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

Lecture 2, 4/5/2020. Scribed by Robin Brown.

2 Scalable algorithms, Scheduling, and a glance at All Prefix Sum

2.1 Types of scaling

Another fundamental quantity to understand is the idea of how much speed-up we can hope to achieve given more processors. There are three different types of scalability:(1) Strong Scaling, (2) Weak Scaling, and (3) Embarrassingly Parallel.

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

We can use the results of last lecture to show that parallel sum of n numbers on p processors, is strongly scalable.

Proof: To see this, recall that work is $T_1 = n$, and depth is $T_\infty = \log_2 n$, so $\texttt{SpeedUp}(p,n) = \frac{n}{\frac{n}{p}+\log_2 n} = \Theta(p)$. Specifically, both n and p are assumed to be going to infinity,but p grows much slower than does n. Hence

$$\frac{n}{\frac{n}{p}+\log_2 n} \geq \frac{n}{\frac{n}{p}+\frac{n}{p}} \geq \frac{p}{2} \qquad \text{and} \qquad \frac{n}{\frac{n}{p}+\log_2 n} \leq \frac{n}{\frac{n}{p}} = p.$$

■

Definition 2.2 (Weakly Scalable) *If* $\texttt{SpeedUp(p,np)} = \frac{T_{1,n}}{T_{p,np}} = \Omega(1)$*, then our algorithm is weakly scalable.*

This metric characterizes the case where, for each processor we add, we add more data as well. This is a useful metric in practice, because oftentimes 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 2.3 (Embarassingly Parallel) *When the DAG representing an algorithm has 0-depth, the algorithms is said to be embarrassingly parallel*

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.

We note here that we have used Brent's theorem to derive the scaling bounds for the parallel sum algorithm. In the previous lecture, we alluded to the fact that Brent's theorem assumes optimal scheduling, which is NP-hard. Fortunately, the existence of a polynomial time constant approximation algorithm for optimal scheduling implies that these bounds still hold.

Figure 1: An Embarrassingly Parallel DAG

2.2 Scheduling

In addition to building algorithms with low depth, clever scheduling is just as important to parallelism. Given a DAG of computations, at any level in the DAG there are a certain number of computations which can be required to execute (at the same time). The number of computations is not necessarily equal to the number of processors you have available to you, so you need to decide how to assign computations to processor—this is what is referred to as scheduling.

Ideally, you wish for all your processors to be busy, however, depending how jobs are assigned to processors, you might end up with processors that are idle and not working. Depending on the size and dependencies of jobs to be scheduled, it may not be possible for all processors to be busy all the time. This then turns into an optimization problem where we try to schedule jobs in a way that minimizes the idle time of processors. This problem turns out to be NP hard.

It is the scheduler's task to schedule things in tandem in such a way minimizes the idle time of processors. We could do this greedily, i.e., as soon as there is any computation to be done, we assign it to a processor. Or, we can look ahead in our DAG to see if we can plan more efficiently.

Spark has a scheduler. Every distributed computing set up has a scheduler. Your operating system and phone's have schedulers. Every computer has processes, and every computer runs in parallel. Your computer might have fifty Chrome tabs open and must decide which one to give priority to in order to optimize performance of your machine.

2.2.1 Problem definition

An important problem in any parallel or distributed computing setting is figuring out how to schedule jobs optimally. I.e. a scheduler must be able to assign sequential computation to processors or machines in order to minimize the total time necessary to process all jobs.

Notation We assume that the processors are identical (i.e. each job takes the same amount of time to run on any of the machines). More formally, we are given p processors and an unordered set of n jobs with processing times $J_1, \ldots, J_n \in \mathbb{R}$. Say that the final schedule for processor i is defined by a set of indices of jobs assigned to processor i. We call this set S_i. The load for processor i is therefore, $L_i = \sum_{k \in S_i} J_k$. The goal is to minimize the *makespan* defined as $L_{max} = \max_{i \in \{1,\ldots,p\}} L_i$.

2.2.2 The simple (greedy) algorithm

The intuition behind the greedy algorithm discussed here is simple: in order to minimize the makespan we don't want to give a job to a machine that already has a large load. Therefore, we consider the following algorithm. Take the jobs one by one and assign each job to the processor that has the least load at that time. This algorithm is simple and is *online*.

```
for each job that comes in (streaming) do
    Assign job to lowest burdened machine
end
```

Algorithm 1: Simple scheduler

Other variants of scheduling We note there are many other variants of scheduling. Jobs can have *dependencies*, i.e. one job must finish before another job can start. Here, the problem is pre-specified by a computational DAG that is known before the time of scheduling. Another variant is that scheduling must happen *online*, i.e. jobs come at you in an order where you cannot look into the future. As jobs come in, you have to schedule it, and you cannot go back and change the schedule. For a comprehensive treatment of variants of scheduling, see Handbook of Scheduling.[1]

2.2.3 Optimality of the greedy approach

In either of the above cases, where jobs have dependencies or must be scheduled online, the problem is NP hard. So, we use approximation algorithms. We claim that the simple (greedy, and online) algorithm actually has an *approximation ratio* of 2. In other words, the algorithm is in the worst-case 2 times worse than the optimal, which is fairly good. For this analysis, we define the optimal makespan (minimal makespan possible) to be OPT and try to compare the output of the greedy algorithm to this. We also define L_{max} as above to be the makespan.

Claim: Greedy algorithm has an approximation ratio of 2.

Proof: We first want to get a handle (lower bound) on OPT. We know that the optimal makespan must be at least the sum of the processing times for the jobs divided amongst the p processors, i.e.[2]

$$\text{OPT} \geq \frac{1}{p} \sum_{i=1}^{n} J_i. \tag{1}$$

A second lower bound is that OPT is at least as large as the time of the longest job, [3]

$$\text{OPT} \geq \max_i J_i. \tag{2}$$

Now consider running the greedy algorithm and identifying the processor responsible for the makespan of the greedy algorithm (i.e. k = $\text{argmax}_i L_i$). Let J_t be the load of the last job placed

[1] To give an idea of another variant, consider the case of distributed computing, where each machine houses a set of local data, and shuffling data across the network is a bottleneck. We may consider scheduling jobs to machines such that no data are shuffled. We'll consider this more in the latter part of the course. This is called *locality sensitive scheduling*

[2] To see this, assume toward contradiction that OPT is able to schedule jobs such that $\text{OPT} < \frac{1}{p} \sum_{i=1}^{n} J_i$. Suppose that instead of OPT assigning jobs to p processors in parallel, we assigned all the work to one processor sequentially. Then of course the total time required given by $p \cdot \text{OPT} < \sum_{i=1}^{n} J_i$ but this is a contradiction, since $\sum_{i=1}^{n} J_i$ exactly represents the amount of work required to process all n jobs on a single processor.

[3] The reason for this is simple: the longest job must be scheduled at some point to be run sequentially on one processor, at which point it will require $\max_i J_i$ time to compute. There may be other processors which bottleneck our makespan, but we know that OPT must take at least as long as any job, and in particular this holds for the largest job.

on this processor. Before the last job was placed on this processor, the load of this processor was thus $L_{\max}$ - J_t. By the definition of the greedy algorithm, this processor must have also had the least load before the last job was assigned. Therefore, all other processors *at this time* must have had load at least $L_{\max} - J_t$, i.e. $L_{\max} - J_t \leq L'_i$ for all i. Hence, summing this inequality over all i,

$$p(L_{\max} - J_t) \leq \sum_{i=1}^{p} L'_i \leq \sum_{i=1}^{p} L_i = \sum_{i=1}^{n} J_i \tag{3}$$

In the second inequality, we assert that although J_t the last job placed on the bottleneck processor, there may still be other jobs yet to be assigned, hence we have that the sum of total work placed on each machine cannot decrease after assigning all jobs. The last equality comes from the fact that the sum of the loads must be equal to the sum of the processing times for the jobs. Rearranging the terms in this expression let's us express this as:

$$L_{\max} \leq \frac{1}{p} \sum_{i=1}^{n} J_i + J_t \tag{4}$$

Now, note that our greedy algorithm actually has makespan exactly equal to $L_{\max}$, by definition. Using equations 1 and 2 along with the fact that $J_t \leq \max_i J_i$, we get that our greedy approximation algorithm has makespan

$$\text{APX} = L_{\max} \leq OPT + OPT = 2 \times OPT. \tag{5}$$

This shows that the greedy algorithm provides us with a scheduling time that is not more than 2 times more than the optimal. ■

What if we could see the future? We note that if we first sort the jobs in descending order and assign larger jobs first, we can naively get a 3/2 approximation. The intuition is that if we first schedule large jobs, we can use the smaller jobs to "fill in the gaps" remaining, i.e. to balance all loads. If we use the same algorithm with a tighter analysis, we get a 4/3 approximation. We'll see later in the course how Spark uses lazy evaluation for exactly this reason: by faking computation until the user takes an *action*, Spark can sort jobs and to obtain a more efficient scheduler.

What's realistic? It may seem that our above assumption, to be able to look into the future and know which jobs are going to be scheduled, is quite unreasonable. In reality, we don't even know how long each job will take. However, we often have a pretty good idea (based on historical data or expectations) how long a particular job will take to run. And further, we may have statistics or expectations on how many jobs of a particular type are going to come in, hence, it may not be such an unrealistic scenario to know (within a certain tolerance) the expected amount of time each job will take as well as what jobs might be in the pipeline.

2.3 All Prefix Sum

Given a list of integers, we want to find the sum of all prefixes of the list, i.e. the running sum. We are given an input array A of size n elements long. Our output is of size $n + 1$ elements long, and its first entry is *always* zero.

As an example, suppose $A = [3, 5, 3, 1, 6]$, then $\texttt{AllPrefixSum}(A) = [0, 3, 8, 11, 12, 18]$.

This feels like an inherently sequential task. The obvious way to do this with a single machine is to have a running sum and write intermediary sums as we iterate through the array in linear time. However, this does not parallelize at all. How can we parallelize this, so that it has low-depth? We'll take a look at this problem in more detail next lecture. For now, try to come up with your own algorithm.

References

[1] Ola Svensson *Approximation Algorithms.* EPFL, January 21, 2013.

[2] Joseph Y-T. Leung. *Handbook of Scheduling.* CRC Press, 2004.

[3] G. Blelloch and B. Maggs. *Parallel Algorithms.* Carnegie Mellon University.