15-499: Parallel Algorithms **Lecturer:** Guy Blelloch
Topic: Introduction **Date:** Jan 13, 2009
Scribe: Kanat, edited by Harsha Simhadri

1.1 Parallelism is ubiquitous

Parallelism exists at most levels in computers from logic gates up to the level of the internet. Parallelism can be found in:

- Logic gates: Implementing any instruction to the processor involves simultaneous operations at millions of logic gates.
- Pipelining: A processor typically executes several instructions at a time to make efficient use of all the functional units on the chip.
- MMX/Vector instructions: Some chips, especially those built for graphics processing, are designed to do vector arithmetic using paralelly.
- Hyperthreading: Running multiple threads on the same core to use multiple functional units and hide latency.
- Multi-cores.
- Shared memory systems.
- Clusters.
- Internet.

This course primarily deals with parallelism above the shared memory systems level and places emphasis on parallelism in multi-core systems.

1.2 The future of the chip

Processor design has already hit the limit in terms of clock speed and increasing the number of processing unts is the only evident method to increase processor performance. The number of cores per processor might soon go in to the hundreds thus requiring tha programs exhibit a high degree of parallelism.

Parallelism also helps increase the performance to power ratio. Decreasing the clock speed and increasing the number of cores in current processors beats increasing clock speeds in terms of the performance-power ratios by significant margins. This makes for the case of using parallelism as a tool to build chips for low power portable devices in addition to high performance computing where it is currently used.

1.3 The sequential machine model

While thinking of sequential algorithms, we normally use the RAM model. In this model, we assume that the machine executes one instruction in a time step and can access any memory location with in the time step. While this does not model real world systems which have large non-uniform memory access latencies, this model helps simplify algorithm design and analysis. In the next lecture, we will look at possible choices for parallel machine models.

1.4 Finding Parallelism

Let us look at how two common algorithmic problems can be parallelised.

1.4.1 Matrix Multiply

To multiply two $n \times n$ size matrices A, B, a normal sequential program would do:

```
1: for $i = 1$ to $n$ do
2:    for $j = 1$ to $n$ do
3:       $C_{ij} = 0$;
4:       for $k = 1$ to $n$ do
5:          $C_{ij} = C_{ij} + A_{ik} \cdot B_{kj}$;
6:       end for
7:    end for
8: end for
```

This algorithm would require $O(n^3)$ time for execution, because each of the 3 loops are executed sequentially. However, we note that the two outer loops can be completely parallelised as computations for different C_{ij}s are completely independent. We can thus run n^2 independent parallel loops, one for each C_{ij}, $1 \leq i, j \leq n$. This would still mean that we need atleast $O(n)$ steps to multiply A and B even if we have arbitrarily large parallel computing power.

We now go on to note that C_{ij} is the vector dot product of row A_i and column B_j and the inner most loop of the above algorithm is just one way to implement a vector dot product operation. However, we can make use of the associativity of the addition operation to implement the dot product much faster. To compute $a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in}b_{nj}$, instead of using the order $(((a_{i1}b_{1j} + a_{i2}b_{2j}) + a_{i3}b_{3j}) + \cdots + a_{in}b_{nj})$, we can use the recursive addition order indicated in fig. 1.4.1. The numbers $a_{ik}b_{kj}, 1 \leq k \leq n$ form the leaves of the tree. The value of each of the nodes can be computed recursively by summing up the values of its children. This recursive computation takes only $O(\log n)$ steps if given enough computing power. Thus, we have a parallel algorithm that can multiply two matrices in logarithmic time.

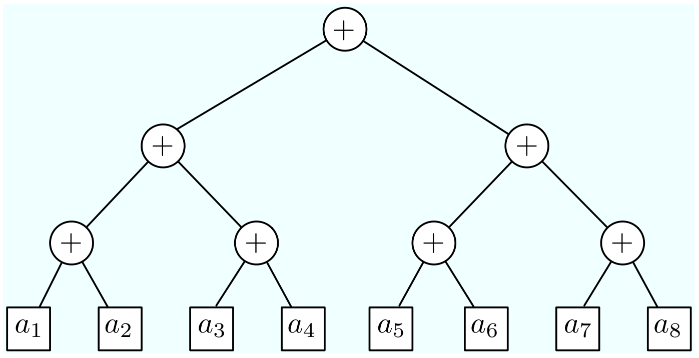

Figure 1.4.1: A perfect binary tree for summing $n = 8$ numbers.

1.4.2 Quick sort

Quick sort is a popular divide-and-conquer, comparison-based sorting algorithm. When the pivots are chosen uniformly at random, we know that quick sort makes $O(n \log n)$ comparisons with high probability[1]. Below is a pseudo-code for sequential quick sort:

```
QSort($A$) =
  **if** $|A| \leq 1$ **return** $A$
  **else**
    $p = A[\text{rand}(|A|)]$;
    **return** QSort($\{x \in A : x < p\}$)++$\{p\}$++QSort($\{x \in A : x > p\}$);
```

We see that the two recursive calls made at each level level of the recursion are independent and can be done in parallel. Assuming that quick sort always splits the array in half every time, this gives us a parallel algorithm that takes $O(n)$ steps (obtained by solving the recursion $T(n) = T(n/2) + n$, n steps being required for partitioning elements larger than p from elements smaller than p at each level of recursion). With a more careful analysis, we can show that this vesion of randomized quick sort takes $O(n)$ time to complete with high probability.

After all this is not so impressive—having infinite processors only yields $\log n$ speed up. When we examine this code further, we see that we could potentially parallelize the partitioning routine. As we will see later in the course, partitioning (i.e., constructing the array $\{x \in A : x < p\}$) can be accomplished in parallel in time time $O(\log n)$ (given enough computing power). This will give us a version of parallel quick sort which completes in $O(\log^2 n)$ time steps with high probability.

[1]An event $\mathcal{E}$ occurs with high probability if $\mathbf{Pr}[\mathcal{E}] \geq 1 - O(n^{-c})$ for some constant $c \geq 1$.