A SYSTEMATIC APPROACH FOR DESIGNING
PIPELINED DATA PARALLEL ALGORITHMS

Chung-Ta King
Department of Computer and Information Science, New Jersey Institute of Technology,
Newark, NJ 07102

Wen-Hwa Chou, Lionel M. Ni
Department of Computer Science, Michigan State University,
East Lansing, MI 48824

ABSTRACT

Pipelined data parallel computations achieve a high degree of parallelism by partitioning the data set among the processors and operating on the data blocks in a pipelined fashion. The resulting algorithms are very suitable for execution on multicomputers. A systematic procedure for designing pipelined data parallel algorithms is introduced. This procedure starts with a nested-loop program, manipulates the dependencies between the loops, and groups related loops to obtain pipelined and data parallel operations. Using this procedure, it is possible to parallelize a nested loop automatically.

Keywords: Parallel processing, pipelining, algorithm design, data parallel algorithms, program restructuring

INTRODUCTION

A data parallel computation divides the set of data among multiple processors. Parallelism is achieved by simultaneously operating upon large sets of data, rather than using multiple threads of control. The resultant algorithms are termed data parallel algorithms (Ref. 2). In terms of programming, programs with loops to handle static and regular data structures are suitable for data parallel computation.

Pipelined data parallel algorithms are a subset of data parallel algorithms, which, in addition to data parallelism, exploit processor-level pipelining among the multiple processors (Ref. 3,4). Pipelined data parallel algorithms are targeted at medium- to large-grain parallel computers, especially multicomputers (Ref. 1).

Multicomputers with a few hundred processors are common. Thus, they hold the promising potential for massive parallelism. However, the need for explicit data exchanges between processors to communicate has become the major system bottleneck. Pipelined data parallel algorithms reduce the effect of communication overhead by regulating data flows in the system so that data can be processed and transmitted between processors in a pipelined fashion.

An Example

A pipelined data parallel algorithm for matrix multiplication is shown in Figure 1 (Ref. 4). Assume that the underlying multiprocessor supports an interprocessor connection of a mesh. Matrices A and B are partitioned along columns and rows. The host first loads submatrices of A into the corresponding processors in the mesh (see Figure 1). Then, submatrices of B are piped into the mesh from the host. During each iteration, the following operations are performed in the processors:

1. receive a submatrix of B from the north;
2. send the submatrix to the south;
3. perform the submatrix multiplication, \( C_{ij} = A_{ij} \times B_{ij} \);
4. receive a C submatrix from the west;
5. add the received C submatrix to the result in (3);
6. send the resultant C submatrix to the east.

Through processor-level pipelining, it can be seen that the activities in the processors are data-driven and highly overlapped. In addition, the communications between processors are accomplished by regular and local data flows. These properties make the resultant pipelined data parallel algorithm very efficient.

Pipelined data parallelism is a very general concept for algorithm design. To apply this concept to general applications, it is necessary to have a systematic procedure of designing this kind of algorithms. Major theme of this paper is thus such a systematic procedure. The basic technique used is the grouping of loop instances.

Organization of the Paper

A general procedure for designing pipelined data parallel algorithms is first outlined in Section 2. Then, the grouping problem is defined in Section 3, followed by various grouping techniques in Section 4. Our conclusion is given in Section 5.

Figure 1. Data flows in pipelined matrix multiplication
A SYSTEMATIC DESIGN PROCEDURE

In this section a systematic design procedure is outlined. Due to the similarity of pipelined data parallelism and systolic arrays, initial stages of our procedure are identical to those used in synthesizing systolic arrays (Ref. 7). However, additional stages are required to control the granularity of the algorithm in order to balance the communication overhead on a multicomputer.

A nested-loop program is shift-invariant if data dependencies between loops do not change with the loop indices. The typical loop for matrix multiplication is shift-invariant:

\[
\text{for } i = 0 \text{ to } 3 \text{ do}
\text{for } j = 0 \text{ to } 3 \text{ do}
\text{for } k = 0 \text{ to } 3 \text{ do}
\]

Using the above program as an example, the design procedure is outlined as follows:

1. **Transform all variables into pipelined variables**

The restructuring involves eliminating broadcast effects and enforcing the single-assignment rule (Ref. 6). After restructuring, the matrix multiplication loop looks like the following, with necessary initialization omitted:

\[
\text{for } i = 0 \text{ to } 3 \text{ do}
\text{for } j = 0 \text{ to } 3 \text{ do}
\text{for } k = 0 \text{ to } 3 \text{ do}
\]

2. **Derive the computational structure**

If the nested-loop program has \( n \) levels, then the computational structure, \( Q \), of the program is a directed graph on an \( n \)-dimensional Cartesian space. Each vertex in \( Q \) represents one loop instance and has a coordinate \((c_0, \ldots, c_{n-1})\) if the corresponding loop instance has a loop index \((c_0, \ldots, c_{n-1})\).

There is an arc (dependence vectors) from one vertex \( v_i \) to the other \( v_j \) if the loop corresponding to \( v_j \) references a variable which is generated in the loop corresponding to \( v_i \). Note that, for a shift-invariant nested loop, all vertices in the corresponding computational structure have the same set of dependence vectors. It follows that such a computational structure \( Q \) can be defined by the two-tuple \((V, D)\), where \( V \) denotes the set of all vertices in \( Q \) and \( D \) is the set of dependence vectors.

The computational structure corresponding to the restructured matrix multiplication program is shown in Figure 2.

3. **Group or project vertices in \( Q \)**

In conventional systolic array synthesis techniques, a space-time projection is sought at this step to produce systolic effects (Ref. 7). All vertices in \( Q \) are projected along a particular direction which represents the time-axis, i.e., the progress of time (Ref. 6). However, projection is just a special case of grouping. In grouping adjacent vertices in \( Q \) are merged together to form larger vertices. By controlling the size of the groups, we can control the granularity of the algorithm.

In Figure 2, the grouping along \( j \) with size 4 (i.e., a projection) and along \( i \) and \( k \) with size 2 respectively will result in a contracted structure as shown in Figure 3. As will be shown later, for many computational structures, certain grouping schemes will introduce extra dependencies between the groups. This implies extra communication in the resultant algorithm. Details of the grouping will be discussed in Section 3 and 4.

4. **Obtain node programs**

In Figure 3, if one group is allocated to one processor in a multicomputer, then the processor at location \((r', k')\), where \( 0 \leq r', k' \leq 1 \), will execute the following program:

\[
l \leftarrow 0 \text{ to } 3 \text{ by } 2 \text{ do}
\text{/* communicate with neighboring processors */}
\text{for } j \leftarrow l \text{ to } l+1 \text{ do}
\text{for } i \leftarrow 2l' \text{ to } 2(l'+1) \text{ do}
\text{for } k \leftarrow 2k' \text{ to } 2(k'+1) \text{ do}
\]

Figure 2. The computational structure of matrix multiplication

Figure 3. The contracted structure of matrix multiplication
For any subset of dependence vectors, integers, and positive integers, respectively. The input to a shift-invariant nested-loop program.

(1) For each subset of dependence vectors, if there exists a grouping along a direction different from the dependence vectors (see Figure 4(b)).

Consider computational structures with one or two dependence vectors. There exist dependence-preserving contracted structures if the grouping is along the dependence vectors (see Figure 4(a)). This is because the dependence vectors are independent in acyclic computational structures. Thus, the resultant groups will not interfere with each other and introduce extra dependencies. Nevertheless, if the base vertices are not properly chosen, then extra dependencies will still be created (see Figure 4(b)).

Suppose the grouping is along a direction different from the dependence vectors of the computational structure. Then, this situation is equivalent to a computational structure, \( Q(V', D') \), with three dependence vectors, \( D = \{ d_0, d_1, d_2 \} \). Since \( Q \) is an acyclic two-dimensional structure, we can assume that

\[
a_2d_2 = a_0d_0 + a_1d_1
\]

where \( a_0, a_1, \) and \( a_2 \) are the smallest positive integers to satisfy Equation 1. A typical example of such a computational structure is shown in Figure 5(a), where \( D = \{ [1,0], [1,1], [0,1] \} \) and \( [1,1] = [1,0] + [0,1] \).

Consider the grouping along \( d_1 \) first. In Figure 5(a), a possible grouping along \( d_1 \) is depicted in dashed boxes. Figure 5(b) shows the corresponding contracted structure with the group ids indicated in the circles. It can be seen that an extra dependence vector, \([0,1] \), is introduced. Thus, the contracted structure is not dependence-preserving. In fact, we have the following theorem:

**Theorem:**

Given a computational structure \( Q \), determine the groupings of \( Q \) which will result in dependence-preserving contracted structures.
<Theorem 1>

Let $Q(V, D)$ be a computational structure. If there exist three dependence vectors, $d_0, d_1, d_2 \in E_D$, where $d_0, d_1,$ and $d_2$ satisfy Equation 1, then the grouping along $d_2$ with size $r > a_2$ is not dependence-preserving.

The proof is given in Ref. 4 and is omitted here.

Theorem 1 states the conditions when a grouping is not dependence-preserving. A more constructive way is to study under what conditions a dependence-preserving structure will be generated. Given an acyclic computational structure $Q(V, D)$ with the dependence set $D = \{d_0, \ldots, d_{m-1}\}$, where $m \geq 3$, we can always find (from linear algebra) two vectors, say, $d_0$ and $d_1$, such that

$$c_i d_i = a_i d_0 + b_i d_1$$

(2)

where $a_i, b_i, c_i \in \mathbb{R}, \quad 2st cm$. Define

$$a_{\text{max}} = \max_{2st cm} \left\{ \frac{a_i}{c_i} \right\} \quad b_{\text{max}} = \max_{2st cm} \left\{ \frac{b_i}{c_i} \right\}$$

The "range of influence" of a vertex $v$ in $Q$ can be defined as the vertex set $X(v)$, where

$$X(v) = \{ w \mid \text{we } V, \quad w = v + s \max d_0 y + y \max d_1, \quad 0 \leq y \leq 1 \}$$

Then, any vertex $w \in V$ which is dependent on $v$ is in $X(v)$.

<Theorem 2>

Let $Q(V, D)$ be a computational structure with $m$ dependence vectors, where $m \geq 3$, and the dependence vectors satisfy Equation 2. Then, the grouping along $d_0$ with size $r > a_{\text{max}}$ and along $d_1$ with size $r > b_{\text{max}}$ is dependence-preserving.

Again, the proof can be found in Ref. 4. The implications of Theorem 2 are: (1) as long as the group size is large enough, there always exist dependence-preserving groupings along $d_0$ and $d_1$, and (2) the resultant contracted structure is a universal planner array (see Figure 6). A universal planner array is the most general systolic arrays in two-dimensional spaces (Ref. 6). It follows that, in the final implementation, each processor only has to communicate with at most three other processors. Note also that $d_0$ and $d_1$ can be any vectors which are not necessary in $D$ as long as Equation 2 is satisfied.

CONCLUSION

We have presented in this paper a systematic procedure for designing pipelined data parallel algorithms from shift-invariant nested loops. This procedure concentrates on grouping loops in the original program so as to reduce the number of communicating processors, control the granularity, and increase the degree of pipelining.

Results presented in this paper are preliminary. A mathematic foundation is needed to abstract the concept of grouping so that results presented here can be applied to higher dimensional spaces. Also, the relationship between projection and grouping needs to be further probed, which may in turn assist in designing more efficient systolic arrays. Ultimately, we could expect results obtain in this research will contribute to intelligent compilers for multicomputers, which parallelize programs (semi-)automatically.

REFERENCE