Predicting Performance

Adrian Clark

2015–16

Speed-up

If a program runs in T_1 seconds on one machine and T_2 on another, we say the speed-up is

$$S = \frac{T_1}{T_2}$$

So if the second computer takes half the time, $T_2 = \frac{T_1}{2}$ and $S = 2$

In a similar vein, if N identical processors are able to spread the computation amongst themselves perfectly, the time taken is $\frac{T_1}{N}$ and $S = N$

Amdahl's law

If there is some fraction f that *must* be performed serially, the fastest we can do with N processors is

$$f + \frac{1-f}{N}$$

so the best speed-up we can achieve is

$$S \leq \frac{1}{f + \frac{1-f}{N}}$$

This is known as Amdahl's law.

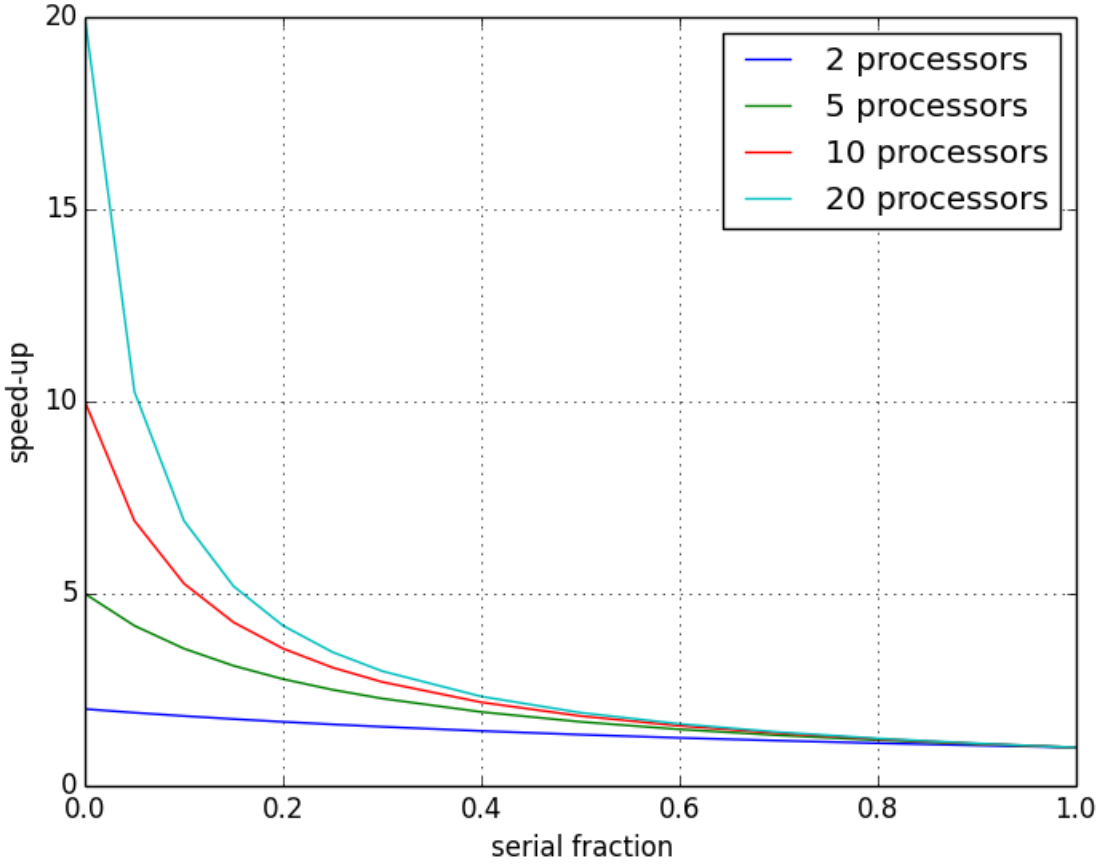

Figure 1:Amdahl's law

The ultimate speed-up is achieved *only* when everything can be done in parallel

Even when a fairly modest proportion of a program cannot be performed in parallel, the performance hit is drastic

Hence, when performing computation over a cluster, any processing that must be performed centrally rather than distributed out is the rate-determining factor, irrespective of the number of processors available

MapReduce

MapReduce has become established as an approach to processing large amounts of data on a cluster, as long as the algorithm can be parallelised and run in MIMD fashion

The name *MapReduce* is inspired by the `Map` and `Reduce` functions commonly found in functional programming, though it is not really the same thing

The **Map** step partitions the overall task into a series of queues and distributes them to worker processors

Workers may themselves partition the problem a distribute them onward, leading to a tree of workers

Each worker node processes its own problem, then passes its result back to the master node

The **Reduce** step combines the results received from the worker nodes into the overall solution to the problem

Apache's *Hadoop* is essentially an implementation of the MapReduce approach

Although MapReduce is claimed to be novel, we can see that the underlying principle is essentially the one used by BOINC, HTCondor and others

In fact, we did a fair amount of work on this kind of computation here at Essex during the 1990s, which we called a *pipeline of processor farms* (PPF) for reasons that shortly will become obvious

Whatever the origin of this concept, it gives us an *approach* for implementation, not a way of working out the best way to distribute the work

To be able to work out how best to distribute work out, we need to understand the different ways in which a task can be speeded up

Types of parallelism

There are actually several different types of parallelism that can be exploited:

- **Data parallelism** or **geometric multiplexing** is when a dataset can be distributed over multiple processors
- **Algorithmic parallelism** is when parts of the same algorithm can be performed on multiple processors
- **Process farming** or **temporal multiplexing** is when different processors can each process a complete task in parallel

The effects of these are perhaps best understood diagrammatically

Exploiting parallelism in a pipeline

Consider a series of processing stages, where the time in each box is how long that stage takes to execute

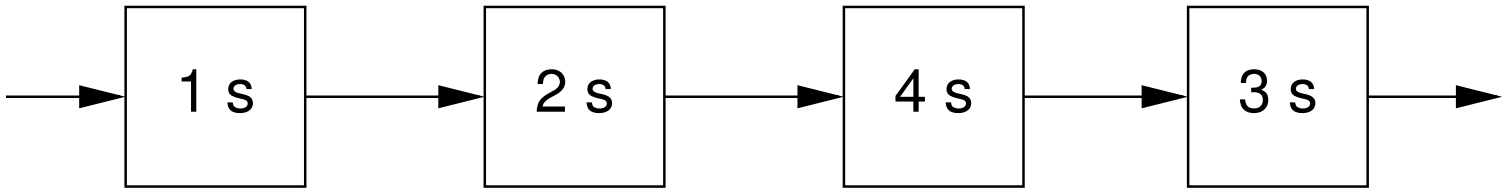

Figure 2:

When this is run on a single processor, the dataset takes 10 sec to be processed (the *latency*), and so the throughput is $\frac{1}{10}$ datasets/sec

If each stage in the pipeline is put onto a separate processor, a dataset is completed every 4 second, and so the throughput of the pipeline is $\frac{1}{4}$ jobs/sec — however, the latency is now 15 sec

When we introduce temporal multiplexing, we are able to run a separate dataset on each worker in the pipeline stages

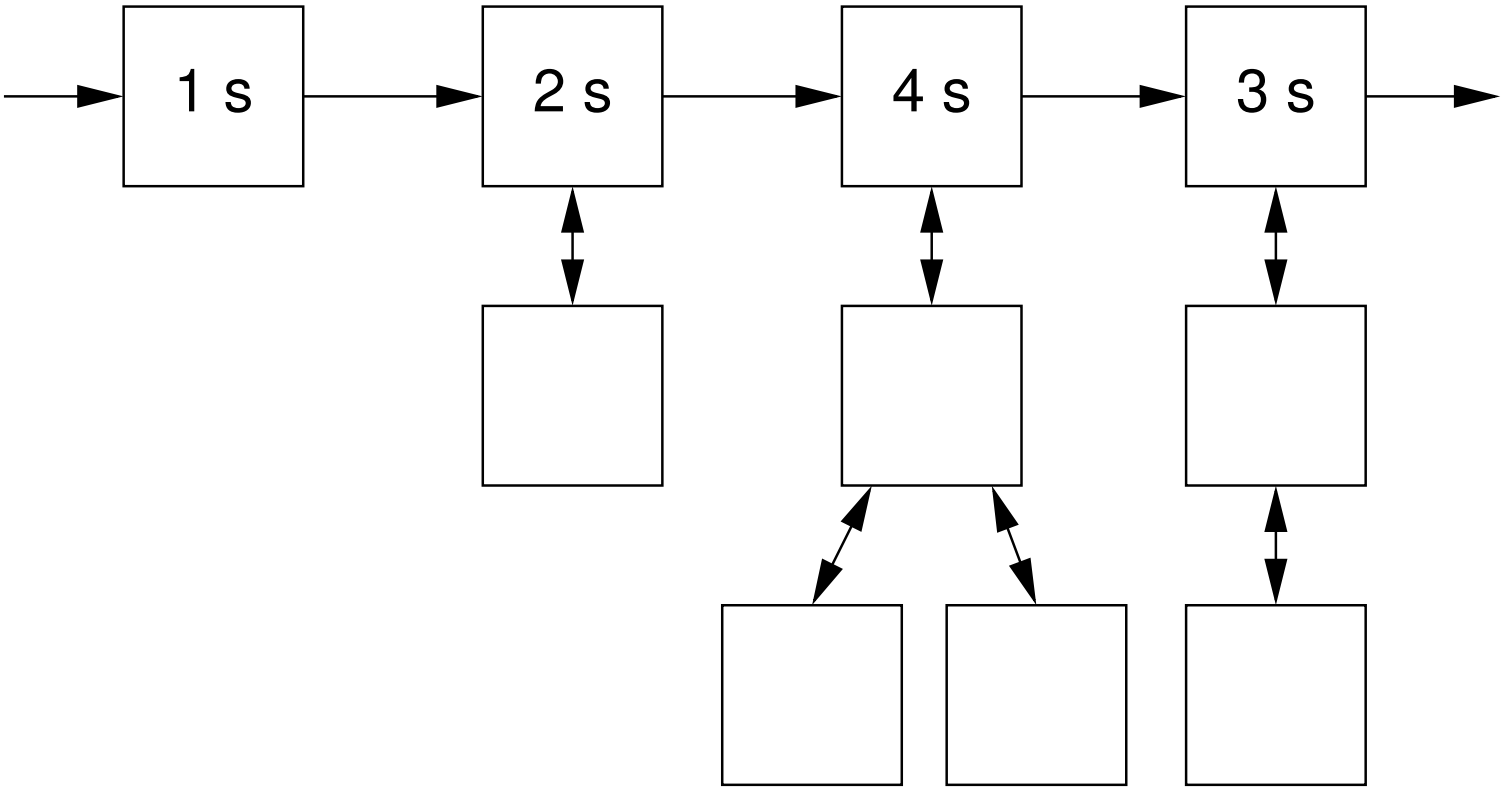

Figure 3:

The throughput is now 1 dataset/sec, and the latency is 10 sec — though we could get the same result by simply having the 10 processors work on a

If, rather than temporal multiplexing, we are able to exploit data or algorithm parallelism and make each stage in the pipeline take the same amount of time to execute, we again have a throughput of 1 dataset/sec but the latency has been reduced to 4 sec

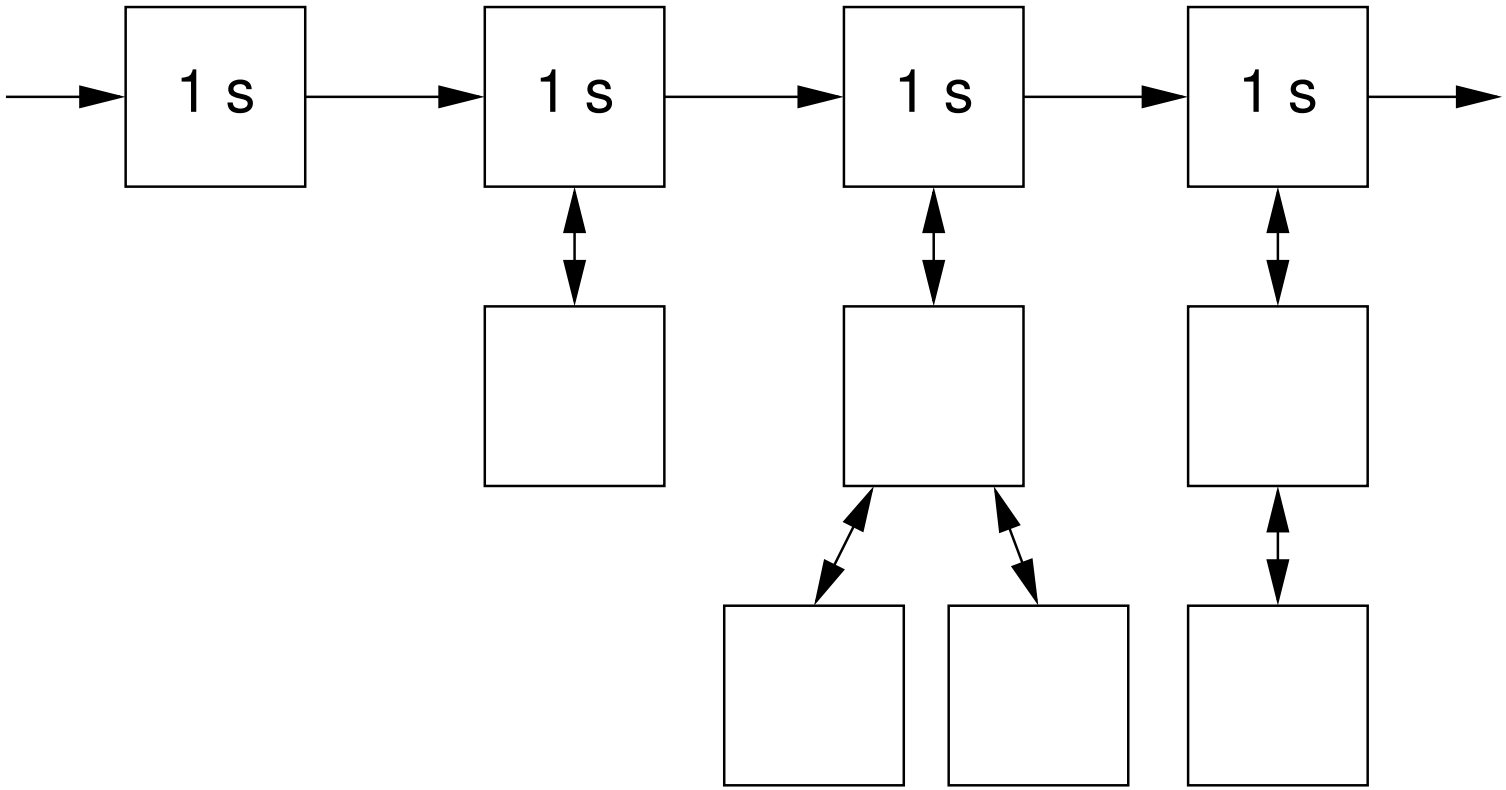

Figure 4:

Synchronous and asynchronous pipelines

If a task can be decomposed into parts that can proceed independently along the pipeline, it can be scheduled independently at each stage and we say it is *asynchronous*

Conversely, in a *synchronous* pipeline, the next stage of processing cannot proceed until the previous one has finished

For example, a job that prints utility bills for houses that were read on the same day would be asynchronous (even though the data may have to be re-ordered at the end of the pipeline)

Conversely, a job that converts the frames of a video sequence into an MPEG movie would be synchronous, as the motion of features within a frame have to be specified relative to their locations in the previous frame

Asynchronous pipelines

If we denote the time taken at each stage i of of S stages arranged in a simple pipeline as T_i, then the slowest stage of a pipeline with no parallelism is

$$T_{max} = \max(T_i, i = 1, \ldots S)$$

The stage at which this occurs is

$$i_{max} = \text{argmin}(T_{max} = T_i, i = 1, \ldots S)$$

The latency is then given by

$$i_{max} T_{max} + \sum_{k=i_{max}+1}^{S} T_k$$

The throughput is given by

$$\frac{1}{T_{max}}$$

Asynchronous pipelines with temporal multiplexing

If we introduce temporal multiplexing with enough processors to prevent waiting at any stage, we have

$$\text{latency} = \sum_{i=1}^{S} T_i$$

$$\text{throughput} = \frac{p_1}{T_1} = \frac{p_2}{T_2} = \ldots \frac{p_S}{T_S}$$

where p_i is the number of processors at stage i

Asynchronous pipelines with geometric multiplexing

If we are able to decompose individual stages in the pipeline by exploiting parallelism so that no task waits, we have as a worst case

$$\text{latency} = \sum_{i=1}^{S} \frac{T_i}{p_i}$$

$$\text{throughput} = \frac{p_1}{T_1}$$

Synchronous pipelines

Let us define:

- $\alpha(i) = 1$ if i is an asynchronous stage
- $\alpha(i) = 0$ if i is a synchronous stage

For a synchronous pipeline with no waiting, we can then write

$$\text{latency} = \sum_{i=1}^{S} \delta(\alpha(i))\, T_i + \sum_{i=1}^{S} \alpha(i) \frac{T_i}{p_i}$$

where $\delta(\cdot)$ is the Kronecker delta function

Incorporating communication

The above expressions assume that communication is instantaneous; in practice, this is not the case

The important time is not the actual transmission of the Ethernet frame but the time involved in setting up the communication at both ends of a link

If T_c is the time taken to perform a calculation and T_x is the transfer time between processors, we can adapt the expressions derived above to include transfers, obtaining expressions like

$$NT_c + (N + 1)T_x$$

for a pipeline in which all processing steps take place in lockstep but communication does not

Partitioning code for clusters

Our consideration to date gives some general principles for mapping programs onto clusters

We have seen that there are different types of parallelism in problems, which need to be accommodated in different ways

The first, vital stage in working out how to map a program onto any non-serial system is to determine how much time is spent where — to profile the code

What a profiler does

Instruments your code so that it can

- count the number of times each function is called
- sample which function execution is in at regular intervals

Counts of function calls are always accurate

Timings of routines in which little execution time is spent are not accurate

How to profile

Many IDEs have profiling built in

For compiled languages, you can use gprof

```
gcc -o myprog myprog.c -g -pg
./myprog   # generates gmon.out
gprof myprog
```

For Python, use `cProfile`

```
def main ():
    ...
cProfile.run('main ()')
```

```
   3032126627 function calls in 12665.551 seconds
   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000 12665.551 12665.551 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 pcov:127(sort_by_value)
        1    0.000    0.000 12665.551 12665.551 pcov:137(main)
      528 12195.198   23.097 12660.026   23.977 pcov:22(coverage)
       11    0.084    0.008 12665.550 1151.414 pcov:42(mean_coverage1)
      528    1.749    0.003    5.441    0.010 pcov:9(load_locations)
        1    0.000    0.000    0.000    0.000 {len}
3029593050  448.954    0.000  448.954    0.000 {math.sqrt}
      528    0.051    0.000    0.051    0.000 {method 'close' of 'file' objects}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profile
        1    0.000    0.000    0.000    0.000 {method 'items' of 'dict' objects}
      528    2.257    0.004    2.257    0.004 {method 'readlines' of 'file' objects
        1    0.000    0.000    0.000    0.000 {method 'sort' of 'list' objects}
  1264623    1.313    0.000    1.313    0.000 {method 'split' of 'str' objects}
      528    0.006    0.000    0.006    0.000 {numpy.core.multiarray.zeros}
      528    0.039    0.000    0.039    0.000 {open}
  1265768   15.899    0.000   15.899    0.000 {range}
```

In the profiler output:

- `tottime` is the time spent in a particular routine
- `cumtime` is the total time spent in a particular routine *and those it invokes*

Output like this is invaluable in helping work out how to distribute computation over a set of machines