CPRE/COMS 583x Adaptive Computing Systems	Fall Semester 1998
October 8, 1998	
Instructor: Arun K. Somani & Akhilesh Tyagi	Scribe: Mark Schlarmann

1 Mapping a Problem onto a Communications Structure

Suppose we have an algorithm that we need to implement in hardware. To achieve an efficient implementation, the communications ability of the computational structure must coincide with the communications requirements of the algorithm. In this section, we will discuss some commonly encountered communications structures and then show how a few algorithms are mapped onto these structures.

2 Commonly Used Communications Structures

A one-dimensional linear array such as the one shown in Figure 1 consists of a linearly linked chain of computational units. Each functional unit can directly communicate with its neighbors only. Therefore, information located in the leftmost node that needs to be processed in the rightmost node must be transmitted through all of the intermediate functional blocks. This process can be very slow for long chains. The limited communications capability of this structure restricts the domain of tasks that can be efficiently implemented on it.

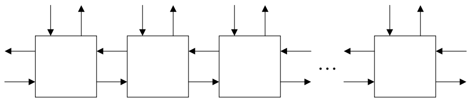

Figure 1: One-dimensional linear array

A simple extension of the linear array to two-dimensions is shown in Figure 2. This structure has improved communication capabilities because each functional unit can communicate with up to four neighbors. Communication can occur both horizontally and vertically.

A structure that facilitates communications with up to six neighbors is shown in Figure 3. This structure is referred to as a hexagonal array.

Sometimes the hexagonal array is drawn in a *squared-up* form as shown in Figure 4. When drawn in this form it is easy to see how the two-dimensional mesh and the hexagonal array differ.

Other possible computation structures include binary trees. Figure 5 contains the traditional diagram of the binary tree and Figure 6 depicts a two-dimensional layout of a binary tree called a *H-tree*. H-tree realizations have an interesting property in that they exhibit the same delay from the root node to nodes that exist on the same level of the hierarchy. As a result, H-trees are commonly used for clock distribution.

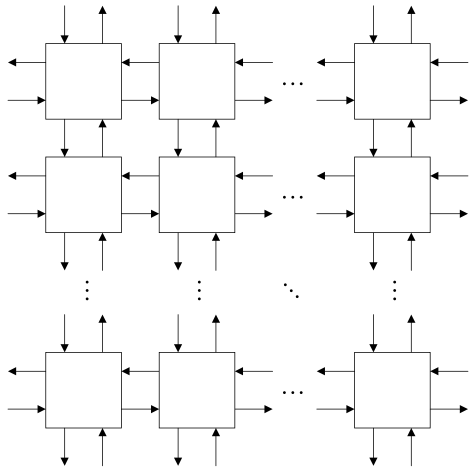

Figure 2: Two-dimensional mesh

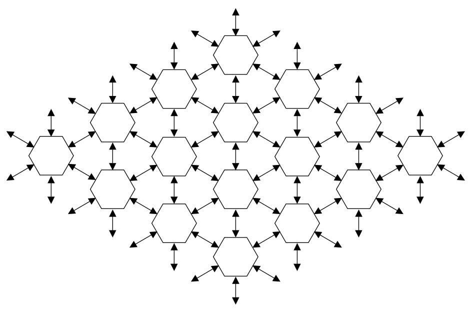

Figure 3: Hexagonal array

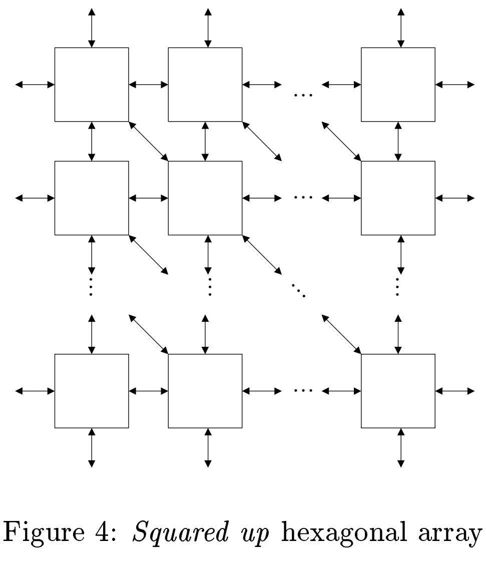

Figure 4: *Squared up* hexagonal array

3 Specific Examples of How Algorithms are Mapped onto Computational Structures

3.1 Matrix-Vector Product

Suppose we want to develop a system that efficiently implements the matrix-vector product shown below:

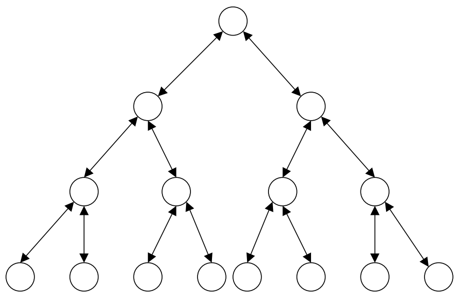

Figure 5: Binary tree

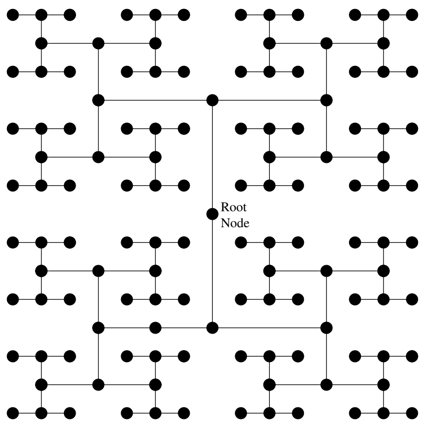

Figure 6: H-tree (2-d layout of a binary tree)

4 Specific Examples of How Algorithms are Mapped onto Computational Structures

4.1 Matrix-Vector Product

Suppose we want to develop a system that efficiently implements the matrix-vector product shown below:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{1}$$

In order to achieve an efficient implementation of this operation, one must arrange the data flow so that it agrees with communications capabilities of the computation structure. The matrix-vector product can be performed on a linear array if the correct data flow is chosen. Figure 7 illustrates the data distribution and flow.

In the configuration of Figure 7 the components of the x vector are assigned consecutively to the functional blocks from left to right. Each of the functional blocks performs a MADD (multiply-addition) operation. The matrix components are fed in a time-delayed fashion into the blocks. The

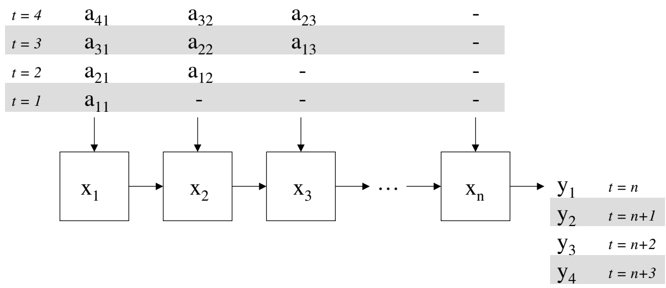

Figure 7: Data distribution and flow for a matrix-vector product implemented on a linear array

partial products are accumulated as the intermediate sum for the i'th row propagates from left to the right. The $n+i-1$'th accumulated sum that emerges from the right side of the array is the i'th row of the result vector y_i. This structure can perform a dense matrix-vector product in $2n$ cycles.

4.2 Banded Matrix-Vector Product

Band matrices are commonly encountered in science and engineering applications. These matrices have a diagonal band of nonzero elements located along the diagonal. All of the other matrix elements are zero. Equation 2 shows an example of a banded matrix-vector product.

$$p\left\{\begin{matrix} \\ \\ \\ \end{matrix}\right. \begin{bmatrix} \overbrace{a_{11} & a_{12}}^{q} & & & & 0 \\ a_{21} & a_{22} & a_{23} & & & \\ a_{31} & a_{32} & a_{33} & a_{34} & & \\ & a_{42} & a_{43} & a_{44} & a_{45} & \\ & & & & \ddots & \\ 0 & & & & & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_n \end{bmatrix} \tag{2}$$

The *lower-bandwidth*, p, refers to the number of diagonal rows below the matrix diagonal including the diagonal. The *upper-bandwidth*, q, refers to the number of diagonal rows above the matrix diagonal including the diagonal. The *bandwidth* of a band matrix is equal to the total number of diagonal rows in the matrix or $p+q-1$. In the example shown in (2) the lower bandwidth is 3, the upper bandwidth is 2, and the overall matrix bandwidth is 4.

As shown in [1], the banded matrix-vector product can be efficiently implemented using a linear array. Figure 8 shows the data flow of the implementation for the example problem posed in (2).

The number of functional blocks required in the array is equal to the bandwidth of the matrix. For the example matrix the bandwidth was 4 so 4 functional blocks appear in Figure 8. Each functional block performs the same basic operations:

$$y_{out} = a_{in}x_{in} + y_{in} \tag{3}$$

and

$$x_{out} = x_{in} \tag{4}$$

The x_i's are fed in the left-hand side every other time step. In accordance with (4), they propagate one position rightward with each clock cycle.

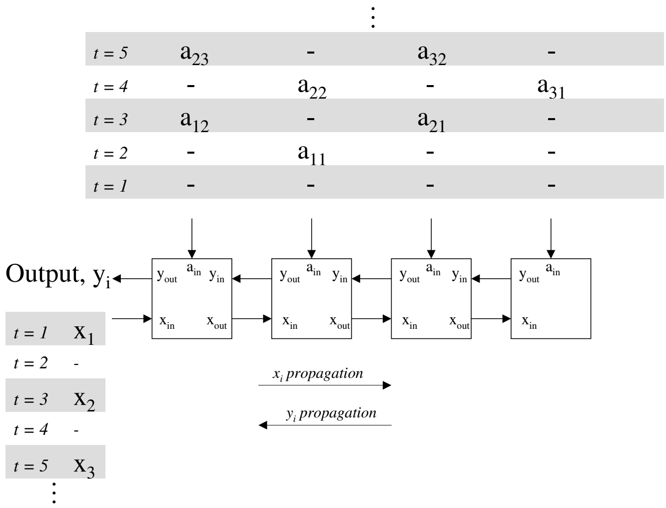

Figure 8: Data flow for a banded matrix-vector product implemented on a linear array

The a_{ij}'s are fed in from the top. They are timed to ensure that a_{ij} appears at one input of a functional block at the same time as x_j appears at the other input. In accordance with (3) the partial products are accumulated as they are passed from the right to the left. The outputs, y_i's, emerge at the left hand side.

4.3 Multiplication of Banded Matrices

Multiplication of banded matrices results in a banded matrix as shown below in (5).

$$\begin{bmatrix} a_{11} & a_{12} & & & 0 \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & a_{43} & \ddots & \\ 0 & & & & \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} & & 0 \\ b_{21} & b_{22} & b_{23} & b_{24} & \\ & b_{32} & b_{33} & b_{34} & \\ & & b_{43} & \ddots & \\ 0 & & & & \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} & & 0 \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & \\ & c_{52} & c_{53} & c_{54} & c_{55} & \\ 0 & & & & & \ddots \end{bmatrix} \tag{5}$$

A hexagonal array can be utilized to perform banded matrix-matrix multiplication [1]. Figure 9 illustrates the data flow through such a system.

Figure 10 shows the designation of the pin names for the functional blocks of Figure 9. Each functional block performs a MADD (multiply-add) operation with each clock cycle:

$$c_{out} = a_{in} b_{in} + c_{in} \tag{6}$$

They also propagate their input data to the next blocks as shown in the following two equations:

$$a_{out} = a_{in} \tag{7}$$

$$b_{out} = b_{in} \tag{8}$$

Referring to Figure 9, the non-zero coefficients of the A and B matrices propagate along diagonal paths through the structure and the C coefficients emerge from the top of the structure. The

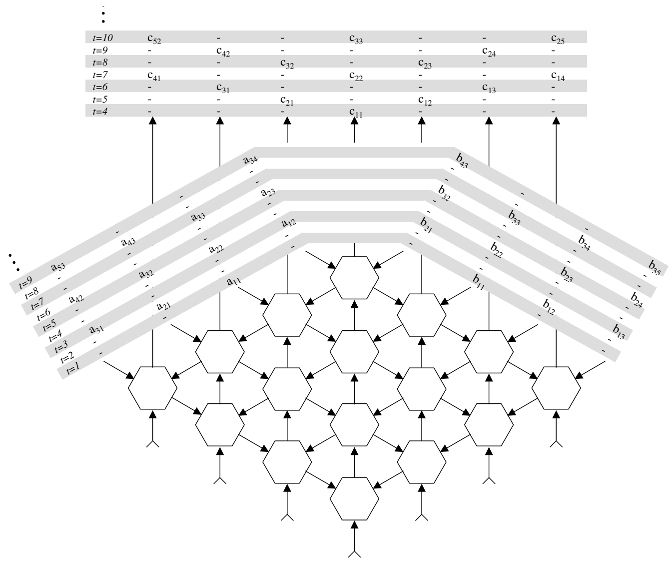

Figure 9: Data flow for the product of 2 banded matrices implemented on a hexagonal array

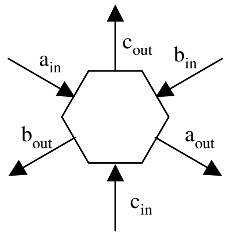

Figure 10: Pin Naming Convention for Functional Blocks in Hex-Array of Figure 9

sequence of input data is chosen to ensure that the correct partial products appear at the right time and in the right column of the structure to ensure that partial products are accumulated correctly.

4.4 Solving Lower-Triangular Systems

Performing a LU factorization is a popular method of solving linear equations. After **A** is factorized into **LU**, solutions for arbitrary y vectors can be inexpensively computed. Part of the solution process is a procedure called *back-substitution.* Back-substitution is the method of solving a lower-triangular set of equations such as those shown in (9).

$$q\left\{\begin{bmatrix} a_{11} & & & & & 0 \\ a_{21} & a_{22} & & & & \\ a_{31} & a_{32} & a_{33} & & & \\ a_{41} & a_{42} & a_{43} & a_{44} & & \\ & a_{52} & a_{53} & a_{54} & a_{55} & \\ 0 & & a_{63} & a_{64} & a_{65} & \ddots \end{bmatrix}\right. \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_n \end{bmatrix} \tag{9}$$

Notice in (9) that we have assumed the matrix is banded with a lower bandwidth equal to q. This assumption makes our solution more general. The case of a dense matrix is the same as a banded matrix with a bandwidth of n.

Solution of (9) is computed by the following recursive relations:

$$
\begin{aligned}
x_1 &= \frac{b_1}{a_{11}} && (10)\\
x_2 &= \frac{b_2 - a_{21}x_1}{a_{22}} && (11)\\
x_3 &= \frac{b_3 - a_{31}x_1 - a_{32}x_2}{a_{33}} && (12)\\
x_4 &= \frac{b_4 - a_{41}x_1 - a_{42}x_2 - a_{43}x_3}{a_{44}} && (13)\\
x_5 &= \frac{b_5 - a_{52}x_2 - a_{52}x_3 - a_{53}x_4}{a_{55}} && (14)\\
&\vdots
\end{aligned}
$$

Figure 11 illustrates how a linear array can be used to implement the relations of (10) through (14) [1].

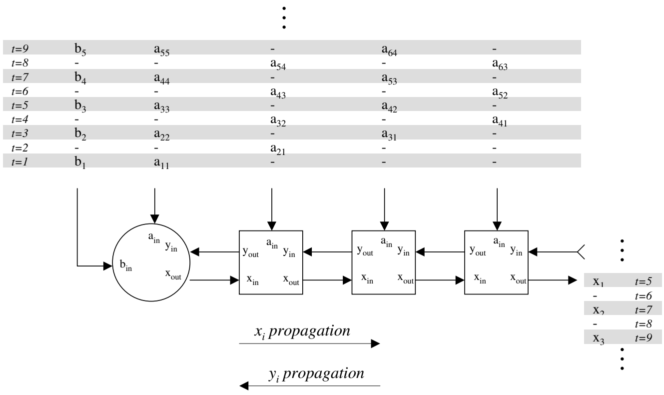

Figure 11: Data flow for the solution of a banded lower-triangular system

The structure consists of one circular block and a linear array of $q-1$ rectangular blocks. The rectangular blocks perform the following operations during each clock cycle:

$$
\begin{aligned}
y_{out} &= a_{in}x_{in} + y_{in} && (15)\\
x_{out} &= x_{in} && (16)
\end{aligned}
$$

and the circular block performs:

$$
x_{out} = \frac{b_{in} - y_{in}}{a_{in}} \quad (17)
$$

$$
(18)
$$

4.5 Ordering Operations/Dependencies

If somebody gives you an algorithm to implement in hardware the problem could be very simple or it could be very complicated depending upon the data dependencies inherent in the algorithm. For example:

```
DO 10 I = 1, 10
10 C[I] = A[I] + B[I]
```

This loop could be implemented with five processors very easily. The processors could concurrently compute the answers because there are no dependencies among the operations. Modern compilers should be able to parallelize loops that are this simple. But what about cases that are more complicated? How can one uncover the dependencies present in an algorithm? There are several techniques that are commonly used.

- Snapshots of the Algorithm - who is doing what at what time?
- Recursive Algorithms - who should be where at what time
- Parallel Code - user identifies parallel portions of the code
- Dependence Graph
- Signal Flow Graph

References

[1] C. Mead and L. Conway, "Introduction to VLSI Systems," Addison-Wesley, Reading, MA, 1980.