CI583: Data Structures and Operating Systems
Algorithms for parallel computing

Outline

Moore's Law slows down

For several decades, computer manufacturers were able to increase the processing power and memory capacity of individual CPUs at an exponential rate.

In 1965 the original expression of Moore's Law stated that the number of components in a CPU would double every year and it proved to be accurate. By the 2000s this had slowed down to a doubling in power every 18 months to two years, and the trend is likely to continue.

Manufacturers increasingly focus their efforts on increasing the number of CPUs available to a single machine.

Parallel computing and the cloud

At the same time, advances in parallel computing make it easier to distribute a problem amongst cores and amongst remote machines.

One of the big contributors to this is the advent of cloud computing and services such as Amazon S3 which provide flexible storage and processing power distributed across large clusters of commodity machines (as opposed to very expensive supercomputers, available only to tiny numbers of researchers).

Popular parallel programming

As an example of how simple parallel computing is becoming, open frameworks such as Hadoop[1] allow the easy creation of large clusters, something which was a research topic in itself quite recently.

The Akka[2] framework (for Java and Scala) allows an algorithm to be distributed over a large number of nodes. Their "Hello World" example is a distributed algorithm for computing π to arbitrary decimal places, and does so in very few lines of code.

---

[1] `http://hadoop.apache.org/`

[2] `http://doc.akka.io/docs/akka/2.0.1/intro/getting-started-first-java.html`

Big Data

At the same time the way we use computers, and the internet especially, means that massive amounts of data are generated every hour, minute, second...

If you had to design an algorithm to mine Twitter for upcoming trends, how would you do it? Would you design it to run on a single machine?

Big Data is the term coined for this environment: sequential algorithms (and other traditional technologies such as relational databases) break down in the face of very large datasets but parallelism can help to make many of the problems easier.

MapReduce

A prominent example of this new paradigm is Google MapReduce. The original paper[3] is well worth reading.

MapReduce is a *design pattern* rather than an algorithm. It depends on breaking down a problem (such as processing a large log file to count the requests from each country, from example) into many small parts with no shared state so that each part can be worked on separately by "worker nodes". The results are later combined by a "master node".

[3] http://research.google.com/archive/mapreduce.html

MapReduce

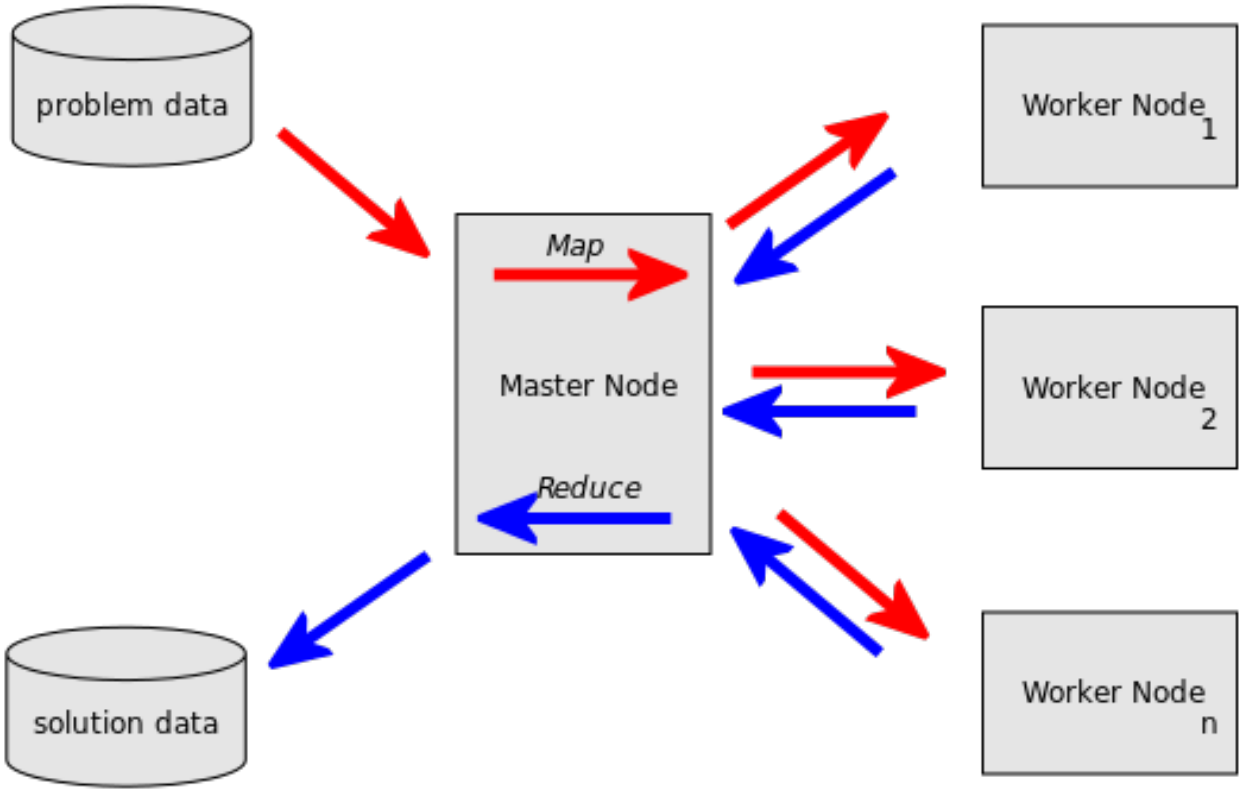

Image ©http://en.wikipedia.org/wiki/MapReduce

A model for parallelism

The main subject of this lecture is not frameworks such as MapReduce or Akka, but the principles of designing algorithms that can run inside them.

We need to develop a model for analysing parallel algorithms. It will include the notions of speed up and cost and scalability.

A model for parallelism

The speed up is the improvement a parallel algorithm provides over an optimal sequential version.

So, we know that sequential MergeSort runs in $O(n \log n)$ time. If we can provide a parallel version that runs in $O(n)$ time then the speed up is $O(\log n)$. Note that we count each processor taking a step as one step – we are measuring time, not work.

The cost of a parallel algorithm is the measure of work: the time it takes multiplied by the processors required. If our $O(n)$ parallel MergeSort requires at least n processors, then its cost is $O(n^2)$.

A model for parallelism

The scalability of a parallel algorithm is related to the number of processors required.

If our parallel MergeSort requires n processors, it is not usable for large values of n. The sequential algorithm has no such size restriction.

We are only interested in scalable parallel algorithms: those for which the numbers of processors required is significantly less than the size of any potential input and where we can increase the input size without needing to add processors.

Summary

Parallel algorithms require careful design, especially in the presence of shared state.

One way around this is to design our algorithms in a functional style, so that they don't require shared state. An algorithm like this should be easy to deploy in a framework like Akka. In fact, libraries like Akka intend to make many of the error-prone parts of parallel programming happen "under the bonnet".

Either way, making use of the multiple cores available on a typical modern machine and of environments like the cloud are increasingly important techniques. Parallel programming is for all of us nowadays!

Advice from a Grand Master Programmer

Rob Pike (inventor of `C`, author of the Go programming language and many other things) has this to say about program complexity[4]:

Pike's Rules of Complexity

Most programs are too complicated - that is, more complex than they need to be to solve their problems efficiently. Why? Mostly it's because of bad design [...]. But programs are often complicated at the microscopic level, and that is something I can address here.

[4] Some of Rob Pike's thoughts on programming:
`http://doc.cat-v.org/bell_labs/pikestyle`

Advice from a Grand Master Programmer

- **Rule 1**. *You can't tell where a program is going to spend its time*. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.
- **Rule 2**. *Measure*. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

Some of Rob Pike's thoughts on programming: `http://doc.cat-v.org/bell_labs/pikestyle`

Advice from a Grand Master Programmer

- **Rule 3**. *Fancy algorithms are slow when n is small*, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.) For example, binary trees are always faster than splay trees for workaday problems.

Some of Rob Pike's thoughts on programming: `http://doc.cat-v.org/bell_labs/pikestyle`

Advice from a Grand Master Programmer

- **Rule 4**. *Fancy algorithms are buggier than simple ones*, and they're much harder to implement. Use simple algorithms as well as simple data structures.
  The following data structures are a complete list for almost all practical programs:
  - array
  - linked list
  - hash table
  - binary tree

  Of course, you must also be prepared to collect these into compound data structures. For instance, a symbol table might be implemented as a hash table containing linked lists of arrays of characters.

Some of Rob Pike's thoughts on programming: `http://doc.cat-v.org/bell_labs/pikestyle`

Advice from a Grand Master Programmer

- **Rule 5**. *Data dominates*. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming. (See The Mythical Man-Month: Essays on Software Engineering by F. P. Brooks, page 102.)
- **Rule 6**. *There is no Rule 6*.

Some of Rob Pike's thoughts on programming: `http://doc.cat-v.org/bell_labs/pikestyle`