Distributed Algorithms and Optimizations

Reza Zadeh

December 16, 2022

Contents

1 Overview, Models of Computation, Brent's Theorem

1.1 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:

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:

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

	Exclusive Read	Concurrent Read
Exclusive Write		
Concurrent Write	Never considered	

The **most popular model** is the concurrent read and exclusive write mode.

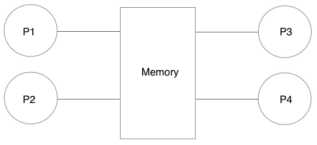
Resolving Concurrent Writes. When dealing with concurrent writes, there needs to be a way to resolve when multiple processors attempt to write to the same memory cell at the same time. Here are the ways to deal with concurrent write:

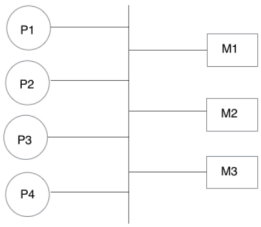
- Undefined/Garbage. Machine could die or results could be garbage

- Arbitrary There is no predetermined rule on which processor gets write-priority. For example, if $p_1, \ldots, p_j$ all try to write to same location, we randomly select arbitrarily **exactly one** of them to give reference to
- Priority There is a predetermined rule on which processors gets to write
- Combination We write a combination of the values being written, e.g., max or logical `or` of bit-values written.

1.2 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 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.





In practice, either **arbitrary** or **garbage** is used.

1.3 Defenestration of bounds on runtime

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

$T_1 =$ amount of (wall-clock) time algorithm takes on one processor
$T_p =$ amount of (wall-clock) time algorithm takes on p processors

1.4 Practical implications of work and depth

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:

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

1.5 Representing algorithms as DAG's

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.

Operations in different layers of a DAG can *not* be computed in parallel. W.L.O.G., we will assume our DAG is a tree, so the levels of the tree are well-defined. Let the root of the tree have depth 0. Suppose m_i denotes the number of operations performed in level i of the DAG. Each of the m_i operations

How an algorithm is executed with an unlimited number of processors At each level i, if there are m_i operations we may use m_i processors to compute all results in constant time. So with an infinitude of processors, the compute time is given by the depth of the tree. We then define **depth** to be

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

1.6 Brent's theorem

Theorem 1.2. *With T_1, T_p, T_∞ defined as above, if we assume optimal scheduling, then*

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

Proof. On level i of our DAG, there are m_i operations. Hence $T_1 = \sum_{i=1}^{n} m_i$ where $T_\infty = n$. For each level i of the DAG, the time taken by p processors is given by

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

□

1.7 Parallel summation

$T_1 = n$. We may partition the integers in each level, therefore

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

$\log_2 n$ is just like $\log_m n$ with some constant m.

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**

Proposition 2.2. *The parallel sum of n numbers on p processors is strongly scalable*

Proof. $T_1 = n$, $T_\infty = \log_2 n$, 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 n, hence

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

□

Note that we have used Brent's theorem to derive the scaling bounds. But 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.

Definition 2.3 (Weakly Scalable)**.** If $\texttt{SpeedUp}(p, np) = \Omega(1)$, then our algorithm is **weakly scalable**

This metric characterizes the case where, for each processor we add, we add more data as well.

Definition 2.4 (Embarrassingly Parallel)**.** When the DAG representing an algorithm has 0-depth, the algorithms is said to be **embarrasingly parallel**.

2.2 Scheduling

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.

2.2.1 Problem definition

Notation 2.5. *We assume that the processors are identical. More formally, we are given p processors and an unordered set of n jobs with processing times $J_1, \dots, J_n \in \mathbb{R}$. 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 $L_{\max} = \max_{i \in \{1,\dots,p\}} L_i$*

2.2.2 The simple (greedy) algorithm

Take the jobs one by one and assign each job to the processor that has the least load at that time.

2.2.3 Optimality of the greedy approach

In either of the 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 algorithm has an **approximation ratio** of 2. For this analysis, we define the optimal makespan 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. Obviously, $OPT \geq \frac{1}{p}\sum_{i=1}^{n} J_i$, and $OPT \geq \max_i J_i$

Now consider running the greedy algorithm and identifying the processor responsible for the makespan of the greedy algorithm. Let J_t be the load of the last job placed on this processor. Before the last job was placed on this processor, the load of this processor was thus $L_{\text{max}} - J_t$. Therefore, all other processors *at this time* must have load at least $L_{\text{max}} - J_t$, i.e., $L_{\text{max}} - J_t \leq L'_i$ for all i. Hence summing the inequality over all i

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

Therefore

$$L_{\text{max}} \leq \frac{1}{p}\sum_{i=1}^{n} J_i + J_t \leq OPT + OPT = 2OPT$$

□

What if we could see in 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. If we use the same algorithm with a tighter analysis, we get a $4/3$ approximation.

What's realistic?

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. Out 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 $R = \texttt{AllPrefixSum}(A) = [0, 3, 8, 11, 12, 18]$

3.1 Algorithm Design

The general idea is that we first take the sums of adjacent pairs of A. So the size of A' is exactly half the size of A.

To compute the running sum for elements whose index is of odd parity in A, i.e., set

$$r_i = r_{i-1} = a_i$$

for $i = 1, 3, 5, ...$ where we by convention let $r_0 = 0$

Algorithm 1 Prefix Sum

1: **if** size of A is 1 **then**
2: **return** only element of A
3: **end if**
4: Let A' be the sum of adjecent pairs
5: Comput $R' = \texttt{AllPrefixSum}(A')$
6: Fill in missing entries of R' using another $\frac{n}{2}$ processors

3.2 Algorithm Analysis

Pairing entries: in line 5, where we let A' be the sum of adjecent pairs, we must perform $n/2$ summations, hence work is $O(n)$.

Recursive call: Line 6 is our recursive call, which is fed an input of half the size of A.

Filling in missing entries: In line 7, filling in missing entries, we can assign each of the $n/2$ missing entries of R to a processor and compute its corresponding value in constant time. Hence line 7 has work $n/2$, and depth $O(1)$.

Total work and depth: Let $T_1 = W(n)$, and $T_\infty = D(n)$,

$$W(n) = W(n/2) + O(n) \Rightarrow W(n) = O(n)$$
$$D(n) = D(n/2) + O(1) \Rightarrow D(n) = O(\log(n))$$

3.3 Mergesort

Suppose we parallelize the algorithm via the obvious divide-and-conquer approach, the work done is then

$$W(n) = 2(W/n) + O(n) = O(n \log n)$$

The depth is

$$D(n) = D(n/2) + O(n) = O(n)$$

By Brent's theorem, we have that

$$T_p \leq O(n \log n)/p + O(n)$$

The bottleneck lies in `merge`.

How do we merge L and R in parallel?

Use binary search to find the rank of an element: Let's call the output of our algorithm M. For an element $x \in R$, define $\text{rank}_M(x)$ to be the index

of element x in output M. For an such element $x \in R$, we know how many elements (say a) in R come before x since we have sorted R. But we don't know immediately.

If we know how many elements (say b) in L are less than x, then we know we should place x in the $(a+b)^{th}$ position in the merged array M. It remains to find b. We can find b by performing a binary search over L. We perform the symmetric procedure for each $l \in L$, so for a call to `merge` on an input of size n, we perform n binary searches, each of which takes $O(\log n)$ time.

3.4 Parallel merge

Algorithm 2 Parallel Merge

1: **Input**: Two sorted arrays A, B each of length n
2: **Output**: Merged array C, consisting of elements of A and B in sorted order
3: **for** each $a \in A$ **do**
4: Do a binary search to find where a would be added into B
5: The final rank of a given by $\text{rank}_M(a) = \text{rank}_A(a) + \text{rank}_B(a)$
6: **end for**

To find the rank of an element $x \in A$ in another sorted B requires $O(\log n)$ work using a sequential processor. Hence in total, this parallel merge routine requires $O(n \log n)$ work and $O(\log n)$ depth.

Hence when we use `parallelMerge` in our `mergeSort` algorithm,

$$W(n) = 2W(n/2) + O(n \log n) \Rightarrow W(n) = O(n \log^2 n)$$
$$D(n) = D(n/2) + \log n \qquad \Rightarrow D(n) = O(\log^2 n)$$

By Brent's Theorem, we get

$$T_p \leq O(n \log^2 n)/p + O(\log^2 n)$$

so for large p we significantly outperform the naive implementation.

3.5 Motivating Cole's mergesort

Can we do better than binary sort?

Let L_m denote the median index of array L. We then find the corresponding index in R using binary search with logarithmic work. We then

observe that all of the elements in L at or below L_m and all of the elements in R at $\text{rank}_R(\texttt{value}(L_m))$ are at most the value of L's median element. Hence if we were to recursively merge-sort the first L_m elements in L along with the first $\text{rank}_R(\texttt{value}(L_m))$ elements in R, and correspondingly for the upper parts of L and R, we may simply append the results together to maintain sorted order. This leads us to Richard Cole[Col88]. He works out all the intricate details in this approach nicely to achieve

$$W(n) = O(n \log n)$$
$$D(n) = O(\log n)$$

4 Divide and Conquer Recipe, Parallel Selection

4.1 General Divide and Conquer Technique

$$T_1 = W(n) = aW(\frac{n}{b}) + w$$
$$T_\infty = D(n) = D(\frac{n}{b}) + t$$

4.2 Parallel Quick Selection

Suppose we have a list of unsorted integers, which we know nothing about. We wish to find the k^{th} largest element of the integers.

Assume that our input array A has unique elements.

Idea: From the input list A pick a value at random called a pivot p. For each item in A, put them into one of two sublists L and R s.t.:

$$x \in L \Leftrightarrow x < p$$
$$y \in R \Leftrightarrow y > p$$

Note that the rank of p is *exactly* $|L|$. To find the k^{th} largest element in A, call it z, note the following:

If $|L| < k$, then $z \notin L$, we discard the values in L.
If $|L| > k$, then $z \notin R$. We discard the values in R.

If we select any of these elements in red as a pivot, *then* both L and R will have size $\leq \frac{3}{4}n$.

So we say that a **phase** in our algorithm ends as soon as we pick a pivot in the middle half of our array. Recognize that in phase k, the array size is at most $n\left(\frac{3}{4}\right)^k$. The maximum number of *phases* before we can hit a base case is given by $\lceil \log_{4/3} n \rceil$

4.3 Analysis - Expected Work

Let X_k denote the number of times `Select` called with array of input size between

$$n\left(\frac{3}{4}\right)^{k+1} \leq |A| < n\left(\frac{3}{4}\right)^k$$

Total work is given by the sum of the work done for each X_k multiplied by the number of calls of that size. Realize that total work done during phase k given by

$$X_k \cdot cn\left(\frac{3}{4}\right)^k$$

for some constant $c \in \mathbb{R}^+$. Total number of phases is $\lceil \log_{4/3} n \rceil$. Let W be a random variable describing the total work done. Then

$$W \leq \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \left(X_k \cdot cn\left(\frac{3}{4}\right)^k \right) = cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \left(X_k \cdot \left(\frac{3}{4}\right)^k \right)$$

We are interested in the expected amount of total work, therefore

$$\mathbb{E}[W] \leq cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \mathbb{E}[X_k]\left(\frac{3}{4}\right)^k$$

We now analyze $\mathbb{E}[X_k]$. Recall that

$$\mathbb{E}[X_k] = \sum_{i=0}^{\infty} i \cdot \Pr(X_k = i)$$

Note that

$$\Pr(X_k = i) = \left(\frac{1}{2}\right)^{i-1} \cdot \frac{1}{2} = \frac{1}{2^i}$$

Hence

$$\mathbb{E}[X_k] = \sum_{i=0}^{\infty} \frac{i}{2^i} = 2$$

Ultimately,

$$\mathbb{E}[W] \leq cn \sum_{k=0}^{\lceil \log_{4/3} n \rceil} \mathbb{E}[X_k] \left(\frac{3}{4}\right)^k \leq 2cn \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k = 8cn = O(n)$$

So in expectation,

$$T_1 = O(n)$$

Applying Markov's Inequality: We can compute a bound on the probability that our total work exceeds a multiple of our expected total work. For example, if we wanted to do analysis that our total work will exceed 5 times our expected work:

$$P(\text{Total Work} \geq 5 \times E[\text{Total Work}]) \leq \frac{E[\text{Total Work}]}{5 \times E[\text{Total Work}]} = \frac{1}{5}$$

4.4 Parallelizing our Select Algorithm

Constructing L and R with Small Depth.

Algorithm 3 Constructing L(or R)

1: Allocate an empty array of size n, with all value initially 0.
2: Construct indicator list $B_L[0, \ldots, n-1]$ where $b_i = 1$ if $a_i < p$
3: Compute **PrefixSum** on B_l ▷ $O(\log n)$ depth
4: Create the array L of size **PrefixSum**$(B_L[n-1])$
5: **for** i=1,2,...,n **do**
6: **if** B[i]=1 **then**
7: L[**PrefixSum**$(B_L[i])] \leftarrow a[i]$ ▷ Can be done in parallel
8: **end if**
9: **end for**

$$D(n) = D\left(\frac{n}{4/3}\right) + O(\log n) \Rightarrow D(n) = O(\log^2 n)$$

5 Memory Management and (Seemingly) Trivial Operations

We assume that we can allocate memory in constant time, as long as we don't ask for the meomory to have special values in it. That is, we can request a large chunk of memory (filled with garbage bit sequences) in constant time. However, requesting an array of zeros already requires $\Theta(n)$ work since we must ensure the integrity of each entry. In the context of sequential algorithms, this is not a concern since reading in an input of n bits, or outputting n bits already requires $\Theta(n)$ work, so zeroing out an array of size n does not dominate the operation count. However, in some parallel algorithms, no processor reads in the entire input, so naively zeroing out a large array can easily dominate the operation time of the algorithm.

6 QuickSort

6.1 Analysis on Memory Management

Algorithm 4 QuickSort

Input: An array A
Output: Sorted A
1: $p \leftarrow$ element of A chosen uniformly at random
2: $L \leftarrow [a \mid a \in A, a < p]$ ▷ Implicitely: $B_L \leftarrow \mathbb{1}\{a_i < p\}_{i=1}^n$, `prefixSum`$(B_L)$
3: $R \leftarrow [a \mid a \in A, a > p]$ ▷ which requires $\Theta(n)$ work and $O(\log n)$ depth
4: **Return** [QuickSort(L), QuickSort(R)]

We denote the size of our input array A by n. To be precise, we can perform step 1 in $\Theta(\log n)$ work and $O(1)$ depth. That is, to generate a number uniformly from the set $\{1, \ldots, n\}$ we can assign $\log n$ processors to independently flip a bit "on" with probability $1/2$.

Allocating storage for L and R: Start by making a call to the OS to allocate an array of n elements; this requires $O(1)$ work and depth, since we do not require the elements to be initialized. We compare each element in the array with the pivot, p, and write a 1 to the corresponding element if the element belongs in L and a 0 otherwise. This requires $\Theta(n)$ work but can be done in parallel, i.e., $O(1)$ depth. We are left with an array of 1's and 0's

indicating whether an element belongs in L or not, call it $\mathbb{1}_L$,

$$\mathbb{1}_L = \mathbb{1}\{a \in A : a < p\}$$

We then apply `PrefixSum` on the indicator array $\mathbb{1}_L$, which requires $O(n)$ work and $O(\log n)$ depth. Then we may examine the value of the last element in the output array from `PrefixSum` to learn the size of L. Looking up the last element in array $\mathbb{1}_L$ requires $O(1)$ work and depth. We can further allocate a new array for L in constant time and depth. Since we know $|L|$ and we know n, we also know $|R| = n - |L|$; computing $|R|$ and allocating corresponding storage requires $O(1)$ work and depth.

Thus allocating space for L and R requires $O(n)$ work and $O(\log n)$ depth.

Filling L and R: Now we use n processors, assigning each to exactly one element in our input array A, and in parallel we perform the following steps. Each processor $1, 2, \ldots, n$ is assigned to its corresponding entry in A.

Suppose we fix attention to the kth processor, which is responsible for assigning the kth entry in A to its appropriate location in either L and R. We first examine $\mathbb{1}_L[k]$ to determine whether the element belongs in L or R. In addition, examine the corresponding entry in `PrefixSum` output, denote this value by $i = \texttt{PrefixSum}(\mathbb{1}_L[k])$. If the kth entry of A belongs in L, then it may be written to the position i in L immediately. If the kth entry instead belongs in R, then realize that index i tells us that exactly i entries "before" element k belong in L. Hence exactly $k-i$ elements belong in array R before element.

The process of filling L and R requires $O(n)$ work and $O(1)$ depth

Therefore we need $O(n)$ work and $O(\log n)$ depth.

6.2 Total Expected Work

The level of our computational DAG is n and there are $\log_{4/3} n$ levels, therefore

$$\mathbb{E}[T_1] = O(n \log n)$$

We define the random indicator variable X_{ij} to be one if the algorithm *does compare* the ith *smallest* and the jth *smallest* elements of input array A during the course of its sorting routine, and zero otherwise. Let X denote the *total* number of comparisons made by our algorithm. Then we have that

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

7 References

References

[Col88] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.