Hardness of parallelization

For many problems, we know of a good single-processor algorithm, but there seems to be no way to adapt the algorithm to make good use of additional processors. A natural question to ask is whether there really is no good way to use additional processors. After all, if we can't prove that efficient parallelization is impossible, how can we know when to stop trying?

There is no known technique for addressing this question perfectly. However, there is an important partial answer worth studying. This answer uses complexity theory, the same field where people study the class ***P***, which includes problems that can be solved in polynomial time, and ***NP***, which includes many more problems, including many that people have studied intensively for decades without finding any polynomial-time solution. Some of these difficult problems within ***NP*** have been proven to be ***NP****-complete*, which means that finding a polynomial-time algorithm for that problem would imply that *all* problems in ***NP*** can also be solved in polynomial time. Since it's unlikely that all those well-studied problems have polynomial-time algorithms, a proof that a problem is ***NP***-complete is tantamount to an argument that one can't hope for a polynomial-time algorithm solving it.

To discuss parallelizability in the context of complexity theory, we begin by defining a new class of problems called ***NC***. (The name is quite informal: It stand's for Nick's Class, after Nick Pippenger, a complexity researcher who studied some problems related to parallel algorithms.) This class includes all problems for which an algorithm can be constructed with a time bound that is polylogarithmic in n(i.e., $O\left((\log n)^d\right)$ for some constant d, where n is the size of the problem input) if given a number of processors polynomial in n.

An example of a problem that is in ***NC*** is adding two big-integers together. We've seen that this problem can be solved with p processors in $O(n / p + \log p)$ time. This isn't itself polylogarithmic in n, but suppose we happened to have n processors. This is of course a polynomial number of processors, and the amount of time taken would be $O(1 + \log n) = O(\log n)$, which is polylogarithmic in n. For this reason, we know that adding big-integers is in ***NC***.

What about sorting? In Section 4 we saw an $O\left((n / p)\left((\log p)^2 + \log n\right)\right)$ algorithm. Again, suppose the number of processors were n, which is of course a polynomial in n. In that case, the time

taken could be simplified to $O((\log n)^2 + \log n) = O((\log n)^2)$, which again is polylogarithmic in n. We can conclude that sorting is also within ***NC***.

Just as with ***P*** and ***NP*** we naturally ask what the relationship is between ***P*** and ***NC***. One thing that is easy to see is that any problem within ***NC*** is also within ***P***: After all, we can simulate all p processors on just a single processor, which ends up multiplying the time taken by p. But since p must be a polynomial in n and we're multiplying this polynomial by a polylogarithmic time for each processor, we'll end up with a polynomial amount of time.

But what is much harder to see is whether every problem within ***P*** is also in ***NC*** — that is, whether every problem that admits a polynomial-time solution on a single processor can be sped up to take polylogarithmic time if we were given a polynomial number of processors. Nobody has been able to show one way or another whether any such problems exist. In fact, the situation is quite analogous to that between ***P*** and ***NP***: Technically, nobody has proven that there is a ***P*** problem that definitely is not within ***NC*** — but there are many ***P*** problems for which people have worked quite hard to find polylogarithmic-time algorithms using a polynomial number of processors: Surely at least one of these problems isn't in ***NC***!

People have been able to show, though, that there are some problems that are ***P*-complete**: That is, if the ***P***-complete problem could be shown to be within ***NC***, then in fact *all* problems within ***P*** lie within ***NC***. (Here, we're talking about ***P***-completeness with respect to ***NC***. For our purposes, this is a minor technicality; it's only significant when you study some of the other complexity classes below ***P***.) Since many decades of research indicate that the last point isn't true, we can't reasonably hope to find a polylogarithmic-time solution to a ***P***-complete problem.

While we won't look rigorously at how one demonstrates ***P***-completeness, we can at least look briefly at a list of some problems that have been shown to be ***P***-complete.

- Given a circuit of AND and OR gates and the inputs into the circuit, compute the output of the circuit.
- Compute the preorder number for a depth-first search of a dag.
- Compute the maximum flow of a weighted graph.

Knowing that a problem is ***P***-complete doesn't really mean that additional processors will never help. For instance, a graph's maximum flow can be

computed in $O(m^2 n)$ time using the Ford-Fulkerson algorithm. (And it happens there's a more complex algorithm that takes $O(m n \log (n^2 / m))$ time.) However, with many processors, we could still hope to find an $O(m^2 n / p + \sqrt{p})$ algorithm without making any major breakthroughs in the ***NC*** = ***P*** question.

Still, knowing about ***P***-completeness gives us some limits on how much we can hope for from multiple processors. Usually parallel and distributed systems can provide significant speedup. But not always. And, as we've seen, the algorithm providing the speedup is often much less obvious than the single-processor algorithm.

Source: http://www.toves.org/books/distalg/index.html#5