DD2440 Advanced Algorithms
Lecture 12: Parallel Algorithms

Lecturer: Johan Håstad
Scribe: Dino Radaković

1 December 2014

1 Introduction

Solve one problem by many processors cooperating

Size of problem: n
Number of processors: $F(n)$
Parallel time [1] : $T(n)$

As we can use one processor to simulate all processors we have that $F(n)T(n)$ is at least the sequential running time (and usually it is a bit larger as it is hard to distribute the execution in a good way).

In theory the most interesting case is when $T(n)$ is much smaller than the sequential running (and thus $F(n)$ is large) while in in practice we often have $F(n)$ in the range 4 to 16.

[1] Assuming synchronized execution where each processor might do something at each time step.

The case of very many processors makes most practical sense in the setting of simple arithmetic functions which we will discuss there. We start by defining our model of computation.

2 General model

PRAM: **P**arallel **R**andom **A**ccess **M**achine

We consider the PRAM model in which our $F(n)$ processors can read and write in the common memory, completely synchronized – no lag, everyone can read and write without any issues.

Let us do something simple in this model:
Compute OR of n bits with n processors.

How long does this take?

Algorithm:
Write 0 in memory cell 0.
Processor i reads bit i.
If bit i is 1, processor i writes 1 in memory cell 0

The algorithm works in three time steps independent of the size of n. This is due to a very generous computational model (that some people might call "cheating") of allowing all processors to write into the same memory cell at the same time. We have the some variants of PRAMs:

- EREW PRAM (exclusive read, exclusive write) – only one processor can read/write in a given memory location at any point in time.
- CREW PRAM (concurrent read, exclusive write) – many processors can read, only a single one can write in a given memory location at any point in time.
- CRCW PRAM (concurrent read, concurrent write) – many processors can read/write in a given memory location at any point in time

The difference between the variants is not huge and if we limit the number of processors to be polynomial in n ther difference is at most a factor $\log n$ in asymptotic complexity.

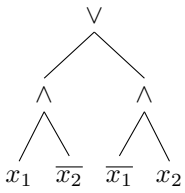
There are in fact a number of variants of CRCW. The most conservative is to demand that all processors that write to the same memory location write the same value (as in our or-computing algorithm above). When allowing different values to be written we must define how this is handled. Versions here include "priority" (the processor with the highest priority succeeds), "arbitrary" (some

processor succeeds) or "undefined" (anything can appear in the memory cell).

3 Boolean circuits

A model closer to actual hardware.

Example (XOR circuit):



Legend:

∧ – AND gate

∨ – OR gate

¬ or $\overline{x_i}$ – negation

We are interested in:

Size: Number of gates, ∼ number of processors, typically a "large" function of the size of the input, such as n or n^2.

Depth: Longest path from input to output, ∼ parallel time, typically a "small" function of the input like $\log n$.

3.1 Adding two n-bit integers using n processors

Allowed operations: OR, AND, negation

Allowing other gates (i.e. parity, NAND, ...) changes the complexity by a constant factor

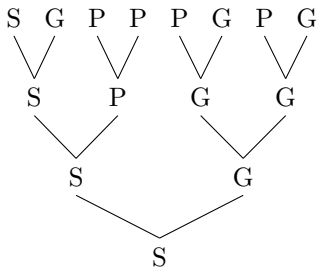
Example:

$$\begin{array}{cccccccccc} & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ + & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 \\ \hline & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{array}$$

$\mathcal{O}(n)$ size – the best you can hope for.
$\Omega(n)$ ($\geq cn$) depth – due to carries as we require the i-th carry to compute the i-th bit and the $(i+1)$-th carry.

Take a look at the intervals of size 2^k.
Do they **S**top, **P**ropagate or **G**enerate a carry?

	0	1	0	0	1	1	1	1
+	0	1	1	1	0	1	0	1
	1	1	0	0	0	1	0	0
	S	G	P	P	P	G	P	G



By visualizing it as a binary tree, we can clearly see what each one of the intervals 2^k does in terms of carry. We can do this in $\mathcal{O}(\log n)$ depth, using 2 bits to code each one of S, P and G.

Easy and obvious way to compute the most significant bit – just go left, starting from the root of the tree.

How do we compute any other bit?

In order to compute any bit, we need to start from the root and move towards the subtree which contains that bit.
When turning left, memorize the symbol (S, P or G) on top of the right subtree (in order to know whether we have a carry incoming from the right side of the bit we're looking to compute).

Going down the tree and computing these takes $\mathcal{O}(\log n)$ steps, meaning that, if we have n processors, addition of two n-bit integers can be done in $\mathcal{O}(\log n)$ depth and $\mathcal{O}(n)$ size. Speaking in terms of parallel time this gives $\mathcal{O}(\log n)$ time and $\mathcal{O}(n)$ processors where the latter can be improved to $\mathcal{O}(n/\log n)$ processors by reusing processors at different time steps.

3.2 Multiplication of n-bit integers using n^2 processors

Hoping to achieve small depth using simple algorithms.

Consider the elementary school multiplication algorithm:

$$
\begin{array}{cccccccccc}
 & 1 & 1 & 0 & \dots & \dots & \dots & 1 & 1 \\
* & 1 & 0 & 1 & \dots & \dots & \dots & 0 & 1 \\
\hline
 & 1 & 1 & 0 & \dots & \dots & \dots & 1 & 1 \\
0 & 0 & 0 & \dots & \dots & \dots & 0 & 0 & \\
\dots & & & & & & & & &
\end{array}
$$

How quickly can we do this in parallel?

Multiplying two numbers in this way consists of adding n n-bit numbers.
Since adding two n-bit numbers can be done in $\mathcal{O}(\log n)$ depth, this can surely be done in $\mathcal{O}(n \log n)$.

However, we can speed this up by adding $n/2$ pairs, each in $\mathcal{O}(\log n)$ depth, recursively. This results in $\mathcal{O}(\log n)$ levels of addition, each of $\mathcal{O}(\log n)$ depth $\implies$ circuit of $\mathcal{O}(\log^2 n)$ depth.

There is, however, a nice way of bringing it down to $\mathcal{O}(\log n)$.
We use that there is an efficient way to given three numbers x_1, x_2 and x_3 find two numbers s_1 and s_2 such $x_1 + x_2 + x_3 = s_1 + s_2$. These two numbers can be found as follows:

- s_1 is the bitwise XOR of x_1, x_2 and x_3
- s_2 is the bit-vector which indicates at which position a carry appears.

Example:

$$
\begin{array}{cccccccc}
 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
+ & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\
\hline
 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1
\end{array}
$$

The sum $x_1 + x_2 + x_3$, written above, yields the same result as $s_1 + s_2$:

$$
\begin{array}{cccccccc}
 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\
+ & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\
\hline
 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1
\end{array}
$$

Computing s_1 and s_2 can be done in $\mathcal{O}(1)$ depth – the i-th bit of s_1, the XOR, and the $(i+1)$-th bit of s_2, the shifted carry vector of the initial three numbers can be seen as a coding of the three bits at the i-th positions of the three numbers (one from each number).

For example, if we are given 101 in one column of addition, we can rewrite it as 01, and do so for every other column, shifting the second row 1 position to the left, in order to get s_1 as the first and s_2 as the second row of the newly computed bits.

How is this useful?

Using this method, we can reduce the sum of n numbers to the sum of $2n/3$ numbers in depth $\mathcal{O}(1)$ (about 4).

We can then iterate the procedure, which will take $\mathcal{O}(\log_{3/2} n)$ iterations in order to reduce the initial problem to the addition of 2 numbers and these can be added to give a single number as the answer, as indicated above. Over this implies that we get a circuits of depth $\mathcal{O}(\log n)$ and $\mathcal{O}(n^2)$ size for multiplying two n-bit integers.

3.3 Two favorite efficient algorithms

Let us mention some results about parallel computation. Let us take two favorite efficient algorithms.

1. Euclidean algorithm to compute the gcd of two n-bit integers.
2. Gaussian elimination – solving $n \times n$ systems of linear equations.

Both algorithms are sequential in nature, where each have have $\sim n$ iterations, each iteration heavily depending on the previous one. It turns out that the situations for the two problems are quite different.

It is unknown whether it is possible to compute gcd by polynomial size circuits of depth $o(n)$ while solving linear systems of equations can be done in depth $\mathcal{O}(\log^2 n)$ and polynomial size. However, this uses other methods, not Gaussian elimination.