\min} + 1)/s_1 \rceil - 1, P_1$
$\vdots$
DO $k_{n-q} = p_{n-q}, \lceil (j_{n-q}^{\max} - j_{n-q}^{\min} + 1)/s_{n-q} \rceil - 1, P_{n-q}$
    DO $k_{n-q+1} = 0, \lceil (j_{n-q+1}^{\max} - j_{n-q+1}^{\min} + 1)/s_{n-q+1} \rceil - 1$
    $\vdots$
    DO $k_n = 0, \lceil (j_n^{\max} - j_n^{\min} + 1)/s_n \rceil - 1$
        $\boxed{\text{Receive and Unpack messages;}}$
        DO $j_1 = j_1^{\min} + k_1 s_1,$
                $\min(j_1^{\min} + k_1 s_1 + s_1 - 1, j_1^{\max})$
        $\vdots$
        DO $j_n = \max(j_n^{\min} + k_n s_n, G_n^1, G_n^2, \cdots),$
                $\min(j_n^{\min} + k_n s_n + s_n - 1, H_n^1, H_n^2, \cdots)$
            $B\prime(j_1, \cdots, j_n)$
        ENDDO
        $\vdots$
        ENDDO
        $\boxed{\text{Pack and Send messages;}}$
    ENDDO
    $\vdots$
    ENDDO
ENDDO
$\vdots$
ENDDO
```

Figure 9. SPMD chain-based parallel code

$(k_1, \cdots, k_{n-q})$. Therefore, the SPMD code for processor $(p_1, \cdots, p_{n-q})$ is a program shown in Figure 9. It is obvious that the program contains all the partitioning and scheduling information that each processor needs at run time.

4 Concluding Remarks

We have presented a series of loops transformations to generate chain-based parallel programs for nested loops on multicomputers. They are loop skewing, loop tiling and normalization, and chain-based partitioning and scheduling. We are currently implementing these transformations in our prototype parallelizing compiler for Fujitsu AP1000.

We have been concentrating on the loop transformations for chain-based partitioning and scheduling. The code generation for message packing and unpacking in the chain-based codes will be discussed elsewhere.

There are a number of factors that will affect the performance of the generated chain-based parallel codes:

Size of tiles. The size of tiles determines the granularity of the parallel computation. The tile size should be such that the execution time of a tile should be roughly equal to the communication time of a message passing between processors. The balance between the computation and communication will hide the communication latency and offer good pipelining for the whole parallel execution.

Regularity of chain length. Since the tile space is trapezoid instead of rectangular due to the possible loop skewing, chosing different controlling loops to form chains will affect the pipelining effect of interprocessor communication. In fact, all the n controlling loops are interchangeable and we can move any of them inwards to form chains. The question is how many and what loops should be moved inwards and used to form chains to achieve the best performance.

Processor space. We need to decide the number of processors needed and the shape of the logical processor space to give the best performance. This problem is directly related to the size of tiles and the shape of the chain space.

The thorough discussions of these issues go beyond the scope of this paper and we will address them in the next report: "Chain-Based Scheduling: Part II – Granularity and Performance".

References

[1] P. Tang and G. Michael, "Chain-Based Partitioning and Scheduling of Nested Loops for Multicomputers," *Proceedings of the 1991 International Conference on Parallel Processing*, vol. II, pp. 243–246, August 1991.

[2] C. -T. King, W. -H. Chou and L. M. Ni, "Pipelined Data-Parallel Algorithms: Part II – Design," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 4, pp. 486–499, October 1990.

[3] J-P. Sheu and T-H. Tai, "Partitioning and Mapping Nested Loops on Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 430–439, October 1991.

[4] M. E. Wolf and M. S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452–471, October 1991.

[5] G. Michael and P. Tang, "Parallel Loop Code Generation for AP1000," in *Proceedings of the Second Fujitsu-ANU CAP Workshop*, Canberra Australia, November 1991, pp. D.1–D.11.

[6] L. Lamport, "The Parallel Execution of DO Loops," *Communications of the ACM*, vol. 17, no. 2, pp. 83–93, February 1974.

[7] F. Irigoin and R. Triolet, "Supernode Partitioning," in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, CA, January 1988, pp. 319–329.

[8] M. Wolfe, "Loop Skewing: the Wavefront Method Revisited," Kuck and Associate, Inc., , 1987.

[9] U. Banerjee, "Unimodular Transformations of Double Loops," in *Advances in Languages and Compilers for Parallel Processing: Proceedings of the Third (1990) Workshop on Languages and Compilers for Parallel Computing*, A. Nicolau, D. Gelernter, T. Gross and D. Padua, Eds. London, UK: Pitman, pp. 192–219, 1991.

[10] K. G. Kumar, D. Kulkarni and A. Basu, "Generalized Unimodular Loop Transformations for Distributed Memory Multiprocessors," Center for Development of Advanced Computing, FG-TR-014, January 1991.

[11] H. Zima and B. Chapman, *Supercompilers for Parallel and Vector Computers.* Addison-Wesley Publishing Company, 1991.