CME 323: Distributed Algorithms and Optimization, Spring 2016
`http://stanford.edu/~rezab/dao`.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 1, 3/28/2016. Scribed by Jane Bae, Sheema Usmani, Andreas Santucci.

1 Overview

Course Outline In the first half of the class, we're going to look at the history of *parallel* computing and pick up the notable breakthroughs from the last several decades. The last half of the class will be focused on the novel algorithmic developments in *distributed* computing. To really talk about distributed computing (which involves network communication overhead), we need to understand parallel computing on a single machine with multiple processors and shared Random Access Memory.

Why focus on parallel algorithms? The regular CPU clock-speed used to double every several years. This phenomena tapered off after CPU's reached around 3 Gigahertz. Now, machines have many cores. Because computation has shifted from sequential to parallel, our algorithms have to change as well.

Efficiency of Parallel Algorithms Even *notions of efficiency* have to change. We can no longer look at the number of operations that an algorithm takes to estimate the wall-clock compute time. Algorithms which work well on a parallel machine are very different from those which work well on a sequential machine.

Course Goals The first lectures will outline measures of complexity for parallel algorithms, focusing on a *work-depth* (single-machine) model. At the end of the course, we'll see how we can string together thousands of machines, each with several CPU's and a shared memory module, and distribute our computations across the network. We'll learn a new module which lets us analyze these communication costs.

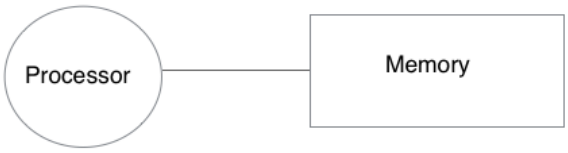
2 Models of Computation on a Single Machine

We first overview our traditional model of computation for a single machine, then discuss a parallel model for computation.

Sequential RAM Model

Random Access Memory (RAM) model is the typical sequential model. We are given a processor p_1, and it's attached to a memory module m_1. The processor p_1 can read and write to (any position in)memory *in constant time*. This is the model we are used when using `loop`s in C or Java.

Figure 1: Sequential RAM Model



We will *not* focus on this model in class.

Parallel RAM Model

In a Parallel RAM (PRAM) Model, we always have multiple processors. But how these processors interact with the memory module(s) may have different variants, explained in the caveat below.

Caveat - Variants of PRAM Models We might have multiple processors which can all access a single large chunk of memory. We can do anything we could in the Sequential RAM Model, with one caveat: when two processors want to access the same location in memory at the same time (whether its read or write), we need to come up with a resolution. What type of resolution we come up with dictates what kind of parallel model we use. The different ways a PRAM model can be set up is the subset of the following combinations:

$$\{\text{Exclusive}, \text{Concurrent}\} \times \{\text{Read}, \text{Write}\}.$$

Of these combinations, the *most popular model* is the concurrent read and exclusive write model. However, sometimes concurrent write is used. [1]

Resolving Concurrent Writes Here are the ways to deal with concurrent write:

- Undefined/Garbage e.g. Machine could die or results could be garbage.
- Arbitrary no specific rule on which processor gets write-priority, e.g. if $p_1, p_2, \ldots, p_j$ all try to write to same location, arbitrarily give preference to *exactly one* of them
- Priority specific rule on which processors gets to write, e.g. preference p_i over $p_j \iff i \leq j$.
- Combination combining values being written, e.g. max or logical `or` of bit-values written.

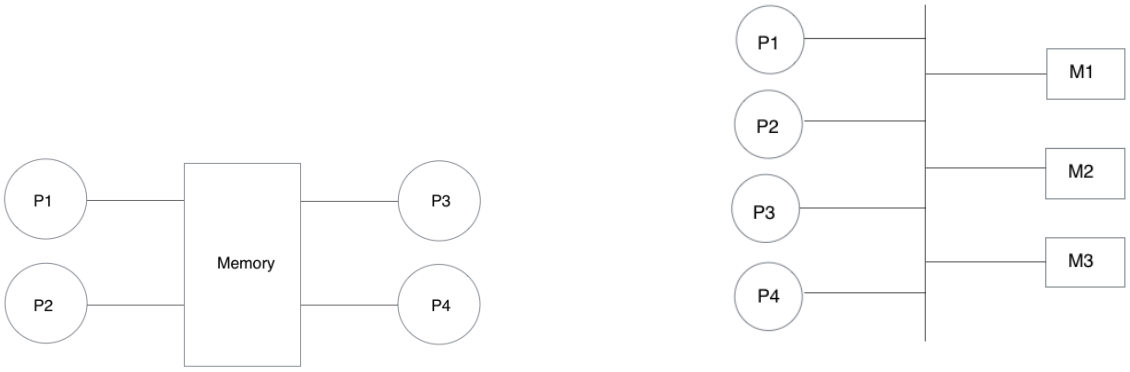
We note that there is still a clock which ticks for all processors at the same time. At each clock-tick, each processor reads or writes to memory. Hence it is in fact possible for several processors to concurrently attempt to access a piece of memory (at the *exact* same time).

[1] We note that some of these combinations don't make sense. For example, exclusive read and concurrent writes.

Pros and Cons of Model Variants In the world of sequential algorithms, the model is fixed. In the parallel world, we have the flexibility to decide which variant of the PRAM model to use. The benefits of this is that the algorithm designer may choose a model which tailors to their application. The downside is that comparing algorithms across models is difficult. We'll see how a parallel algorithm's efficiency can be orders of magnitude different using different models for parallel computation. As an algorithm *designer*, you should advertise the model which you think your algorithm which will be used on the most.

Generic PRAM Models Below, we picture two examples of PRAM models. On the left, we have a *multi-core* model, where multiple processors can access a single shared memory module. This is the model of computation used in our phones today. On the right, we have a machine with multiple processors connected to multiple memory modules via a bus. This generalizes to a model in which each processor can access memory from *any* of the memory modules, or a model in which each processor has its own memory module, which cannot be directly accessed by other processors but instead may be accessed indirectly by communicating with the corresponding processor.

Figure 2: Examples of PRAM



In practice, either *arbitrary* or *garbage* is used; these are both reasonable. We will not tie ourselves to what any one manufacturer chooses for their assembly-level instruction set.

Further Complications - Cache In reality, there are additionally (several layers of) *cache* in between each processor and each memory module, which all have different read-write speeds. This further complicates things. We won't press this last point much further.

3 Brent's Theorem

We now move toward developing theory on top of these models. In RAM Models, we typically count the number of operations and assert that the time taken for the algorithm is proportional to the number of operations required. I.e.,

$$\text{number operations} \propto \text{time taken}$$

This *no longer holds* in a parallel model. In a PRAM, we have to wait for the slowest processor to finish all of its computation before we can declare the entire computation to be done. This is known as the *depth* of an algorithm. Let

$$T_1 = \text{amount of (wall-clock) time algorithm takes on one processor, and}$$
$$T_p = \text{amount of (wall-clock) time algorithm takes on } p \text{ processors.}$$

Practical Implications of Work and Depth In general, work is the most important measure of the cost of an algorithm. We argue as follows: the cost of a computer is proportional to the number of processors on that computer. The cost for purchasing time on a computer is proportional to the cost of the computer multiplied by the amount of time used. Hence the total cost of a computation is proportional to the number of processors in the computer times the amount of time required to complete all computations. This last product is the *work* of an algorithm.

Relating Work and Depth Note that in the best case, the total work required by the algorithm is evenly divided between the p processors; hence in this case the amount of time taken by each processor is evenly distributed as well. Fundamentally, T_p is lower bounded by

$$\frac{T_1}{p} \leq T_p,$$

where the equality gives the best case scenario.[2]

This is the lower bound we are trying to achieve as algorithm designers. We now build some theory in order to determine a useful upper bound on T_p.

Relating Depth to PRAM with Infinite Processors In establishing theory, it's relevant to ask: what happens when we have infinitude of processors? One might suspect the compute time for an algorithm would then be zero. But this is often not the case, because algorithms usually have an *inherently* sequential component to them.

Representing Algorithms as DAG's It is natural to represent the dependencies between operations in an algorithm using a directed acyclic graph (DAG). Specifically, each fundamental unit of computation is represented by a node. We draw a directed arc from node u to node v if computation u is required as an *input* to computation v. We trivially note that our resulting

[2]To see this, assume toward contradiction that $T_p < T_1/p$, i.e. that $T_p \cdot p < T_1$. Note that T_p describes the time it takes for the entire algorithm to finish on p processors, i.e. the processor with the most work takes T_p time to finish. There are p processors. Hence if we were to perform all operations in serial it should require at most $T_p \cdot p$ time. But if $T_p \cdot p < T_1$, then we get a contradiction, for T_1 should represent the time it takes to complete the algorithm on a sequential machine.

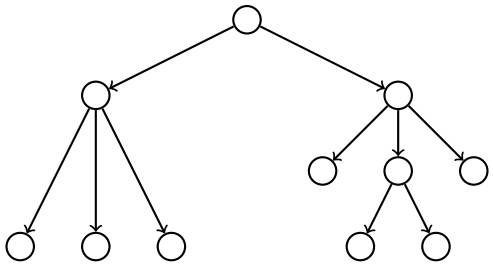


Figure 3: Example Directed Acyclic Graph (DAG)

graph surely ought to be connected (otherwise the isolated node[s] represent useless computations), and by definition it's acyclic. Hence its a tree. The root of the tree represents the output of our algorithm, and its children are the dependencies.

Since our DAG is a tree, it's natural to talk about levels of the tree. Let the root of the tree (i.e. the output of the algorithm) have depth 0, its children depth 1, and so on. Suppose m_i denotes the number of operations (or nodes) performed in level i of the DAG. Each of the m_i operations may be computed concurrently, i.e. no computation depends on another in the same layer. Operations in different levels of the DAG *may not* be computed in parallel. For any node, the computation cannot begin until *all* its children have finished their computations.

In a sequential machine, it's obvious what we must do to execute an algorithm. We look for the *leaves* of the tree, since these depend on no prior computations. We may evaluate all of the leaf nodes and continue through each layer of the DAG until we reach the root node (i.e. return the output).

How long does it take to execute the algorithm sequentially? Clearly, it takes time proportional to the number of nodes in the graph (assuming each node represents a fundamental unit of computation which takes constant time). So, we define *work* to be

$$T_1 = \text{number of nodes in DAG.}$$

How much time would it take to execute the algorithm given an infinitude of processors? Clearly, the depth of the DAG. At each level i, if there are m_i operations we may use m_i processors to compute all results in constant time. We may then pass on the results to the next level, use as many processors as required to compute all results in parallel in constant time again, and repeat for each layer in the DAG. Note that we cannot do better than this.[3] So, with an infinitude of processors, the compute time is given by the depth of the tree. We then define *depth* to be

$$T_\infty = \text{depth of computation DAG.}$$

[3]One feature that is considered universal across all machines is that at the fundamental hardware level, there is a discrete clock tick, and each operation executes exactly in sync with the clock. This feature, combined with the properties of our DAG, explain why the depth is a lower bound on compute time with infinite processors.

What's the point of T_∞? Nobody has an infinitude of processors available to them! It's useless in its own right.

Theorem 3.1 (Brent's theorem) *We claim that with T_1, T_p, T_∞ defined as above,*

$$\frac{T_1}{p} \leq T_p \leq \frac{T_1}{p} + T_\infty.$$

Since T_1/p optimal, we see that T_∞ is in fact useful for giving us a handle on how far off our algorithm performs relative to the best possible version of the parallel algorithm. In a sense, T_∞ itself measures *how parallel* an algorithm is.

This theorem is powerful, it tells us that if we can figure out how well our algorithm performs on an infinite number of processors, we can figure out how well it performs for any arbitrary number of processors. Ideally, we would go through each T_p to get a sense of how well our algorithm performs, but this is unrealistic.

Proof: On level i of our DAG, there are m_i operations. Hence by that definition, since T_1 is the total work of our algorithm,

$$T_1 = \sum_{i=1}^{n} m_i$$

where we denote $T_\infty = n$. For each level i of the DAG, the time taken by p processors is given as

$$T_p^i = \left\lceil \frac{m_i}{p} \right\rceil \leq \frac{m_i}{p} + 1.$$

This equality follows from the fact that there are m_i constant-time operations to be performed at m_i, and once all lower levels have been completed, these operations share no inter-dependencies. Thus, we may distribute or assign operations uniformly to our processors. The ceiling follows from the fact that if the number of processors not divisible by p, we require exactly one wall-clock cycle where some but not all processors are used in parallel. Then,

$$T_p = \sum_{i=1}^{n} T_p^i \leq \sum_{i=1}^{n} \left(\frac{m_i}{p} + 1 \right) = \frac{T_1}{p} + T_\infty$$

■

So, if we *analyze* T_∞, and if we understand the sequential analysis of our algorithm which gives us T_1, we have useful bounds on how well our algorithm will perform on any arbitrary number of processors.

We lastly note that using more processors *can't worsen* run-time. That is, if $p_1 > p_2$, then $T_{p_1} \leq T_{p_2}$ (since we can always allow some processors to idle). Hence, the goal in the *work-depth model* is to design algorithms which work well on an infinitude of processors, since this minimizes T_∞, which in turn gets us closer to our desired optimal bound of T_1/p.[4]

[4]So although processors are a valuable resource, the work-depth model encourages us to imagine we have as many as we want at our disposal.

4 Scalable algorithms

One important notion is the idea of how much speed-up we can hope to achieve given more processors. We have three different types of scalability.

Let $T_{1,n}$ denote the run-time on one processor given an input of size n. Suppose we have p processors. We define the speed up of a parallel algorithm as

$$\texttt{SpeedUp}(p, n) = \frac{T_{1,n}}{T_{p,n}}.$$

Definition 4.1 *If* $\texttt{SpeedUp}(p, n) = \Theta(p)$*, we say that the algorithm is strongly scalable.*

There is another notion of scalability. We sometimes are interested to consider the case where for each processor we add, we add more data as well. This is actually quite realistic, since often the only time we can afford to add more processors or machines is when we are burdened with more data than our infrastructure can handle.

Definition 4.2 *If* $\texttt{SpeedUp(p,np)} = \frac{T_{1,n}}{T_{p,np}} = \Omega(1)$*, then our algorithm is weakly scalable.*

The last notion of scalability is the case when our algorithm is embarrassingly parallel. This is true if our DAG has nodes 0-depth, e.g.

Figure 4: An Embarrassingly Parallel DAG

That is, there is no dependency between our operations. It's scalable in the most trivial sense, e.g. flipping as many coins as possible at the same time.

Example: Summation

How do we add up a bunch of integers? Sequentially, the summation operation on an array can be described by an algorithm of the form

```
1 s ← 0 for i ← 1, 2, ..., n do
2 |   s+ = a[i]
3 end
4 return s
```

Algorithm 1: Sequential Summation

For this summation algorithm $T_1 = n$. So the work of the algorithm is $O(n)$. What's T_2 on this algorithm? The only correct answer is $T_2 = n$ as well, since we haven't written the code in a way which is parallel. Further, realize that the depth of the algorithm is $O(n)$, since we have written

it in a sequential order. In fact, $T_\infty = n$. So, if we increase the number of processors but keep the same algorithm, the time of algorithm does not change, i.e.

$$T_1 = T_2 = \cdots = T_\infty = n.$$

This is clearly not optimal. How can we redesign the algorithm? Instead of

$$\big((a_1 + a_2) + a_3\big) + a_4 + \ldots$$

we instead assign each processor a pair of elements from our array, such that the union of the pairs is the array and there is no overlap. At the next level of our DAG, each of the summations from the leaf-nodes will be added by assigning each pair a processor in a similar manner. Note that if the array length is not even, we can simply pad it with a single zero. Realize that this results in an algorithm with depth $T_\infty = \log_2 n$. Hence by Brent's theorem,

$$T_p \leq \frac{n}{p} + \log_2 n.$$

As $n \to \infty$, our algorithm does better since n/p dominates. As $p \to \infty$, our algorithm does worse, since all that remains is $\log_2 n$.

We remark that the analysis above works for any *associative binary operator*. Finally, we consider the `max` operator. It's associate and binary, and hence we may use a similar analysis as above to get the same efficiency. However, we can actually achieve $T_\infty = \log\log n$ *if* we allow one processor to arbitrarily write in the event of a conflict.

References

[1] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[2] G. Blelloch and B. Maggs. *Parallel Algorithmw.* Carnegie Mellon University.