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

Lecture 3, 04/04/2016. Scribed by Tuan Nguyen, Alex Adamson, Andreas Santucci.

1 Matrix multiplication: Strassen's algorithm

We've all learned the naive way to perform matrix multiplies in $O(n^3)$ time.[1] In today's lecture, we review Strassen's sequential algorithm for matrix multiplication which requires $O(n^{\log_2 7}) = O(n^{2.81})$ operations; the algorithm is amenable to parallelizable.[4]

A variant of Strassen's sequential algorithm was developed by Coppersmith and Winograd, they achieved a run time of $O(n^{2.375})$.[3] The current best algorithm for matrix multiplication $O(n^{2.373})$ was developed by Stanford's own Virginia Williams[5].

Idea - Block Matrix Multiplication The idea behind Strassen's algorithm is in the formulation of matrix multiplication as a recursive problem. We first cover a variant of the naive algorithm, formulated in terms of block matrices, and then parallelize it. Assume $A, B \in \mathbb{R}^{n\times n}$ and $C = AB$, where n is a power of two.[2]

We write A and B as block matrices,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix},$$

where block matrices A_{ij} are of size $n/2 \times n/2$ (same with respect to block entries of B and C). Trivially, we may apply the definition of block-matrix multiplication to write down a formula for the block-entries of C, i.e.

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Parallelizing the Algorithm Realize that A_{ij} and $B_{k\ell}$ are smaller matrices, hence we have broken down our initial problem of multiplying two $n \times n$ matrices into a problem requiring 8 matrix multiplies between matrices of size $n/2 \times n/2$, as well as a total of 4 matrix additions.

[1] Refresher, to compute $C = AB$, we need to compute c_{ij}, of which there are n^2 entries. Each one may be computed via $c_{ij} = \langle a_i^T, b_j \rangle$ in $2n - 1 = \Theta(n)$ operations. Hence total work is $O(n^3)$.

[2] If n is *not* a power of two, then from a theoretical perspective we may simply pad the matrix with additional zeros. From a practical perspective, we would simply use un-equal size blocks.

There is nothing fundamentally different between the matrix multiplies that we need to compute at this level relative to our original problem.

Further, realize that the four block entries of C may be computed independently from one another, hence we may come up with the following recurrence for *work:*

$$W(n) = 8W(n/2) + O(n^2)$$

By the Master Theorem,[3] $W(n) = O(n^{\log_2 8}) = O(n^3)$. So we have *not* made any progress (other than making our algorithm parallel). We already saw in lecture two that we can naively parallelize matrix-multiplies very simply to yield $O(n^3)$ work and $O(\log n)$ depth.

Strassen's Algorithm We now turn toward Strassen's algorithm, such that we will be able to reduce the number of sub-calls to matrix-multiplies to 7, using just a bit of algebra. In this way, we bring the work down to $O(n^{\log_2 7})$.

How do we do this? We use the following factoring scheme. We write down C_{ij}'s in terms of block matrices M_k's. Each M_k may be calculated simply from products and sums of sub-blocks of A and B. That is, we let

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})\,(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22})
\end{aligned}$$

Crucially, each of the above factors can be evaluated using exactly *one* matrix multiplication. And yet, since each of the M_k's expands by the distributive property of matrix multiplication, they capture additional information. Also important, is that these matrices M_k may be computed independently of one another, i.e. this is where the parallelization of our algorithm occurs.

It can be verified that

$$\begin{aligned}
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M3 + M_6
\end{aligned}$$

[3] Case 1: $f(n) = O(n^2)$, so $c = 2 < 3 = \log_2(8)$.

Realize that our algorithm requires quite a few summations, however, this number is a constant independent of the size of our matrix multiples. Hence, the work is given by a recurrence of the form

$$W(n) = 7W(n/2) + O(n^2) \implies W(n) = O(n^{\log_2 7}).$$

What about the depth of this algorithm? Since all of our recursive matrix-multiplies may be computed in parallel, and since we can add matrices together in unit depth,[4] we see that depth is given by

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

By Brent's theorem, $T_p \leq \frac{n^{2.81}}{p} + O(\log n)$. In the years since Strassen published his paper, people have been playing this game to bring down the work required marginally, but nobody has come up with a fundamentally different approach.

1.1 Drawbacks of Divide and Conquer

We now discuss some bottleneck's of Strassen's algorithm (and Divide and Conquer algorithms in general).

- We haven't considered communication bottlenecks; in real life communication is expensive.
- Disk/RAM differences are a bottleneck for recursive algorithms, and
- PRAM assumes perfect scheduling.

Communication Cost Our PRAM model assumes zero communication costs between processors. The reason is because the PRAM model assumes a shared memory model, in which each processor has fast access to a single memory bank. Realistically, we never have efficient communication, since often times in the real world we have clusters of computers, each with its own private bank of memory. In these cases, divide and conquer is often impractical.

It is true that when our data are split across multiple machines, having an algorithm operate on blocks of data at a time can be useful. However, as Strassen's algorithm continues to chop up matrices into smaller and smaller chunks, this places a large communication burden on distributed set ups because after the first iteration, it is likely that we will incur a shuffle cost as we are forced to send data between machines.

[4] We note that to perform matrix addition of two $n \times n$ matrices $X + Y = W$, we may calculate each of the n^2 entries $W_{ij} = X_{ij} + Y_{ij}$ in parallel using n^2 processors. Each entry requires only one fundamental unit of computation, hence the work for matrix addition is $O(n^2)$ and the depth is $O(1)$.

Caveat - Big $\mathcal{O}$ and Big Constants One last caveat specific to Strassen's Algorithm is that in practice, the $O(n^2)$ term requires $20 \cdot n^2$ operations, which is quite a large constant to hide. If our data is large enough that it must be distributed across machines in order to store it all, then really we can often only afford to pass through the entire data set one time. If each matrix-multiply requires twenty passes through the data, we're in big trouble. Big $\mathcal{O}$ notation is great to get you started, and tells us to throw away egregiously inefficient algorithms. But once we get down to comparing two reasonable algorithms, we often have to look at the algorithms more closely.

When is Strassen's worth it? If we're actually in the PRAM model, i.e. we have a shared memory cluster, then Strassen's algorithm tends to be advantageous only if $n \geq 1,000$, *assuming no communication costs*. Higher communication costs drive up the n at which Strassen's becomes useful very quickly. Even at $n = 1,000$, naive matrix-multiply requires $1e9$ operations; we can't really do much more than this with a single processor. Strassen's is mainly interesting as a theoretical idea. For more on Strassen in distributed models, see [1].

Disk Vs. RAM Trade-off What is the reason that we can only pass through our data once? There is a big trade-off between having data in ram and having it on disk. If we have tons of data, our data is stored on disk. We also have an additional constraint that with respect to *streaming data*, as the data are coming in they are being stored in memory, i.e. we have fast random access, but once we store the data to disk retrieving it again is expensive.

2 Mergesort

Merge-sort is a very simple routine. It was fully parallelized in 1988 by Cole.[2] The algorithm itself has been known for several decades longer.

Algorithm 1: Merge Sort

Input : Array A with n elements
Output: Sorted A

```
1 n ← |A|
2 if n is 1 then
3 |   return A
4 end
5 else
  |   // (In Parallel)
6 |   L ← MERGESORT(A[0,...,n/2))    // Indices 0, 1, ..., n/2 − 1
7 |   R ← MERGESORT(A[n/2,...,n))    // Indices n/2, n/2 + 1, ..., n − 1
8 |   return MERGE(L,R)
9 end
```

It's critical to note how the `merge` sub-routine works, since this is important to our algorithms work and depth. We can think of the process as simply "zipping" together two sorted arrays.

Algorithm 2: Merge

Input : Two sorted arrays A, B each of length n
Output: Merged array C, consisting of elements of A and B in sorted order

```
a ← pointer to head of array A (i.e. pointer to smallest element in A)
b ← pointer to head of array B (i.e. pointer to smallest element in B)
while a, b are not null do
    Compare the value of the element at a with the value of the element at b
    if value(a) < value(b) then
        add value of a to output C
        increment pointer a to next element in A
    end
    else
        add value of b to output C
        increment pointer b to next element in B
    end
end
if elements remaining in either a or (exclusive) b then
    Append these sorted elements to our sorted output C
end
return C
```

Since we iterate over each of the elements exactly one time, and each time we make a constant time comparison, we require $\Theta(n)$ operations. Hence the `merge` routine on a single machine takes $O(n)$ work.

2.1 Naive parallelization

Suppose we parallelize the algorithm via the obvious divide-and-conquer approach, i.e. by delegating the recursive calls to individual processors. The work done is then

$$\begin{aligned} W(n) &= 2W(n/2) + O(n) \\ &= O(n \log n) \end{aligned}$$

by case 2 of the Master Theorem.

As you'll recall from earlier algorithms classes, the canonical implementation of the `merge` routine involves simultaneously iterating over L and R: starting at the first index of each, we merge them by placing the smaller of the currently pointed-to elements of L and R at the back of a new list and advance the pointer in the list that the just-placed element belonged to, and continue until we reach beyond the end of one list. Crucially, `merge` has depth $O(n)$. The depth is then

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

again by the Master Theorem.

Using Brent's theorem, we have that

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

Therefore $W(n) = O(n \log n)$ and $D(n) = O(n)$. Note that the bottleneck lies in `merge`, which takes $O(n)$ time. That is, even though we have an infinitude of processors, the time it takes to merge two sorted arrays of size $n/2$ on the first call to `mergeSort` dominates the time it takes to complete the recursive calls.

2.2 Improved parallelization

How do we merge L and R in parallel? The `merge` routine we have used is written in a way that is inherently sequential; it is not immediately obvious how to interleave the elements of L and R together even with an infinitude of processors.

Let us call the output of our algorithm M. For an element x in R, let us define $\text{rank}_M(x)$ to be the index of element x in output M. For any 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 immediately know the rank of an element x in M.

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 $l \in L$ (i.e. we find how many elements in R are less than it), so for a call to `merge` on an input of size n, we perform n binary searches, each of which takes $O(\log n/2) = O(\log n)$ time.

$$\text{rank}_M(x) = \text{rank}_L(x) + \text{rank}_R(x)$$

Algorithm 3: Parallel Merge

Input : Two sorted arrays A, B each of length n

Output: Merged array C, consisting of elements of A and B in sorted order

```
1 for each a ∈ A do
2 |   Do a binary search to find where a would be added into B,
3 |   The final rank of a given by rank_M(a) = rank_A(a) + rank_B(a).
4 end
```

To find the rank of an element $x \in A$ in another sorted B requires $O(\log n)$ work using a sequential processor. Notice, however, that each of the n iterations of the for loop in our algorithm

is independent of the previous, hence our binary searches may be performed in parallel. That is, we can use n processors and assign each a single element from A. Each processor then performs a binary search with $O(\log n)$ work. 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, we realize the following work and depth, by the master theorem:

$$\begin{aligned} W(n) = 2W(n/2) + O(n \log n) &\implies W(n) = O(n \log^2 n), \\ D(n) = D(n/2) + \log n &\implies D(n) = O(\log^2 n). \end{aligned}$$

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! The best known implementation (work $O(n \log n)$, depth $O(\log n)$) was found by Richard Cole[2].

Motivating the Next Step We notice that we use many binary searches in our recently defined parallel merge routine. Can we do better? Yes.

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 (1988).[2] He works out all the intricate details in this approach nicely to achieve

$$\begin{aligned} W(n) &= O(n \log n) \\ D(n) &= O(\log n) \end{aligned}$$

3 Coming Up: Quick-Sort and Scheduling

We briefly recall how quick-sort operates. We arbitrarily pick an element as a pivot. In $O(n)$ time, we place all elements smaller than the pivot on the Left Hand Side of the array, and all elements larger than pivot on Right Hand Side. This is an inherently sequential process.

With regard to scheduling, we have assumed that processors are assigned tasks in an optimal way. However, the process of assigning tasks to processors is actually a non-trivial problem in and of itself. So, suppose you have a program which is very large and recursive in nature. At the end of the day, it's 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). It's possible that the number of computations to be more (or less) than the number of processors you have available to you. Ideally,

you wish for all your processors to be busy. Depending on how you schedule the operations, you sometimes may end up with processors which are idle and not working.

It is the schedulers task to schedule things in tandem in such a way that you look ahead a little bit and minimize 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 could be a little bit clever about it, and perhaps look ahead further to our DAG to see if we can plan more efficiently.

We will talk about scheduling after we are done with Divide and Conquer algorithms. 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.

References

[1] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz, *Communication-optimal parallel algorithm for strassen's matrix multiplication*, CoRR, abs/1202.3173 (2012).

[2] R. Cole, *Parallel merge sort*, SIAM J. Comput., 17 (1988), pp. 770–785.

[3] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, J. Symbolic Computation, 9 (1990), pp. 251–280.

[4] V. Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik, 13, pp. 354–356.

[5] V. V. Williams, *Multiplying matrices in* $o(n^{2.373})$ *time*, Stanford University, (2014).