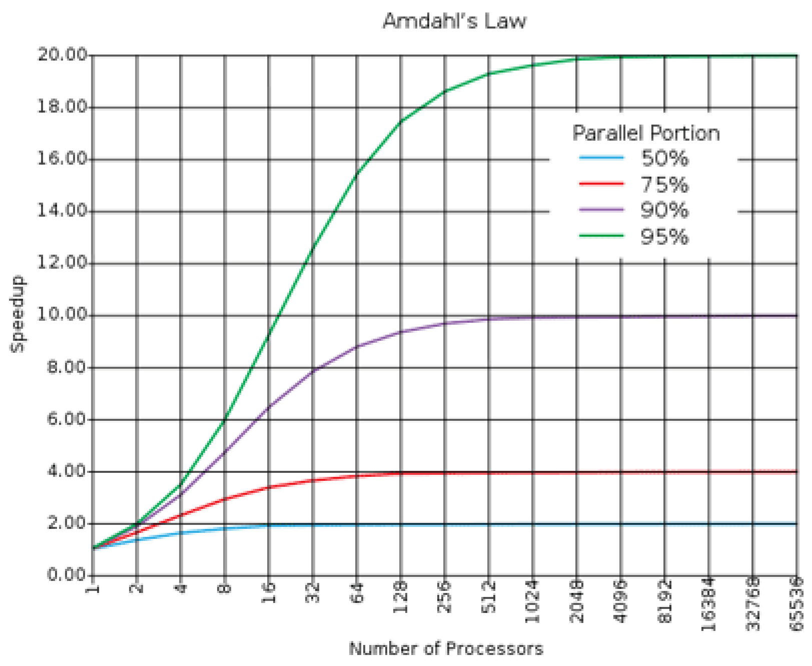
NOT ENTIRELY UNLIKE LINEAR SCALING

The difficulty of obtaining linear scaling is not due to the language itself, but rather to the nature of the problems to solve. Problems that scale very well are often said to be *embarrassingly parallel*. If you look for embarrassingly parallel problems on the Internet, you're likely to find examples such as ray-tracing (a method to create 3D images), brute-forcing searches in cryptography, weather prediction, etc.
From time to time, people then pop up in IRC channels, forums or mailing lists asking if Erlang could be used to solve that kind of problem, or if it could be used to program on a GPU. The answer is almost always 'no'. The reason is relatively simple: all these problems are usually about numerical algorithms with lots of data crunching. Erlang is not very good at this.
Erlang's embarrassingly parallel problems are present at a higher level. Usually, they have to do with concepts such as chat servers, phone switches, web servers, message queues, web crawlers or any other application where the work done can be represented as independent logical entities (actors, anyone?). This kind of problem can be solved efficiently with close-to-linear scaling.

Many problems will never show such scaling properties. In fact, you only need one centralized sequence of operations to lose it all. **Your parallel program only goes as fast as its slowest sequential part**. An example of that phenomenon is observable any time you go to a mall. Hundreds of people can be shopping at once, rarely interfering with each other. Then once it's time to pay, queues form as soon as there are fewer cashiers than there are customers ready to leave.
It would be possible to add cashiers until there's one for each customer, but then you would need a door for each customer because they couldn't get inside or outside the mall all at once.

To put this another way, even though customers could pick each of their items in parallel and basically take as much time to shop whether they're alone or a thousand in the store, they would still have to wait to pay. Therefore their shopping experience can never be shorter than the time it takes them to wait in the queue and pay.

A generalisation of this principle is called Amdahl's Law. It indicates how much of a speedup you can expect your system to have whenever you add parallelism to it, and in what proportion:



According to Amdahl's law, code that is 50% parallel can never get faster than twice what it was before, and code that is 95% parallel can theoretically be expected to be about 20 times faster if you add enough processors. What's interesting to see on this graph is how getting rid of the last few sequential parts of a program allows a relatively huge theoretical speedup compared to removing as much sequential code in a program that is not very parallel to begin with.

Don't drink too much Kool-Aid:
Parallelism is *not* the answer to every problem. In some cases, going parallel will even slow down your application. This can happen whenever your program is 100% sequential, but still uses multiple processes.
One of the best examples of this is the *ring benchmark*. A ring benchmark is a test where many thousands of processes will pass a piece of data to one after the other in a circular manner. Think of it as a game of telephone if you want. In this benchmark, only one process at a time does something useful, but the Erlang VM still spends time distributing the load accross cores and giving every process its share of time.
This plays against many common hardware optimizations and makes the VM spend time doing useless stuff. This often makes purely sequential applications run much slower on many cores than on a single one. In this case, disabling symmetric multiprocessing (`$ erl -smp disable`) might be a good idea.

Source : http://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency