

## Data Dependency

One of the fundamental limitations restricting whether operations can proceed concurrently is the concept of **data dependency**, e.g., a program statement can not proceed until the data values used in that statement are available. There are several types of dependency; some you are “stuck with” and some you are not.

**Flow dependence** This type of dependence is also known as **data dependency**, a.k.a. **true dependency**, a.k.a **read-after-write**. It occurs when a statement (or an instruction) depends on the result of a previous statement (or instruction). For example:

```
//      Assume values for x and y are already computed.
(1)    a = x + y ;
(2)    b = 2 * a ;
```

In this example, statement (2) uses the value **a** produced in statement (1). In this example, there is nothing that can be done to enable statements (1) and (2) to run simultaneously.

**Anti-dependence** Anti-dependence is also known as **write-after-read**. For example:

```
//      Assume values for x, y, and z are already computed.
(1)    a = x + y ;
(2)    b = 2 * a ;
(3)    a = z + 1 ;
```

Statement (3) is anti-dependent on statement (2). Notice that statements (2) and (3) can not be done in reverse order. However, anti-dependence is an example of **name dependency**. I.e., the dependency can be removed by re-naming some variables. For example:

```
//      Assume values for x, y, and z are already computed.
(1)    a = x + y ;
(2)    b = 2 * a ;
(3)    a2 = z + 1 ;

//      Replace all subsequent references to 'a' by 'a2'.
```

**Output dependence** Output dependence is also known as write-after-write. For example:

```
(1)    b = 7      ;
(2)    a = 2 * b  ;
(3)    b = 11     ;
```

Statement (1) is output dependent on statement (3). Notice that the order of these three statements can be not be changed without changing the final value of either **a** or **b**. Output dependence is another form of name dependence. The dependency can be removed by re-naming some variables. For example:

```
(1)    b2 = 7      ; // Where b2 is a new name, not used
(2)    a = 2 * b2 ; // anywhere else in the program.
(3)    b = 11     ;
```

**Control dependence** We start with a very simple example of control dependence:

```
(1)   if ( a == b ) {  
(2)       c = 2 * a ;  
(3)       a = a + b ;  
      }  
(4)   b = a + b      ;
```

The outcome of statement (1) will determine whether statements (2) and (3) will be executed.

Formally, a statement  $S_2$  is said to be control dependent on another statement  $S_1$  if and only if:

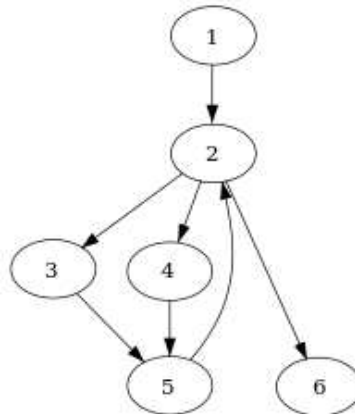
- there exists a path  $P$  from  $S_1$  to  $S_2$  such that every statement  $S_i \neq S_1$  within  $P$  will be followed by  $S_2$  in every possible path to the end of the program, and
- there is an execution path from  $S_1$  to the end of the program that does not go through  $S_2$ .

Control dependence can be expressed more concisely using the concepts of *control flow graphs*, *dominance* and *post-dominance*.

**Definition** A basic block is a sequence of statements with a single entry point, and no branching statements, except possibly the last statement.

**Definition** A control flow graph is a directed graph illustrating the possible paths that execution may take. The nodes of the graph are basic blocks. If it is possible that control may be transferred from block  $b_i$  to  $b_j$  then there exists an edge  $(b_i, b_j)$  in the control flow graph.

An example control flow graph is illustrated below:



**Definition** A node  $d$  dominates a node  $n$  if every path from the entry node to  $n$  must go through  $d$ . Notationally, this is written as  $d \text{ dom } n$  (or sometimes  $d \gg n$ ). By definition, every node dominates itself.

In the example above, node 2 dominates nodes 3, 4, 5, and 6. Notice that node 4 does not dominate node 5.

There are a number of related concepts:

**Definition** A node  $d$  strictly dominates a node  $n$  if  $d$  dominates  $n$  and  $d$  does not equal  $n$ .

**Definition** The immediate dominator or idom of a node  $n$  is the unique node that strictly dominates  $n$  but does not strictly dominate any other node that strictly dominates  $n$ . Every node, except the entry node, has an immediate dominator.

**Definition** The dominance frontier of a node  $d$  is the set of all nodes  $n$  such that  $d$  dominates an immediate predecessor of  $n$ , but  $d$  does not strictly dominate  $n$ . It is the set of nodes where  $d$ 's dominance stops.

**Definition** A dominator tree is a tree where each node's children are those nodes it immediately dominates. Because the immediate dominator is unique, it is a tree. The start node is the root of the tree.

**Definition** A node  $z$  is said to post-dominate a node  $n$  if all paths to the exit node of the graph starting at  $n$  must go through  $z$ .

**Definition** The immediate post-dominator of a node  $n$  is the post-dominator of  $n$  that doesn't strictly post-dominate any other strict post-dominators of  $n$ .

We now re-consider our definition of control dependence:

**Alternate Definition** A statement  $S_2$  is said to be control dependent on another statement  $S_1$  if and only if:

- there exists a path  $P$  from  $S_1$  to  $S_2$  such that every statement  $S_i \neq S_1$  within  $P$  is post-dominated by  $S_2$ , and
- $S_2$  does not post-dominate  $S_1$ .

**Loop-Carried Dependence** A loop-carried dependence is a situation where a computation is data dependent on a value which was created in a previous loop iteration. For example, Newton's Method is an efficient approximation technique for finding the zero crossing<sup>1</sup> of a function. I.e., let  $f : \mathcal{R} \rightarrow \mathcal{R}$  be a scalar function. Then to solve  $f(x) = 0$ , we make an initial guess  $x_0$  and then iterate on  $i$  using the formula:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

where  $f'(x)$  denotes the derivative of  $f$  with respect to  $x$ .

In the case of Newton's method, little can be done about the loop-carried dependence, and in general, one-variable Newton's method does not lend itself well to parallel implementation.

But, loop carried dependence can be deceptive. Consider the prefix sum problem. Given a sequence of numbers:

$$[a_0, a_1, a_2, \dots, a_{n-1}]$$

the **prefix sum** is a sequence:

$$\begin{aligned} s_0 &= a_0 \\ s_i &= \sum_{j=0}^i a_j \quad \text{for } i = 1, 2, 3, \dots, n-1 \end{aligned}$$

---

<sup>1</sup>Trivia question: What 70's musical group has a song that refers to a zero crossing ?

A simple way to compute the sequence  $s_0, s_1, \dots, s_{n-1}$  is as follows:

```
void prefix_sum( int n, int a[], int s[] )
{
    s[0] = a[0] ;
    for ( int i = 1 ; i < n ; i++ ) {
        s[i] = s[i-1] + a[i] ;
    }
}
```

We note that the computational complexity of function `prefix_sum()` is obviously  $\mathcal{O}(n)$ .

Function `prefix_sum()` clearly has a loop-carried dependence: the computation of  $s[i]$  requires the value  $s[i-1]$  which was computed on the previous loop iteration. It appears that there is little hope of parallelizing this computation.

But, appearances can be deceiving. One aspect of the art of designing parallel algorithms is to re-think the problem from the beginning. One of the pitfalls in parallel computation is that our minds can become fixated on an obvious sequential approach that does not lead to a parallel version. It is often too easy to convince ourselves that nothing more can be done.

For the prefix sum problem, we can make use of commutativity and associativity of sums.

## Parallel Prefix Sum

Figure 1 illustrates the operations needed to compute parallel prefix sums with  $n = 16$ . The computation proceeds in a sequence of stages. At each stage, some addition operations can be performed in parallel. Notice there are  $2 \log_2(n) - 1$  stages, so the parallel time complexity is  $\mathcal{O}(\log n)$ .

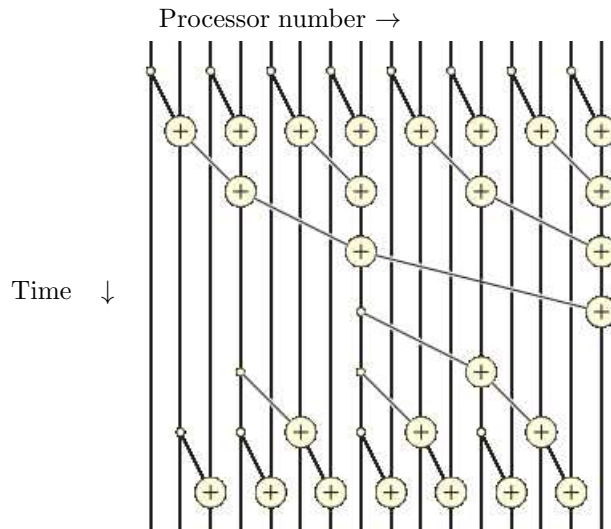


Figure 1: Parallel prefix computation with  $n = 16$ .

The careful reader will notice that some operations in the lower half of Figure 1 do not have to wait until all of the operations in the upper portion are complete. In fact, only  $\log_2(n)$  stages are needed as shown in the following pseudocode parallel prefix algorithm.

## Parallel Prefix Sum

Input: A number  $n$  of the form  $n = 2^k$  for some  $k \geq 0$  and an array  $A$  of length  $n$ .

Output: The sequence of prefix sums stored in array  $S$ .

Number of Processors:  $n$ . Processors ID numbers are  $0, 1, 2, \dots, n-1$

Private variables: Processor ID number,  $p$   
Loop control variable  $i$   
Temporary variable  $t$

Shared variables: Arrays  $A[]$  and  $S[]$

Method:

```
Each processor  $p$  performs the following in parallel:
S[p] = A[p] ;
synchronize ; // Barrier synchronization
for  $i = 1$  to  $k$  do { // Recall  $k = \log_2(n)$ 
    if (  $p \geq 2^i$  ) {
         $t = S[p] + S[p - 2^i]$  ;
    }
    synchronize ; // Barrier synchronization
    if (  $p \geq 2^i$  ) {
         $S[p] = t$  ;
    }
    synchronize ; // Barrier synchronization
}
output S
```

## Discussion Question

The *Fibonacci* numbers are defined by:

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \quad \text{for } n \geq 2 \end{aligned}$$

A simple sequential implementation is as follows:

```
void fib( int n, int f[] )
{
    f[0] = 0 ;
    f[1] = 1 ;
    for ( int i = 2 ; i <= n ; i++ ) {
        f[i] = f[i-1] + f[i-2] ;
    }
}
```

Function `fib()` has a loop carried dependence over two consecutive prior iterations.

**Can the Fibonacci numbers be computed in parallel ?**