CW5

- Timing:
  - Issued today : https://github.com/HPCE/hpce-2016-cw5
  - Due Fri 25th Oct (2 weeks)
- Pair work:
  - You'll work from two private repositories
  - Give each other push permission to your private repos
    - Keep the two repositories in sync
  - I'll infer the graph of pairs from access rights
- If you don't have a pair sorted out...
  - Hang around after the lecture

Suggestions

1. Read the code
   – What are the dependencies, what is the complexity?
2. Parallelism – Low-hanging fruit
   – Parallel for, recursive parallelism
3. Obvious inefficiencies
   – Excessive copying or re-initialisation (sloppy code)
   – Caching / eliminating computation (sloppy algorithms)
4. Parallelism – Larger-scale changes
   – Moving code to GPU
   – More aggressive re-structuring of code
5. Deep analysis
   – Algorithmic opportunities (change complexity)
   – Batching of computation
   – Maximise compute per data transfer (memory bandwidth)

Feasibility:

Learning to say no

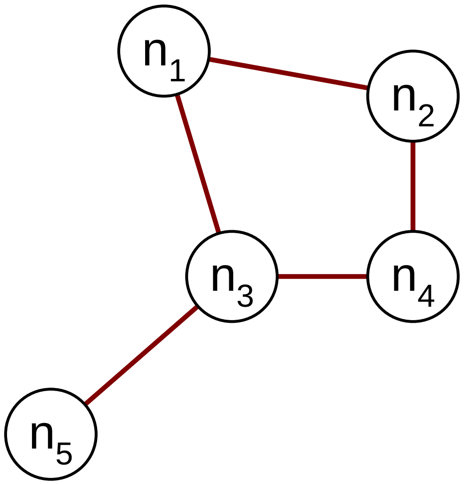
Making sure metrics are meaningful

- Some things are quantifiable, but not very useful
  - CPU performance: MHz is not the same as performance
  - Cameras: Mega-Pixels is not the same as quality
- Consistent and quantifiable metrics provide open competition
  - Suppliers of systems always want to use the “best” metrics
  - Metrics should be defined by users, not suppliers
- People will optimise for metrics (it’s what they are for!)
  - Poor metrics lead to poor design and optimisation
  - Part of the specification problem

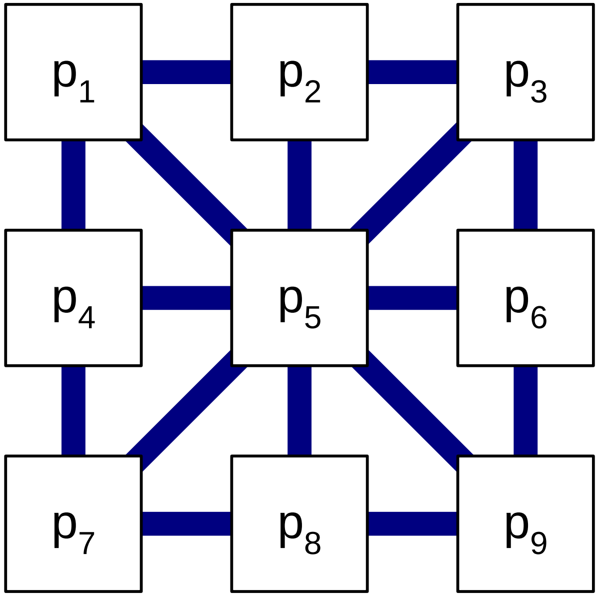
Feasibility studies

- People come up with demands
  - "*I want real-time spectral analysis of a 0hz-1GHz signal*"
  - "*We must process HD video within a latency of 1ms*"
  - "*This base-station must beam-form 32 channels*"
- Is it feasible to meet those demands?
  - Will it be easy?
  - Will it require optimisation?
  - Will it require a specialised platform?
  - Is it fundamentally impossible?
- You need some estimates before you start implementation
  - Execution time is the most basic check to be made

Circuit Placement



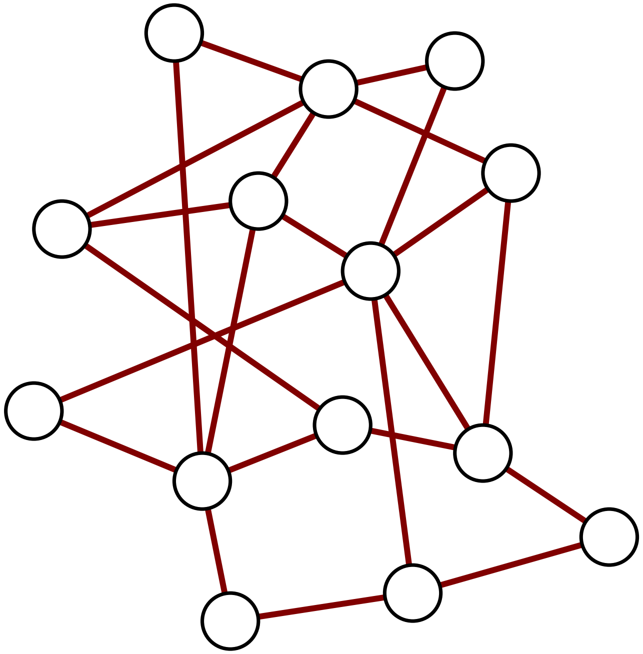
Logical components in circuit

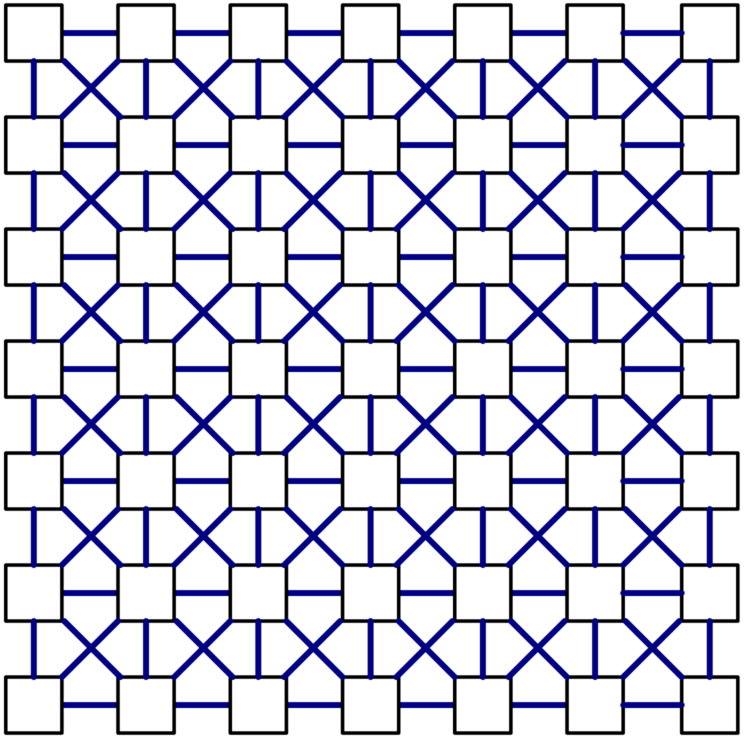


Physical resources in device

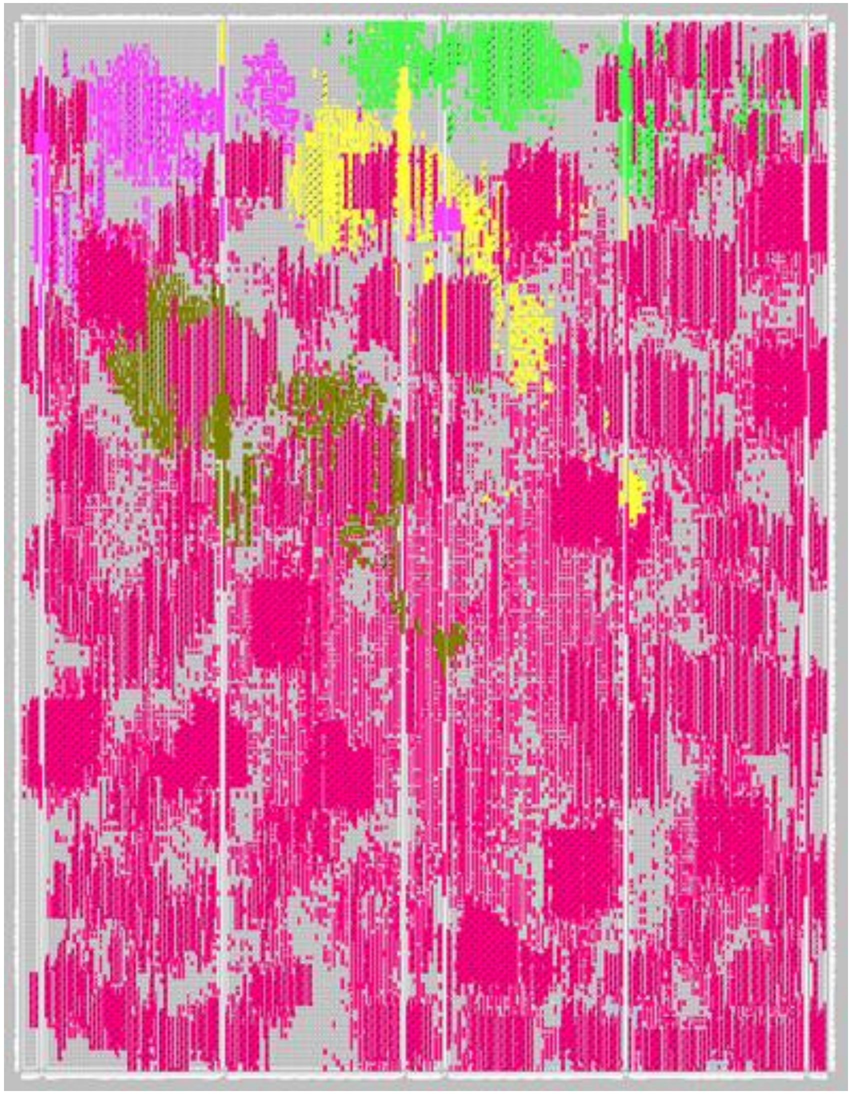
- Take the graph of circuit, and find a valid placement onto physical resources
- Make sure that all logical ***components*** have a unique physical location
- Make sure that all logical ***connections*** map to a physical channel

Circuit Placement



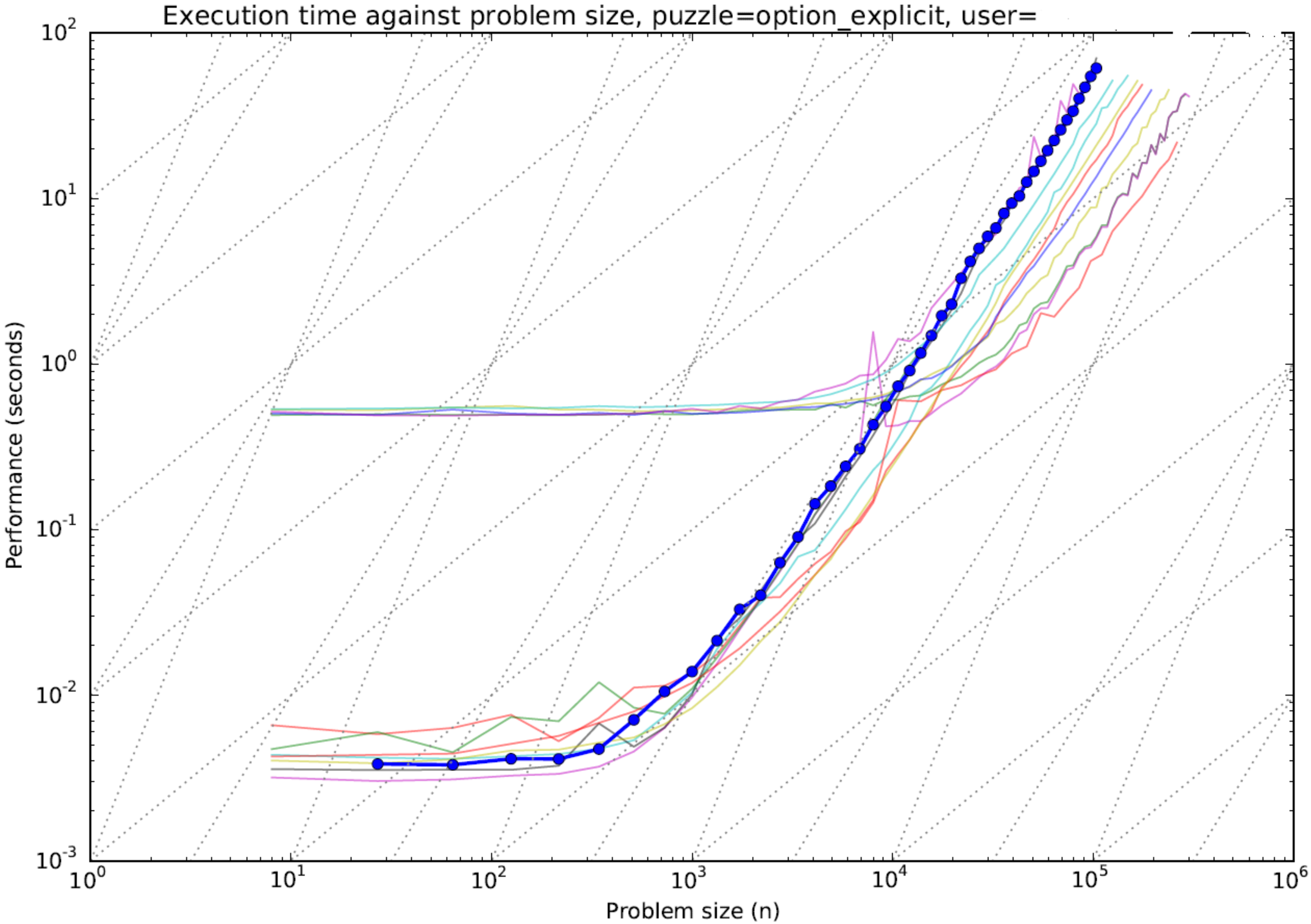


How difficult is a big problem?

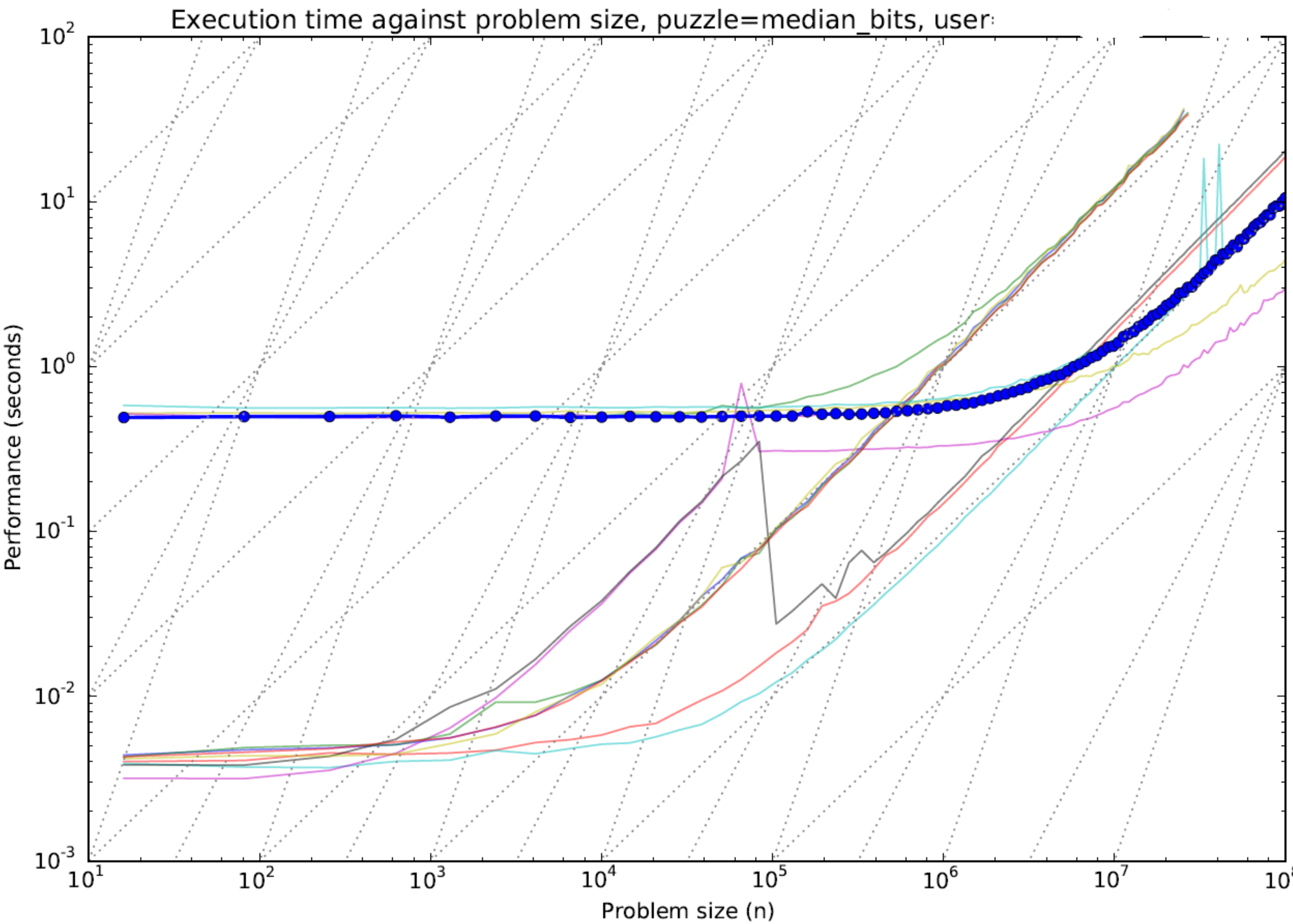


Trying to measure complexity

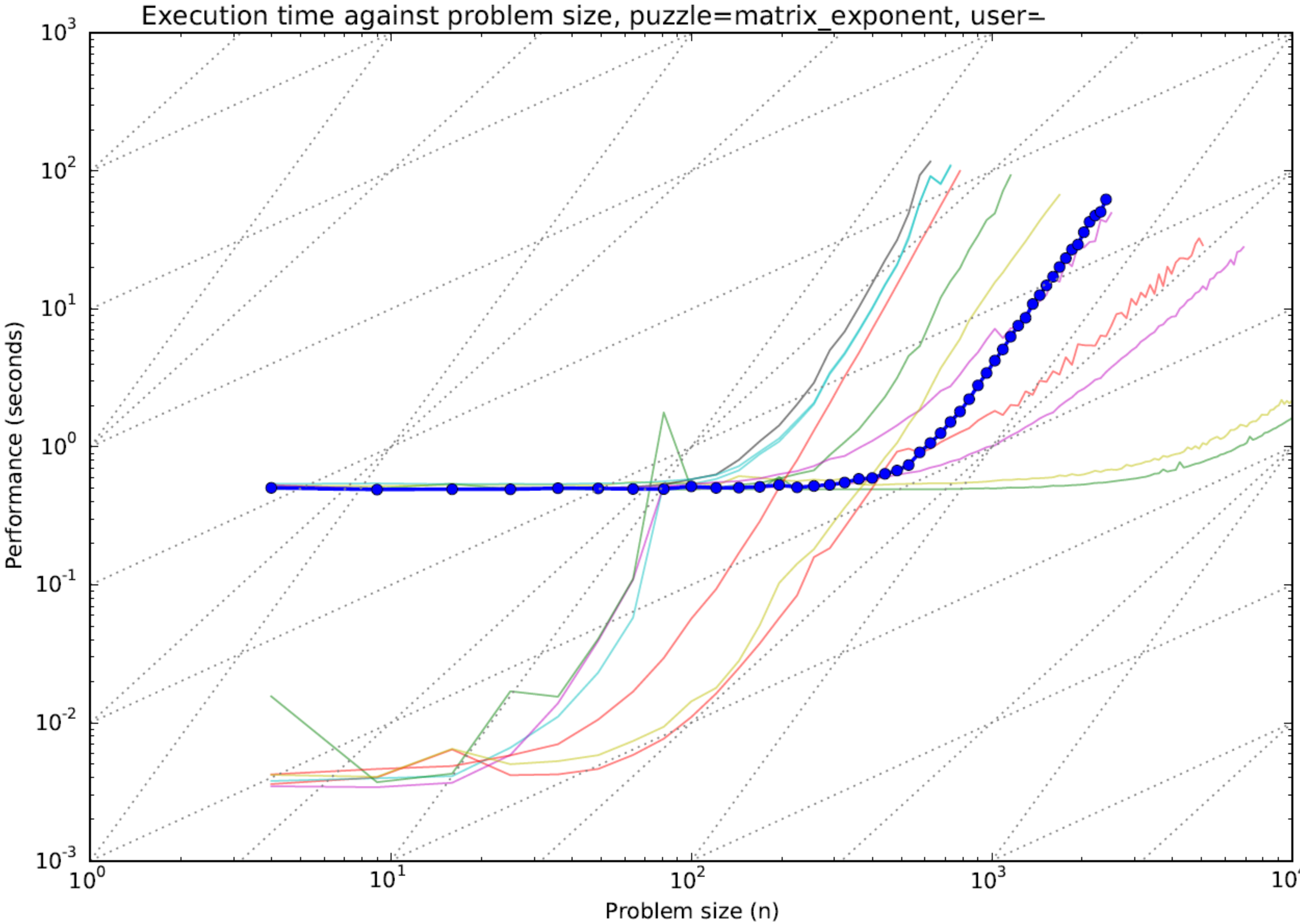
- Want to capture complexity of a task using equations
  - ***Time complexity***: how many “steps” does it require
  - ***Space complexity***: how much “storage” does it require
- We could derive exact equations for each
  - How many instructions does the task take in total?
  - How many bytes of memory are allocated during execution
- Means we have to worry about lots of details
  - Language: did you use C++, Fortran, VHDL?
  - Compiler: what optimisation flags were used?
  - Architecture: are integers 32-bit or 64-bit?
- Exact equations are sometimes possible, but often impractical



Execution time against problem size, puzzle=option_explicit, user=
Performance (seconds)
Problem size (n)
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}



Execution time against problem size, puzzle=median_bits, user
Performance (seconds)
Problem size (n)
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
10^{8}


Execution time against problem size, puzzle=matrix_exponent, user=
Performance (seconds)
Problem size (n)
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}

Complexity and Big-O (recap)

- Let's assume the existence of a function *g*(*n*)
  - *g*(*n*) specifies *exactly* how many steps are taken
    - *Note: this function doesn't need to be stated explicitly!*
  - *n* is the "size" of the problem; e.g. an input vector of length *n*

- Goal: find a simple function *f*(*n*), such that *g*(*n*) ϵ O(*f*(*n*))

$$g(n) \in \mathrm{O}(f(n)) \quad \text{iff} \quad \exists n_c > 0, m > 0 : \left[\forall n > n_c : \left[g(n) < m \times f(n)\right]\right]$$

"*There must exist a critical value n_c, and a positive constant m, such that for all $n>n_c$ the relation g(n) ≤ m×f(n) holds*"

"*For increasing n, eventually you'll reach a point where f(n) times a constant is always bigger than g(n)*"

Complexity and Big-O

- $O(f(n))$ is a set of functions with the same or lower complexity
  - $n \in O(n)$ $\quad\quad$ $n^2 \notin O(n)$
  - $n \in O(n^2)$ $\quad\quad$ $n^2 \in O(n^2)$
  - $O(n) \subset O(n^2) \subset O(n^3)$
- It is sort of correct to claim that everything is $O(\infty)$
  - But it's really not very useful...
- Try to find the smallest complexity class containing a function
  - $n^2 + 2 \in O(n^2)$
  - $100\ n^2 + 0.1\ n^3 - 100 \in O(n^3)$
  - $2^n + n^4 \in O(2^n)$
- Find the fastest growing component, and choose that

Reduction Rules

```
function F(n:integer) : integer
begin
  for i = 0..n/2
    acc = acc + init(i)
    for j = 64..n/3
      tmp = tmp + func(j,acc)
    next j
  next i
  return tmp
end
```

Executes n/2 times: O(n)

Executes (n/3-64)*n/2

$= n^2 / 6 - 32\,n \in O(n^2)$

$O(a \times g(n)) \equiv O(g(n))$, a is independent of n

$O(a + g(n)) \equiv O(g(n))$

$O((a \times n)^2) \equiv O(n^2)$

Common complexity classes

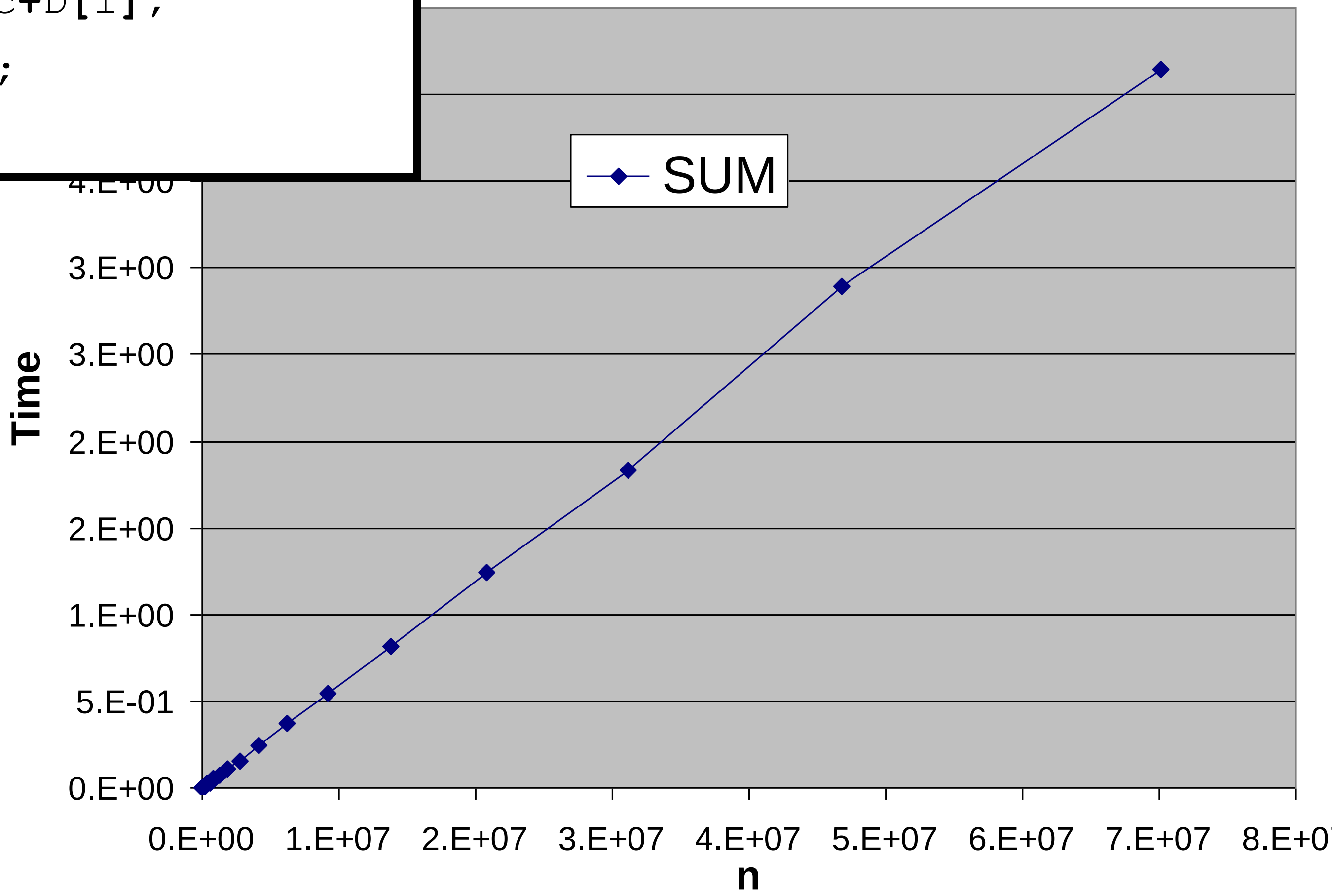
```
int SUM(int N)
{
    int acc=0;
    for(int i=0;i<N;i++)
        acc=acc+D[i];
    return acc;
}
```

```
function [R]=randMMM(N)
    A=rand(N,N);
    B=rand(N,N);
    R=A*B;
end
```

```
function [R]=randFFT(N)
    A=rand(N,1);
    B=rand(N,1);
    R=fft(A,B);
end
```

```
int Ack(int N)
{
    int A(int m, int n)
    {
        if(m==0) return n+1;
        if(n==0) return A(m-1,1);
        return A(m-1,A(m,n-1));
    }

    return A(N,N);
}
```

```
int SUM(int N)
{
    int acc=0;
    for(int i=0;i<N;i++)
        acc=acc+D[i];
    return acc;
}
```



Matrix-Matrix Multiply

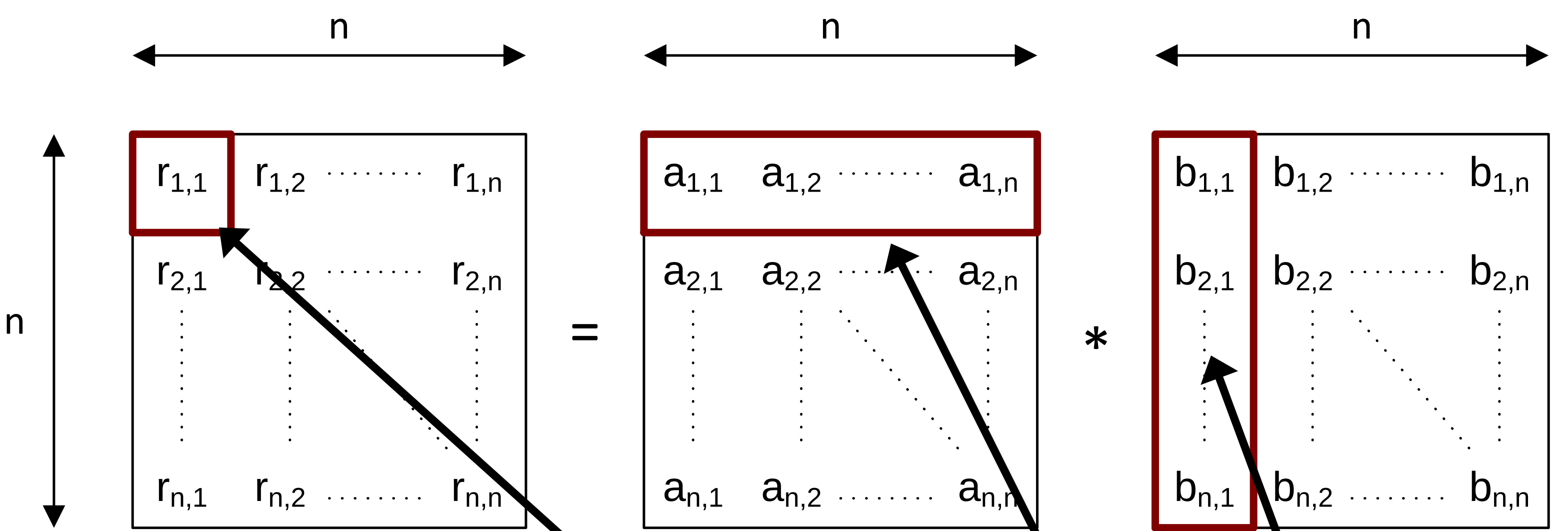
$$\begin{bmatrix} r_{1,1} & r_{1,2} & \cdots & r_{1,n} \\ r_{2,1} & r_{2,2} & \cdots & r_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n,1} & r_{n,2} & \cdots & r_{n,n} \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix} * \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,n} \end{bmatrix}$$

n n n n

```
function[R]=randMMM(N)
   A=rand(N,N);
   B=rand(N,N);
   R=A*B;
end
```

```
function [R]=Mul(A,B)
  for r=1:n
    for c=1:n
      R(r,c)=sum(A(r,:).*B(:,c));
    end
  end
end
```

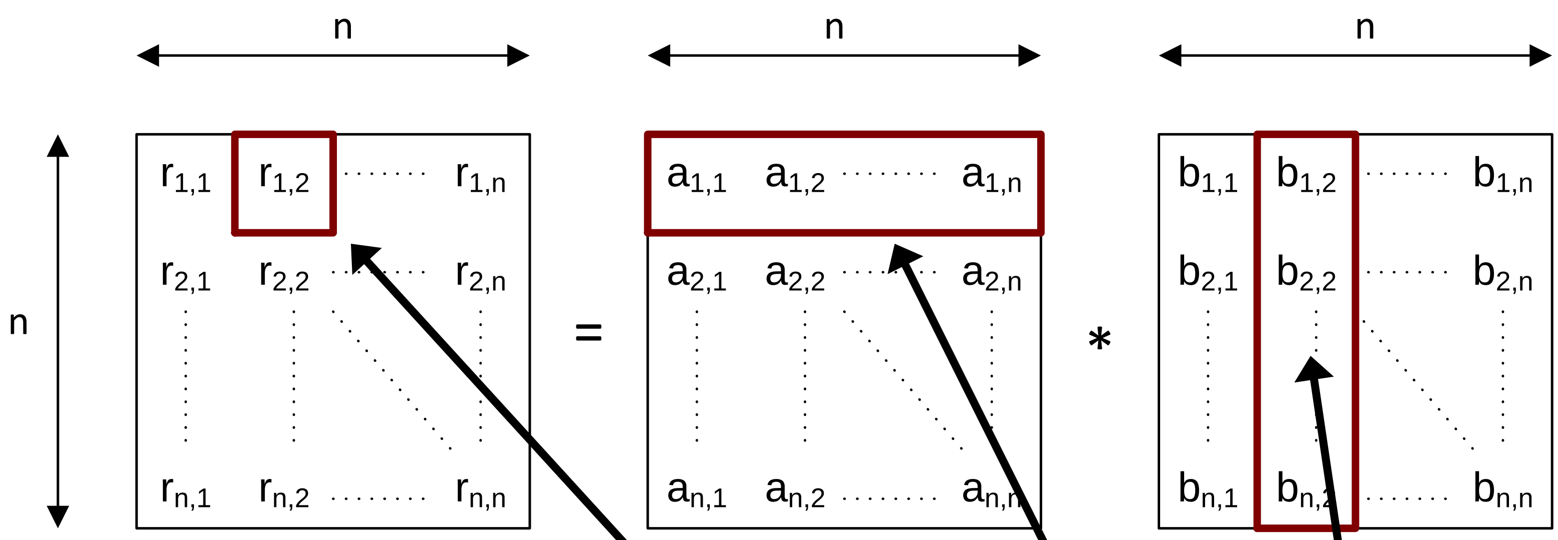
Matrix-Matrix Multiply




```
function[R]=randMMM(N)
    A=rand(N,N);
    B=rand(N,N);
    R=A*B;
end
```

```
function [R]=Mul(A,B)
  for r=1:n
    for c=1:n
      R(r,c)=sum(A(r,:).*B(:,c));
    end
  end
end
```

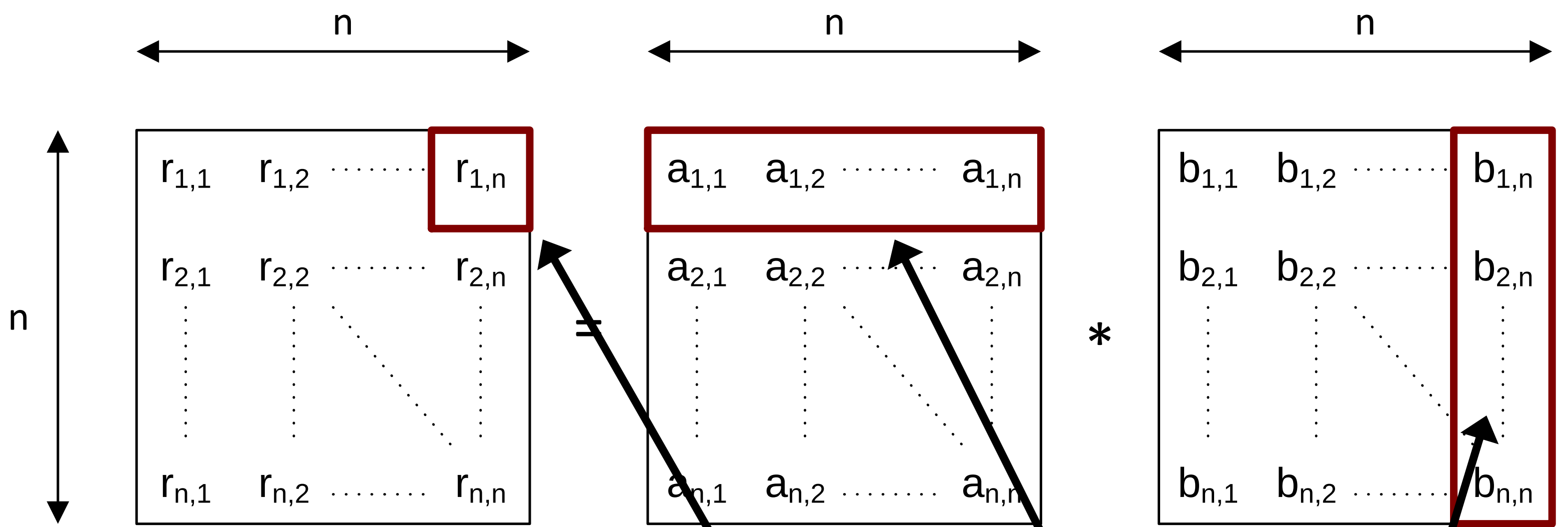
Matrix-Matrix Multiply




```
function[R]=randMMM(N)
    A=rand(N,N);
    B=rand(N,N);
    R=A*B;
end
```

```
function [R]=Mul(A,B)
  for r=1:n
    for c=1:n
      R(r,c)=sum(A(r,:).*B(:,c));
    end
  end
end
```

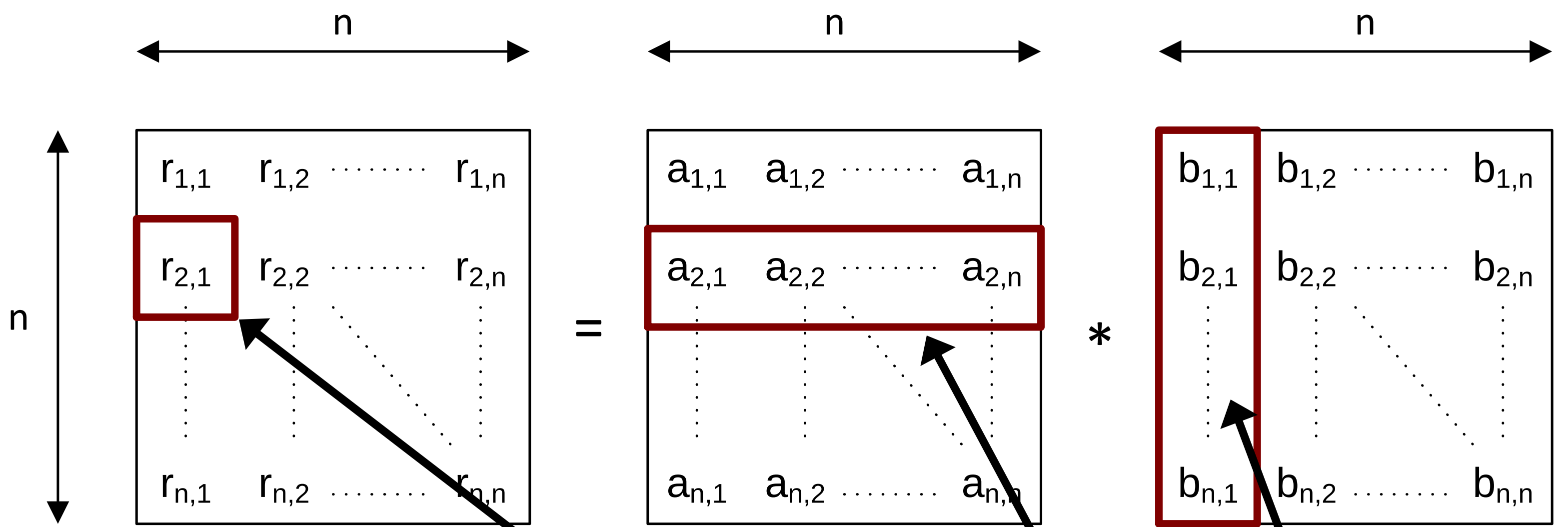
Matrix-Matrix Multiply




```
function[R]=randMMM(N)
   A=rand(N,N);
   B=rand(N,N);
   R=A*B;
end
```

```
function [R]=Mul(A,B)
  for r=1:n
    for c=1:n
      R(r,c)=sum(A(r,:).*B(:,c));
    end
  end
end
```

Matrix-Matrix Multiply

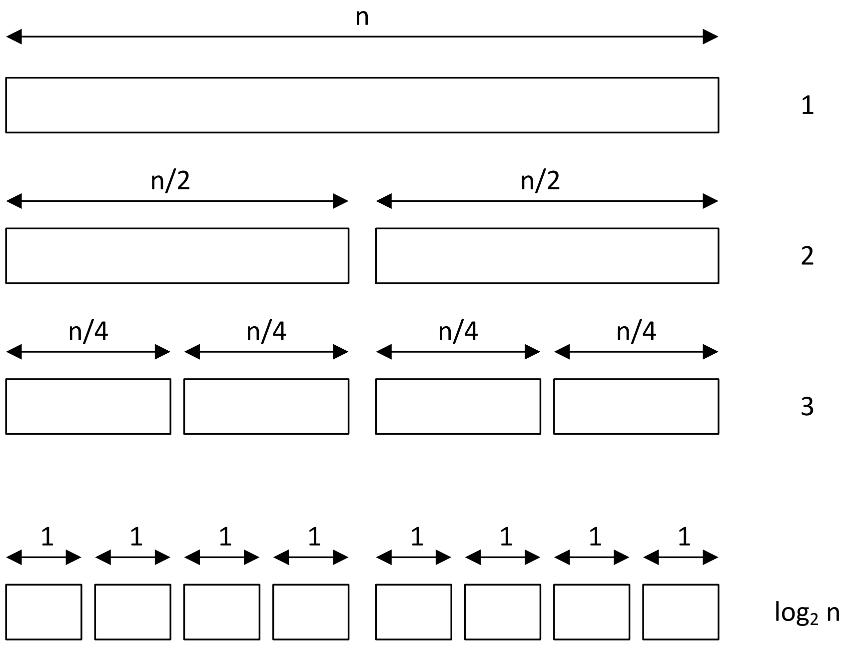



```
function[R]=randMMM(N)
    A=rand(N,N);
    B=rand(N,N);
    R=A*B;
end
```

```
function [R]=Mul(A,B)
  for r=1:n
    for c=1:n
      R(r,c)=sum(A(r,:).*B(:,c));
    end
  end
end
```

Quick-Sort

```
void Qsort(int N, int *data)
{
   partition(N, data); // O(n)
   if(N>1){
      Qsort(N/2, data);
      Qsort(N/2, data+N/2);
   }
}
```



How fast do functions grow?

$O(1)$

$O(n^4)$

$O(n)$

$O(1.01^n)$

$O(n!)$

$O(2^n)$

$O(n \log n)$

Polynomial Algorithms

- Polynomial-time algorithms are considered “easy”
  - That’s mathematically easy, not necesarily practical
- O(1) – ***Constant time***
  - *The best complexity class! Not much interesting in it though...*
  - Read an item from RAM
- O(n) – ***Linear time***
  - Vector addition
  - Search through an un-ordered list
- $O(n^2)$ – ***Quadratic time***
  - Matrix-vector multiply
- $O(n^3)$ – ***Cubic time***
  - Dense matrix-matrix multiply
  - Gaussian elimination

In theory it’s lower, but in practise it often isn’t – see Strassen’s algorithm

Log and Log-Linear Algorithms

- Algorithms which recursively sub-divide some space
  - These are *actually* easy, not just mathematically
- O(log n) – ***Logarithmic time***
  - Find an element in a ***sorted*** list
  - Root finding through bi-section
  - See if an element belongs to a set / add an element to a set
- O(n log n) – ***Log-Linear time*** *(also called linearithmic)*
  - Sorting a vector of items
  - Fast-Fourier-Transform (FFT)
- Both are huge improvements over closest polynomial
  - O(log n) preferred to O(n)
  - O(n log n) is *massively* better than O(n^2)

Exponential Algorithms

- Algorithms which explore some multi-dimensional space
- Class of algorithms with complexity $O(a^n)$ for $a>1$
  - Brute-force search of all length-n binary patterns : $O(2^n)$
  - Brute-force search of all length-n decimal strings : $O(10^n)$

- Exponential time algorithms are generally very bad
  - Scale extremely poorly with n

- Occasionally useful as long as you are careful
  - $O(2^n)$ algorithm can be useful for $n<32$
  - $O(a^n)$ algorithm with *a* close to 1 is sometimes feasible

Combinatorial Algorithms

- Looking at permutations and combinations of things
- Lots of specific sub-classes, but generally O(n!)

- Optimal mapping: bind abstract resources to physical ones

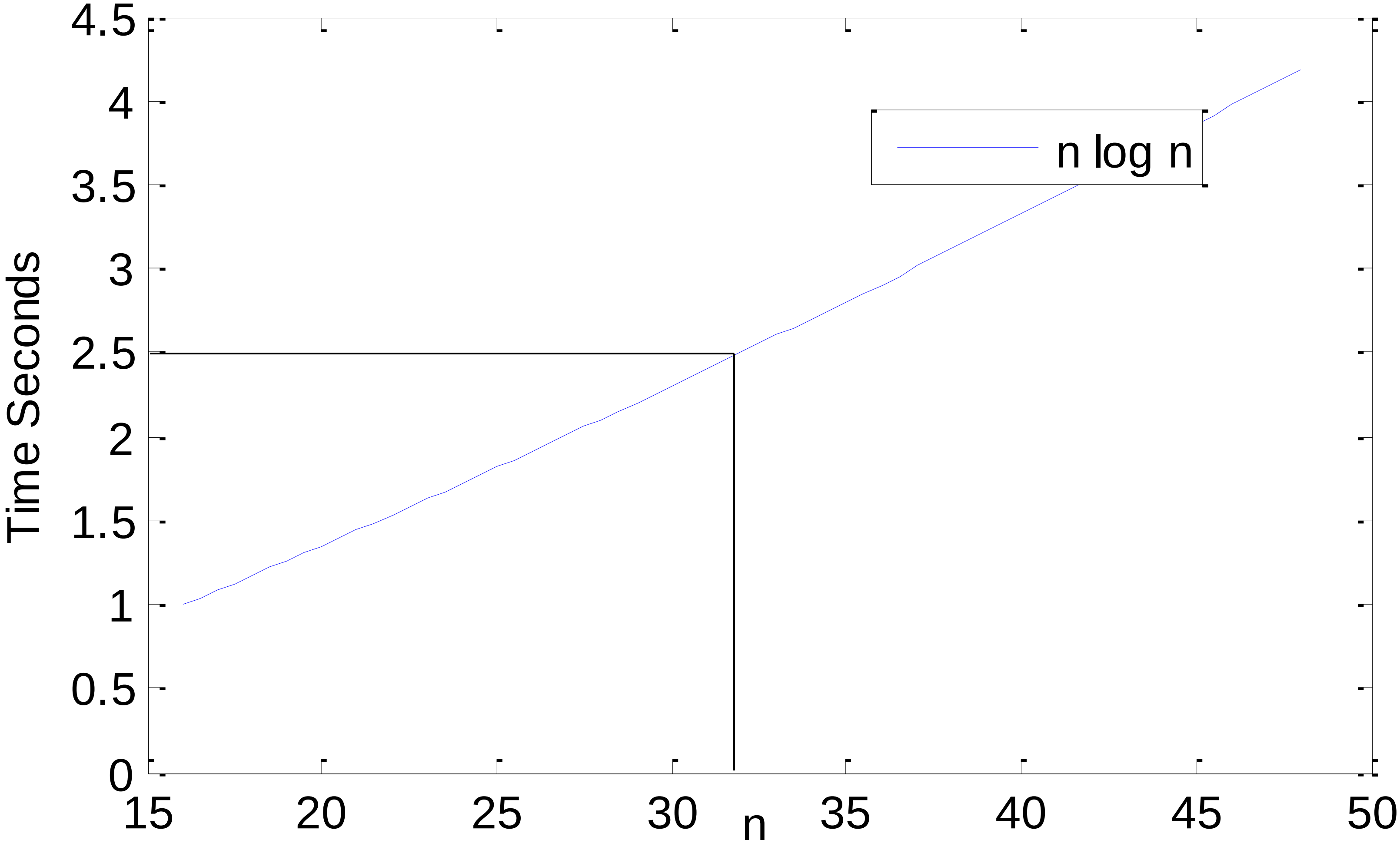
- Find the best sub-set from a larger set of resources
  - Many interesting engineering problems are combinatorial

Avoid anything more than polynomial

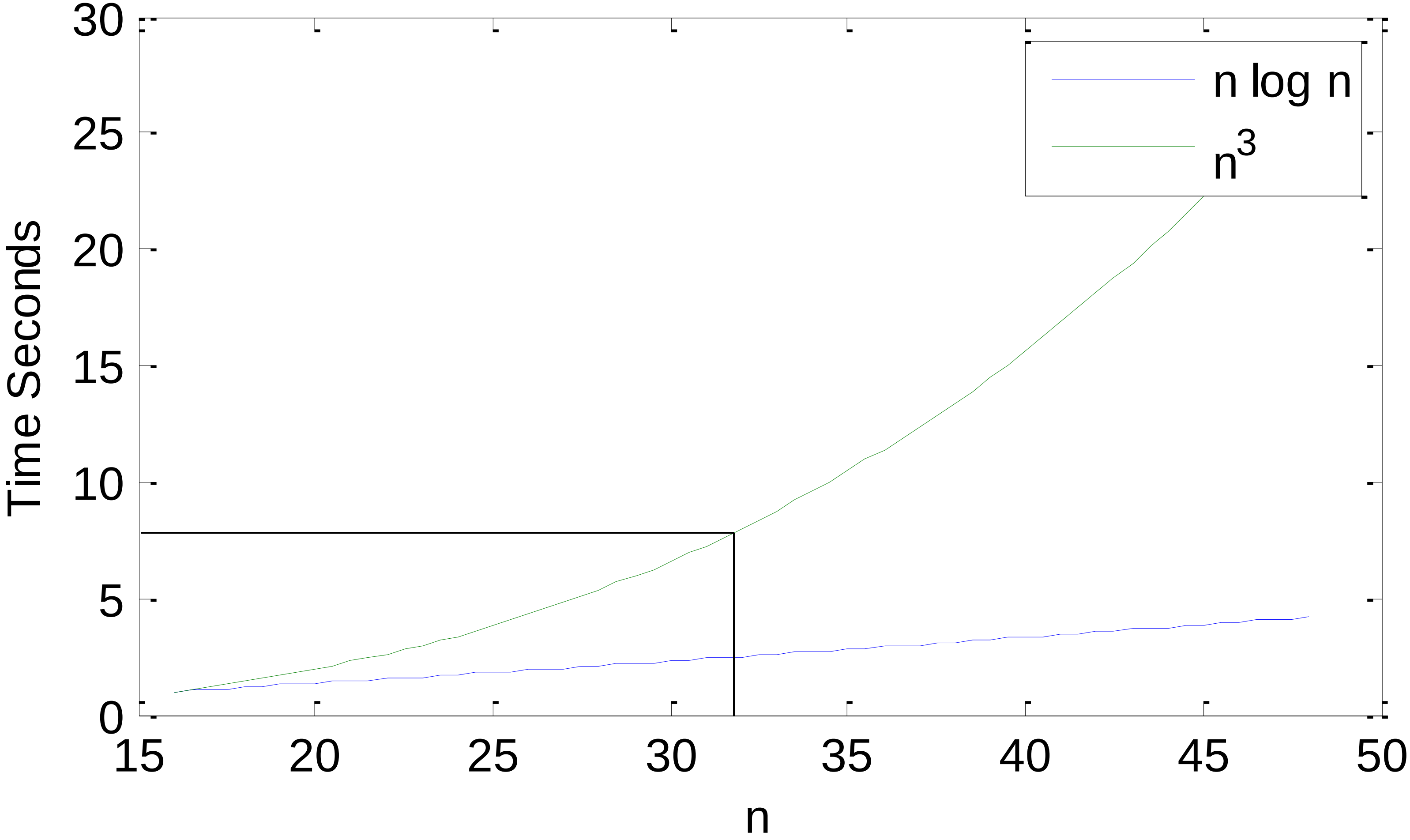
http://www.win.tue.nl/~gwoegi/P-versus-NP/homer2.jpg, Copyright Fox

Thought Experiment

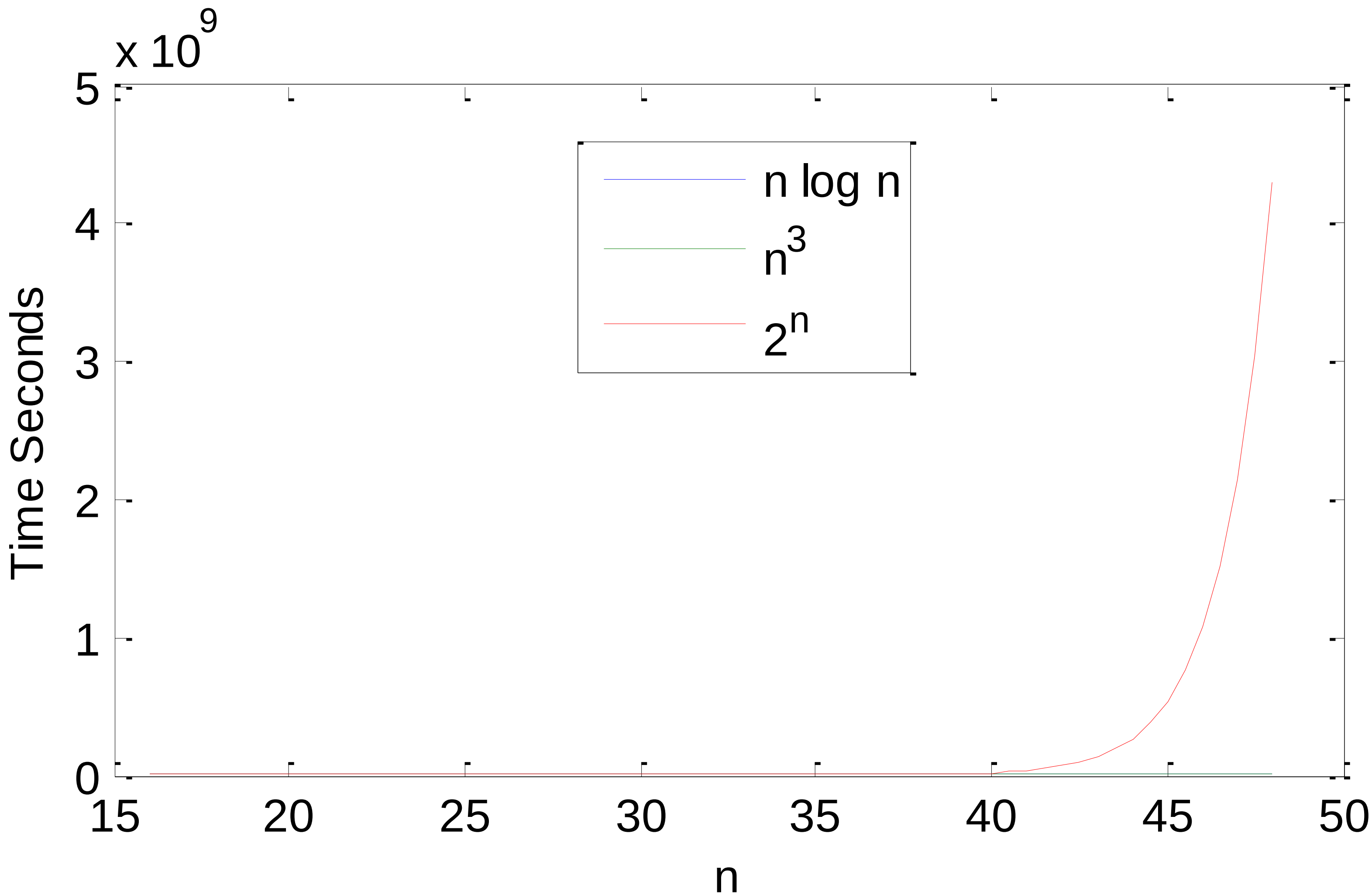
- We have four applications
  - All currently run in one second
  - All currently handle problems of “size” 16
- Each application has different complexity
  - Log-linear: $O(n \log n)$
  - Cubic: $O(n^3)$
  - Exponential: $O(2^n)$
  - Combinatorial: $O(n!)$
- The customer wants to handle problems of twice the size
  - How much do we need to accelerate the existing applications?



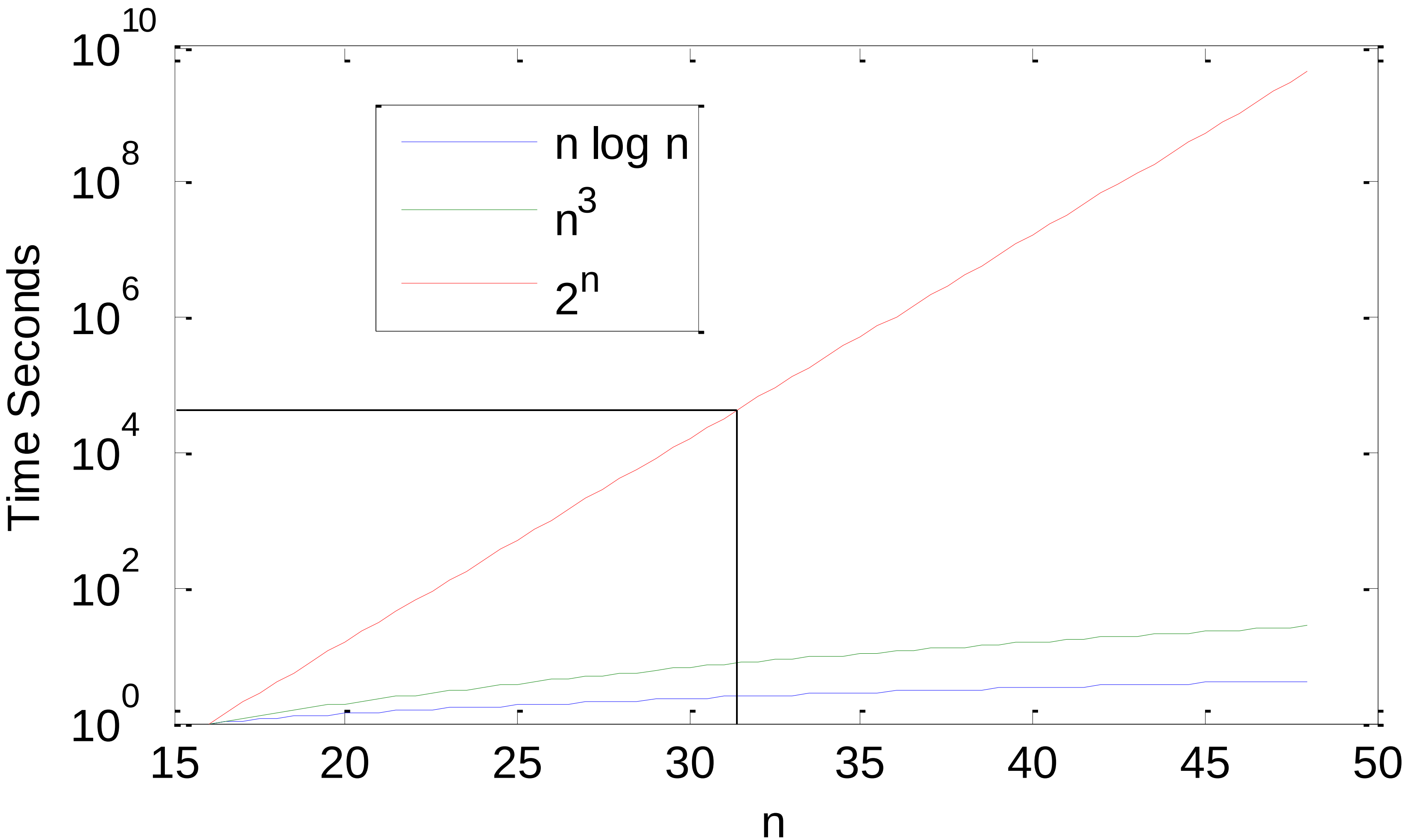
Speedup by about 2.5 times – probably use multi-core



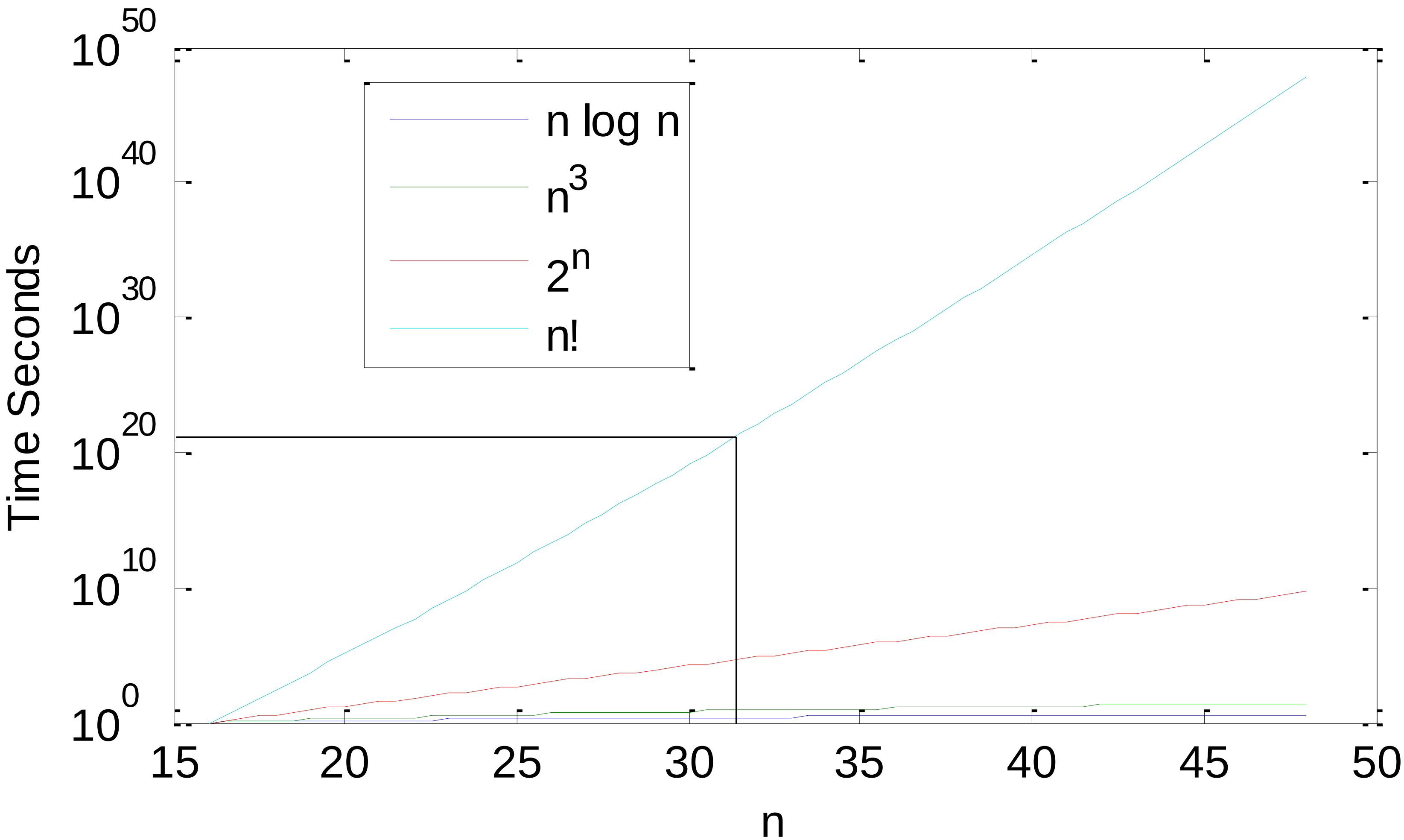
Speedup by about 8 times – maybe use a GPU?



Umm.....



Speedup by 65000 times – err, cluster of FPGAs? Cloud?



Speedup by 10^{22} times – turn every atom in 1kg of iron into a Pentium?

The limits of computation

- New systems are not magic
  - Multi-core CPUs: ~16x speedup
  - GPUs: ~500x speedup
  - FPGAs: ~1000x speedup
  - Cloud: ~10,000x speedup
  - O(n!), from n=16 to n=32 ~10^{22} required

- Lots of problems are completely intractable
  - Travelling Salesman: find shortest path to visit n cities
  - Bin packing: pack objects into the minimum number of bins
  - Boolean satisfiability: find values to make equation true

How to deal with intractable problems?

- Circuit place-and-route has ridiculously high complexity
  - But we regularly create designs with millions of logic gates...
- Must make decision about quality versus runtime
  - Wait 1 hour : design runs at 250MHz
  - Wait 10 hours : design runs at 310MHz
  - Wait ? hours : design runs at 317 MHZ
- Some algorithms are progressive and approximate
  - Quality of solution improves as more compute time applied
  - Monte-Carlo, Genetic Algorithms, Simulated Annealing, ...
  - No guarantee of optimality – ***but at least you get an answer***

Why parallelism fails: Amdahl's Law

- Split a given compute task **X** into two portions
  - **A** : The parts that cannot be easily optimised or accelerated
  - **B** : The parts that can be sped up significantly
- Assume we achieve a speed-up of $S_\mathbf{B}$ times to part **B**
  - What is the speed-up $S_\mathbf{X}$ for the entire task?

$$T_\mathbf{X} = T_\mathbf{A} + T_\mathbf{B} \qquad \text{Original execution time}$$

$$T_\mathbf{X'} = T_\mathbf{A} + T_\mathbf{B} / S_\mathbf{B} \qquad \text{New execution time}$$

$$S_\mathbf{X} = T_\mathbf{X} / T_\mathbf{X'} \qquad \text{Achieved speedup}$$

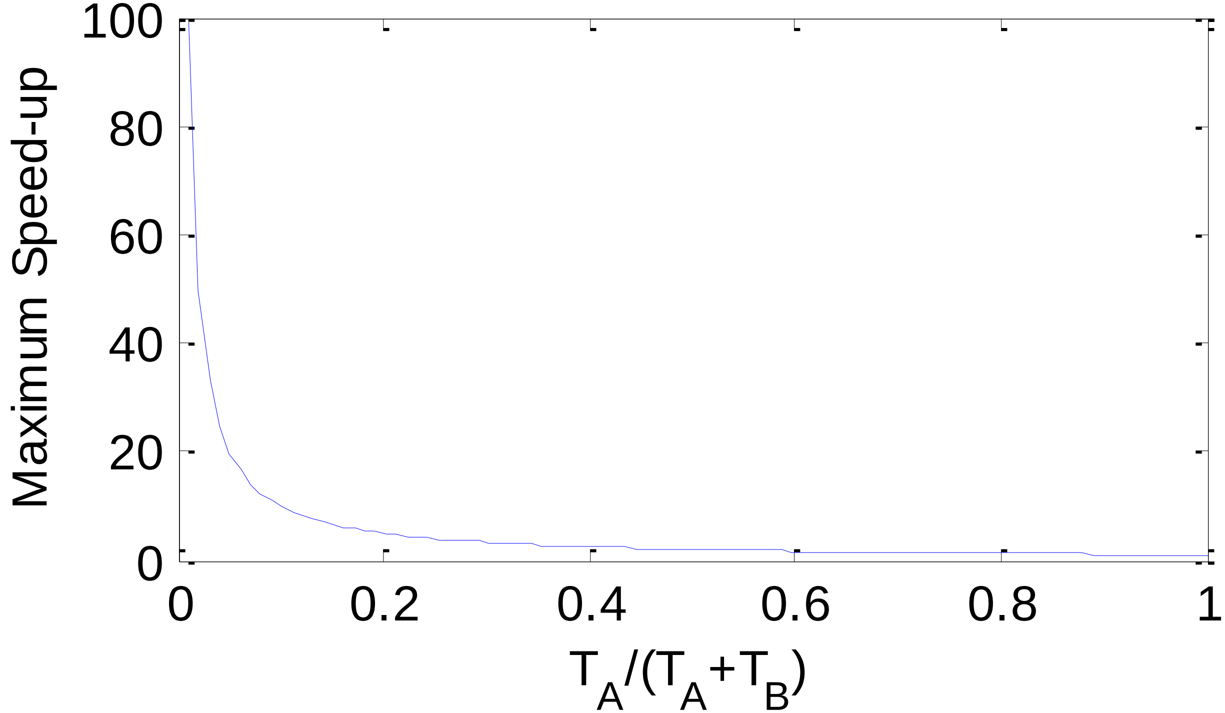
$$= \frac{T_\mathbf{A} + T_\mathbf{B}}{T_\mathbf{A} + {}^{T_\mathbf{B}}\!/_{S_\mathbf{B}}}$$

$$= \frac{T_\mathbf{A} + T_\mathbf{B}}{T_\mathbf{A}} \quad \text{as} \quad \xrightarrow{S_\mathbf{B}} \infty$$

- Maximum speed-up is limited by the ***serial fraction*** : $T_A/(T_A+T_B)$

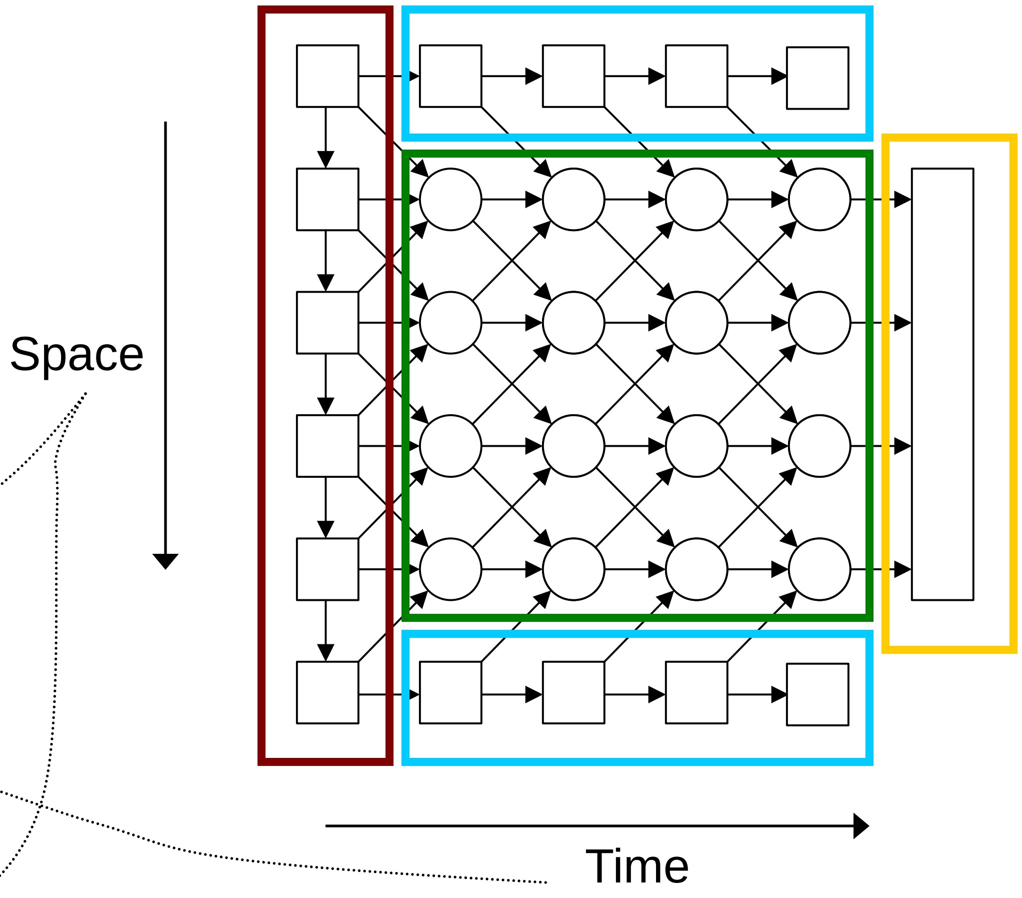
- Need a *tiny* serial fraction to achieve big speed-ups

- Are 1000x speed-ups realistic then?

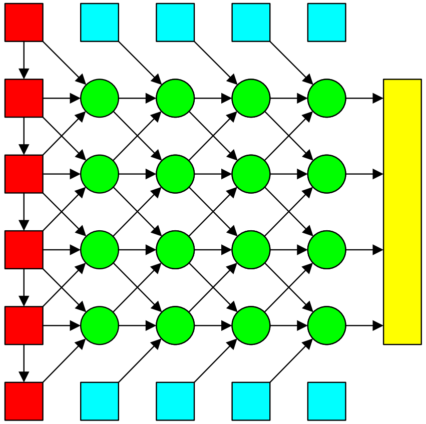


Practical example

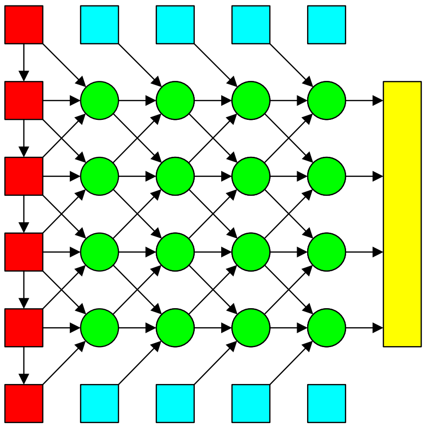
- Finite difference applications (fluid-mechanics, physics)
  - Discretise continuous space into cells
  - Discretise continuous time into distinct time-steps
- Goal of acceleration is to support finer resolution solutions
  - Usually increase resolution of space and time axis together
  - Let’s take ***n*** as the resolution along each axis
- Tasks within finite-difference
  - Initialisation: initialise the n items in the first column
  - Processing: advance each column through n steps in time
  - Collection: retrieve answers from final column



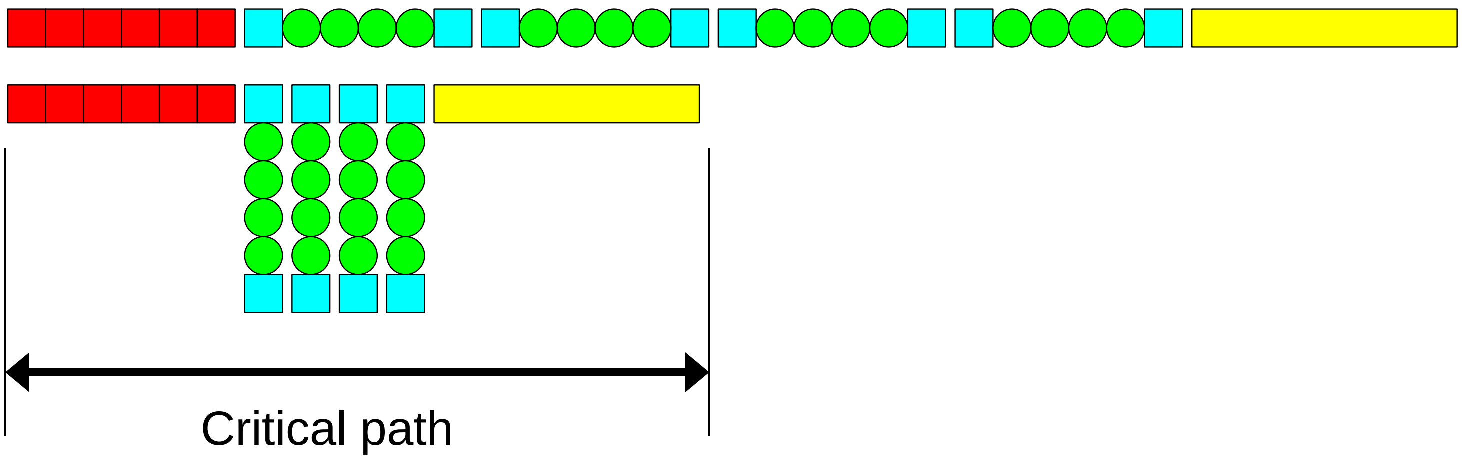

```
g[1,1]=F1(0)
for s=2..n
  g[s,1]=F1(g[s-1,1])
end

for t=1..n-1
  g[1,t+1]=g[1,t]
  for s=2..n-1
    g[s,t+1]=F2(g[s-1,t],g[s,t],g[s+1,t])
  end
  g[n,t+1]=g[n,t]
end

return F3(g[1..n,t])
```



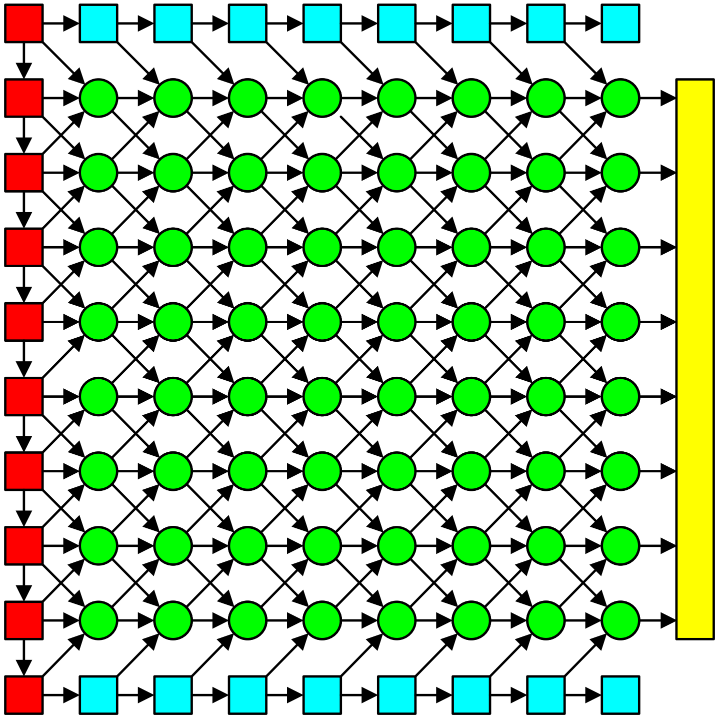
- Total work: $(n+1)(n+2) + C \in O(n^2)$



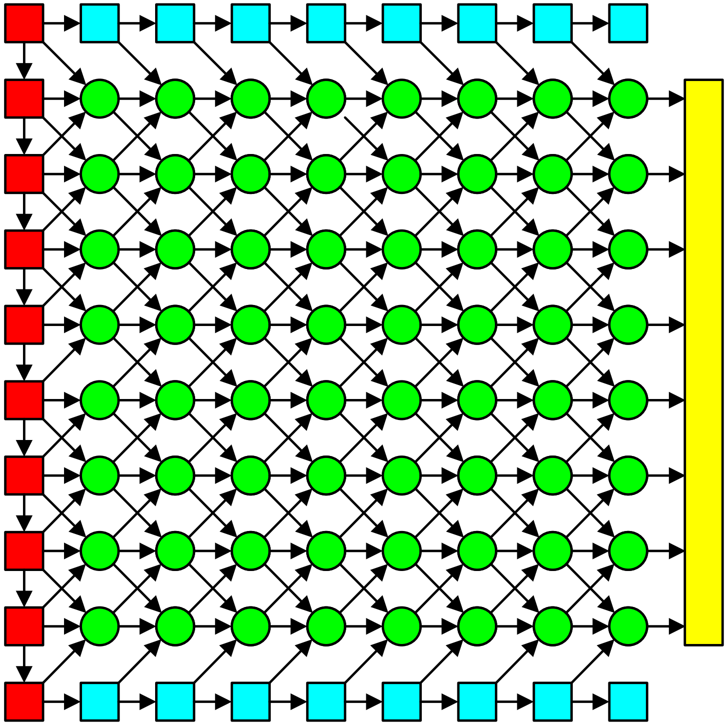
- Total work: $(n+1)(n+2) + C \in O(n^2)$



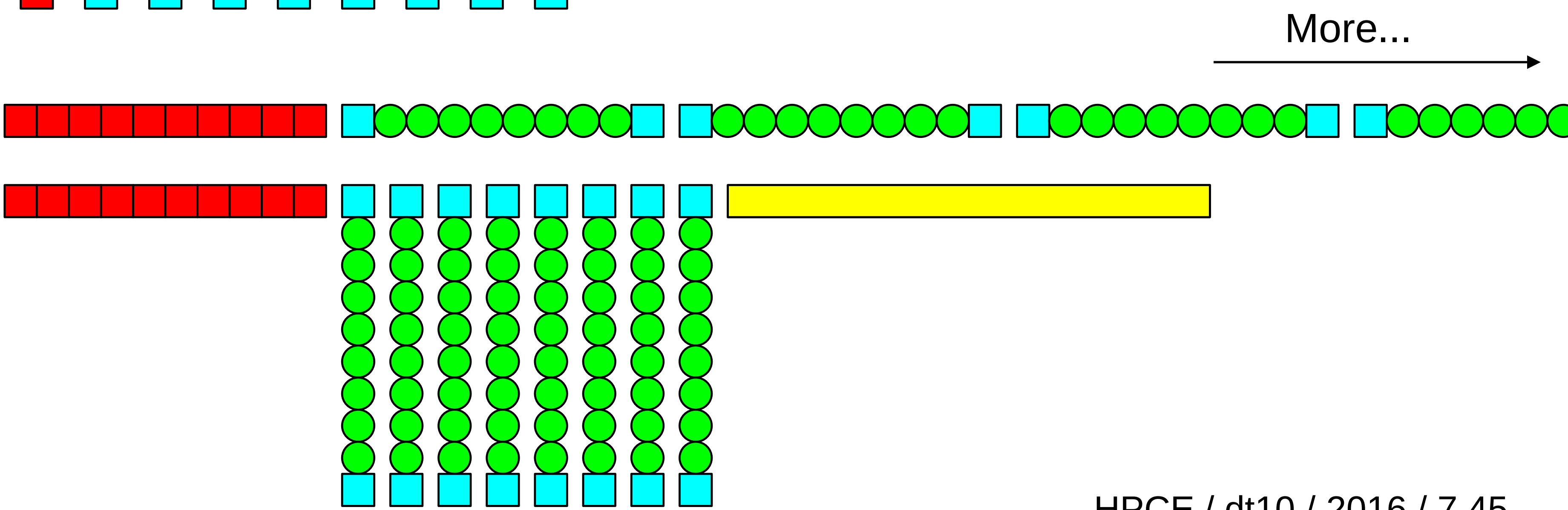
- ***Critical path***: longest path through dependency graph
  - *Assume infinite processors, and zero communication overhead*

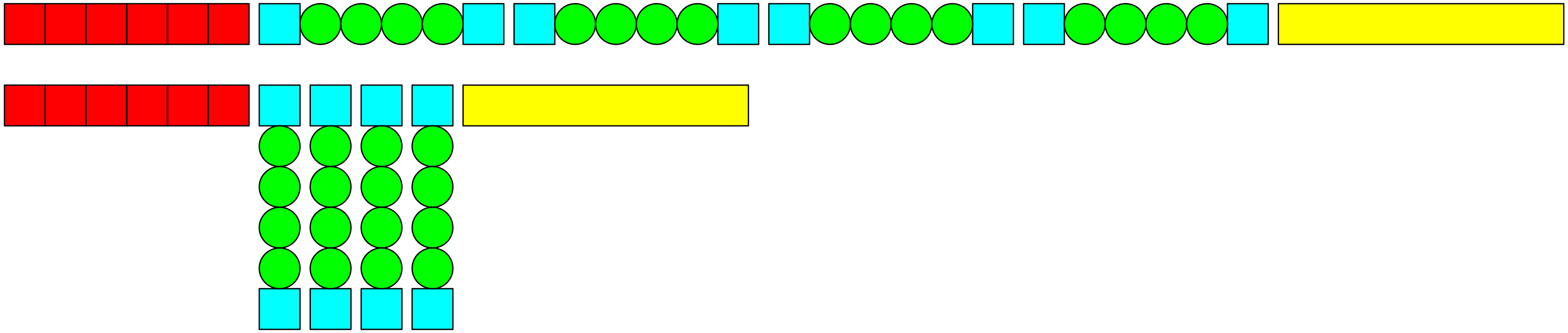


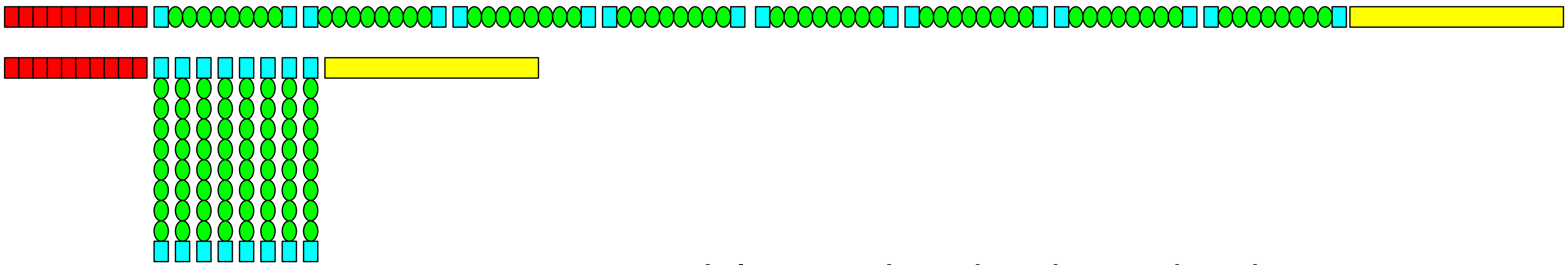
- Double the resolution of the grid
  - Increase resolution in both time and space
  - Model things of smaller size
  - Model things that happen faster
- Solution is better, but more compute intensive
  - Need High-Performance Computing!



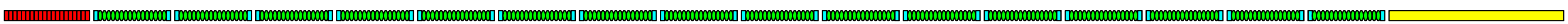
- Double the resolution of the grid
  - Increase resolution in both time and space
  - Model things of smaller size
  - Model things that happen faster
- Solution is better, but more compute intensive
  - Need High-Performance Computing!

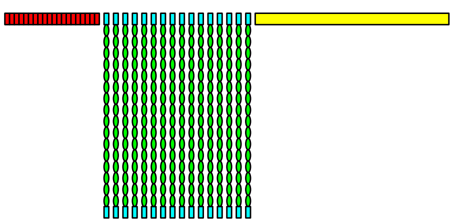






Serial execution: $(n+2) + n\times(n+2) + C$





Parallel execution: $(n+2) + n + C$

Speedup: $[\, n^2 + 3n + 2 + C] \,/\, [\, 2n + 2 + C]$

Speedup is $O(n)$ - increases linearly with problem size

Why parallelism works: Gustafson's Law

- Split a task X into two portions **A** and **B**
  - **A** cannot be accelerated, while **B** can be parallelised
  - But now the execution time of **A** and **B** depends on problem size
  - $T_A(n)$: time to perform part **A** for problem of size n

$$T_{\mathbf{X}}(n) = T_{\mathbf{A}}(n) + T_{\mathbf{B}}(\mathrm{n}) \qquad \text{Original execution time}$$

$$T_{\mathbf{X}'}(n) = T_{\mathbf{A}}(n) + T_{\mathbf{B}}(n) / S_{\mathbf{B}} \qquad \text{New execution time}$$

$$\begin{aligned} S_{\mathbf{X}}(n) &= T_{\mathbf{X}}(n) / T_{\mathbf{X}'}(n) \qquad \text{Achieved speedup} \\ &= \frac{T_{\mathbf{A}}(n) + T_{\mathbf{B}}(n)}{T_{\mathbf{A}}(n) + T_{\mathbf{B}}(n) / S_{\mathbf{B}}} \\ &= S_{\mathbf{B}} \quad \text{as} \xrightarrow{n} \infty \text{ if } \; O(T_{\mathbf{A}}(n)) \prec O(T_{\mathbf{B}}(n)) \end{aligned}$$