7
Parallel Definitive Systems

7.1 Parallel Computer Architectures

The common parallel computer architectures fall into two categories: *shared-memory multiprocessor systems* and *multicomputer systems*. The shared-memory multiprocessor system (see Figure 7-1) comprises a collection of CPUs connected by a bus to a common pool of memory. If a processor wants to access the memory, it must first get permission from other processors. As more processors are added to the system, more competition arises among processors to get access to the memory. This decreases the performance of the system since the processors spend more time on waiting for the memory. This is the so-called von Neumann bottle-neck.

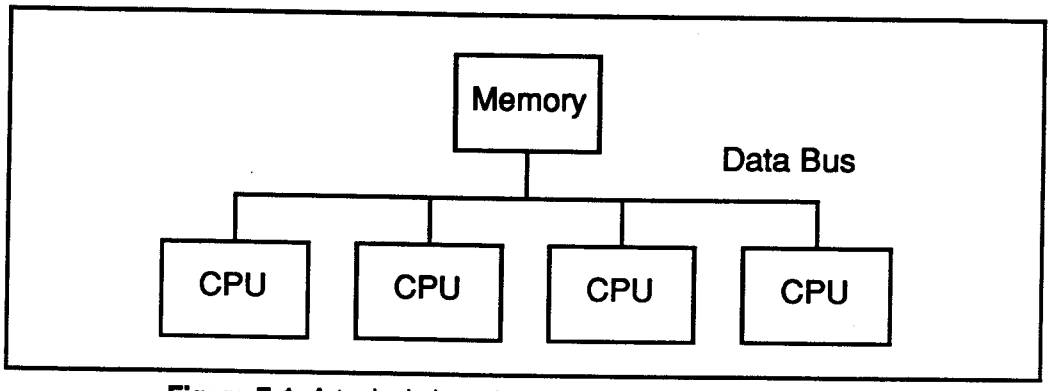


Figure 7-1: A typical shared-memory multiprocessor system
CPUs share a common pool of memory connected to a common data bus.

The multicomputer differs from the multiprocessor in several ways. Each processor of the multicomputer system has its own memory connected to a local data bus. This combination of processor and memory is called a *node*. Processors share their data through high speed serial links (see Figure 7-2). Since the processors do not share a common bus with any other node, there is no bottle-neck. The multicomputer's bandwidth rises linearly with the number of nodes.

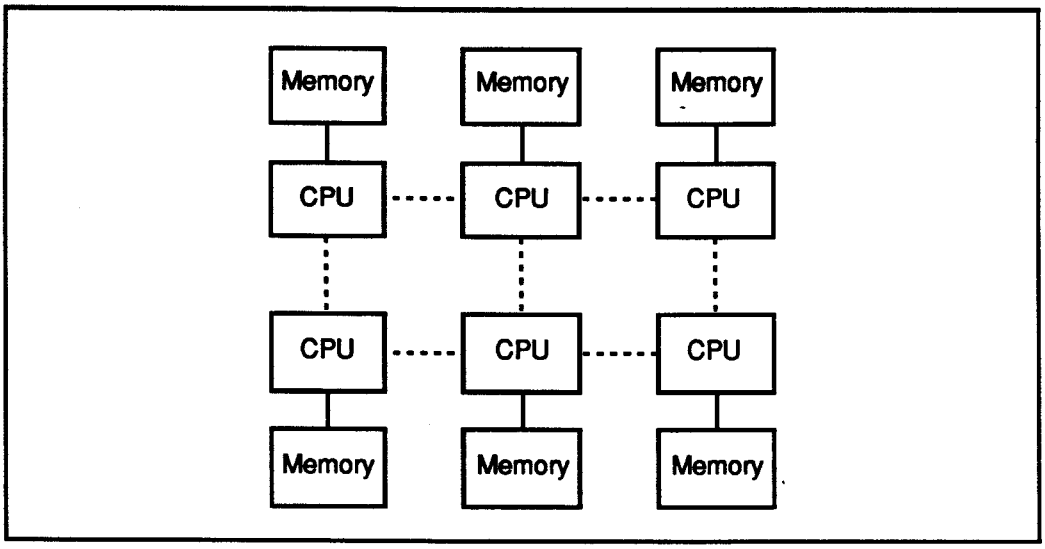


Figure 7-2: A typical multicomputer system
Each node consists of a CPU and memory. Processors communicate with each other through bidirectional serial links.

7.2 Concurrent Programming Languages

> *"It is now rather easy to build the hardware of a multiprocessor computer, but still quite difficult to program it to do useful work. I argue that a key cause of this problem is that the models of computation on which current programming languages are based are inadequate for describing parallelism."* — [Ba87]

Imperative languages closely reflect the von Neumann machine architecture. A program in an imperative language explicitly describes how the computer changes its own state to produce a desired result. The next state is computed from the current state which, in turn, is the result of previous states. Such a program is difficult to analyse to produce the data dependency information which is the key factor of partitioning a program into concurrent processes. Nonetheless, imperative languages can be extended to explicitly describe parallelism.

Declarative languages are based on some mathematical formalisms. These languages avoid the explicit description of the state changes. The data relationships are described in terms of equations or relations. Parallelism can be detected (by hand or by machine) more easily in these languages than in imperative languages. Thus, declarative languages have high potential for implicitly describing parallelism. However, most of the declarative languages are still mainly experimental or not fully exploited in parallel computation [Ba87].

7.3 Parallelism in Definitive Languages

In this section, we try to exploit the potential of parallel computation in definitive languages, especially EDEN.

A definition in definitive languages can be viewed as a composition of *data* and *program*:

$$definition = data + program$$

where *program* is the method of evaluating the *data* and is expressed in mathematical terms. The *program*, theoretically, has no interference with other definitions since the only thing affected is the value (*data*) of the *definition*. This perspective on a definition matches the abstract view of the *node* in the multicomputer system:

$$node = memory + CPU$$

where *memory* stores *data* and *CPU* executes *program*. The data dependency of definitions describes the links of the nodes.

This is different from the abstract model of data-flow machine in which a *node* corresponds to an operator and data flows as a stream from one node to the others through the links.

These differences make it difficult in a data-flow language to re-program (re-connect the links of) a single node at run-time because it will break the data streams. In contrast, the nodes (variables) of definitive languages store the data rather than passing the data to other nodes. Data is sent to other nodes if a node receives query messages from other nodes. Therefore, a parallel definitive system can be interactively modified without restarting execution of the entire system.

A parallel evaluation scheme was presented in §5.3. In this scheme, we use the term "message" to describe inter-processor control signals which synchronize processors' activities.

The *program* of a definition can also be parallelized if the functions in the expression can be evaluated in parallel. If this is not possible, as in the case of EDEN which uses imperative statements to implement functions, a definition

can be broken down into smaller definitions to achieve parallel computation. For instance, consider the following definition:

$$f = ax^3 + bx^2 + cx + d$$

It can be broken down into simpler definitions, ax^3, bx^2, cx, and so forth:

$$\begin{aligned} ax^3 &= ax \cdot x^2 \\ &\text{Â} \quad ax = a \cdot x \\ &\text{Â} \quad x^2 = x \cdot x \end{aligned}$$

$$\begin{aligned} bx^2 &= b \cdot x^2 \\ &\text{Â} \quad x^2 = x \cdot x \end{aligned}$$

$$cx = c \cdot x$$

By eliminating duplicated definitions, we get 8 definitions,

$$\begin{aligned} f &= G + H \\ G &= AXXX + BXX \\ H &= CX + d \\ AXXX &= AX \cdot XX \\ AX &= a \cdot x \\ XX &= x \cdot x \\ BXX &= b \cdot XX \\ CX &= c \cdot x \end{aligned} \tag{7-1}$$

Figure 7-3 shows the dependency graph of these definitions:

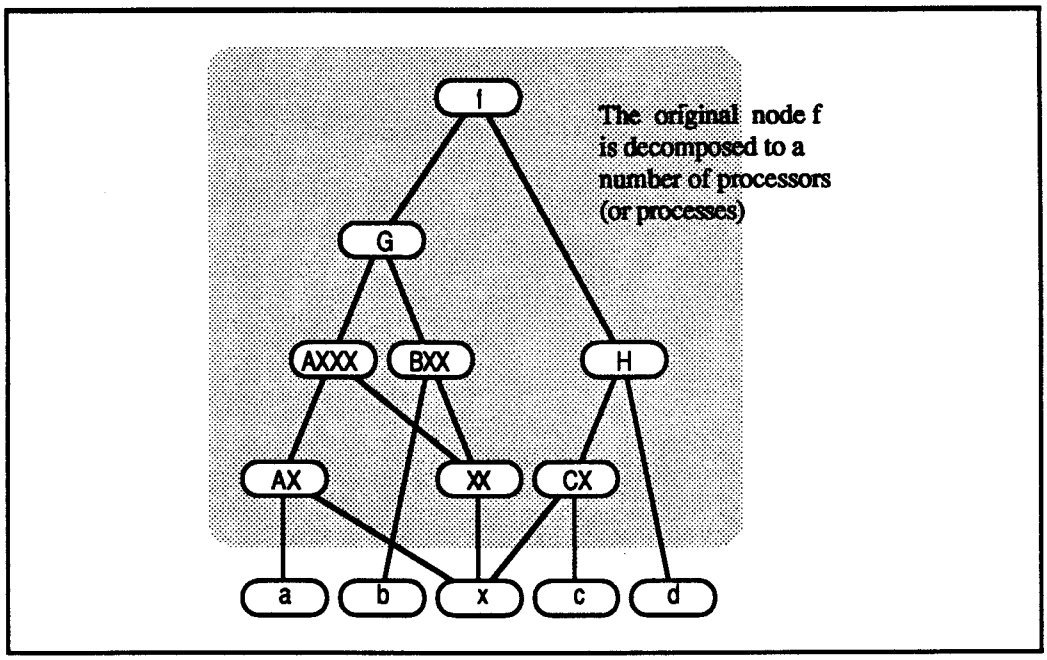


Figure 7-3: The arrangement of processors of the parallel computation of $f = ax^3 + bx^2 + cx + d$

Each node represents a processor node in the multicomputer system. The original node f *is replaced by 8 nodes so that the computation of* $ax^3 + bx^2 + cx + d$ *can now be parallelized.*

From Figure 7-3, it is clear that AX, XX, and CX can be evaluated in parallel because they are not dependent upon each other. Similarly, AXXX, BXX, and H can be evaluated in parallel. However, a, AX, AXXX, G, and f must be evaluated sequentially.

The decomposition of definition can be automated if the compiler knows some mathematical properties, like associative and distributive rules of addition and multiplication.

7.4 Hard-wired Definitive System

The definition in the above example is decomposed into a number of simple definitions, each of which has only one operator (either addition or multiplication) and two operands. This set of definitions can be compiled to silicon level, i.e. hard-wired. The connectivity of the circuit has already been specified by the arcs of the dependency graph. The auto-routing techniques are beyond of the scope of this thesis.

Suppose each operator spends the same amount of time to compute an answer, by plugging in some dummy definitions (e.g. "B = b") and rewriting some definitions (e.g. replace "BXX = b · XX" by "BXX = B · XX"), we can

produce a data pipeline which can be hard-wired (c.f. the example in §4.3). These dummy definitions are *delayers*. The locations of delayers can be found from the dependency graph. An extended topology sort can tell us which nodes can run concurrently. Hence, we can partition the nodes into different *stages* (nodes having the same stage number can run in parallel). There are 4 stages in the polynomial example:

stage 0:	a	b	c	d	e
stage 1:	AX	XX	CX		
stage 2:	AXXX	BXX	H		
stage 3:	G				
stage 4:	f				

If there is an dependency arc joining two nodes M and N, and the stage numbers of M and N, denoted as S_M and S_N respectively, have a difference greater than 1, then $(|S_M - S_N| - 1)$ delayers must be inserted. For example, BXX depends on b, and the stage numbers of BXX and b are respectively 2 and 0. Thus, the number of delayers required is |2–0|–1=1. Similarly, delayers are required between f and H, and between H and d (see Figure 7-4).

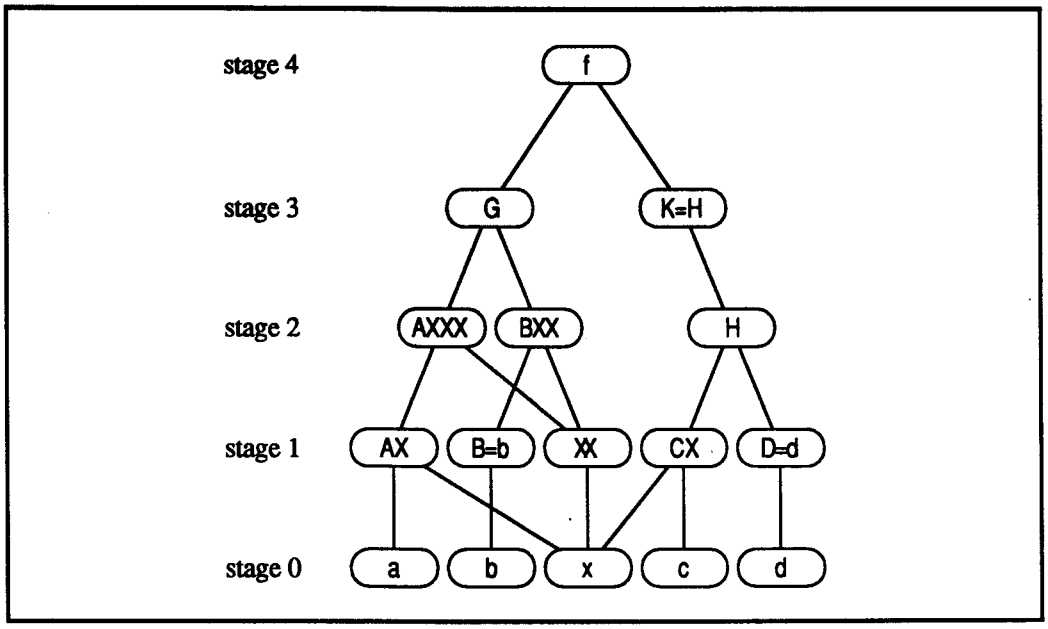


Figure 7-4: The arrangement of a data pipeline derived from Figure 7-3
*Three delayers (*B, D, *and* K*) are inserted to hold data between subsequent stages. (c.f. §4.3)*

If the assumption that each operator requires the same amount of time to compute an answer is false, then some computation time is not utilized. More analysis must be done in this case.

7.5 Problems of Automatic Parallelism Detection

In principle, the decomposition of definition can be done automatically by the compiler. However, the efficiency of the resultant set of definitions is a matter of question. For example, another decomposition of bx^2 in the previous example, might be:

$$bx^2 = bx \cdot x$$
$$\hat{A}\ \ bx = b \cdot x$$

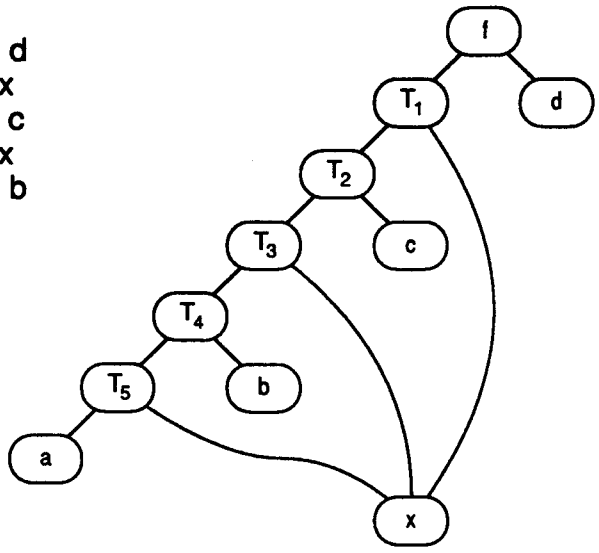
There are then 9, instead of 8, definitions. In other words, we require one more processor node in the actual implementation.

Generally, different ways of decomposition produce different dependency constructions. In the worst case, a concurrent system does no better than a sequential system. For example, the polynomial:

$$f = ax^3 + bx^2 + cx + d$$
$$= ((ax + b)x + c)x + d$$

produces six definitions:

$$\begin{aligned} f &= T_1 + d \\ T_1 &= T_2 \cdot x \\ T_2 &= T_3 + c \\ T_3 &= T_4 \cdot x \\ T_4 &= T_5 + b \\ T_5 &= a \cdot x \end{aligned} \qquad (7\text{-}2)$$



In the special situation, where the values of a, b, c and x are seldom changed but the value of d varies frequently, the decomposition of (7-2) is more efficient than that of (7-1) because it requires only one time unit to compute the answer while (7-1) requires two time units. Hence, it is very difficult to judge whether one decomposition is better than another.

7.6 Related Works

The papers [BSY88] and [Be88] describes an abstract machine model for parallel computation based on definitions. As an example, a systolic array that does matrix multiplication is modelled by a set of definitions and actions.

LSD [Be86b] is a definitive notation for communicating systems that uses concepts originating from **SDL** [BH87]. (A comparison of **LSD** and **SDL** can be found in [BN87].) In the paper [BNS88], an example is presented to illustrate modelling and simulating concurrent systems using definitions.

The paper [RF87] presents a technique for systematically deriving systolic architectures from a general class of recurrence equations. The synthesis process relies on analysing data dependency.

Papers [GL87] and [SSM88] describe some partitioning methods for the high-level data-flow language **SISAL** and translation mechanisms into **OCCAM** [Do88] programs. The partitioning process optimises the data-flow graph generated by the translator.

[Ka87] has some discussion on the translation of processes into circuits.

7.7 Conclusion

Definitions are a natural way to implicitly specify parallel computation. Each definition corresponds to a processor in a multiprocessor system. In principle, we can, manually or automatically, decompose a definition into simpler definitions to further exploit parallelism, and generate software simulations and hardware circuits from the same set of definitions. However, the circuits we can model are restricted to pipeline architectures due to the acyclic property of definitions.