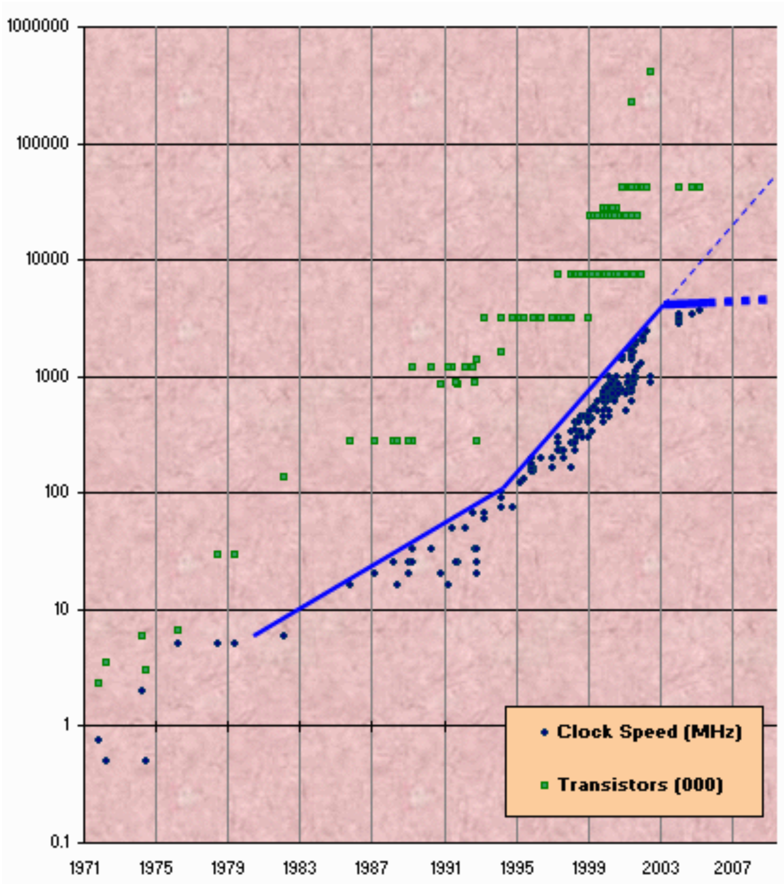
Parallel Patterns

Objectives

- Why should I care about parallelism?
- Types of parallelism
- Introduction to Parallel Patterns

The free lunch is over



- Herb Sutter, Dr Dobb's Journal 30th March 2005

The continual need for speed

- For a given time frame X
  - Data volume increases
    - More detail in games
    - Intellisense
  - Better answers
    - Speech recognition
    - Route planning
    - Word gets better at grammar checking

Expectations

- Amdahl's Law
  - If only n% of the compute can be parallelised at best you can get an n% reduction in time taken.
  - Example
    - 10 hours of compute of which 1 hour cannot be parallelised.
    - So minimum time to run is 1 hour + Time taken for parallel part, thus you can only ever achieve a maximum of 10x speed up.
- Not even the most perfect parallel algorithms scale linearly
  - Scheduling overhead
  - Combining results overhead

Coarse grain parallelism

- Large tasks, little scheduling or communication overhead
  - Each service request executed in parallel, parallelism realised by many independent requests.
  - DoX,DoY,DoZ and combine results. X,Y,Z large independent operations
- Applications
  - Web Page rendering
  - Servicing multiple web requests
  - Calculating the payroll

Fine grain parallelism

- Fine grain parallelism
  - A single activity broken down into a series of small co-operating parallel tasks.
    - f(n=0…X) can possibly be performed by many tasks in parallel for given values of n
    - Input => Step 1 => Step 2 => Step 3 => Output
  - Applications
    - Sensor processing
    - Route planning
    - Graphical layout
    - Fractal generation (Trade Show Demo)

Parallel Patterns

- Fine grain parallelism
  - A single activity broken down into a series of small co-operating parallel tasks.
    - $f(n=0...X)$ can possibly be performed by many tasks in parallel for given values of n
    - Input => Step 1 => Step 2 => Step 3 => Output
  - Applications
    - Sensor processing
    - Route planning
    - Graphical layout
    - Fractal generation (Trade Show Demo)

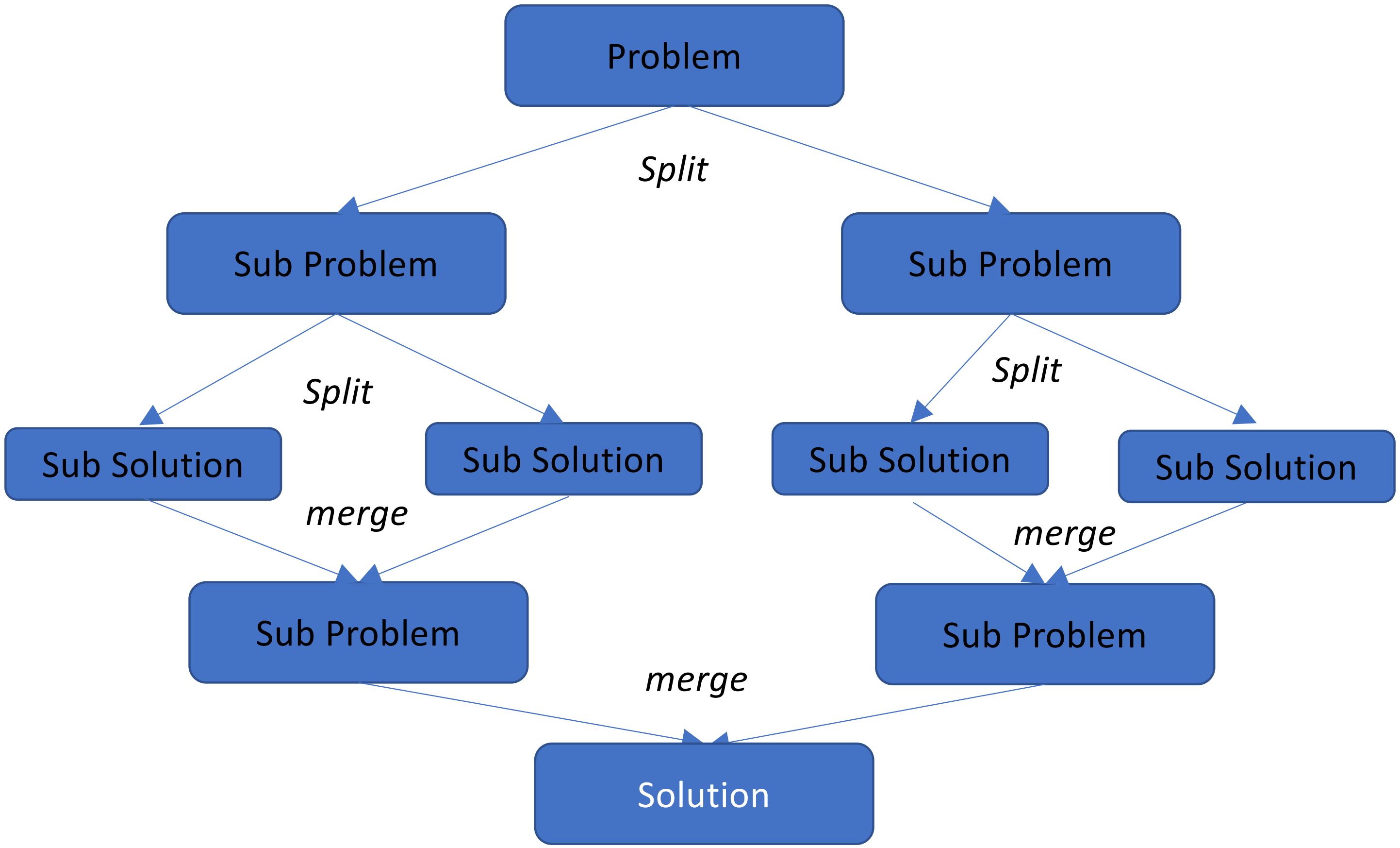
Fork and join

- Book Hotel, Book Car, Book Flight
- Sum X + Sum Y + Sum Z

```
Parallel.Invoke( BookHotel , BookCar , BookFlight );

// Won't get here until all three tasks are complete
```

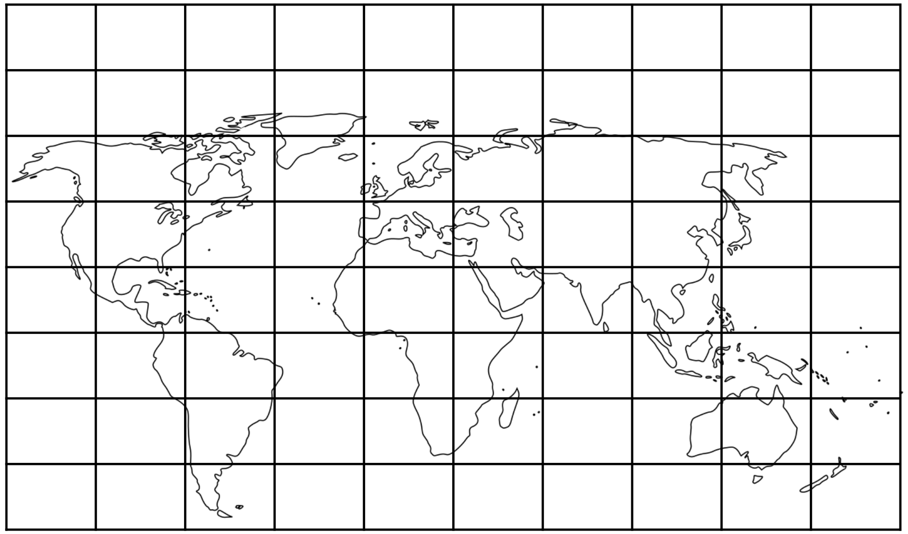
```
Task<int> sumX = Task.Factory.StartNew(SumX);
Task<int> sumY = Task.Factory.StartNew(SumY);
Task<int> sumZ = Task.Factory.StartNew(SumZ);

int total = sumX.Result+ sumY.Result+ sumZ.Result;
```

Divide and Conquer



Geometric decomposition

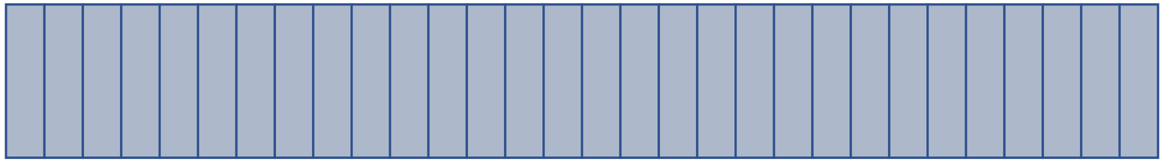
- Task parallelism is one approach, data parallelism is another
- Decompose the data into a smaller sub set
- Have identical tasks work on smaller data sets.
- Combine results.



How to partition?

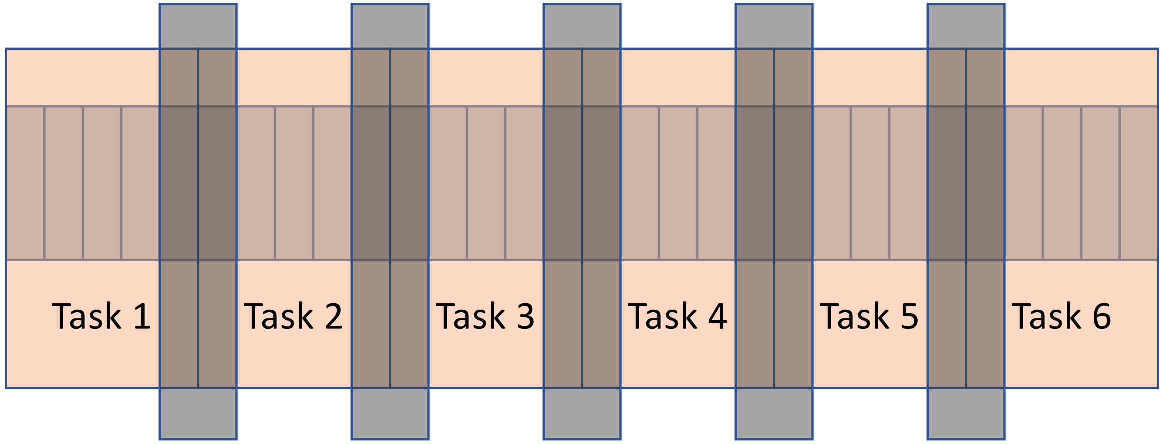
- How many parts
- How to divide
  - Partition 0, 0,4,8,12 …
  - Partition 1,1,5,9,13 …
  - Partition 2,2,6,10,14 …
  - Partition 3,3,7,11,15 …
- Or
  - Partition 0, 0 – 25 %
  - Partition 1, 25 – 50%
  - Partition 2, 50 – 75%
  - Partition 3, 75 – 100%

Array of data



Edge complications

- Each task runs on its own sub region, often they need to share edge data
- Update/Reading needs to be synchronised with neighbouring tasks



Parallel Loops

- Loops can be successfully parallelised IF
  - The order of execution is not important.
  - Each iteration is independent of each other.
  - Each Iteration has sufficient work to compensate for scheduling overhead.
- If insufficient work per loop iteration consider creating an inner loop.

Cost of loop parallelism

$$\text{Time} = \frac{\text{Number of Tasks} * (\text{ Task Scheduling Cost} + (\text{ IterationCost} * \text{IterationsPerTask })}{\text{Number of Cores}}$$

- Task Scheduling Cost = 50
- Iteration Cost = 1
- Total Iterations = 100,000
- Sequential Time would be 100,000

Number of Cores	Iterations Per Task	Number of Tasks	Time
2	100	1000	75000
2	50000	2	50050
4	100	1000	37500
4	25000	4	25050
256	100	1000	586
256	391	256	440

Functional programming ideal for parallelism

- Input => f() => f2() => f3() => output
- Immutable data

LINQ is a form of functional programming

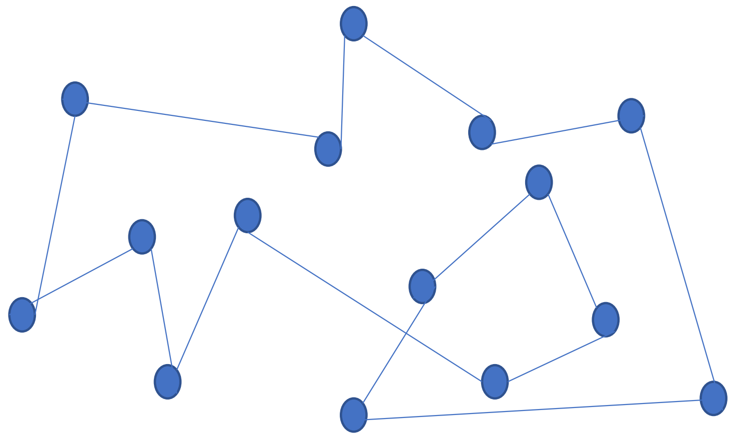
- LINQ is readable
- PLINQ as readable, but in parallel
  - Free lunch
- Works with IEnumerable<T>
  - Better with Array and List<T>
  - Best when not order sensative

PLINQ

- Embarrassingly parallel
- PLINQ not always faster
- LINQ is NOT a performance technology
  - Boosting it doesn’t mean best
- Brings parallelism to the masses

Travelling salesman problem

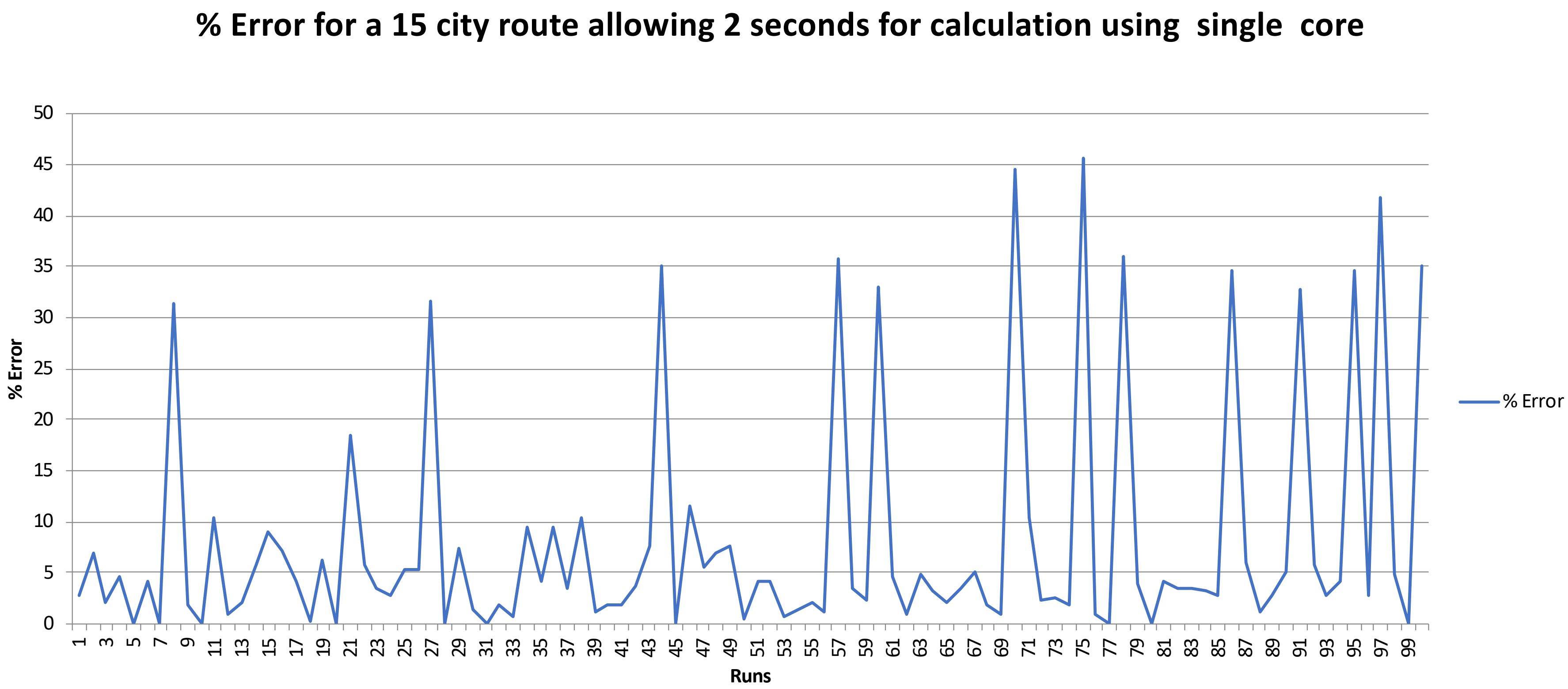
- Calculate the shortest path, between a series of towns, starting and finishing at the same location
- Possible combinations *n* Factorial
- 15 City produces 1307674368000 combinations
- Took 8 Cores appx. 18 days to calculate exact shortest route.
- Estimate 256 Cores would take appx. 0.5 days



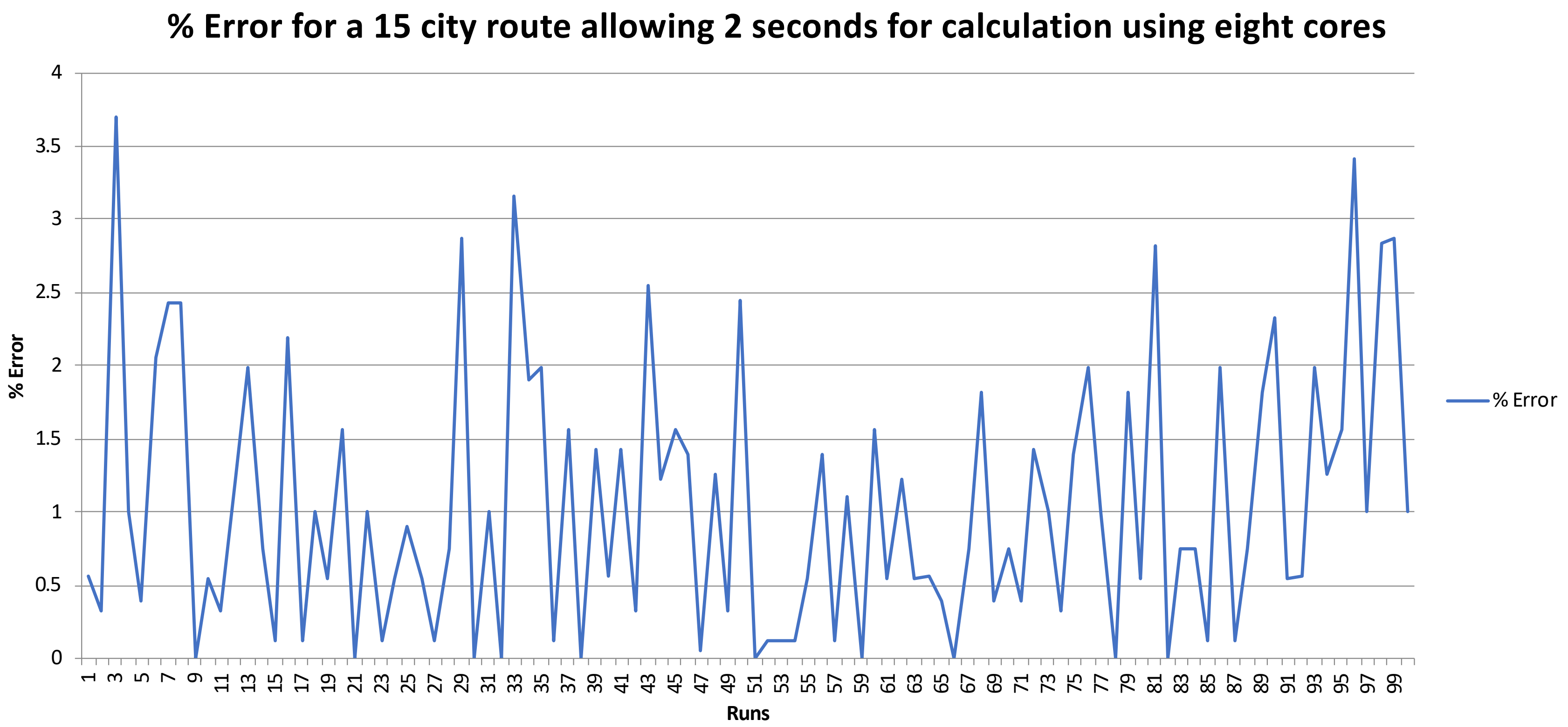
Monte Carlo

- Some algorithms even after heavy parallelisation are too slow.
- Often some answer in time T is often better than a perfect answer in T+
  - Real time speech processing
  - Route planning
- Machines can make educated guesses too..

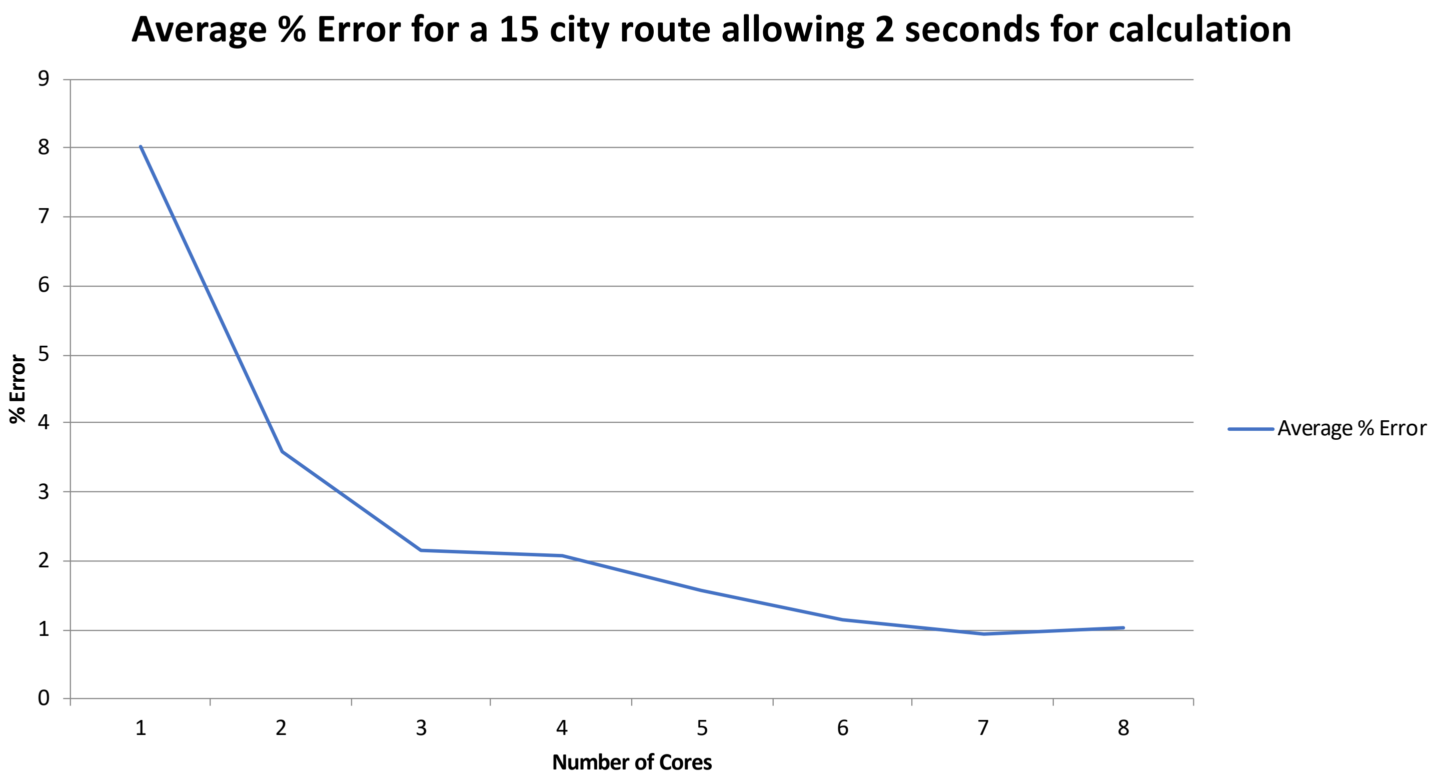
Monte Carlo - Single Core



Monte Carlo – 8 Cores



Average % Error



Monte Carlo

- Each core runs independently making guesses.
  - Scales, more cores greater the chance of a perfect guess.
  - As we approach 256 cores…looks very attractive.
- Strategies
  - Pure Random
  - Hybrid, Random + Logic

Summary

- Lots of ways to parallelise
  - Frameworks makes things easier, but not trivial
  - PLINQ enables safe parallel programming to the masses
- Always benchmark
- For best performance tune to hardware