36. Parallel Computations

1 Parallel Computations: Why

Need to speed up computations. NP-hardness means that – unless P = NP – some real-life problems require too much computation time. What should we do?

If a person is given a task that requires too much time – e.g., cleaning a big house for tomorrow's party – a natural idea is ask others for help. If several people work in parallel, they can do this task faster.

Similarly, if a task requires too much time for a single computer, a natural idea is to have several computers working in parallel.

This speeds up computations – although not always, since, in general, some time is needed for communication between the computers.

What we do in this lecture. In this lecture, we consider the simplified approach, when:

- we assume that we have unlimited number of processors, and
- we can ignore communication time, and only consider computation time.

Let us first have some examples.

2 Examples

Adding 4 numbers. Suppose that we want to add 4 numbers, i.e., compute the sum $x_1 + x_2 + x_3 + x_4$. If we do it sequentially, then we need to perform 3 additions:

- first, we compute $x_1 + x_2$,
- then, we compute $(x_1 + x_2) + x_3$, and
- finally, we compute $(x_1 + x_2 + x_3) + x_4$.

In parallel, we can compute this sum faster:

- first, the 1st computer computes $x_1 + x_2$ and, at the same time, the 2nd computer computes $x_3 + x_4$;

- then, the 1st computer computes the sum of the two numbers $x_1 + x_2$ and $x_3 + x_4$.

In this case, we use 2 computers and 2 moments of time.

Adding 6 numbers. Suppose that we want to add 6 numbers, i.e., compute the sum $x_1 + x_2 + x_3 + x_4 + x_5 + x_6$. Then:

- we have the first two computers to compute the sum $x_1 + x_2 + x_3 + x_4$ of the first 4 numbers in 2 moments of time;
- simultaneously, another computer computes the sum $x_5 + x_6$ of the remaining 2 numbers in 1 moment of time;
- finally, we use the 1st computer to add the two sums and get the desired sum of all 6 numbers.

Here, we use $2 + 1 = 3$ computers and $2 + 1 = 3$ moments of time.

Adding 8 numbers. Suppose that we want to add 8 numbers. Then:

- we have the first two computers to compute the sum $x_1 + x_2 + x_3 + x_4$ of the first 4 numbers in 2 moments of time;
- simultaneously, another pair of computers computes the sum

  $$x_5 + x_6 + x_7 + x_8$$

  of the remaining 4 numbers in 2 moments of time;
- finally, we use the 1st computer to add the two sums and get the desired sum of all 8 numbers.

Here, we use $2 \cdot 2 = 4$ computers and $2 + 1 = 3$ moments of time.

Adding 16, 32, ..., 2^m numbers. In general, when we double the number of elements to add, from n to $2n$:

- we have the first group of computers to compute the sum of the first n numbers;
- simultaneously, another group of computers computes the sum of the second group of n numbers;
- finally, we use the 1st computer to add the two sums and get the desired sum of all $2n$ numbers.

Thus, when we double the number of elements to add, we double the number of computers and increase time by 1 cycle. So:

- to add up 16 numbers, we need 8 computers and 4 moments of time,
- to add up 32 numbers, we need 16 computers and 5 moments of time,

- in general, to add up $n = 2^m$ numbers, we need $2^{m-1} = n/2$ computers and m moments of time.

By definition of a logarithm, if $n = 2^m$, then $m = \log_2(n)$. Thus, to add n numbers, we need $n/2$ processors and $\log_2(n)$ time.

Similar examples. Similar procedure can be used if we need to compute:

- the product of n numbers,
- the maximum (largest) of n numbers, and
- the minimum (smallest) of n numbers.

Adding matrices. For matrices, addition is defined component-wise:

$$\begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \ldots & \ldots & \ldots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix} + \begin{pmatrix} b_{11} & \ldots & b_{1n} \\ \ldots & \ldots & \ldots \\ b_{n1} & \ldots & b_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} + b_{11} & \ldots & a_{1n} + b_{1n} \\ \ldots & \ldots & \ldots \\ a_{n1} + b_{n1} & \ldots & a_{nn} + b_{nn} \end{pmatrix}.$$

In other words, each of $n \times n = n^2$ components c_{ij} of the sum matrix is simply the sum $c_{ij} = a_{ij} + b_{ij}$ of the corresponding components a_{ij} and b_{ij} of the two matrices. If we have n^2 computers, then we can compute all these sums in 1 step.

Multiplying matrices. For matrices, the product is defined as follows:

$$\begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \ldots & \ldots & \ldots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & \ldots & b_{1n} \\ \ldots & \ldots & \ldots \\ b_{n1} & \ldots & b_{nn} \end{pmatrix} = \begin{pmatrix} c_{11} & \ldots & c_{1n} \\ \ldots & \ldots & \ldots \\ c_{n1} & \ldots & c_{nn} \end{pmatrix},$$

where $c_{ij} = a_{i1} \cdot b_{1j} + \ldots + a_{in} \cdot b_{nj}$.

For each i and j, we can first use n processors to compute all n products $a_{i1} \cdot b_{1j}$, $\ldots$, $a_{in} \cdot b_{nj}$ in 1 moment of time. Then, we can use $n/2$ processors to compute the sum c_{ij} of these products in time $\log_2(n)$. Thus, overall, to compute each value c_{ij}, we need n processors and time $\log_2(n) + 1$.

The computations for each of n^2 pairs of indices (i, j) can be done in parallel. Thus, overall, we need $n^2 \cdot n = n^3$ processors and time $\log_2(n) + 1$.

3 General Case

General case: analysis. In all these cases, we need a polynomial number of processors and time proportional to logarithm of the input size. There are more complex algorithms in which parallelization needs $\log_2(n)$ stages each of which

takes time $\log_2(n)$ – to the overall time is $\log_2(n) \cdot \log_2(n) = (\log_2(n))^2$. In some cases, we need $(\log_2(n))^3$. These examples lead to the following definition.

Definition. We say that a problem belongs to the class NC if it can be solved in polylog time – i.e., in time bounded by a polynomial of $\log_2(n)$ – on polynomial number of processors.

Comment. In contrast to meaningful abbreviations such as P for polynomial and NP for non-deterministic polynomial, NC simply means Nick's Class, after a person who first proposed this definition.

NC is a subclass of P. Every parallel algorithm can be implemented on a sequential machine. To simulate 1 step of a parallel machine with p processors working in parallel, we need p computational steps:

- first, we simulate one step of the first processor,
- then, we simulate one step of the second processor, etc.

So, if a machine with p processors solves the problem in time t, then a sequential machine can solve it in time $p \cdot t$.

For problems from the class NC, time is bounded by a polynomial of logarithm – and is, thus, smaller than n. Thus, when we multiply n by the polynomial number p of processors, we get a polynomial sequential time. Thus, all such problems belong to the class P.

Natural question: is P = NC? A natural question is: can all problems from the class P be efficiently parallelized? In other words, if P equal to NC?

The answer is very similar to the question of whether P is equal to NP: no one knows, it is still an open problem, *but* there are problems to which all problems from the class P can be reduced. Such problems are known as *P-complete.*

These problems are the most difficult to parallelize:

- if we can parallelize this problem,
- then, because of the reduction, we can parallelize all the problems from the class P.

Example of a P-complete problem: linear programming. Historically the first example of a P-complete problem is a *linear programming* problem: find a solution to a system of linear inequalities.

This problem is useful in economics. For example, it is used to determine the diet for prisoners. By international conventions, prisoners must get nutritious food, with enough calories, enough proteins, enough vitamins, etc. – and at the same, the food service must stay within the budget. How can we describe this in precise terms?

We need to find the daily amounts of different food products: the daily amount x_1 of rice, x_2 of bread, x_3 of meat, x_4 of bananas, etc.

- The overall number of calories in this diet can be computed as

$$c_1 \cdot x_1 + c_2 \cdot x_2 + \ldots,$$

  where c_1 is the amount of calories in 1 kg of rice, c_2 is the amount of calories in 1 kg of bread, etc. The overall daily amount of calories must be greater than or equal to the required norm c_0:

$$c_1 \cdot x_1 + c_2 \cdot x_2 + \ldots \geq c_0.$$

- Similarly, the need to have at least the required amount p_0 of proteins means that we must have

$$p_1 \cdot x_1 + p_2 \cdot x_2 + \ldots \geq p_0,$$

  where p_i is the amount of proteins in 1 kg of product i, etc.

- For money, the inequality is opposite, since the overall amount of money per prisoner cannot exceed the given amount m_0:

$$m_1 \cdot x_1 + m_2 \cdot x_2 + \ldots \leq m_0,$$

  where m_i is the (wholesale) price of 1 kg of product i, etc.

We want to find the values $x_1, x_2, \ldots$ that satisfy all these inequalities – or generate a message that it is not possible to satisfy all the given inequalities.

Linear programming: general case.

- *given:* values a_{ij} $(j = 1, \ldots, n)$, a_i, b_{kj}, and b_k;
- *find:* the values $x_1, \ldots, x_n$ for which:

$$a_{i1} \cdot x_1 + \ldots + a_{in} \cdot x_n \geq a_i \text{ for all } i, \text{ and}$$

$$b_{k1} \cdot x_1 + \ldots + b_{kn} \cdot x_n \leq b_k \text{ for all } k.$$

Comment. Such problems are ubiquitous in economics, where we need to stay within a given budget (and/or within a given amount of electricity etc.).

Why programming? This problem was formulated in the 1950s and 1960s, when programming was a hot topic (similar to what data science or deep learning are now), so everyone called his or her inventions programming, even when it has nothing to do with actual programming. With an attractive name, researchers had a better chance to get published, to get grants, etc. This explains why we also have dynamic programming, quadratic programming, etc. – none of these has anything to do with programming.

A brief history of linear programming. For linear programming, reasonable efficient algorithms were known already in the 1950s – their authors got a Nobel

Prize in Economics for that. However, these algorithm required exponential time 2^n for some inputs. For some time, researchers even thought that this problem may be NP-complete.

The first breakthrough happened in 1979, when Alex Khachiyan from Moscow, Russia, proved that linear programming can be solved in polynomial time. Interestingly:

- while his algorithm was polynomial in the formal sense of this word,
- this algorithm was *not* practically feasible,

just like the examples that we had in class.

The second breakthrough occurred in 1984, when Narendra Karmarkar from IBM provided a practically feasible polynomial-time algorithm for solving linear programming problems.