Amdahl's Law

- Gene Amdahl — designer of the IBM 360 mainframe
- Any task can be split into parts that can be parallelized and parts that can't
- Serial parts — setup of problem by reading in data, generating statistics after each iteration
- Parallel parts — numerical solver, integration of Newton's laws

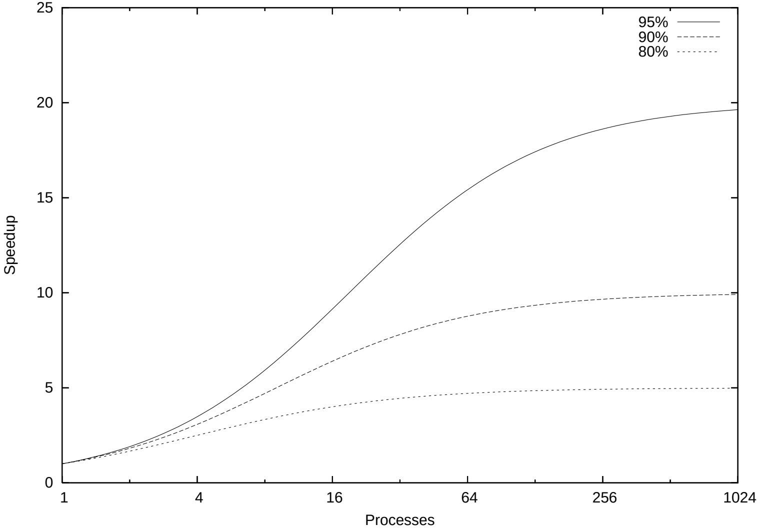
Amdahl's Law

- Say we have task of which 95% can be executed in parallel
- Even if we use an infinite number of processes on the parallel part we still need 5% of the original time to execute the serial part
- Maximum speed up is given by the formula

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

where S is the proportion of the code to be executed in serial and N is the number of processes in the parallel part

Amdahl's Law



Amdahl's Law

- This suggests that there is no point in writing code for more than 8-16 processes
- Not true!
- As you run on larger problem sizes often the serial part scales linearly but the parallel part scales with n^2 or greater
- By tackling larger problems a 95% parallel problem can become a 99% parallel problem and eventually a 99.999% parallel problem
- Plugging in the figures a 99.9% problem on a 1024 way system gives a 506 speedup

Amdahl's Law

- Of course Amdahl's law assumes the parallel part is perfectly parallelizable
- It does not take into consideration the time spent passing data between processes
- There is a more complex form that takes transfer times and network latencies into account
- For our purposes the simple version is sufficient

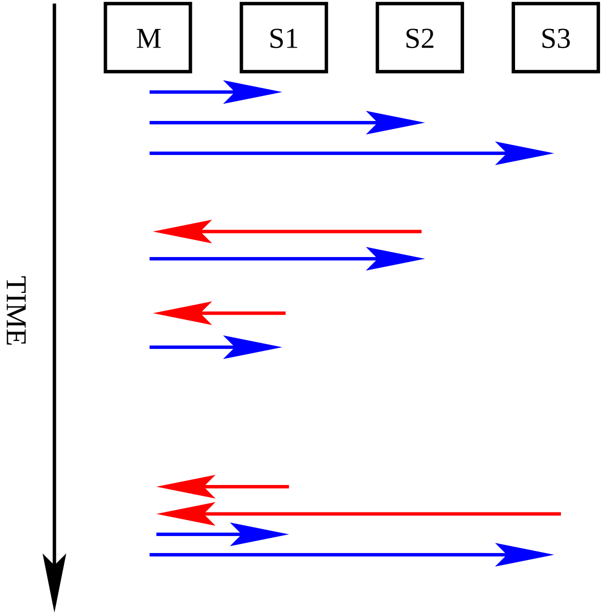
Master-Slave model

- The Master-Slave model is one of the standard models in parallel programming
- It is used when the problem is *embarassingly parallel* i.e. each task is independant of the others
- Effectively a 100% parallel problem from Amdahl's perspective
  - Brute force attacks in cryptography
  - Non-realtime rendering for animation
  - Ray tracing
  - Pretty much all of bio-informatics
  - Genetic algorithms
- The model used in things like BOINC / *-at-home
- Sometimes called the manager-worker model to be slightly more politically correct

Master-Slave model

- One process in the calculation (usually rank 0) is designated the master
- It generates work and then allocates it to the other processes
- Each slave carries out its given task and sends the results back to the master
- If there is further work to do the master sends the slave a new task
- If there is no more work then the slave waits

Master-Slave model



Master-Slave model

- There is no mapping of work to processes — any task can be carried out by any process
- Need to make sure that the master is not a bottleneck. Can occur if
  - If the tasks are too small
  - Too many workers
- Choose size of tasks so the time taken to do the task is much larger than time taken to transfer work to slave and result back to master
- Waiting times are also reduced if the time taken for each task is somewhat random

Master-Slave model

Master

```
if (rank == 0) {
    for (i=1;i<size;i++)
        send_next_job(i);
    while (jobs_left) {
        get_result(s);
        send_next_job(s);
    }
    for (i=1;i<size;i++) {
        get_results(s);
        no_more_jobs(s);
    }
```

Slave

```
} else {
    while (1) {
        get_next_job();
        if (no_job)
            break;
        do_work();
        send_result();
    }
}
```

Master-Slave model

- Need to have some agreed message to indicate the work has all been completed
  - Can use the tag field for this
- Master needs to know who has send him work
  - Use *MPI_ANY_SOURCE* in the recv
  - Use the *MPI_STATUS* object to determine the actual source

Genetic Algorithms

- Genetic algorithms are used to search a solution space for the optimum
- They are based on Darwinian "survival of the fittest" concepts
- The solution space is modelled using a string — the chromosone
- A population of chromosones is generated
- They are selected according to their "fitness"
- They may crossover to form new offspring
- Random mutation may occur in new offspring

Genetic Algorithms

- Chromosones are usually modelled as strings of 1s and 0s
- In C we usually use the bit representation of a char or int
- For initialisation we often just randomly fill the population
- Need to make sure they represent valid solutions to the problem
- The crossover operation is the key to GA
  - Choose a location in the string
  - Exchange the subsequences after this locus between two chromosones to create two offspring
  - Given parents 10000100 and 11111111 to be crossed after the third bit
  - Offspring are 10011111 and 11100100

Genetic Algorithms

- Mutation occurs very rarely (0.1%) but are important in escaping local maximums
- The rate is sometimes varied as the population becomes more homogeneous
- Each iteration is called a generation
- Normally a GA is run for 100 or more generations

Genetic Algorithms

- Using chromosone of size 8 with a fitness function that counts the number of 1s in the string
-

Label	String	Fitness
A	00000110	2
B	11101110	6
C	00100000	1
D	00110100	3

- Might select B and D to be 1st set of parents and B and C to be the 2nd set — parents are chosen randomly by fitness (roulette wheel method)
- B and D crossover after first bit to give new offspring E (10110100) and F (01101110)
- B and C do not crossover

Genetic Algorithms

- Next mutation occurs. Say E is mutated at location 6 and B is mutated at location 1
- This gives us a population for the next generation
-
Label	String	Fitness
$\hat{E}$	10110000	3
F	01101110	5
C	00100000	1
$\hat{B}$	01101110	5

- Even though the highest fitness value is now 5 compared to 6 previously, the overall fitness of the population has increased from 12 to 14
- Eventually, if iterated often enough, this will result in a population of srings with all 1s

Genetic Algorithms

- Where are genetic algorithms useful?
- Quite often used in scheduling and timetabling problems
- Problems with a huge solution space — can't evaluate all possible solutions
- Problems with complex solutions spaces where normal techniques (simplex/hill climbing) might get stuck
- Protein folding and docking
- Travelling salesman problem

Prisoners' Dilema

- A simple two-person game invented in the 1950s
- Two people (Alice and Bob) are held in separate cells in the cop shop accused of committing a crime
- A deal is offered to each — testify against the other, you will get off free and the other will get 5 years
- If both people take the deal then both have discredited the other's testimony and both get 4 years
- If both people don't take the deal then both are convicted of lesser charges and get 2 years each
- What should each person do?

Prisoners' Dilema

- Create a reward matrix. Base this on the number of years off their prison time
-
	Co-operate	Defect
Co-operate	3,3	0,5
Defect	5,0	1,1

- What is the best strategy to maximize your own payoff?
- Take the deal! Your average time off is $\frac{5+1}{2} = 3$ as opposed to $\frac{3+0}{2} = 1.5$
- The game becomes more interesting if this process is iterated — you play several games in a row
- Both players always taking the deal leads to a much lower payoff than both not taking the deal
- How can you make your opponent co-operate for mutual benefit?
- Act based on how the previous games went

Prisoners' Dilema

- Use a genetic algorithm to evolve strategies
- How to encode the problem into a string of bits?
- 4 possibilities from previous game
  - CC — both players co-operate with each other
  - CD — player 1 co-operates but player 2 stabs
  - DC — player 2 co-operates but player 1 stabs
  - DD — both players stab each other
- Invent a rule to apply based on what happened in the previous game
- If you look over the prevous two games there are 16 possibilities
  - CC CC
  - CC CD
  - ...
  - DD DD

Prisoners' Dilema

- To evaluate the fitness of each strategy, play a round-robin tournament between each strategy
- The fitness is the total payoff from all the games
- Then use this fitness value to select the next generation, perform cross-over and mutation and keep going

Parallel GA

- Genetic algorithms are very good targets for parallelising using the Master-Slave paradigm
- Generally fitness evaluation is the most expensive part of the GA
- Most of the time the evaluation of fitness can be split into independent operations
  - Counting bits in a string — each chromosone can be evaluated separately
  - Prisoners Dilema' — fitness of chromosones are inter-dependant but matches between sets of chromosones can be evaluated separately
- While the crossover/mutation code can be a bit messy once you have written it once it can be reused and it executes quickly