Automatic Derivation of Parallel and Systolic Programs

Automatic Derivation of Parallel and Systolic Programs

Lilei Chen[1]
Center for Research in Computing Technology
Division of Applied Sciences
Harvard University

October 18, 1994

Abstract

We present a simple method for developing parallel and systolic programs from data dependence. We derive sequences of parallel computations and communications based on data dependence and communication delays, and minimize the communication delays and processor idle time. The potential applications for this method include supercompiling, automatic development of parallel programs, and systolic array design.

1 Introduction

Given a sequential program consisting of a loop, or a set of equations that recursively define an array, we want to develop a parallel program for a shared or distributed memory parallel computer. There are two problems here. First we have to reveal the parallelism and find the sequences of parallel computations. Second, for distributed memory parallel computers, we want to schedule the computations and communications at compile time so that all data needed arrive before a computation is scheduled to start, and minimize processor idle time and overall communications. Given enough parallel processors, parallelism is limited by two factors. The first is the data dependence. The second is the delays in the arrival of dependent data, which we call communication delays. We will introduce simple ways to

- reveal the parallelism by finding the sequences of parallel computations allowed by data dependence;
- minimize communication delays and make possible systolic data movements;
- produce systolic programs by combining the computation sequences and communication delays.

F. Irigoin and R. Triolet [8] showed how to find the sequences of parallel computations allowed by a data dependence using a hyperplane method. Their method reduces the problem to that of solving a set of linear inequalities. The solution of the set of inequalities reveals

[1]this work is supported by DARPA under contract N00039-88-C-0163 and F19628-92-C-0113

all possible parallel sequences that the data dependence allows. We [4, page 16] showed that it is not obvious to find the best sequence from the solution set of an inequality. By going after all solutions, the best sequence is clouded by an infinite number of absolutely useless ones, making this method inefficient for very simple parallelization problems. Our method does not reveal *all* the sequences allowed by a data dependence. Neither do we guarantee maximum parallelism. The sequences we produce are the most obvious and natural ones, which often give maximum parallelism. In return, we get a much simpler and cost effective method.

The communication delays are based on very simple and straight forward data to processor mappings. We assume a network topology that is natural to the problem being solved, and show that the communication delays can be combined to the computation sequences to easily produce systolic algorithms. We will use our method to produce the classic systolic program for matrix multiplication, and also produce a very efficient systolic program for the shortest path problem that is very hard to write by hand. We make no restriction on either network topology of the parallel machine or number of processors here. Different systolic algorithms may result if we add more restrictions on the parallel machine or change the mapping. While it may be interesting to experiment with those, our focus here is to expose the maximum parallelism allowed by data dependence while minimize the overall communication required by the problem.

There are many related works in the area of automatically producing parallel and systolic programs. Tseng [12] has described a systolic array parallelizing compiler which compiles programs in the sequential language AL into systolic programs for the ten cell linear systolic array Warp. Fortes and Moldovan [6] proposed some index transformation techniques called time and space transformations which compute the computation sequences and processor allocations that may result in systolic algorithms, and some techniques to find computation sequences for limited class of data dependencies. The basic idea of time and space transformations is to reshape the indices for the computations such that one coordinate represents the computation sequences, which is the time transformation, and the rest of the coordinates index the corresponding processor, which is the space transformation. Time transformation is based solely on data dependencies while space transformation is based both on data dependencies and parallel processor topologies.

Many results in the area of parallelization are developed from supercompilers that parallelize Fortran programs for shared memory parallel computers by Kuck *et al.* [9,10] and Kennedy *et al.* [1,2]. The basic idea is to replace sequential loops with semantically equivalent vector and array operations wherever it is determined that no data dependencies exist there. There are also some compiling systems [13] that target distributed memory parallel computers. ParaScope[3] is such a system that takes some extension of Fortran programs which allows programmers to specify the parallel features.

Sequential program parallelizing techniques are usually not directly applicable to recursive equations because they are based on pre-existing explicit computation sequences which do not exist in recursive equations. Most existing equational languages require the programmers to specify the sequences of computations and communications one way or the other. In the recursive equational language Crystal [5] implemented at Yale University, the programmer has to provide the compiler with domain morphism, which is in essence sequence of parallel computations and data layout. In the equational programming language EPL

[11] implemented at Rensselaer Polytechnic Institute, iterations are implied in the recursive equations. So the recursive equations are actually equivalent to the sequential loops, and the parallelization techniques such as those for Fortran parallelizing compilers can be applied. There are also some functional languages with extended parallel meta-language features [7] making it partly programmer's responsibility to specify the parallel behaviors.

Our method could potentially fill the need for automatically producing the sequences of parallel computations and communications for the equational languages. Our system consists of two parts. The first part uses *sequence constraints* to find the sequences of parallel computations based on data dependence. The second part combines the computation sequences with communication sequences to minimize the overall communication and computation time. Since our algorithm is only based on data dependence, it can be applied to recursive equations with no control sequences, sequential programs with loops, or programs in other forms as long as data dependence can be extracted.

2 Sequence Constraints and Computation Sequences

First we need to clarify what a *computation* is. When we talk about a computation here, it corresponds to an instance of a statement, either in a loop or in a set of equations. In a sequential loop, each iteration of a statement corresponds to a unique computation. In a recursive equation, each instance of the equation corresponds to a unique computation. We are interested in parallelism in the statement level, parallelism between instances of a statement or different statements.

Suppose S represents a statement or an equation, $(i_1, \ldots, i_n)$ represents an iteration in a loop or an instance of a recursive equation, then $S_{i_1,\ldots,i_n}$ represents a computation. The sequences of parallel computations for S are limited by the recurrent data dependence between them.

Suppose we have a data dependence relation:

$$S_{i_1,\ldots,i_n} \leftarrow S_{i_{11},\ldots,i_{1n}}, \ldots, S_{i_{m1},\ldots,i_{mn}},\ l_1 \le i_1 \le u_1 \wedge \ldots \wedge l_n \le i_n \le u_n$$

where each computation on the left side is dependent on the computations on the right side. We want to find a parallel program in the following form:

for $s = L$ **to** U
  compute $S_{i_1,\ldots,i_n}$,
    forall $l_1 \le i_1 \le u_1 \wedge \ldots \wedge l_n \le i_n \le u_n$ **such that** $\lfloor \mathrm{ds}(i_1, \ldots, i_n) \rfloor = s$

The program computes S in $U - L + 1$ steps. We can see from the program that the value of $\mathrm{ds}(i_1, \ldots, i_n)$ determines which step $S_{i_1,\ldots,i_n}$ is to be computed. It maps an index to a step. So the sequence of parallel computations in the program is defined by the function ds. We call this function *sequence function*, or *dependence sequence*. And we say that the computation can be sequenced by the function ds. We limit ds to linear functions.

In order to satisfy the data dependence, we have to make sure that

$$\forall k \in (1, m) : \mathrm{ds}(i_{k1}, \ldots, i_{kn}) - \mathrm{ds}(i_1, \ldots, i_n) \ge 1$$

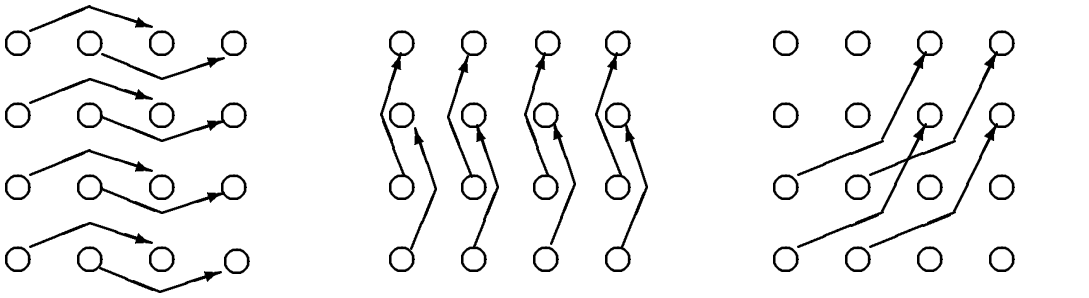


Figure 1: One to One Dependence

i.e., the computations on the right side of the data dependence must be completed at least one step ahead that of the left side.

To find the sequence functions, we start from the simple data dependencies in figure 1, where circles represent computations and arrows point to the flows of data dependencies. Based on the first data dependence, the computations can go column by column, two columns at a time, from left to right. The sequence function for this is:

$$\mathrm{ds1}(i,j) = \frac{j}{2}$$

With the second dependence, the computations can go row by row, two rows at a time, from bottom to top. The sequence function for this is:

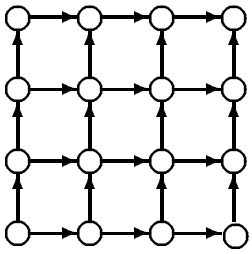
$$\mathrm{ds2}(i,j) = \frac{i}{-2}$$

With the third dependence, the computations can go both ways. Therefore, it can use either one of these sequence functions. Of course there are many other sequence functions that would work based on these data dependencies. One example is:

$$\mathrm{ds3}(i,j) = \frac{i}{-2} + \frac{j}{2}$$

which starts the computation from lower left, going diagonally up to upper right, two diagonal lines at a time. It is easy to check that ds3 works for all three dependences. But ds1 and ds2 are simpler, and more efficient considering the numbers of steps and processors required.

We found the sequence functions ds1 and ds2 by looking at the data dependence along each dimension. Along each dimension, the direction of the data dependence gives an indication of the direction of the sequence of parallel computations. And the dependence distance gives an upper bound of the amount of parallel computations in each step. The dependence direction and the dependence distance can be represented by the sign and the value of a number. This number summarizes the constraints imposed by the data dependence on the sequence of parallel computations along the dimension. For the first dependence, 2 summarizes the parallelism allowed along j, *i.e.*, the computation can go in increase direction along j, with a parallel distance of 2. For the second dependence, -2 summarizes the parallelism allowed along i, *i.e.*, the computation can go in decrease direction along i, with a parallel



$$S_{i,j} \leftarrow S_{i,j-1}, S_{i+1,j}, 0 \leq i,j \leq n$$

Figure 2: One to Many Dependence

distance 2. For the third dependence, 2 summarizes the parallelism allowed along j, and -2 summarizes the parallelism allowed along i. We call these numbers *sequence constraints* along the indices, written as $(i,-2)$ and $(j,2)$.

When there is more than one computation on the right side of the dependence relation, we can find the constraints from each dependence and combine the results. Let us look at the data dependence in figure 2. From the dependence on $S_{i,j-1}$, we know the computations can go column by column from left to right, with sequence constraint $(j,1)$ and sequence function j. From the dependence on $S_{i+1,j}$, we know the computations can go row by row from bottom to top, with sequence constraint $(i,-1)$ and sequence function $-i$. We combine the above sequence constraints and represent it as $(i,-1)\widehat{\wedge}(j,1)$ ($\widehat{\wedge}$ reads as "and"). It corresponds to the sequence function:

$$\text{ds4} = -i + j$$

which is exactly the same as adding the two sequence functions together. As we can verify, ds4 starts the computations from lower left, going up diagonally to upper right, one diagonal line at a time. It is the natural sequence we would come up by looking at the dependence graph.

When two dependence fall in the same direction, such as

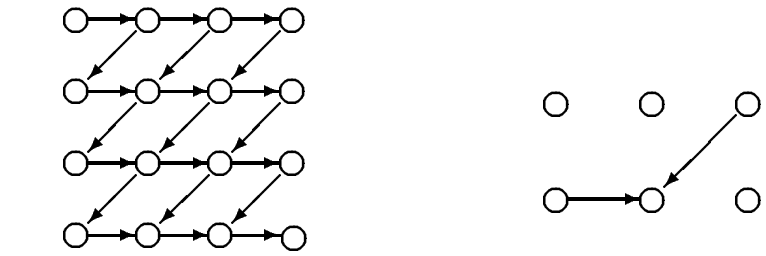
$$S_{i,j} \leftarrow S_{i-1,j}, S_{i-2,j},\ 0 \leq i,j, \leq n$$

the two dependence give the sequence constraints $(i,1)$ and $(i,2)$ respectively. Combining them, we have

$$(i,1)\widehat{\wedge}(i,2) = (i, \min(1,2)) = (i,1)$$

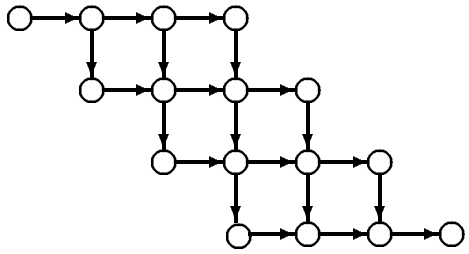
And the sequence function is $\text{ds}(i,j) = i$.

It does not always work to just combine the sequence constraints to obtain the sequence functions. When we have a data dependence relation that "spreads out", then the above simple approach does not work. Figure 3 is an example of this. We show on the left the data dependence and its graph. Now if we focus on the dependence for one particular element, as shown on the right, we will notice that the two dependences do not fall into the same corner. If we project the data dependence onto index variable j, the two dependences are in opposite



$S_{i,j} \leftarrow S_{i,j-1}, S_{i-1,j+1}, 0 \leq i,j \leq n$

Figure 3: "Spread Out" Dependence



$S_{i',j'} \leftarrow S_{i',j'-1}, S_{i'-1,j'}, 0 \leq i' \leq n \wedge 0 \leq j' \leq 2n$

Figure 4: Transformed Dependence

directions, which is the red flag. Looking back at the sequence constraints, dependence on $S_{i,j-1}$ provides the constraint $(j,1)$ and and sequence function j. Dependence on $S_{i-1,j+1}$ provides the constraints $(i,1)$ or $(j,-1)$, and sequence functions i or $-j$. Combining the constraints $(j,1)$ with $(i,1)$ produces a function $i+j$, which is wrong since it places $S_{i,j}$ and $S_{i-1,j+1}$ in the same step. The constraints $(j,1)$ and $(j,-1)$ are in fact conflict and cannot be combined.

This problem can be solved by index transformations. With the new index variables i', j' where $i' = i, j' = i + j$, the dependence and its graph of figure 3 become that in figure 4. With this dependence, the constraints are $(j',1)$ and $(i',1)$. And the sequence functions are j' and i' respectively. Combined constraint and sequence function are $(i',1)\hat{\wedge}(j',1)$ and $i' + j'$. Now we can transform the index back and get the sequence function $2i + j$, which gives the maximal parallelism for the above dependence.

In general, if conflict occurs when we project a data dependence onto an index variable, say j, we can adjust j against another index variable, say i, where

$$\begin{aligned} i' &= i \\ j' &= ci + j \end{aligned}$$

such that no conflict occurs in the new index. The direction or amount of this adjustment,

or index transformation, can be controlled by c. The sign of c decides the direction of the transformation, while the value decides the amount.

When there is no conflict along any dimension, *i.e.*, all the right hand elements fall on the same corner in the dependence graph, we can be sure that the sequence functions from the combined sequence constraints give the right sequences of parallel computations.

3 Communication Delays and Data movements

In a distributed memory parallel computer with message passing, we have to take into consideration data movements and communication delays in scheduling the computations, which depend on the communication network of the parallel processors, the mapping from data to processors, and the data distances.

We assume the communication network is a multidimensional grid, since the index of $S_{i_1,\ldots,i_n}$ forms an n dimensional grid. With similar computation and parallel machine structures, we can focus on the kind of data mappings and movements that brings efficient systolic algorithms. And we will only use nearest neighbor communications in the output programs.

Given the data dependence

$$\begin{aligned} S_{i_1,\ldots,i_n} \quad \leftarrow \quad & S_{i_{11},\ldots,i_{1n}}, \ldots, S_{i_{m1},\ldots,i_{mn}}, A^{(1)}_{j_{11},\ldots,j_{1n_1}}, \ldots, A^{(k)}_{j_{k1},\ldots,j_{kn_k}}, \\ & l_1 \le i_1 \le u_1 \wedge \cdots \wedge l_n \le i_n \le u_n \end{aligned}$$

where $S_{i_1,\ldots,i_n}$ refers to specific instances of statement S as before, and $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$ is a predefined data element with the restriction that $(j_{y1},\ldots,j_{yn_y})$ is a subset of $(i_1,\ldots,i_n)$.

We need to map $S_{i_1,\ldots,i_n}$ to the parallel processors for their computations, map any predefined data to the appropriate processors, and arrange necessary data movements.

3.1 Data to Processor Mapping

Based on the recurrent data dependence part, we can find a sequence function according to the previous section, which slices the index space of $S_{i_1,\ldots,i_n}$ into pieces. All the computations on the same piece can be computed in parallel, while different pieces have to be computed in sequence. For example, the sequence functions k and $i+j$ slice the index space $S_{i,j,k}$ into pieces as shown in figure 5. In both cases, we have to do the computations one piece at a time in sequence. Therefore, we can project all the pieces onto an $n-1$ dimensional grid of parallel processors.

In other words, we first find a dimension where every elements on it belong to different pieces and have to be computed in sequence. It is k in the first example and i or j in the second example. We then map all the elements on that dimension onto a single processor. For the first example, we map $S_{i,j,k}$ to processor (i,j). For the second example, we can choose the mappings $S_{i,j,k} \rightarrow (j,k)$ or $S_{i,j,k} \rightarrow (i,k)$. All three mappings are clearly shown by the arrows in figure 5. Generally speaking, if i_t appears in the sequence function, then for all i_t, $S_{c_1,\ldots,c_{t-1},i_t,c_{t+1},\ldots,c_n}$ have to be computed in sequence. We can map them onto processor $(c_1,\ldots,c_{t-1},c_{t+1},\ldots,c_n)$, which does the computations in sequence according to the sequence function.

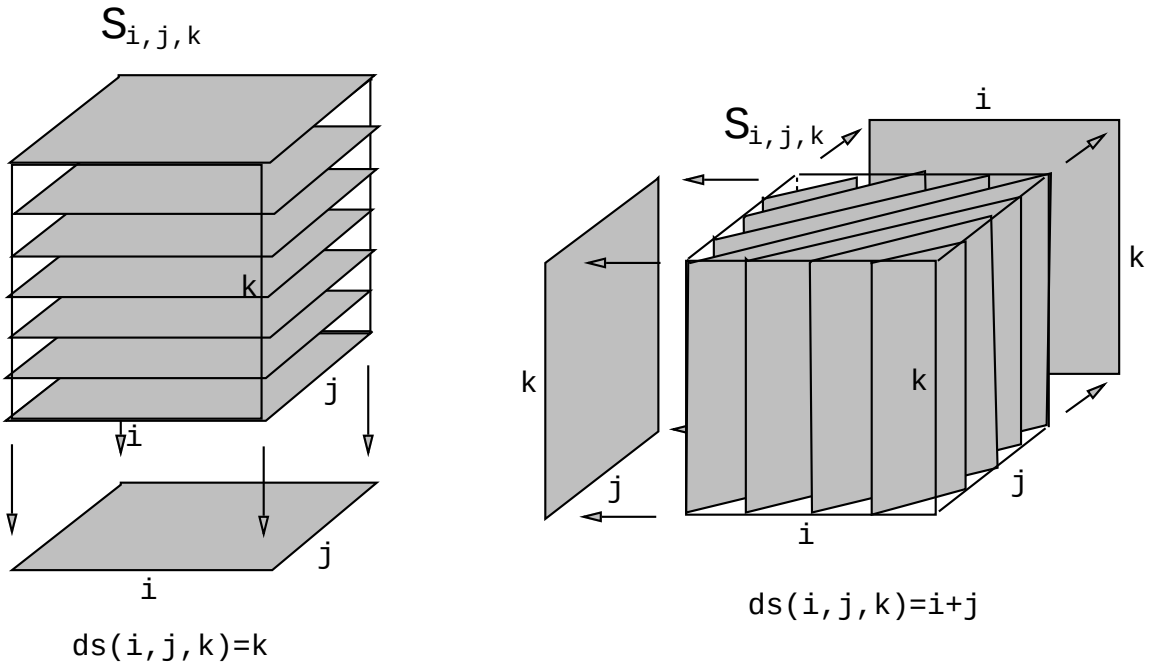


Figure 5: Index Spaces Divided by Sequence Functions

More interesting is the mapping for $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$. If there is only one processor that needs $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$, then we want to map it to that processor. If there is a group of processors that need it, then we want to map it to the edge or corner of that group so that $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$ can be spread to the whole group one step at a time during the computations. So we map $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$ to

$$(p_1,\ldots,p_{t-1},p_{t+1},\ldots,p_n)$$

where $p_x = i_x$ if i_x appears in $(j_{y1},\ldots,j_{yn_y})$, or p_x is an edge of i_x if i_x does not appear in $(j_{y1},\ldots,j_{yn_y})$, which means $p_x = \min(i_x)$ or $p_x = \max(i_x)$.

The mappings we describe are actually very natural ones that we would use when writing parallel programs by hand. For the matrix multiplication problem

$$\begin{aligned} c^{(0)}_{ij} &= 0,\ 1 \le i,j \le n \\ c^{(k)}_{ij} &= c^{(k-1)}_{ij} + a_{ik}b_{kj},\ 1 \le i,j,k \le n \end{aligned}$$

The mappings can be

$$c^{(k)}_{ij} \to (i,j),\ a_{ik} \to (i,1),\ b_{kj} \to (1,j)$$

which is shown in figure 6.

Our mappings make the data close to where they are needed, thus minimize the total communications.

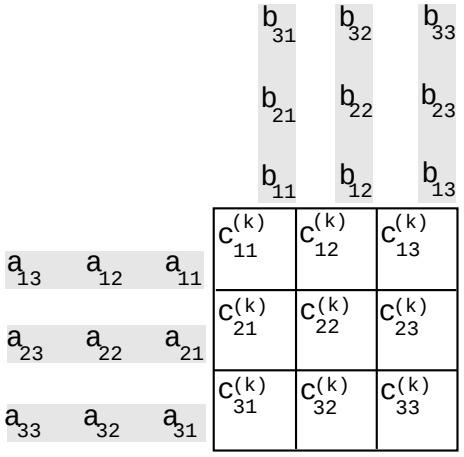


Figure 6: Mapping for Matrix Multiplications

3.2 Data Movement

We can envision an ordering among the processors along the dimensions $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$ will traverse. The closer a processor is to where $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$ is placed initially, the earlier it starts its computations. We call such orderings communication delays or *communication sequences*. Suppose $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$ is place on lower edge of i_x, then the communication sequence can be characterized as i_x, since the processor with smaller i_x will receive $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$ first and thus start its computations first. Similarly, if $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$ is placed on upper edge of i_x, the the communication sequence can be characterized as $-i_x$. We get the total communication sequence $\mathrm{cs}(i_1,\ldots,i_n)$ by adding the communication sequences for each dimension caused by the right hand side elements in the dependence relation. For matrix multiplication, with the mapping we chose above, it is j caused by a_{ik} and i caused by b_{kj}. And the communication sequence is $i+j$.

Because of the mapping we use for $A^{(y)}_{j_{y1},\ldots,j_{yn_y}}$, it will only move in one direction along any dimension, either in ascending direction or descending direction. So we have very simple representation of communication sequences for it. However, if the result of $S_{i_{y1},\ldots,i_{yn}}$ is needed by a group of processors, then it may need to move in both directions. For example, in the case of shortest path problem:

$$d^{(k)}_{ij} = \min(d^{(k-1)}_{ij}, d^{(k-1)}_{ik} + d^{(k-1)}_{kj}),\ 1 \le i,j,k \le n$$

the data dependence is:

$$S_{i,j,k} \leftarrow S_{i,j,k-1}, S_{i,k,k-1}, S_{k,j,k-1},\ 1 \le i,j,k \le n$$

With the mapping

$$S_{i,j,k} \rightarrow (i,j)$$

$S_{i,k,k-1}$ at processor (i,k) has to move to $S_{i,j,k}$ at processor (i,j) for all $1 \le j \le n$. This means $S_{i,k,k-1}$ has to move from (i,k) to $(i,k+1),(i,k+2),\ldots$ as well as to $(i,k-1),(i,k-2),\ldots$. In this case, the delay caused by $S_{i,k,k-1}$ is

the distance between $S_{i,j,k}$ *and* $S_{i,k,k-1}$
plus
the delay for $S_{i,k,k-1}$ *to become available*

The closer $S_{i,j,k}$ is to $S_{i,k,k-1}$, the earlier it will be computed. We can think of $S_{i,k,k-1}$ as causing a wave of computations along the second dimension, with itself being in the center of the wave. This is actually the case as we will show later. The function $s(i,j,k) = k$ gives the series of centers of the waves along the second dimension. The delay for $S_{i,k,k-1}$ to be available is $s(i,j,k)$ if $s(i,j,k) \geq s(i,k,k-1)$ or $-s(i,j,k)$ if $s(i,j,k) < s(i,k,k-1)$. This condition is necessary since all waves must occur in the right order. So the communication sequence from $S_{i,k,k-1}$ is

$$|j-k|+k$$

From the dependence and communication sequences, we can easily produce systolic programs.

4 Systolic Programs

Given the data dependence

$$\begin{aligned} S_{i_1,\ldots,i_n} \quad \leftarrow \quad & S_{i_{11},\ldots,i_{1n}}, \ldots, S_{i_{m1},\ldots,i_{mn}}, A^{(1)}_{j_{11},\ldots,j_{1n_1}}, \ldots, A^{(k)}_{j_{k1},\ldots,j_{kn_k}}, \\ & l_1 \leq i_1 \leq u_1 \wedge \cdots \wedge l_n \leq i_n \leq u_n \end{aligned}$$

We can obtain a sequence constraint

$$\mathrm{CS} = (i_{x_1}, d_{x_1}) \widehat{\wedge} \ldots \widehat{\wedge} (i_{x_r}, d_{x_r})$$

by choosing a sequence constraint from each of the S on the right side of the dependence and combining them. The dependence sequence from the sequence constraint is

$$\mathrm{ds}(i_1, \ldots, i_n) = \sum_{y=1}^{n} c_y i_y$$

where

$$c_y = \begin{cases} 0 & \text{if } i_y \text{ does not appear in CS} \\ 1/d_y & \text{if } (i_y, d_y) \text{ appears in CS} \end{cases}$$

We can then produce a mapping according to any index variable that appears in $\mathrm{ds}(i_1, \ldots, i_n)$. Let us call this index variable i_t. The mapping is

$$\begin{aligned} S_{i_1,\ldots,i_n} \quad & \rightarrow \quad (i_1, \ldots, i_{t-1}, i_{t+1}, \ldots, i_n) \\ A^{(y)}_{j_{y1},\ldots,j_{yn_y}} \quad & \rightarrow \quad (p_{y1}, \ldots, p_{y\,t-1}, p_{y\,t+1}, \ldots, p_{yn}) \end{aligned}$$

where

$$p_{yx} = \begin{cases} i_x & \text{if } i_x \text{ appears in } (j_{y1}, \ldots, j_{yn_y}) \\ \min(i_x) \text{ or } \max(i_x) & \text{otherwise} \end{cases}$$

According to the mapping, we can then find the communication sequences

$$\mathrm{cs}(i_1,\ldots,i_n) = \sum_{y=1}^{n} q_y$$

where

$$q_y = \begin{cases} 0 & y = t \text{ or there is no movement along } i_y \\ i_y & \text{data move from lower edge to the upper edge} \\ -i_y & \text{data move from upper edge to the lower edge} \\ |i_y - s(i_1,\ldots,i_n)| + s(i_1,\ldots,i_n) & \text{data move away from } s(i_1,\ldots,i_n) \text{ and} \\ & \quad \forall x \in (1,m) : s(i_1,\ldots,i_n) \geq s(i_{x1},\ldots,i_{xn}) \\ |i_y - s(i_1,\ldots,i_n)| - s(i_1,\ldots,i_n) & \text{data move away from } s(i_1,\ldots,i_n) \text{ and} \\ & \quad \forall x \in (1,m) : s(i_1,\ldots,i_n) < s(i_{x1},\ldots,i_{xn}) \end{cases}$$

We can then produce a systolic program

$\mathrm{map}(S_{i_1,\ldots,i_n})$,
$\mathrm{map}(A^{(y)}_{j_{y1},\ldots,j_{yn_y}})$
  forall $1 \leq y \leq k$
for $s = \min(\lfloor \mathrm{fs}(i_1,\ldots,i_n) \rfloor)$ **to** $\max(\lfloor \mathrm{fs}(i_1,\ldots,i_n) \rfloor)$
  data movements for $A^{(1)}_{j_{11},\ldots,j_{1n_1}},\ldots,A^{(k)}_{j_{k1},\ldots,j_{kn_k}}$,
  data movements for $S_{i_{11},\ldots,i_{1n}},\ldots,S_{i_{m1},\ldots,i_{mn}}$,
  compute $S_{i_1,\ldots,i_n}$,
    forall $l_1 \leq i_1 \leq u_1 \wedge \ldots \wedge l_n \leq i_n \leq u_n$ **such that** $\lfloor \mathrm{fs}(i_1,\ldots,i_n) \rfloor = s$

where $\mathrm{fs}(i_1,\ldots,i_n) = \mathrm{ds}(i_1,\ldots,i_n) + \mathrm{cs}(i_1,\ldots,i_n)$. All the movements and computations in the same parallel step can be done in parallel. The data move one processor each step towards their destinations. We have proofs [4] that the program satisfies the data dependence and all data arrive right before the computation is scheduled to start. We will show how this works in the following examples.

5 Examples

Matrix Multiplication

The recursive definition for the matrix multiplication problem is:

$$\begin{aligned} c^{(0)}_{ij} &= 0,\ 1 \leq i,j \leq n \\ c^{(k)}_{ij} &= c^{(k-1)}_{ij} + a_{ik}b_{kj},\ 1 \leq i,j,k \leq n \end{aligned}$$

Let us call the second equation S. Its data dependence is:

$$S_{i,j,k} \leftarrow S_{i,j,k-1}, a_{ik}, b_{kj},\ 1 \leq i,j,k \leq n$$

From the recursive dependence part, we can easily get the sequence constraint

$$(k,1)$$

and the dependence sequence function based on data dependence:

$$\mathrm{ds}(i,j,k) = k$$

We can map $S_{i,j,k}$ to processor (i,j). a_{ik} can be mapped to processor $(i,1)$ or (i,n) since j does not appear in the index. And b_{kj} can be mapped to processor $(1,j)$ or (n,j). Suppose we choose the mapping:

$$S_{i,j,k} \rightarrow (i,j), \;\; a_{ik} \rightarrow (i,1), \;\; b_{kj} \rightarrow (1,j)$$

Then the communication sequence is:

$$\mathrm{cs}(i,j,k) = i + j$$

Combining the dependence sequence and communication sequence, we get the final sequence:

$$\mathrm{fs}(i,j,k) = \mathrm{ds}(i,j,k) + \mathrm{cs}(i,j,k) = i + j + k$$

We use $c_{ij}^{(k)}(i,j)$ for $c_{ij}^{(k-1)}$ at processor (i,j), the corresponding parallel program is

$c_{ij}^{(k)} \rightarrow (i,j),\ a_{ik} \rightarrow (i,1),\ b_{kj} \rightarrow (1,j)$
  forall $1 \le i,j,k \le n$
for $s = 3$ **to** $3n-1$
  $c_{ij}^{(k)}(i,j) = c_{ij}^{(k-1)} + a_{ik} \times b_{kj}$,
  $a_{ik}(i,j) \rightarrow (i,j+1)$,
  $b_{kj}(i,j) \rightarrow (i+1,j)$,
    forall $1 \le i,j,k \le n$ **such that** $i+j+k = s$

Figure 7 shows step by step the data layout of running this program for $n = 3$. In the first step, a_{11} and b_{11} are available to $c_{11}^{(1)}$ and $c_{11}^{(1)}$ is computed. The data move one processor at each step. In the sth step, for all i,j,k such that $i+j+k = s$, a_{ik} and b_{kj} are available to $c_{ij}^{(k)}$ and $c_{ij}^{(k)}$ is computed. The data move like a wave starting from the upper left of the processor array towards the lower right. In each processor, the computations start as the data wave arrives and finishes as the wave leaves. For processor (i,j), data arrive in the sequence $(a_{i1}, b_{1j}), (a_{i2}, b_{2j}), (a_{i3}, b_{3j})$, and the sequence of the computation is

$$c_{ij}^{(1)} = c_{ij}^{(0)} + a_{i1}b_{1j}, \;\; c_{ij}^{(2)} = c_{ij}^{(1)} + a_{i2}b_{2j}, \;\; c_{ij}^{(3)} = c_{ij}^{(2)} + a_{i3}b_{3j}$$

Shortest Path Problem

A systolic program for shortest path problem is not obvious to write. Through the combination of computation sequence and data distance, we can obtained a very efficient one automatically. Its recursive specification is:

$$S:\ d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}),\ 1 \le i,j,k \le n$$

where $d_{ij}^{(0)}$ is the length of the direct path between i and j, and $d_{ij}^{(n)}$ is the shortest path between i and j. The data dependence is:

$$S_{i,j,k} \leftarrow S_{i,j,k-1}, S_{i,k,k-1}, S_{k,j,k-1},\ 1 \le i,j,k \le n$$

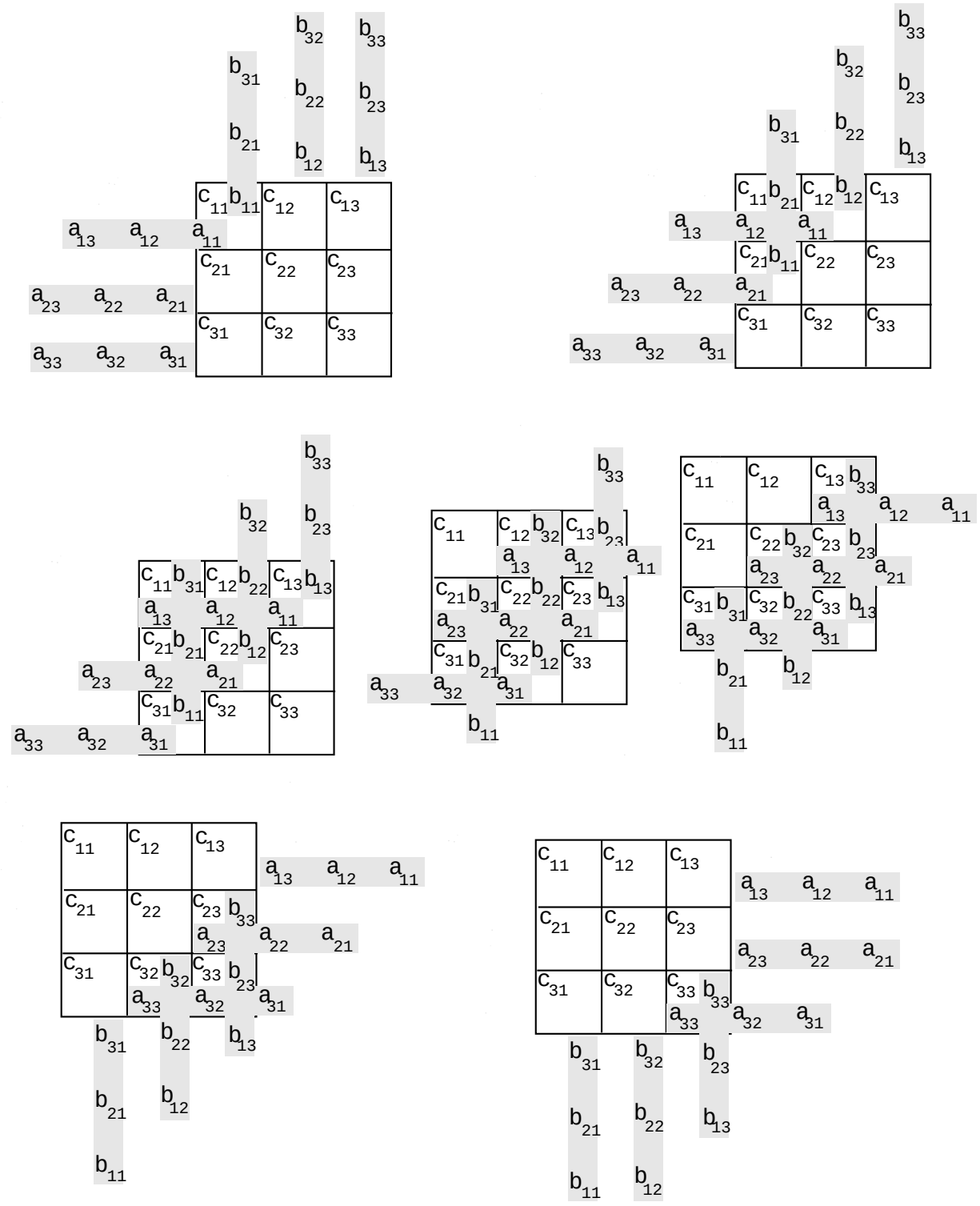


Figure 7: Data movements for 3 by 3 matrix multiplications

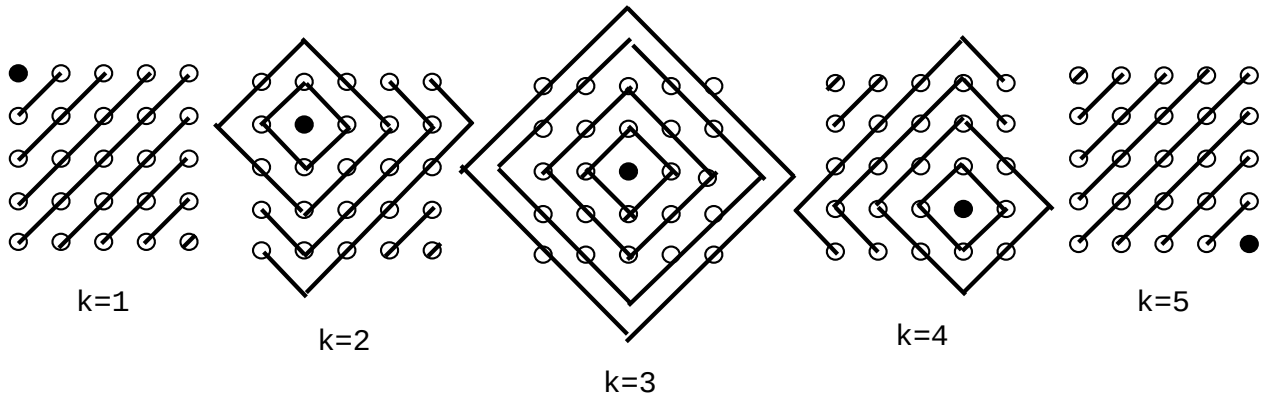


Figure 8: Computation Waves for 5 Node Shortest Path

The sequence constraint and the dependence sequence are $(k, 1)$ and k, since all three dependent elements agree on them. The mapping is $S_{i,j,k} \rightarrow (i, j)$.

The data movements are as follow:
$d_{i,j,k-1}$ has no movement.
$d_{i,k,k-1}$ moves from (i, k) to $(i, k-1), (i, k-2), \ldots, (i, 1)$ and $(i, k+1), (i, k+2), \ldots, (i, n)$.
$d_{k,j,k-1}$ moves from (k, j) to $(k-1, j), (k-2, j), \ldots, (1, j)$ and $(k+1, j), (k+2, j), \ldots, (n, j)$.
The communication delay is

$$|i-k| + k + |j-k| + k$$

The final sequence is

$$|i-k| + |j-k| + 3k$$

The parallel program is

$d_{i,j}^{(k)} \rightarrow (i, j)$, **forall** $1 \leq i, j, k \leq n$
for $K = 3$ **to** $5n - 2$
  $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$,
    forall $1 \leq i, j, k \leq n$ **such that** $|i-k| + |j-k| + 3k = K$
  $d_{ik}^{(k-1)}(i, j) \rightarrow (i, j+1)$,
    forall $1 \leq i, k \leq n \wedge k < j < n$ **such that** $|i-k| + |j-k| + 3k = K$
  $d_{ik}^{(k-1)}(i, j) \rightarrow (i, j-1)$,
    forall $1 \leq i, k \leq n \wedge 1 < j < k$ **such that** $|i-k| + |j-k| + 3k = K$
  $d_{kj}^{(k-1)}(i, j) \rightarrow (i+1, j)$,
    forall $1 \leq j, k \leq n \wedge k < i < n$ **such that** $|i-k| + |j-k| + 3k = K$
  $d_{kj}^{(k-1)}(i, j) \rightarrow (i-1, j)$,
    forall $1 \leq j, k \leq n \wedge 1 < i < k$ **such that** $|i-k| + |j-k| + 3k = K$

Figure 8 show the waves of computations for $n = 5$. There are five waves as shown, each starting where the black dot is. Each wave starts as soon as the previous wave clears out of the way, which is exactly 3 steps after the previous wave starts. So it takes a total of $3 \times (n-1) + 2n - 1 = 5n - 4$ steps to complete the computations.

6 Conclusion

We have presented a method for finding the sequences of parallel computations and communications based on data dependence. We have focused on parallelizing one statement. If there is more than one statement, we can simply find the sequences of parallel computations and communications for each statement and merge them according to their interdependence. For a subclass of problems where the recursive specifications satisfy certain condition, we can produce optimal programs with minimal communications and maximal processor utilizations. These are treated in elsewhere [2].

Because the method is only based on data dependence, it can be used to produce parallel and systolic programs from equational programs or sequential loop programs. The method allow programmers to write simple and elegant programs in equational form and produce efficient parallel and systolic programs automatically. A system based on this method is being developed.

References

1. Allen, R., Callahan, D., and Kennedy, K. Automatic decomposition of scientific programs for parallel execution. *14th ACM Symp. Principles of Programming Languages (POPL).* (1987), pp. 63–76.

2. Allen, R., and Kennedy, K. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9, 4 (Oct. 1987), 491–542.

3. Callahan, D., and Kennedy, K. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2 (1988), 151–169.

4. Chen, L. Deriving parallel and systolic programs from data dependence. Ph.D. thesis, Center for Research in Computing Technology, Harvard University, May 1992.

5. Chen, M., and Choo, Y. Compiling crystal for massively parallel machines. *Proc. the Workshop on Compilation of (Symbolic) Languages for Parallel Computers.* Argonne National Laboratory Report ANL-91/34. Nov. 1991.

6. Fortes, J. A. B., and Moldovan, D. I. Parallelism detection and transformation techniques useful for VLSI algorithms. *J. Parallel and Distrib. Comput.* 2, 3 (August 1985), 277–301.

7. Hudak, P. Para-functional programming in haskell. In Szymanski, B. (Ed). *Parallel Functional Languages and Compilers.* ACM Press, 1991, pp. 159–196.

8. Irigoin, F., and Triolet, R. Dependence approximation and global parallel code generation for nested loops. In Consnard, M., Robert, Y., Quinton, P., and Raynal, M. (Eds.). *Parallel and Distributed Algorithms.* Elsevier Science Publishers B.V., The Netherlands, 1989, pp. 297–308.

9. Kuck, D. J., Kuhn, R. H., Leasure, B., and Wolfe, M. The structure of an advanced retargetable vectorizer. In Hwang, K. (Ed). *Tutorial Supercomputers: Design and applications*. IEEE Society Press, Silver Spring, MD, 1984, pp. 967–974.

10. Padua, D. A., and Wolfe, M. J. Advanced compiler optimizations for supercomputers, *Communications of the ACM*, 29, (Dec. 1986), 1184–1201.

11. Szymanski, B. EPL — parallel programming with recurrent equations. In Szymanski, B. (Ed). *Parallel Functional Languages and Compilers*. ACM Press, 1991, pp. 51–104.

12. Tseng, P. A systolic array parallelizing compiler. *J. Parallel and Distrib. Comput.* 9, 2 (June 1990), 116–127.

13. Zima, H., and Chapman, B. *Supercompilers for Parallel and Vector Computers*. ACM Press, 1990.