15-499: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Parallel Models II **Date:** 01/20/09
Scribe: Philippe Ajoux

3.1 Programming Models

For the first portion of the class we are concerned with "Nested Parallel" algorithms and programs. The following programming constructs allow us to describe such parallelism.

- **Parallel Loop**: To show a parallel loop on the items of A,

  ```
  for x in A { ... }
  ```

- **Parallel Evaluation**: To show that two statements, s_1 and s_2, can be run in parallel,

  ```
  s1 || s2
  ```

Note that these constructs lead to *series parallel DAGs* as described in the previous lecture. Below are some programing languages that provide such constructs.

- **Nesl** - A functional programing language.
- **Cilk++** - A C++ based language with a race condition check (in beta).
- **X10** - An IBM Java type language.
- **OpenMP** - Widely used but "flawed".

Example: Merge sort.

```
function msort(A)
{
    if |A| <= 1 then A;
    else
    {
        b = msort(bot(A)) || t = msort(top(A));
        return merge(b,t);
    }
}
```

Assuming that *merge* takes $O(n)$ work and $O(\log^2 n)$ depth, we can write the recursion relations for depth and work of *msort*:

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) = O(n \log n) \\ D(n) &= D(n/2) + O(\log^2 n) = O(\log^3 n) \end{aligned}$$

To calculate the depth, we must take the maximum of the recursive calls. However, each call operates on the same size data and takes the same time complexity. Thus, we simply add the work of one of the recursive *msort* calls. We must also add the depth of the *merge*. A simple unrolling yields $D(n) = O(\log^3 n)$.

3.2 Correctness

Correctness of a parallel algorithm is important. If an algorithm doesn't work, there is not point to optimize it. In order to discuss the correctness of parallel algorithms we will introduce some definitions.

- **Ordered Operations** - Two operations A and B are ordered if there is a path in the DAG from $A \rightarrow B$ (in which case, we use the notation $A \prec B$) or $B \rightarrow A (B \prec A)$.
- **Sequential Ordering** - A sequential ordering of a parallel program is an any sequence S which includes all the instructions of the parallel program such that if $A \prec B$ in the parallel program, then instruction A occurs before B in the sequence S. Intuitively, it is any instruction-by-instruction execution of the paralllel program that does not violate the constraints imposed by the program's DAG. Consider, for instance, fig. 3.2. Both $\langle A, B, C, D\rangle$ and $\langle A, C, B, D\rangle$ are valid sequential orderings, where as $\langle A, B, D, C\rangle$ is not as it violates the constraint $C \prec D$. As a more concrete example, in our *msort* code, the parallel recursive calls could be executed from left to right in sequence to get a sequential ordering.
- **Concurrent Operations** - Two operations A and B are concurrent if there does not exist a path in the DAG from $A \rightarrow B$ or $B \rightarrow A$.

  Example: Consider the two operations A and D in the DAG in fig. 3.2. There is a path (red) from $A \rightarrow D$. Thus, A and D are *ordered*. However, B and C are *concurrent* because there is no path from $B \rightarrow C$ or $C \rightarrow B$.
- **Concurrent Read (CR)** - Two concurrent operations read from the same memory location.
- **Concurrent Write (CW)** - Two concurrent operations write to the same memory location.
- **Concurrent Read-Write (CRW)** - Two concurrent operations, access the same location; one does a read, and the other a write.

Race conditions are conditions where the order of execution of instructions in the parallel machine affects the outcome of the computation. Since a parallel program does not impose any constraints on the order of execution of concurrent operations, a parallel machine could schedule them in any order it chooses. If such concurrent instructions happen to read and write to the same memory location, then the order in which they are executed dictates the values that have been read/written by the concurrent instructions. We can now make some remarks about parallel programs.

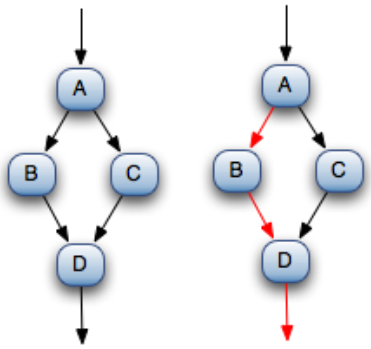

Figure 3.2.1: Parallel Program DAG

- *A parallel program is **race free** if it has no CWs or CRWs.*
- *A parallel program has **write races** if it has no CRWs.*

★ *In a **race free** parallel program, we only need to consider a sequential ordering to argue correctness. This is because, in the abscence of races, any valid execution order of the instructions yields the same output. So we can take a simple ordering, sequential, and argue that it does exactly what the program is intended to do. This would guarantee the correctness of the parallel program.*

So, if we ensure a parallel program is race free, then proving its correctness is the same as proving correctness for a sequential algorithm, which is well understood. How can we guarantee a race free parallel algorithm?

- Use a functional programing language (Nesl, ML). There is no concept of writes/side-affects.
- Use a race-detector to check if the program has races (Cilk++).

3.3 Efficiency

In order to discuss efficiency, we define some new terms:

- **Work Efficient** - A parallel algorithm is work efficient if $W(n)$ is asymptotically the same as the best known $T(n)$ for a sequential algorithm for the same problem.
- **Parallelism** $= \frac{W}{D}$ - A measure of the number of processor that can be used for the algorithm.
- **Scalable** - A parallel program is *scalably parallel* if,

$$\lim_{n \to \infty} \frac{W(n)}{D(n)} = \infty$$

Example: Quick sort without a parallelized partion.

$$\begin{aligned} W(n) &= O(n \log n) \\ D(n) &= O(n) \\ \lim_{n\to\infty} \frac{n \log n}{n} &= \lim_{n\to\infty} \log n = \infty \end{aligned}$$

This is scalable. However, it does not scale as fast as quick sort with a parallelized partition that has a $D(n) = O(\log n)$.

Generally, when designing a parallel algorithm, one should follow the two steps:

1. Make sure it is work efficient (or close to it).
2. Minimize the depth.

Aside. If a *greedy scheduler* with P processor is used, then we know that,

$$T \leq \frac{W}{P} + D \qquad T \geq \max\left(D, \frac{W}{P}\right)$$

Notice that these are relatively tight bounds. The lower bound will be within a factor of 2 of the upper bound if $D = W/P$. However, this is usually not the case, and the lower bound is closer than a factor of 2 from the upper bound.

Note: This will be proved later in class.

Example: It is important when considering efficiency to get an algorithm *work efficient* before minimizing depth. The following example from the early days of parallelism (80s) shows this mistake.

A company developed a "logic simulator" (useful for companies like INTEL) on a 64,000 processor machine. The idea was to have each processor simulate a logic gate. Each processor would, at each step, process the input of the logic gate and produce an output. The output would then be relayed as input to the appropriate processor. After completion of this logic simulator, it was found that it ran about 10x slower than a single-processor sequential simulator! At the time, the company could not understand why this was happening. The reason lay behind the *work efficiency*. While a logic circuit is "parallel" there is an implicit sequential operation as each logic gate must wait for input from another logic gate. As it turns out, for each step, only about 2-3% of the gates would flip bits during a step. On a sequential simulator, only those logic gates were processed. However, the huge 64,000 simulator would process everything.