9

CME 323: Distributed Algorithms and Optimization, Spring 2020
http://stanford.edu/~rezab/dao.
Instructor: Reza Zadeh, Matroid and Stanford.

Lecture 1, 3/31/2020. Scribed by Andreas Santucci, Edited by Robin Brown.

1 Overview, Models of Computation, Brent's Theorem

1.1 Overview

The first half of the class will be focused on the history of *parallel* computing, and the second on *distributed* computing, with an emphasis on notable algorithmic breakthroughs from the last several decades.

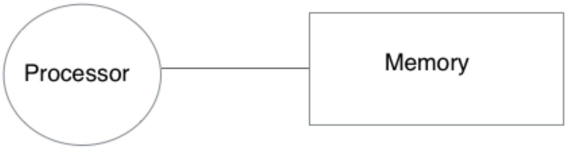
Parallel computing refers to computation with multiple processors and shared memory on a *single* machine. Although closely related, parallel and distribute computation both present unique challenges—chiefly, management of share memory in the case of parallel computation and network communication overhead in the case of distributed computation. Understanding the models and challenges of parallel computation is foundational for understanding distributed computation. The course content reflects this by first covering computation and graph algorithms in a parallel setting, and then covering the same topics in a distributed setting. The aim is to emphasize the unique challenges that each setting brings.

1.2 Introduction to Parallel Algorithms

Why focus on parallel algorithms? The regular CPU clock-speed used to double every several years which meant our algorithms would run faster. This phenomena tapered off after CPU's reached around 3 Gigahertz; we reached fundamental limits on how much heat can be transferred away from the CPU. This is in contrast with RAM and hard-disk space, which has not stopped getting bigger, yet; each year we can buy larger amounts of memory for the same price. CPU's are different, and so hardware manufacturers decided to have many cores instead of faster cores. Because computation has shifted from sequential to parallel, our algorithms have to change.[1]

Efficiency of Parallel Algorithms Even *notions of efficiency* have to adapt to the parallel. Typically, the efficiency of algorithms is assessed by the number of operations needed for it to terminate (such as in CME 305), with the assumption that this is a good proxy for wall-clock compute time. However, this is no longer an appropriate metric in the case of parallel algorithms. An efficient parallel algorithm may be inefficient when emulated on a single processor and vice-versa. Consequently, we will introduce a new measure of complexity for parallel algorithms, *work-depth*, and illustrate how algorithms can be designed to optimize work-depth.

Figure 1: Sequential RAM Model



Different types of hardware A machine made with an Intel instruction set will have dozens of cores in it, each capable of complex operations; but notice it's only dozens, e.g. 16 or 32. NVidia, on the other hand, pursues GPU computing, in which there are thousands of cores, each of which can only do basic floating point operations; at each time-step all of the cores must perform the same operation, with the caveat that they are feeding off of different data. These are known as `SIMD` chips, "single instruction multiple data". There are many other variants as well.

1.3 Sequential RAM Model

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

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

1.4 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:

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

[1] One might think we can take existing sequential algorithms and magically run a compiler to make it a parallel program. It turns out that writing such a compiler is actually an NP hard problem, something we will prove later on in this course.

	Exclusive Read	Concurrent Read
Exclusive Write	Every memory cell can be read or written to by only one processor at a time	Multiple processors can read a memory cell but only one can write at a time
Concurrent Write	Never considered	Multiple processors can read and write

Of these combinations, the *most popular model* is the concurrent read and exclusive write model. However, sometimes concurrent write is used. [2]

Resolving Concurrent Writes When dealing with concurrent writes, there needs to be a way to resolve when multiple processors attempt to write to the same memory cell at the same time. Here are the ways to deal with concurrent write:

- Undefined/Garbage Machine could die or results could be garbage.
- Arbitrary There is no predetermined rule on which processor gets write-priority. For example, if $p_1, p_2, \ldots, p_j$ all try to write to same location, we randomly select arbitrarily*exactly one* of them to give preference to.
- Priority There is a predetermined rule on which processors gets to write, e.g., we give p_i over p_j if $i \leq j$.
- Combination We write a combination of the values being written, e.g., max or logical `or` of bit-values written. Note that that unlike priority or arbitrary, the combination method relies of the *values* being written, not the processors doing the writing.

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).

Pros and Cons of Model Variants In the world of sequential algorithms, the model is fixed. However, in the parallel world, we have the freedom to decide which variant of the PRAM model to use. The benefit of this is that the algorithm designer may choose the model that is well-tailored 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.

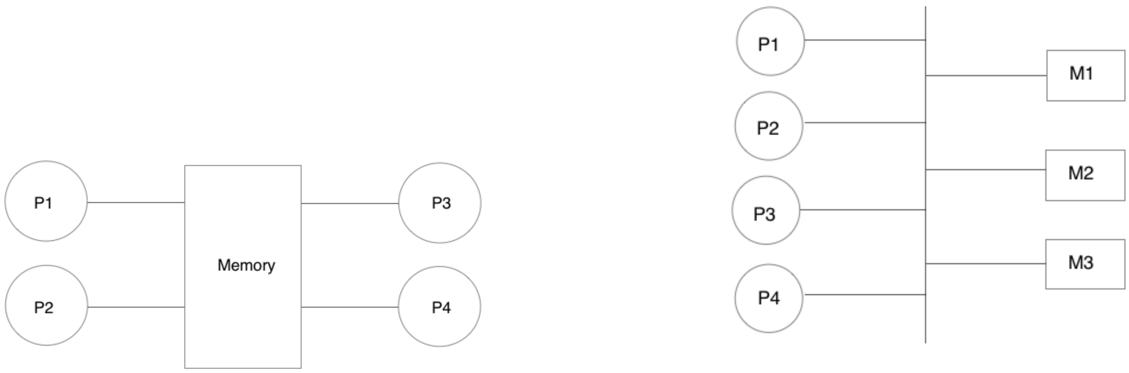
1.5 Generic PRAM Models

Below, we depict 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

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

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.

1.6 Defenestration of bounds on runtime

We now move toward developing theory on top of these models. In RAM Models, we typically count the number of operations needed for an algorithm to terminate. This implicitly makes the assumption that the clock-time taken for the algorithm is proportional to the number of operations required. I.e.,

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

This assumption is no longer valid 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. We define

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

1.7 Practical implications of work and depth

Defining work In general, work is the most important measure of the cost of an algorithm. The reasoning is 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.

Definition 1.1 (Work) *The work of an algorithm is defined to be the amount of time required complete all computations times the number of processors used.*

Fundamental lower bound T_p 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.[3] This is the lower bound we are trying to achieve as algorithm designers. We would now like to determine a useful upper bound for 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. For example, suppose we represent our algorithm as a collection of computations, then if the output of one computation is used as the input to another, the first must complete before the second can begin.

1.8 Representing algorithms as DAG's

The above intuition can be made rigorous by representing the dependencies between operations in an algorithm using a directed acyclic graph (DAG).

Constructing a DAG from an algorithm 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. The resulting graph is not guaranteed to be connected—it is possible to have calculations that are completely independent of each other. However, each connected component is acyclic and thus must be an anti-aborescence.

[3] 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.

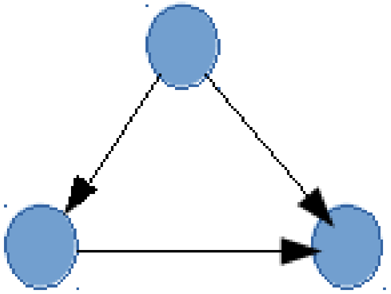


Figure 3: An example of a non-tree DAG

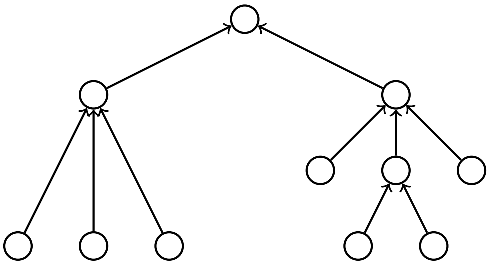


Figure 4: Example of a directed dcyclic graph (DAG)

Operations in different layers of a DAG can *not* be computed in parallel Without loss of generality, we will assume our DAG is a tree, so the levels of the tree are well defined. 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.

How an algorithm is executed on a single machine 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 an algorithm is executed with an unlimited number of processors How much time would it take to execute the algorithm given an unlimited number 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.[4] So, with an infinitude of processors, the compute time is given by the depth of the tree. We then define *depth* to be

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

Realistically, the number of processors will be limited, so what's the point of T_∞? It is important as a fundamental lower-bound on the time an algorithm takes, and will be used in upper-bounding T_p.

1.9 Brent's theorem

Theorem 1.1 *With T_1, T_p, T_∞ defined as above, if we assume optimal scheduling* [5]*, then*

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

Since T_1/p optimal, we see that T_∞ allows us to assess how far off our algorithm performs relative to the best possible version of the parallel algorithm. T_∞ can be interpreted as *how parallel* an algorithm is.

This theorem is powerful because 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:(of Brent's Theorem) 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,

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

[4] 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.

[5] This is a non-trivial assumption because optimal scheduling is NP-hard. However, as we will see in Lecture 2, there is a constant approximation algorithm for optimal scheduling. This will allow us to reason about the asymptotic scaling of these algorithms despite not having an algorithm for optimal scheduling

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.[6]

1.10 Parallel 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

```
$s \leftarrow 0$ for $i \leftarrow 1, 2, \ldots, n$ do
    $s += $ a[i]
end
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.

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

[6] 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.

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

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

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 $\log_2 n$.

1.11 A remark on associative binary operators

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 Algorithms.* Carnegie Mellon University.