SCIENCE AND COMPUTER SCIENCE

ERNesto Gomez

1. Introduction - A Personal Viewpoint

We need to introduce some context here, so we can understand what we will be studying this term. So I will talk for a bit about what Computer Science is, and how it relates to other things you may already know.

Computer Science:

- Includes programming, but it is not programming.
- Involves a lot of Math and Physics, but is not part of Mathematics or Physics.
- Studies how to build computers, but is not Engineering.
- Involves the study of language, but is not Linguistics.
- Involves a lot of logic, but is not part of Philosophy.

Computer Science is, I think, an intersection of all the above, with a particular viewpoint of studying devices (computers!) that can do some of our work for us. Therefore we study algorithms - specifications of how to solve problems - and programming, which is a way of writing down an algorithm so a computer can 'understand' it. We study languages, so we can clearly and without ambiguity express what we want to the computer. We study Mathematics and Logic to analyze what is the best and fastest way of solving a problem, and to prove that our solutions are correct. We study Physics and Engineering so we can design the computers that execute our programs.

One of the biggest concerns of Computer Science is studying which problems can be solved on a computer and which can’t, which problems can be solved efficiently and which can’t. This sounds complicated enough, but it gets worse - to what extent do the limits of what can be done depend on the computer? Could it be that some problems can’t be solved, or can’t be solved efficiently, because we don’t have the right kind of computers?

This is both a challenge and an opportunity. If we were, for example, aeronautical engineers, studying what an airplane can do, we are limited by the laws of aerodynamics and Physics. As Computer Scientists, however, we not only get to consider what a program can do on a particular computer, we get to design new systems that maybe get around the limitations of our original system. It’s as if the aeronautical engineer could sometimes change the laws of Physics to improve the performance of his design! But we can’t change everything - we can build faster computers, but we can’t give one an infinite clock rate. We can change the organization and quantity of memory in a computer, but we can’t have infinite memory, and as we increase the memory size it is eventually going to get slower. We can change the playing field (the computer) to favor our programs, but there are limits to what we can do, and the limits on the hardware interact with the limits on the software.
In this course, we are going to look at one particular way of pushing the limits of both hardware and software, look at why we think it might work, what benefits we may get, and what limits we are still likely to encounter.

2. LIMITS - A PROBLEM WE CAN'T SOLVE

Let us consider an example of a problem we can't solve - consider: given a computer program P - we run it on a computer. Will P ever stop? If P runs for (1 hour, 1 day, 1 year, 1 million years) - is it simply taking a long time to solve whatever it is supposed to solve, or is it going to continue running forever?

Now, it may seem straightforward to read a program and figure out if it will finish - and this is true for some programs, but not all, or even most programs. Any program that takes any noticeable time to run on a computer has to be in a 'loop' - that is, it is repeating some set of instructions a lot of times. How do we know? Well, even a basic entry-level computer runs at over 1 Gigahertz - that is, more than one billion cycles per second, and typically executes one instruction per cycle. The largest program ever written (Windows, with 30 to 40 million lines of code) translates to less than one billion instructions - if Windows were not repeating instructions in loops, you would start it and it would finish running in less than a second. The same is true for any shorter program - that is, every other program ever written.

The problem with loops is, how do we know we will ever stop repeating the same instructions? That is, how do we know the program is not in an 'infinite loop'? You might think we could just set some very large limit for any loop and just force the program to stop repeating itself. Suppose we say 10 billion seems like a reasonable number of repetitions - we could build this into the language - if the program repeats the same instruction 10 billion times, then it has to stop. The difficulty is, then somebody will come up with a problem that requires us to loop ten billion and one times, and we won't be able to solve that problem. So we can't just limit loops, we have to allow them to repeat an arbitrarily large number of times, up to infinity.

The only general way we have to determine if a program halts or does not halt is to run it. If it does halt - in one hour, or ten years, or 15 million years - then we can answer 'Yes, it halts'. But we can never be sure that it doesn't halt. If, after a billion years, it is still running, all we know is it didn't halt in a billion years - it might halt in the next 5 minutes! OK, maybe we can say that it does not halt in some reasonable time, and just stop it because it is not practical to keep running.

3. HARD PROBLEMS

The problem 'Does program P halt?' for any program P, is called the halting problem, and it is insoluble. In practical terms, of course, we don't really care if a program will never halt, since we will stop it if it runs too long anyway. But, what do we mean by 'too long'? For the kinds of programs you write when you first learn how to program, too long is probably anything over ten seconds - if a program runs for one minute, you've made a mistake in writing it and you should just kill it and try again. But for other, more elaborate programs, hours, or even over a year may be reasonable. For example, a program I was involved with when I was a graduate student, which calculated how explosions happen when gas falls on a neutron star would run for up to a month on the fastest computer we could
get (this was a supercomputer called ASCI RED, belonging to the Department of Energy, which had 4000 Pentium processors working together).

A big concern of Computer Science is being able to estimate how long a program is likely to take when it is run. We need to do this both to see if it is going to be practical to use our program and, while testing the program, to have some idea of how long to let it run without producing results before we conclude there is something wrong with it. We make this estimate by trying to figure out what we call the 'computational complexity' - that is, we try to figure out how many operations need to be done to solve a problem, in relation to how big the problem is.

Consider sorting - putting a collection of things - for example, names, in some order - for example, alphabetical. Suppose we have a list of 10,000 names. Well, we have to do at least 10,000 operations, because we have to read the whole list (we might get lucky - it might already be sorted, but we don't know without reading it!). Suppose we didn't get lucky - we now have to compare each name with the other names. If we use the dumb, brute force approach (which at least has the virtue of being easy to program) we compare every name with every other name. So we need to read 10,000 names and perform 10,000 comparisons for each name (OK, 9,999 comparisons, but we might as well work with round numbers here - it doesn't make much of a difference). Anyway, this is 10,000 x 10,000 = 100,000,000 operations!

Another way of looking at this is that (using a dumb, brute force method), sorting a list of N items requires N x N operations, that is N squared operations. Notice that this means that, if we double the list size, we actually quadruple (2 squared = 4) the number of operations, and the time it takes to sort it. So if our program takes 1 second to sort a list of 1000 names, which seems perfectly OK for a test, how long will it take to sort the L.A. phone book? Suppose there are 1,000,000 names in the L.A. phone book. This is 1000 times longer than our list, so it will take 1000 squared = 1,000,000 times more operations. So if sorting our test list took 1 second, it takes 1,000,000 seconds to sort the L.A. phone book, or about two weeks (on the same computer, of course - we might use a faster one, but unless we have a computer 1,000,000 times faster, it is still going to take a while).

We can sort better than this - the best sort algorithms do work proportional to N x logarithm( N ) not N squared, which means that if our list is 1000 times bigger our program actually only does about 10,000 times more work, rather than 1,000,000 times. So we could sort the L.A. phone book in about three hours, not two weeks. Notice, though, that the amount of work required, and the time it takes to do it, increases faster than the problem size. This is a common situation, and something we come across in everyday life as well. For example, writing a ten page report usually takes more than ten times longer than writing a one page essay. (Or, if you do jigsaw puzzles - a 1000 piece puzzle is a lot more than twice as hard as a 500 piece puzzle!)

For some problems, the amount of work increases exponentially with the problem size, or worse. Consider an example of this: suppose a problem of size 1 takes one second, a problem of size 2 takes 2 seconds, a problem of size 3 takes 4 seconds ... every time we add 1 to the problem size we double the time. This is an exponential increase, which we express with the formula 2^N, where N is the problem size. (Lots of things increase this way - population of bacteria for example, since each
bacterium reproduces by splitting in two, so you start with 1, then 2, then 4 ... and so on.) This seems fairly innocuous, but as \( N \) gets bigger, it increases very quickly - for example, when \( N = 32, 2^N \) is four billion! If a problem of size 1 takes 1 second to solve, a problem of size 32 would take just under 127 years. Now, here is the real difficulty - we might think, since computers are getting faster all the time, all we have to do is wait a few years and we can solve this kind of hard problem with newer, faster machines.

Suppose we had a computer 1000 times faster - now our 127 year problem takes about a month to solve, which would be doable (although still inconveniently long). But, then a problem of size 33 takes twice as long - 2 months. A problem of size 34 takes 4 months. ... with our thousand times faster computer, we can only go up to a problem of size 42 before it takes over a century to solve again. Multiplying our computer power by a factor of 1000 only lets us increase the problem size from 32 to 42 (and not even that - these were problem sizes we could really not do because they took over a century). We say exponential problems are impractical to solve - no matter how fast our computers get, we can still only solve simple, small cases.

Do we really need to deal with these problems? We could hope that problems that require exponential amounts of work to do (or worse - there are problems that require even more than exponential work!) are mathematical oddities, that we would never encounter them in real applications. But no such luck - it turns out that a lot of real, important, practical problems need exponential work to solve. For example, the optimal layout of the wires on a circuit board, the most efficient path that connects a list of cities, the best way to park a truck or a space shuttle, the best way to schedule classes into classrooms in a university - all these are exponential! In many cases we can come up with decent approximate solutions in reasonable time, but sometimes we can't do even that. Problems like how a protein folds itself, which determines how various drugs will react with it or affect it, are at least exponential - maybe worse, and we don't have any good way of solving them.

4. Making computers faster

We can see that dealing with the really hard exponential problems requires computers much faster than anything we have at present. This also applies, by the way, to really large problems, like climate modeling, and the problem I mentioned before, of explosions in stars.

One key to making computers faster is making them smaller. This is due to the speed of electric current, and ultimately, to the speed of light. We know (from Physics) that nothing can go faster than light - electric current is fast, but not quite that fast (maybe 1/2 to 1/4 the speed of light). But let's take light - we have actually built computers in which signals were carried by light (or microwaves, which is the same thing). So how does this limit us? The speed of light is approximately 300,000 Kilometers/second - it is usually more convenient to write this as \( 3 \times 10^8 \)Km/sec, which means 3 followed by 5 zeroes. In more convenient units, this is \( 3 \times 10^8 \) meters/second - for those of you more accustomed to inches and feet, a meter is a bit over 39 inches, just over a yard. What does this mean in terms of computers? A typical, inexpensive, modern computer has a 2 GHz clock - this means it "ticks" at 2 Gigahertz, which is 2 billion times per second. Now, this means that things in a computer happen at the rate of the clock - one operation in the central processing unit takes 1/2 a billionth of a second. Memory isn't that fast (and disk is a lot
slower!), but we need it to be close to that speed - otherwise our super fast CPU will spend most of its time just sitting there, waiting for data and we have wasted our money buying a computer that fast. So consider - how fast does light travel in one half billionth second (we can write this as \(0.5 \times 10^{-9}\), meaning \(0.5\), but we shift the decimal 9 places to the left, giving .0000000005)? We get the distance by multiplying the velocity and the time - in this case, \(3 \times 10^8 \times 0.5 \times 10^{-9}\) which gives 1.5 \(\times 10^{-1}\), or (shifting the decimal) 0.15 meters - this is about 6 inches.

If our computer is bigger than 6 inches across, it can’t possibly respond to anything the CPU does in half a nanosecond (a billionth of a second is called a ’nanosecond’, which is easier to say than ’billionth’). Already we have a problem - even a typical laptop is bigger than that! (And you can see why the computers filling a room that appeared in old science fiction movies are a thing of the past). Our CPU chips are still OK though - they are less than one inch across, and we can build some very fast memory, put it on the same chip with the CPU (we call this cache), and arrange things so that most of the data we need gets moved into cache before we have to use it.

What happens, though, when we want to make a computer a million times faster than what we have now? We will want to build such a thing - going back to our exponential problem example, that is what we would need to handle a problem of size 50 in less than a century (which is still not very good…). Now we need our whole computer to fit into something about one micron in size (one millionth of a meter) or less. Set aside the problem of how do we connect all the needed wiring to something we need a microscope to even see (we actually can do this - the wiring inside our current chips is that small). The problem is, once we get into things that small - if the whole computer is one micron, then the actual circuits inside the computer chips would have to be almost as small as single atoms. Even if we could build, for example, a transistor (the basic unit inside many computer circuits) out of only a few atoms, things behave differently on very small scales. The physics we’re used to no longer works as we expect it to - we have to deal with what we call quantum effects. Maybe we can take advantage of this - maybe we can use some of the strange (and somewhat spooky) things that we see on atomic scales to build different, possibly more powerful computers than anything we have now.

First, though, let’s see if we can perform some extra trickery to get even more speed out of our current technology.

5. Parallelism

One thing we have been doing for the last 20 years or so, in dealing with really big problems is parallelism. That is, we split up the problem in many tiny chunks, and solve each chunk on a separate computer, at the same time. Sometimes this works very well - for example, computer animation often gets done this way - it would take a long time to generate a whole movie on a single computer, but by generating separate frames on each of, say, 1000 machines, you can do in days what would otherwise take years. Parallelism doesn’t always work - if one woman can produce one baby in 9 months, how long will it take 9 women to produce 1 baby? But when it works, it can be very useful. The difficulty is, for some really large problems we need too many computers for parallelism to be practical, particularly if they need to communicate with each other.
Some systems are what we call trivially parallel - SETI (Search For Extraterrestrial Intelligence) At Home is this kind of problem - they get signals from radio telescopes pointed at different regions of the sky, in different frequencies of radio, and split them up into little independent chunks of data (figuring that, if there is an alien message it comes from ONE direction in ONE frequency). They send the data out to volunteers who run screen savers that analyze the data to see if it is random noise or could be an intelligent signal. Since each chunk of data is independent, there is no need for all the volunteer computers to be running at the same time or to talk to each other, so the system can (and does) use several million computers - SETI At Home is probably the most powerful computer system on Earth, but it is only useful for one particular problem.

If we want to tackle things like protein folding using parallelism, we need a lot of computers, but we need them to run at the same time and communicate with each other while they are solving the problem. IBM is currently building a machine to do just that - it is called Blue Gene, and will have one million Pentium-equivalent processors working together when it is complete. This, though, is a brute force approach, and can’t expand very well. Specifically - if we have a big organic molecule, with up to one million atoms, we’re OK - we assign one atom to each Pentium. But if we need to handle something a lot bigger than that (DNA, maybe?) then we’re stuck - we’re back into problems that grow exponentially.

What we’d like is a system in which the number of things we can do in parallel increases as the problem size does - effectively, the computer gets bigger for bigger problems - the work increases exponentially, but the computer grows exponentially too, so the amount of time it takes to solve the problem stays the same. Sounds like magic? (Should we also wish for something that turns lead into gold or maybe oil?). It turns out, though, that the same quantum effects that we run into when making computers very small might allow us to do exactly this!

It looks like, no matter what we do to speed up computers, we come back to using quantum effects. So let us consider exactly what this means.

6. Experiments

Examples

**double slit:**

waves in water (view from top) entering bay
bullets shot at double slit steel plate
light waves
electrons

**photoelectric effect:**

6.1. **Implications.** results of : wave/particle duality, measurement -> collapse of wave function

**uncertainty principle**

**collapse of wave function - entanglement**