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

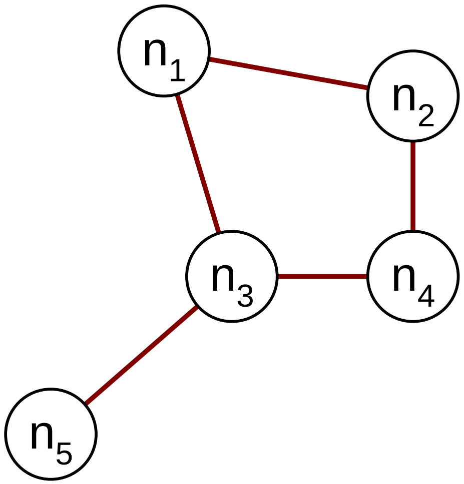
Feasibility:

Learning to say no

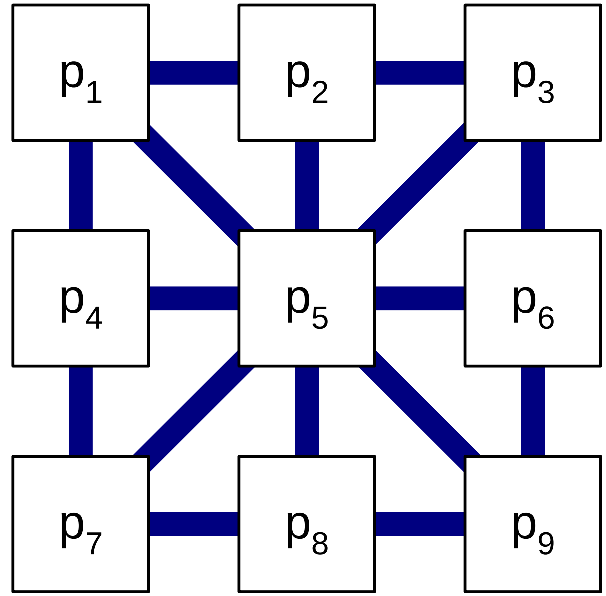
Feasibility studies

- 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



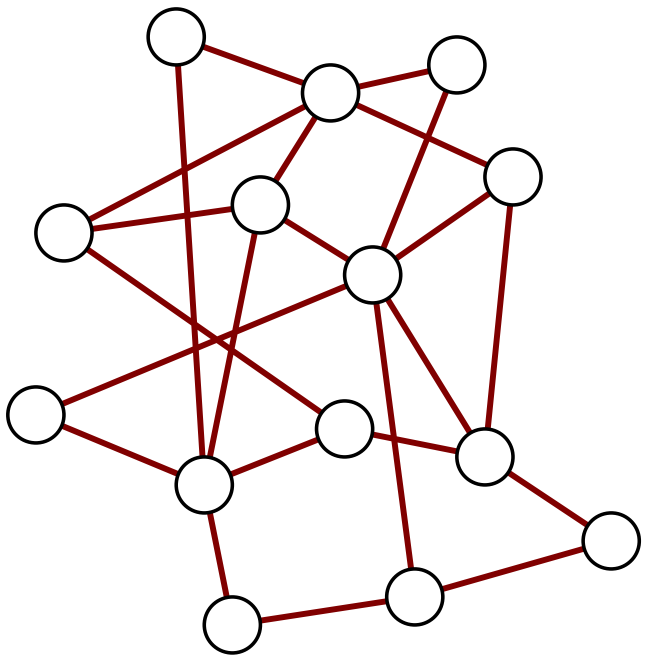
Logical components in circuit

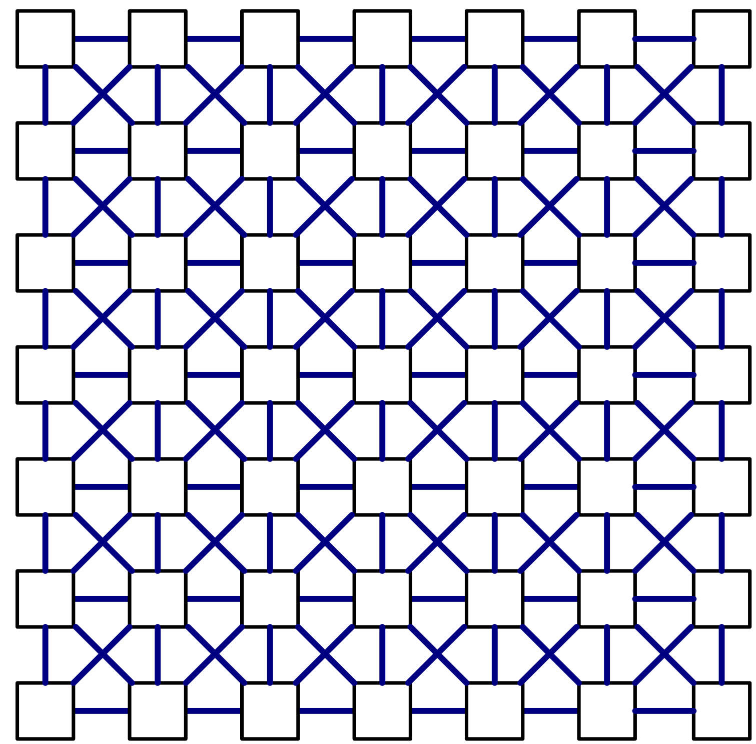


Physical resources in device

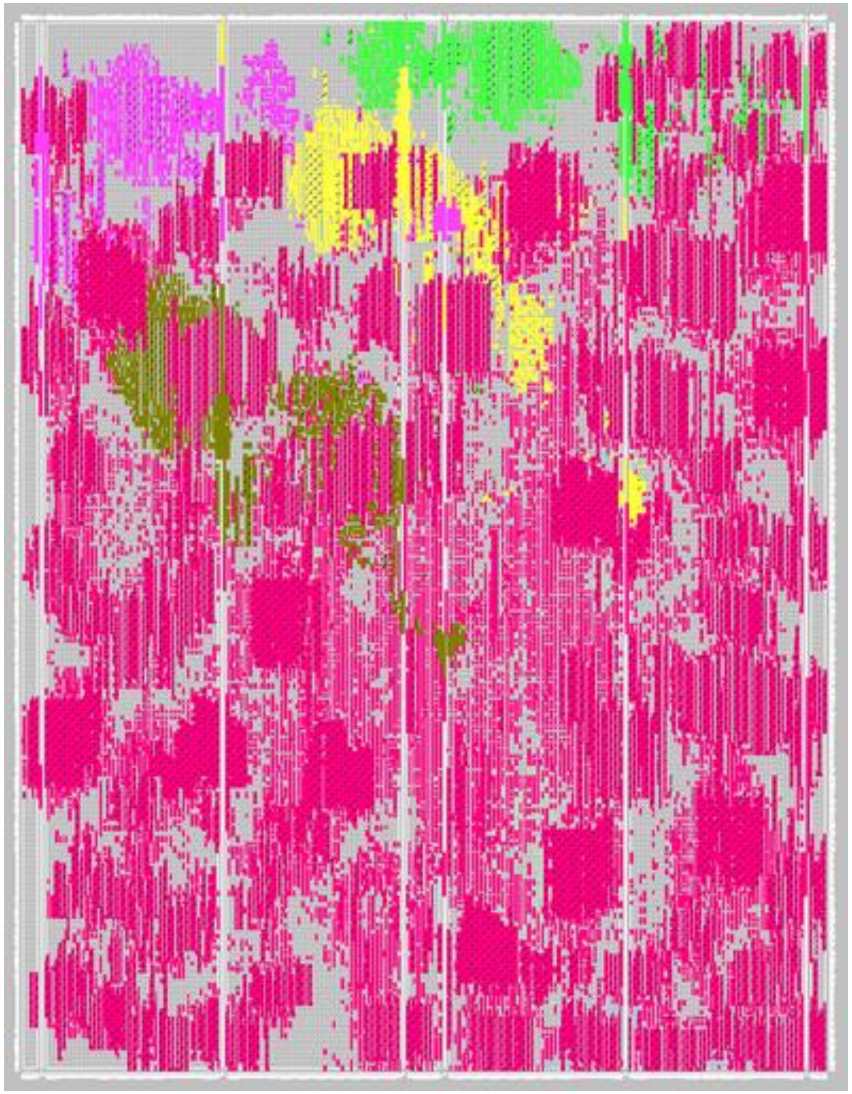
- Take the graph of circuit, and find a valid placement onto physical resources
- Make sure that all logical ***components*** have a unique physical location
- Make sure that all logical ***connections*** map to a physical channel

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

Recap: Complexity and Big-O

- 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
  - $n \in O(n)$ $\quad$ $n^2 \notin O(n)$
  - $n \in O(n^2)$ $\quad$ $n^2 \in O(n^2)$
  - $O(n) \subset O(n^2) \subset O(n^3)$

- 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

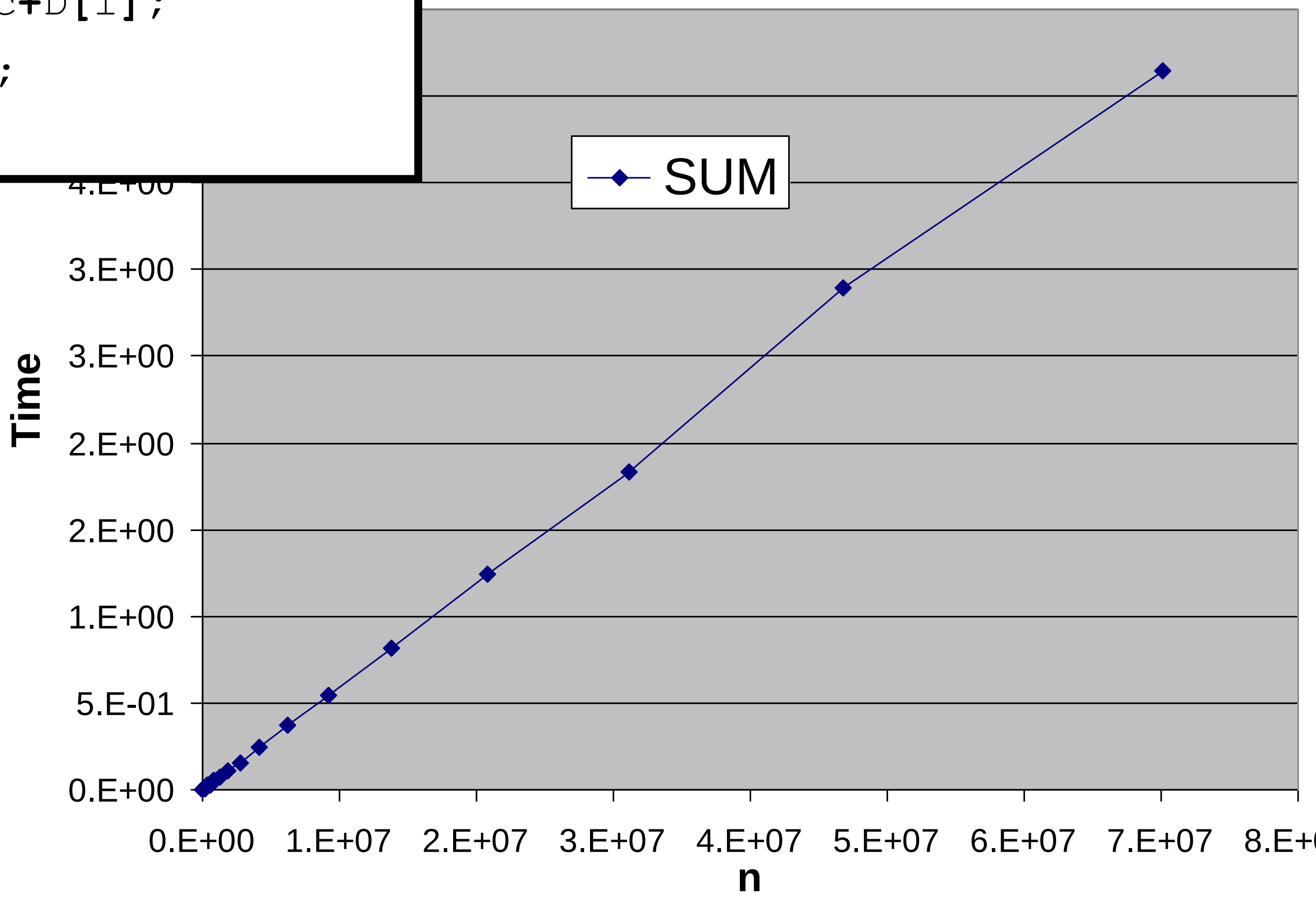
```
int SUM(int N)
{
    int acc=0;
    for(int i=0;i<N;i++)
        acc=acc+D[i];
    return acc;
}
```

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

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

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



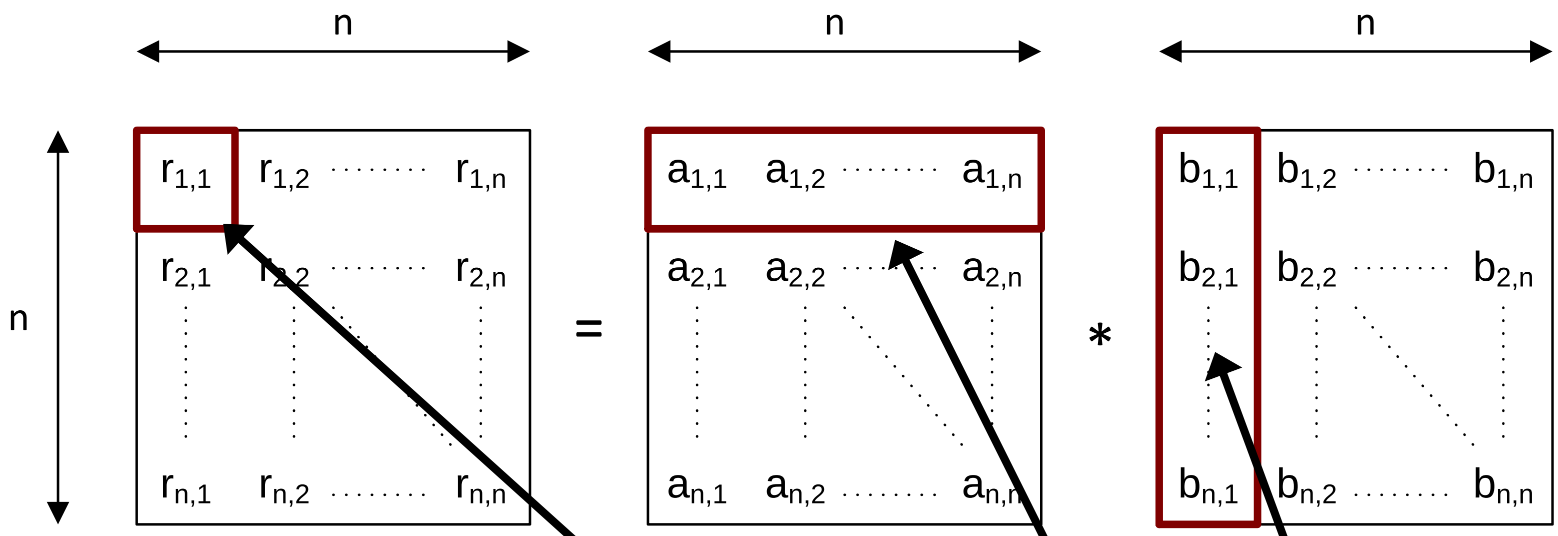
Matrix-Matrix Multiply

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

```
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

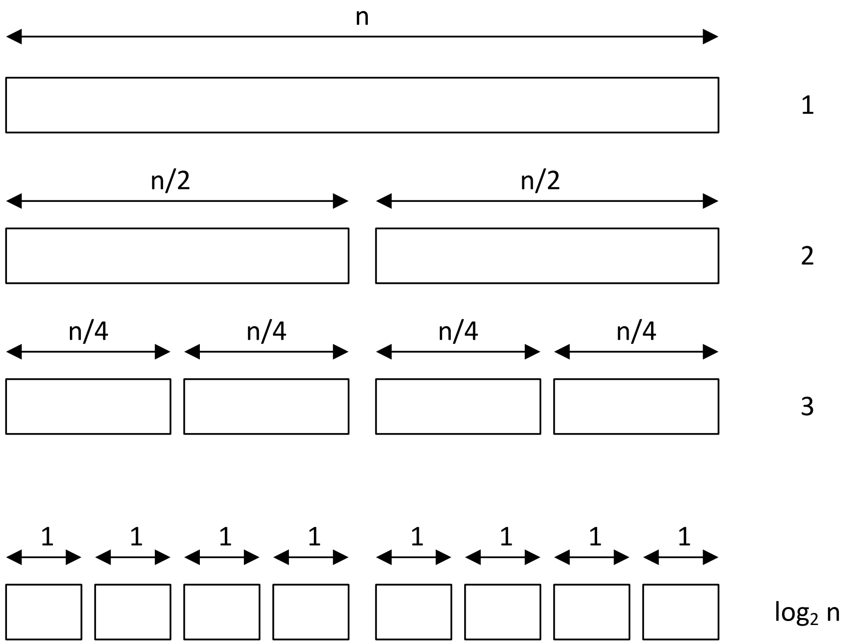



```
function[R]=randMMM(N)
    A=rand(N,N);
    B=rand(N,N);
    R=A*B;
end
```

```
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) much better than O(n^2)

Exponential Algorithms

- Algorithms which explore some multi-dimension 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

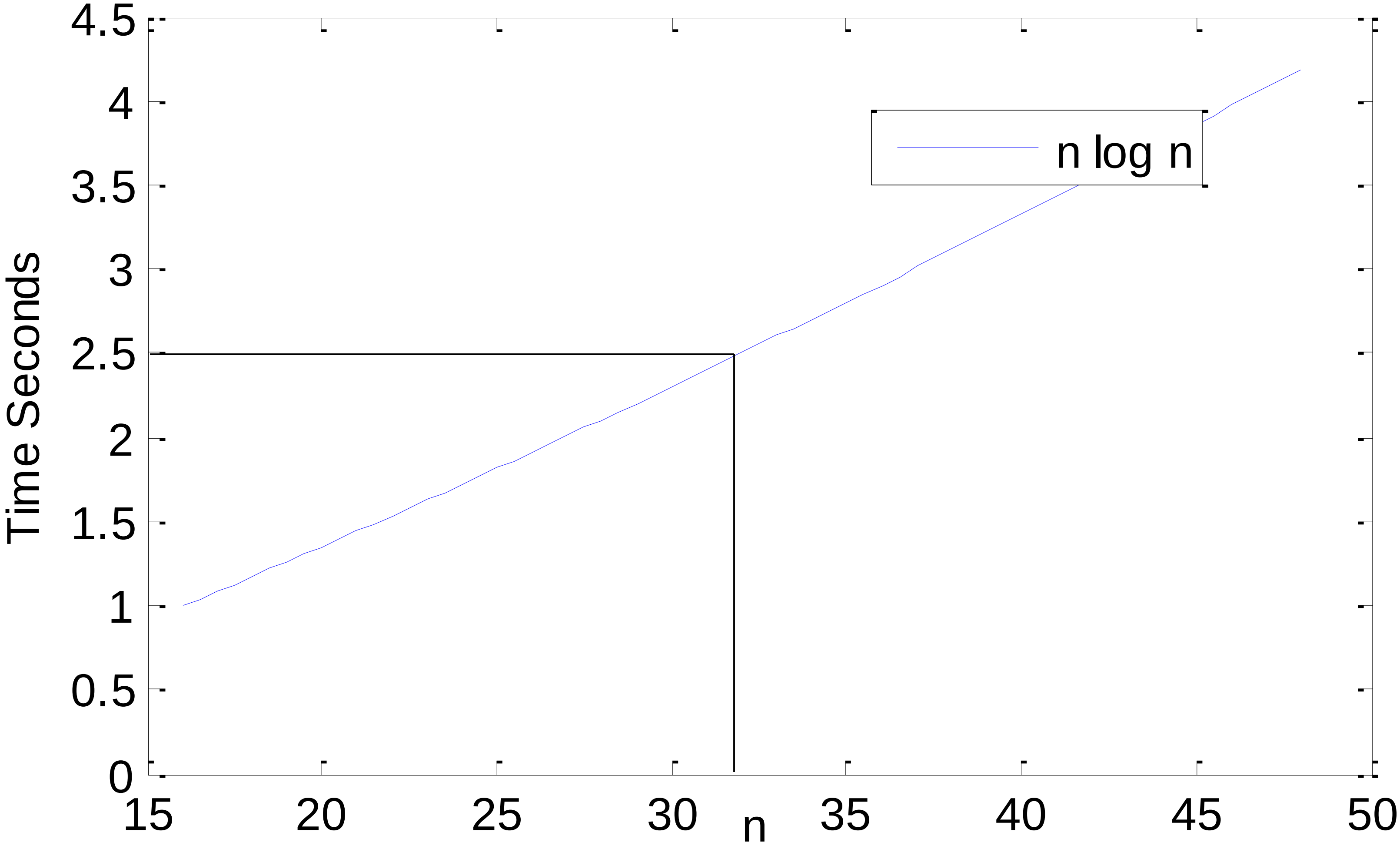
- Find the best sub-set from a larger set of resources
  - Many interesting engineering problems are combinatorial

Avoid anything more than polynomial

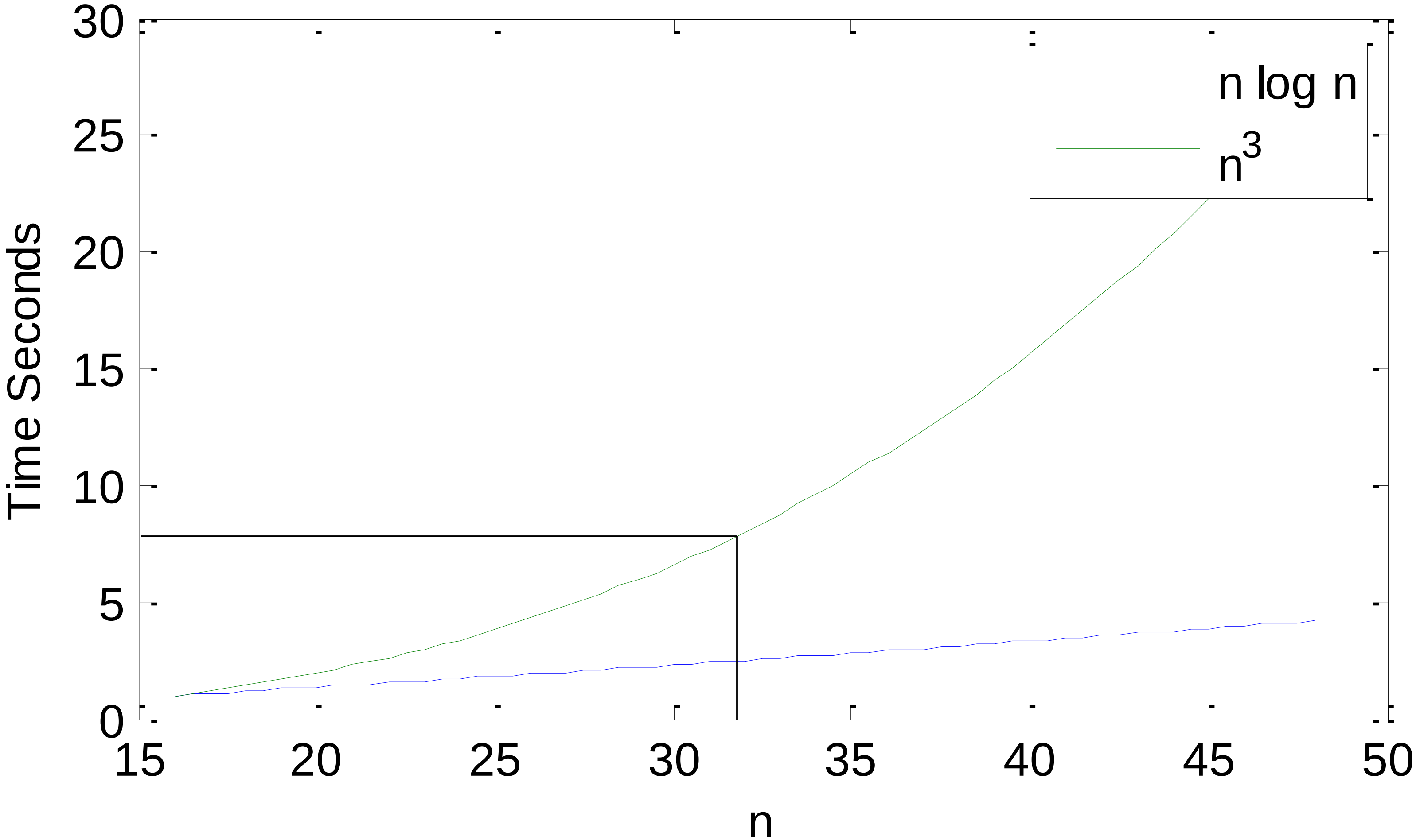
http://www.win.tue.nl/~gwoegi/P-versus-NP/homer2.jpg, Copyright Fox

Thought Experiment

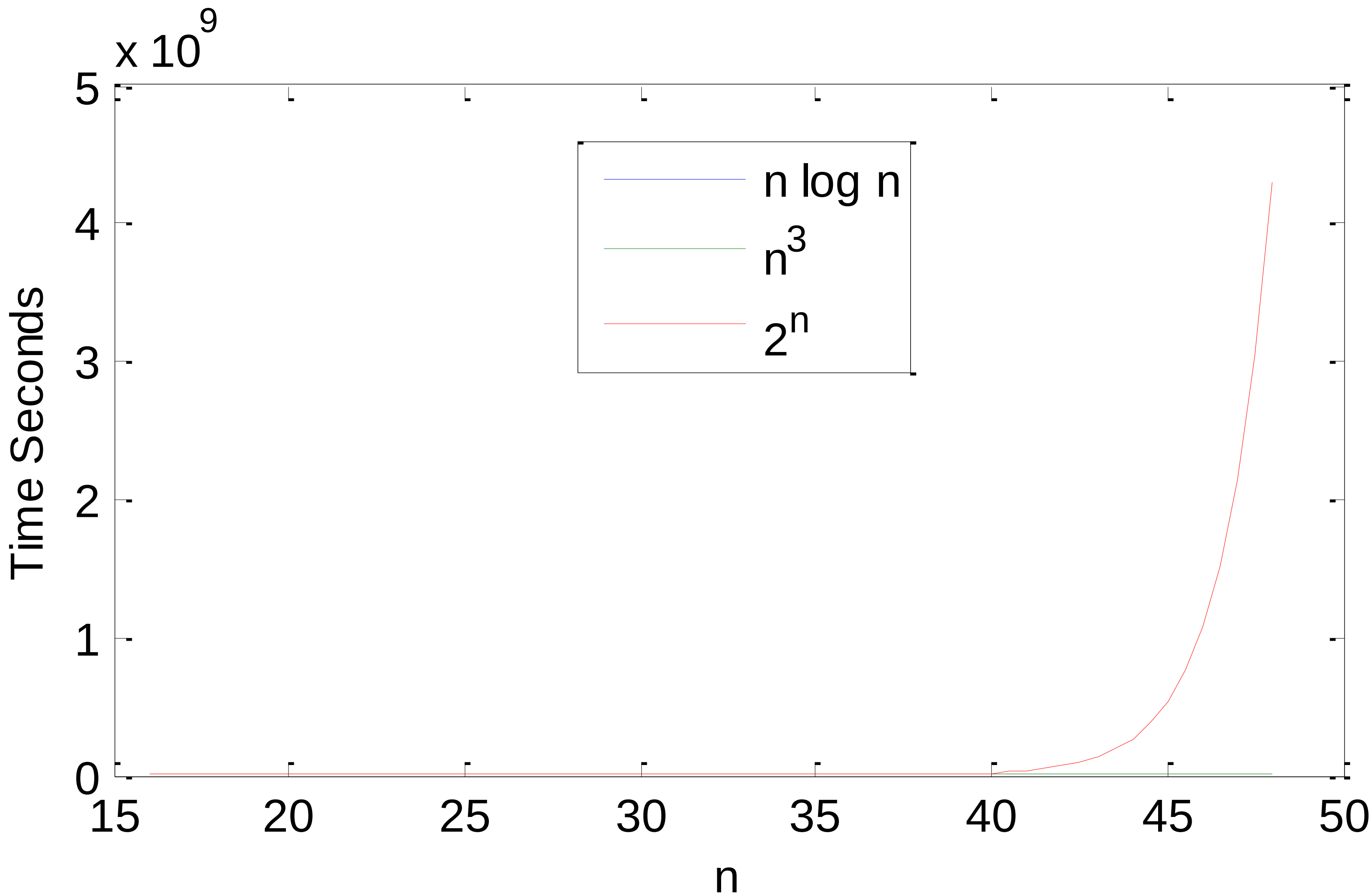
- We have four applications
  - All currently run in one second
  - All currently handle problems of “size” 16
- Each application has different complexity
  - Log-linear: $O(n \log n)$
  - Cubic: $O(n^3)$
  - Exponential: $O(2^n)$
  - Combinatorial: $O(n!)$
- The customer wants to handle problems of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



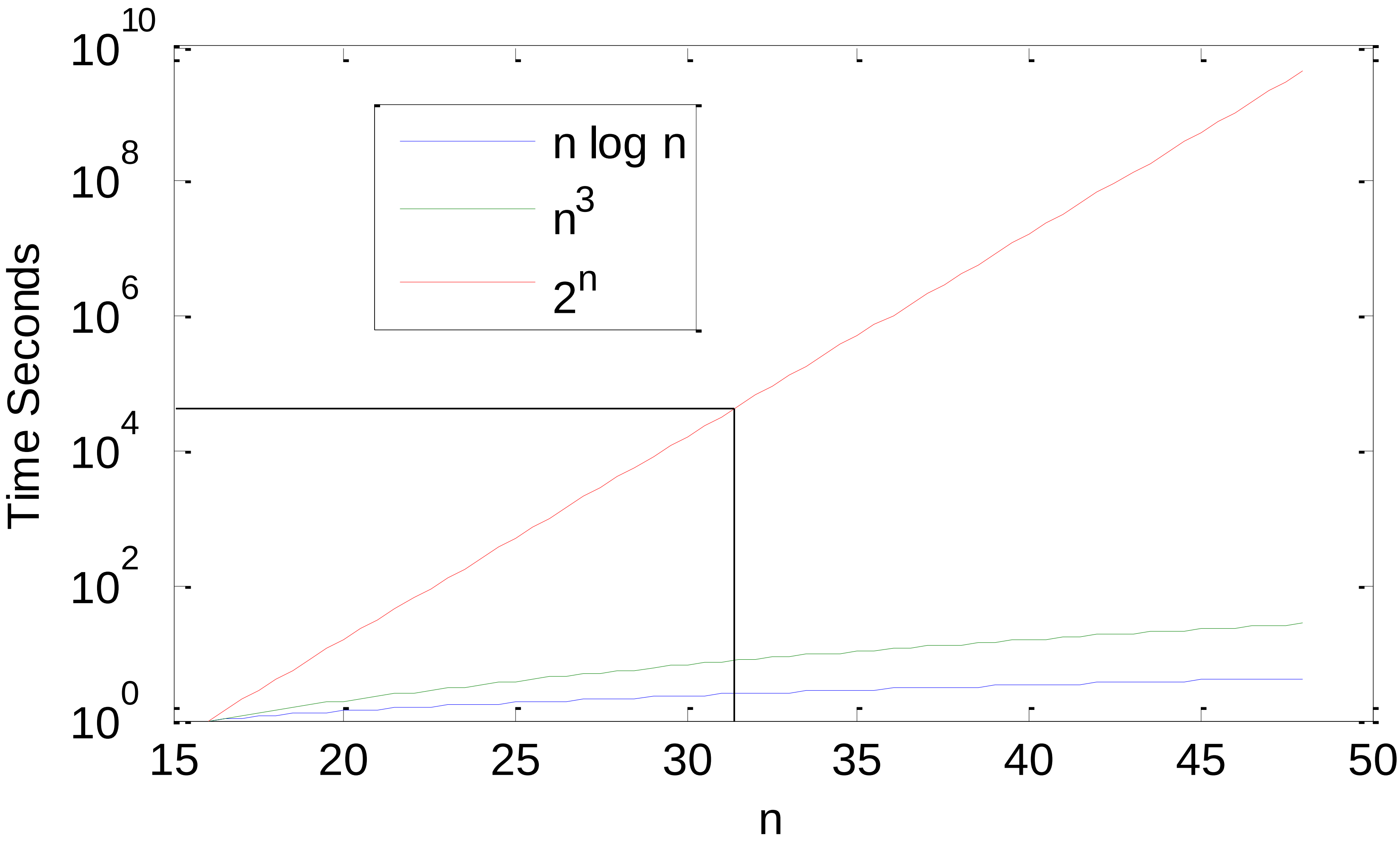
Speedup by about 2.5 times – probably use multi-core



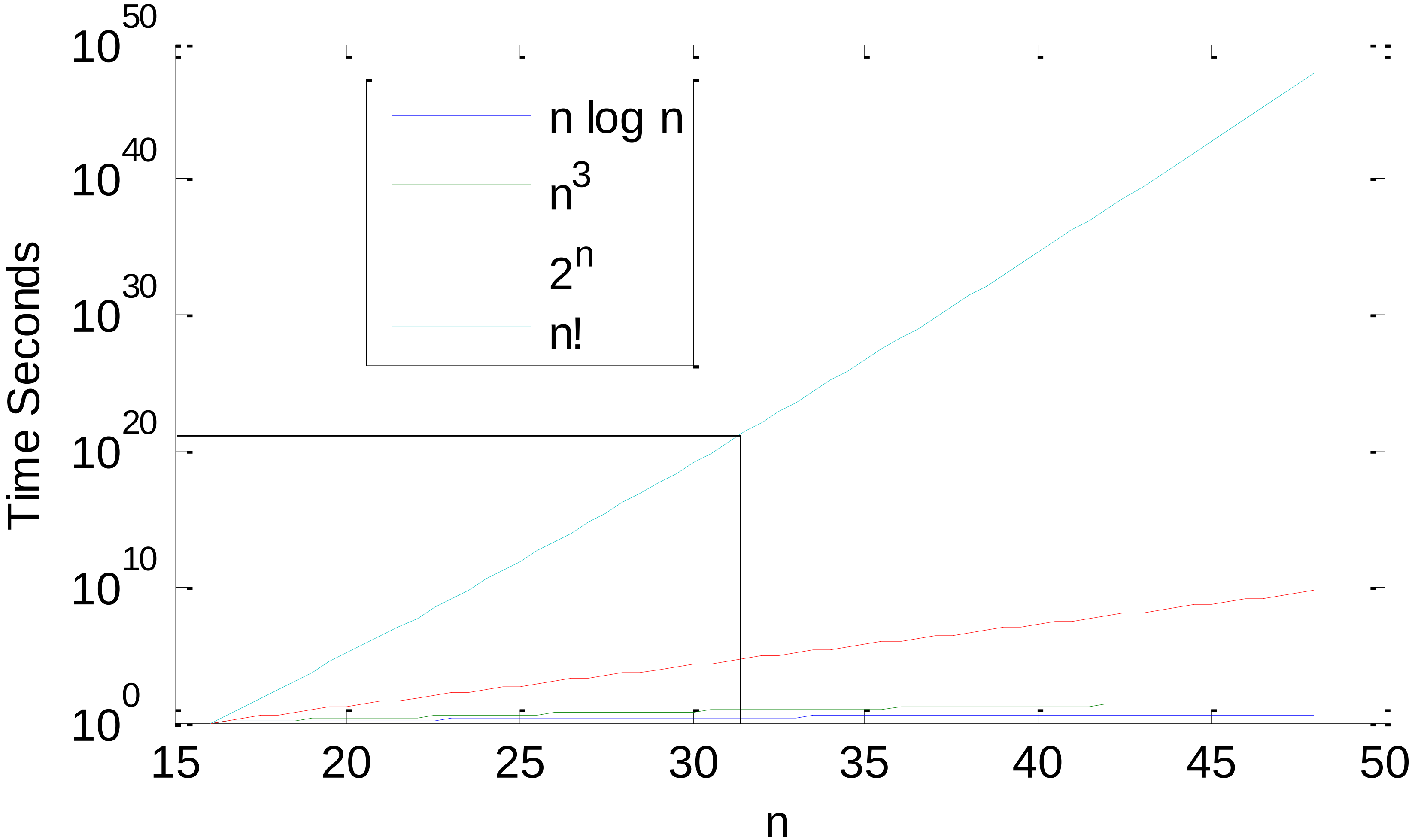
Speedup by about 8 times – maybe use a GPU?



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
  - Circuit placement: optimal place and route

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?

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

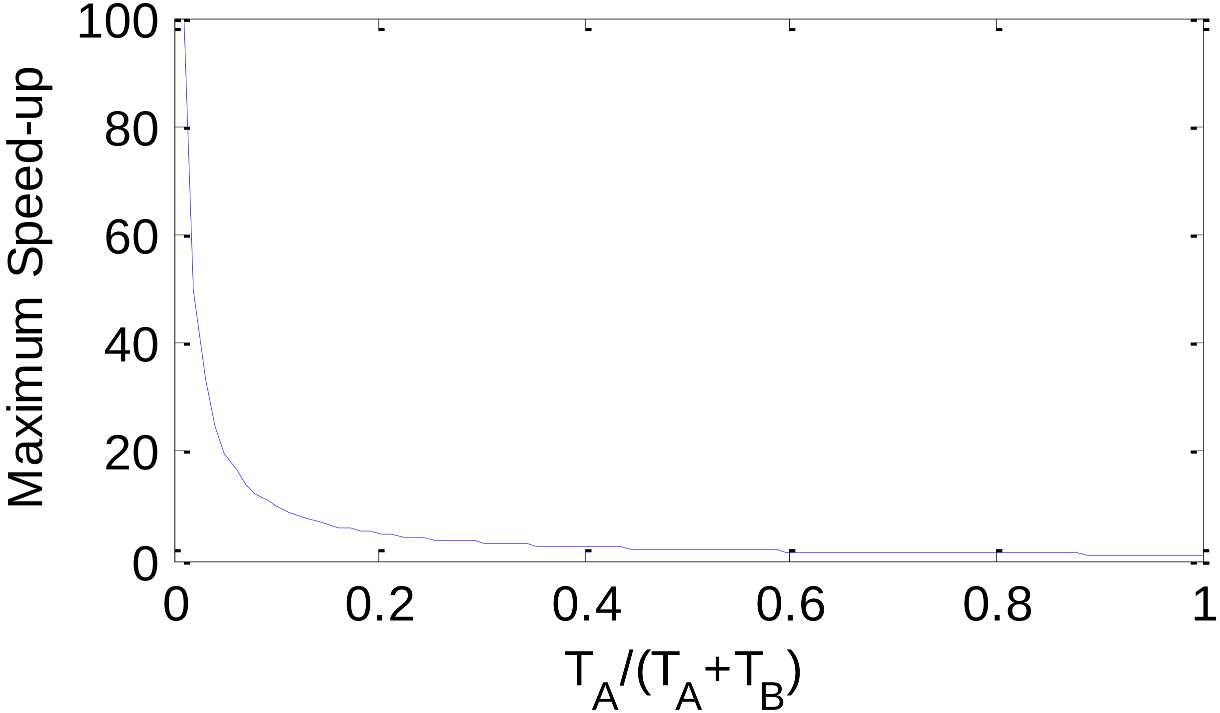
$$T_{\mathbf{X'}} = T_{\mathbf{A}} + T_{\mathbf{B}} / S_{\mathbf{B}} \qquad \text{New execution time}$$

$$\begin{aligned} S_{\mathbf{X}} &= T_{\mathbf{X}} / T_{\mathbf{X'}} \qquad \text{Achieved speedup} \\ &= \frac{T_{\mathbf{A}} + T_{\mathbf{B}}}{T_{\mathbf{A}} + {}^{T_{\mathbf{B}}}\!/_{S_{\mathbf{B}}}} \\ &= \frac{T_{\mathbf{A}} + T_{\mathbf{B}}}{T_{\mathbf{A}}} \quad \text{as} \quad \xrightarrow{S_{\mathbf{B}}} \infty \end{aligned}$$

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

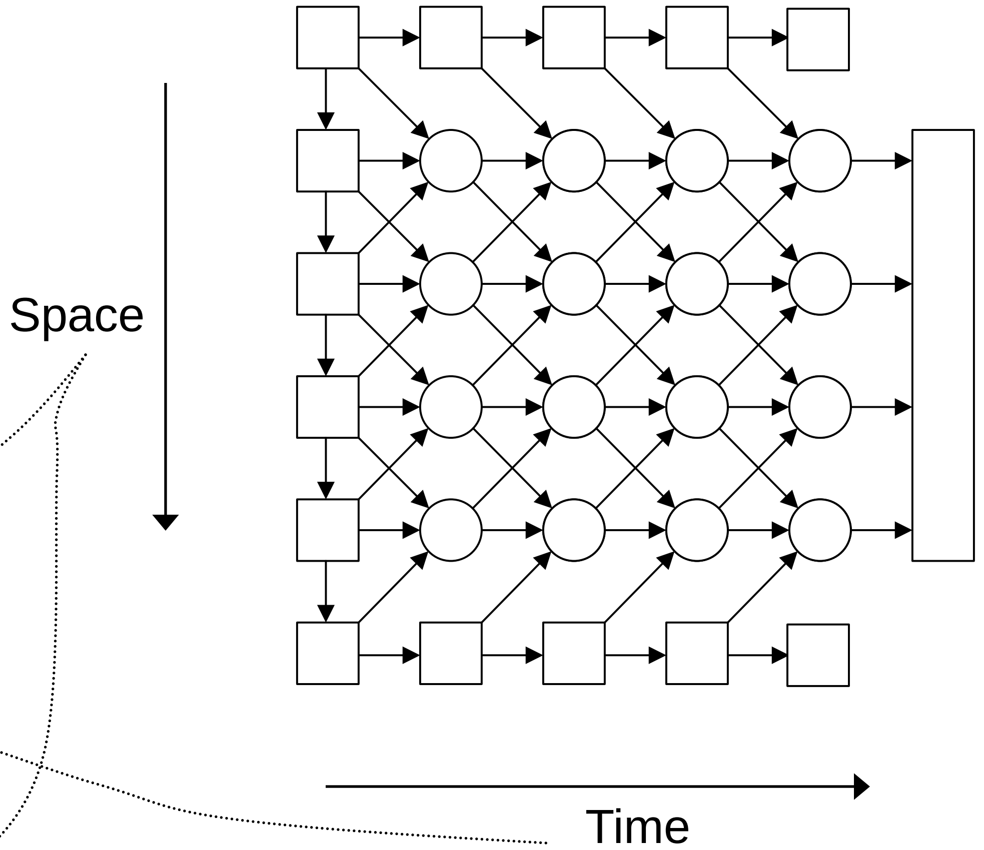
- Need a *tiny* serial fraction to achieve big speed-ups

- Are 1000x speed-ups realistic then?

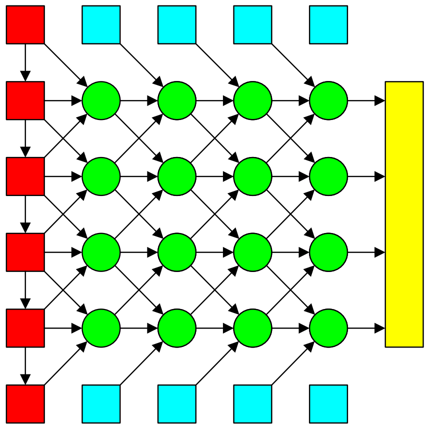


Practical example

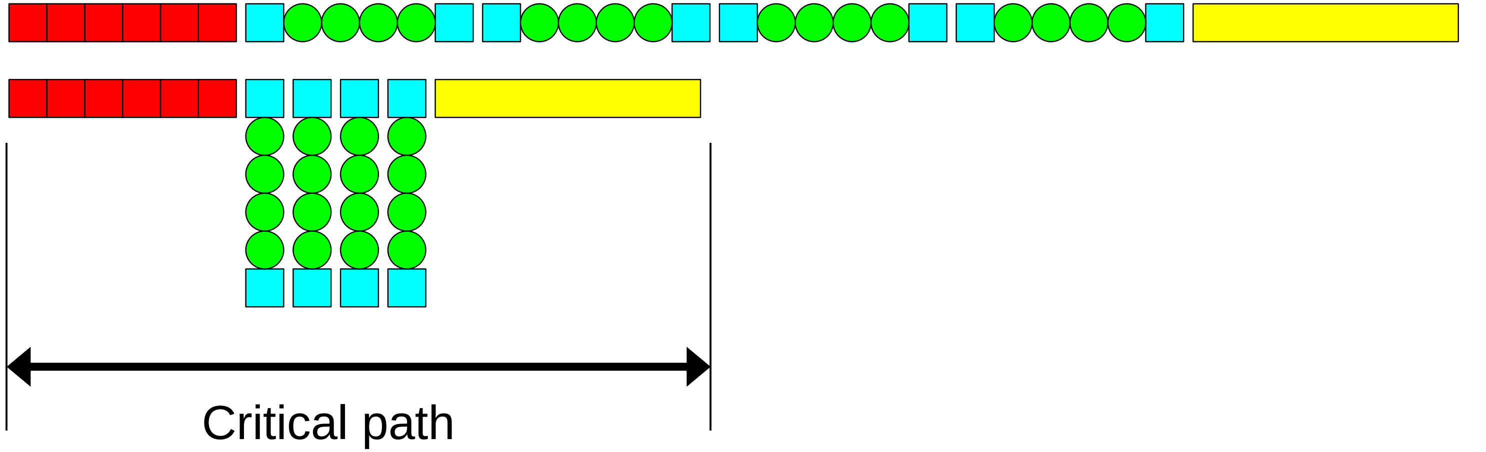
- Finite difference applications (fluid-mechanics, physics)
  – Discretise continuous space into cells
  – Discretise continuous time into distinct time-steps
- Goal of acceleration is to support finer resolution solutions
  – Usually increase resolution of space and time axis together
  – Let’s take ***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])
```

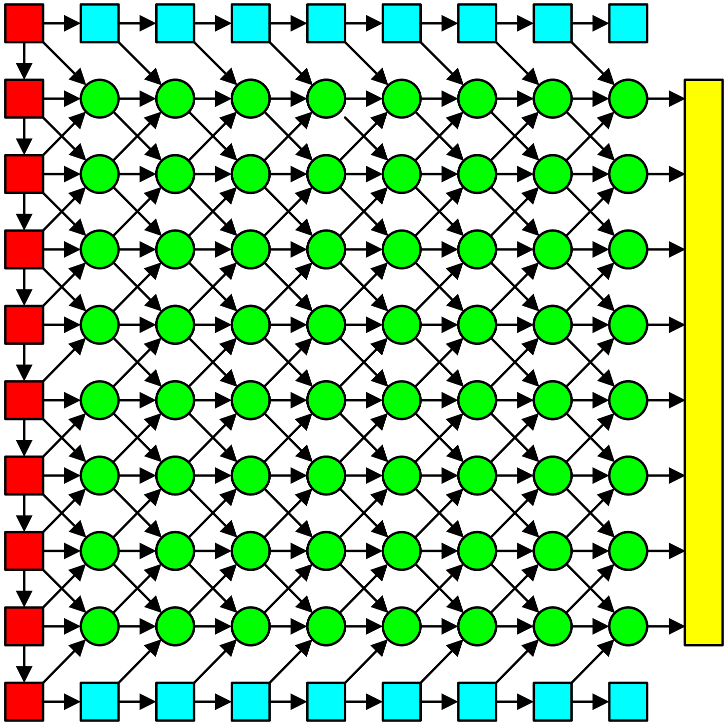


•Total work: $(n+1)(n+2) + C \in O(n^2)$

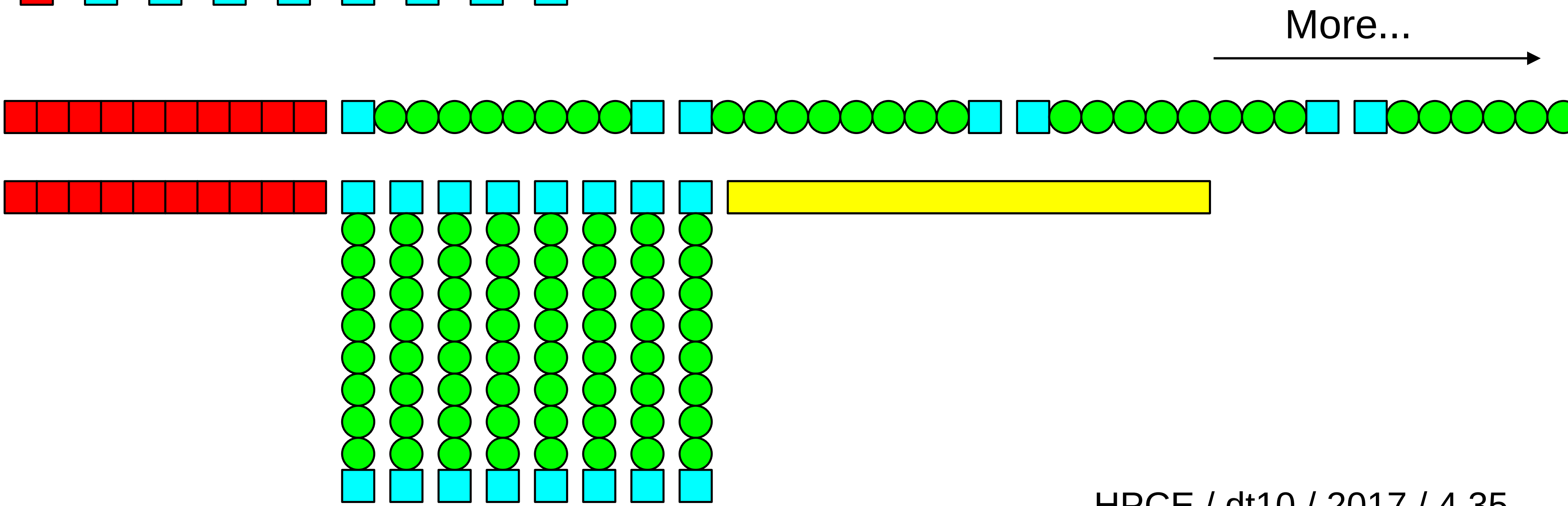


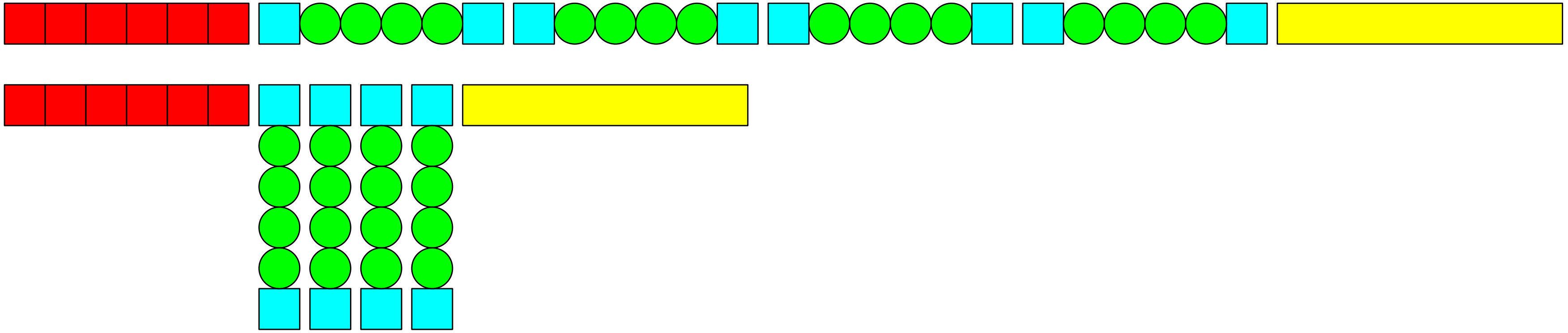
Critical path

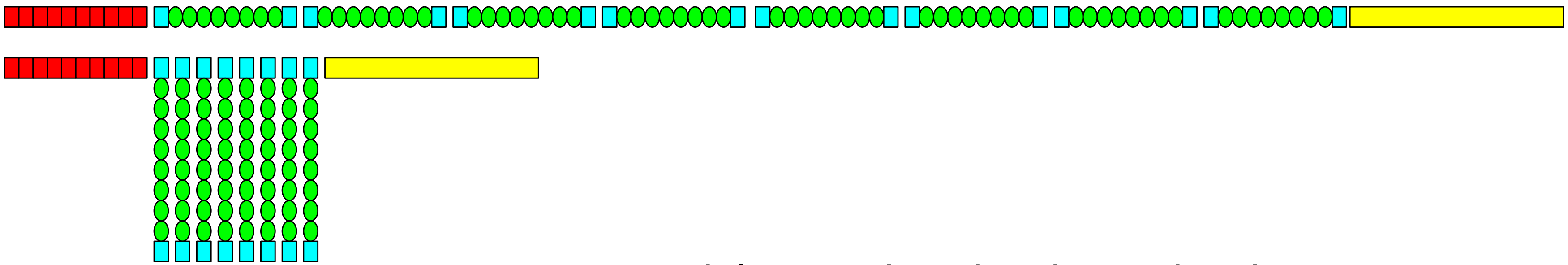
- ***Critical path***: longest path through dependency graph
    - *Assume infinite processors, and zero communication overhead*



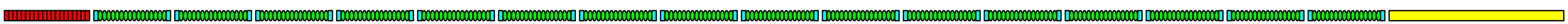
- Double the resolution of the grid
  - Increase resolution in both time and space
  - Model things of smaller size
  - Model things that happen faster
- Solution is better, but more compute intensive
  - Need High-Performance Computing!

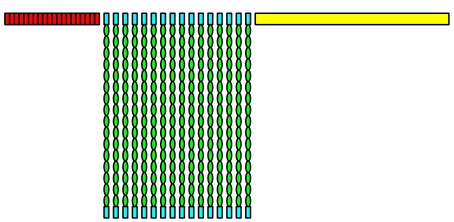






Serial execution: (n+2) + n×(n+2) + C





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

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

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

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

Tasks and Dependencies

- Parallelism can sometimes be detected
  - Compilers: re-order statements during optimisation
  - Super-scalar CPUs: issue instructions in parallel
- Sometimes parallelism can be made explicit
  - Programmer: manually schedule exact order of threads
- Problems with both approaches
  - *Detecting parallelism*: expensive and may miss opportunities
  - *Explicit parallelism*: difficult to program, and often sub-optimal

- Solution we have used: permissive task-based parallelism
  - Indicate or imply which parts ***may*** execute in parallel
  - Make the specific scheduling **`Somebody Else's Problem`**

Questions arising

- *When should I stop splitting tasks?*
- *If I have P processors, why not split the work into P tasks?*
- *It works well on 4 processors, but will it work on 64?*
- *How can I ensure linear scaling?*
- *Should I pre-calculate on the CPU or the GPU?*

A concrete treatment of tasks

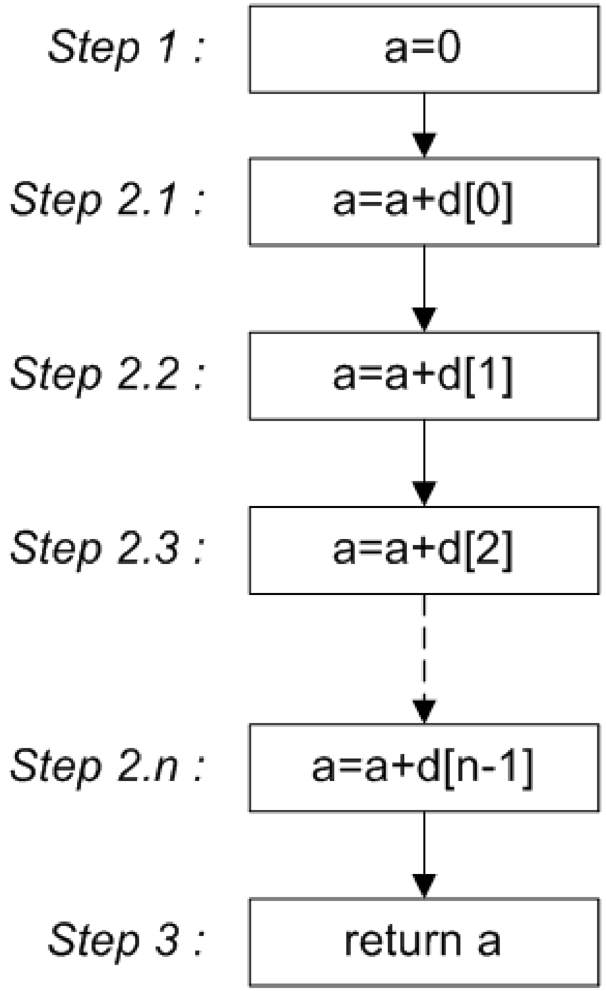
- We have looked at decomposing programs into tasks
  - Dependencies: task A must execute before task B
  - Scheduling: find an order of execution that respects dependencies
  - ASAP scheduling: As Soon As Possible

- Now we'll briefly look at a formal task-based system : *Cilk*
  - Very influential academic project started in the early 90s
  - Good combination of theory and practical results
  - Eventually bought by Intel, now incorporated into Intel compiler
    - http://software.intel.com/en-us/articles/intel-cilk-plus/
  - Basic concepts used in TBB, Microsoft TPL, ...

- Some of the structure from this lecture is adapted from Leiserson and Prokop, "A Minicourse on Multithreaded Programming", 1998.

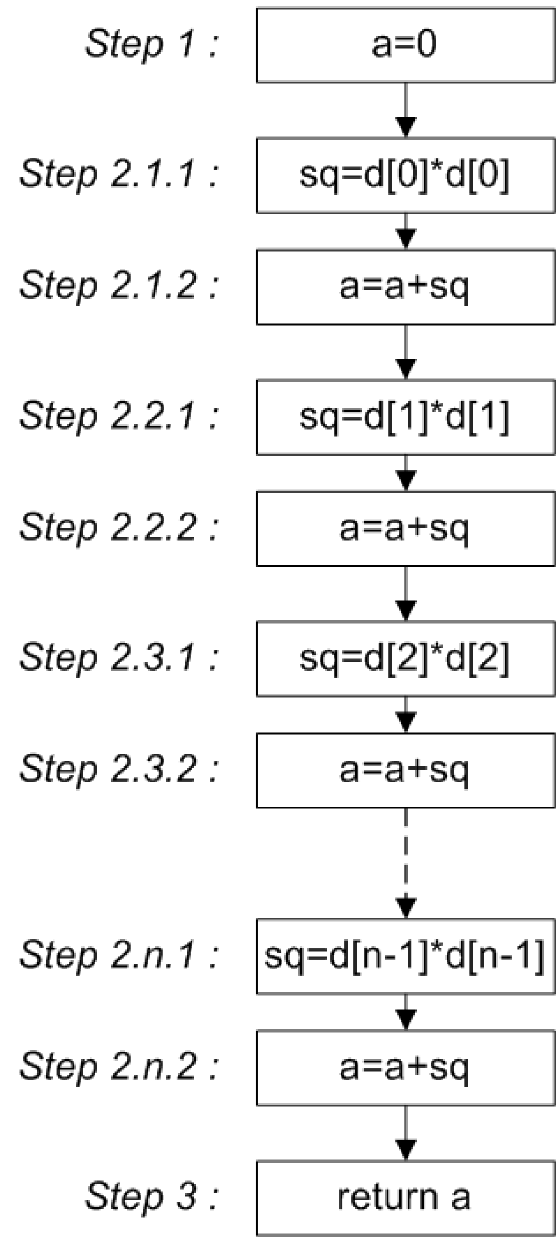
Algorithms as steps

```
int Sum(int n, int *d)
{
   a=0;                          // 1
   for(int i=0; i<n; i++){       // 2
      a = a + d[i];              // 2.i
   }
   return a;                     // 3
}
```

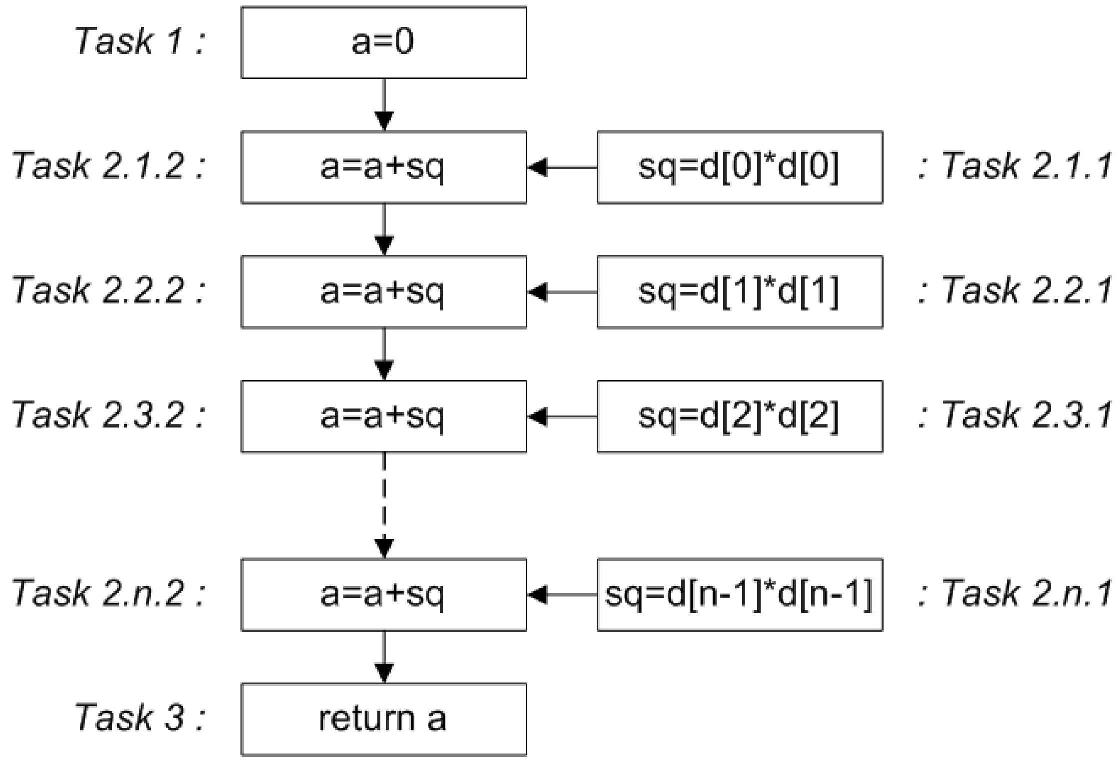


- Interpret algorithms as a sequence of steps or tasks
- What constitutes a "step" depends on the context: *instruction*; *statement*; *function*
- C-like languages impose a strict ordering on the sequence of steps

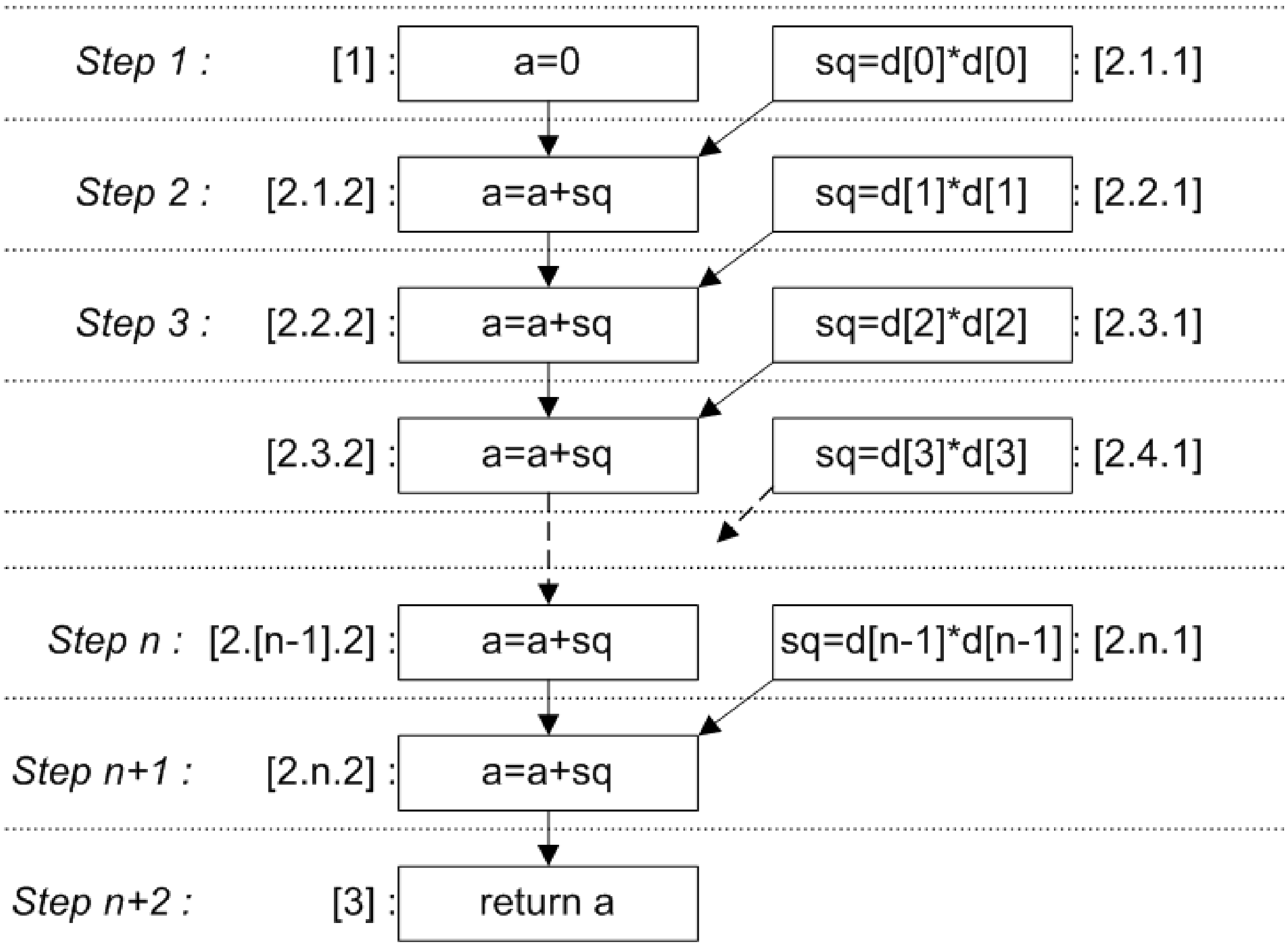
```c
int SumOfSquares(int n, int *d)
{
  a=0;                          // 1
  for(int i=0; i<n; i++){       // 2
    sq = d[i]*d[i];             // 2.i.1
    a  = a + sq;                // 2.i.2
  }
  return a;                     // 3
}
```



- Each arrow indicates a dependency
  - Step 2.1.1 ***depends on*** Step 1
- Tasks must be ***scheduled*** such that the dependency order is preserved
  - Can’t execute Step 2.3.1 before Step 2.2.2
- Traditional C code imposes dependencies based on loops and sequences
  - e.g. Step 2.i.1 depends on 2.[i-j].2, j>0

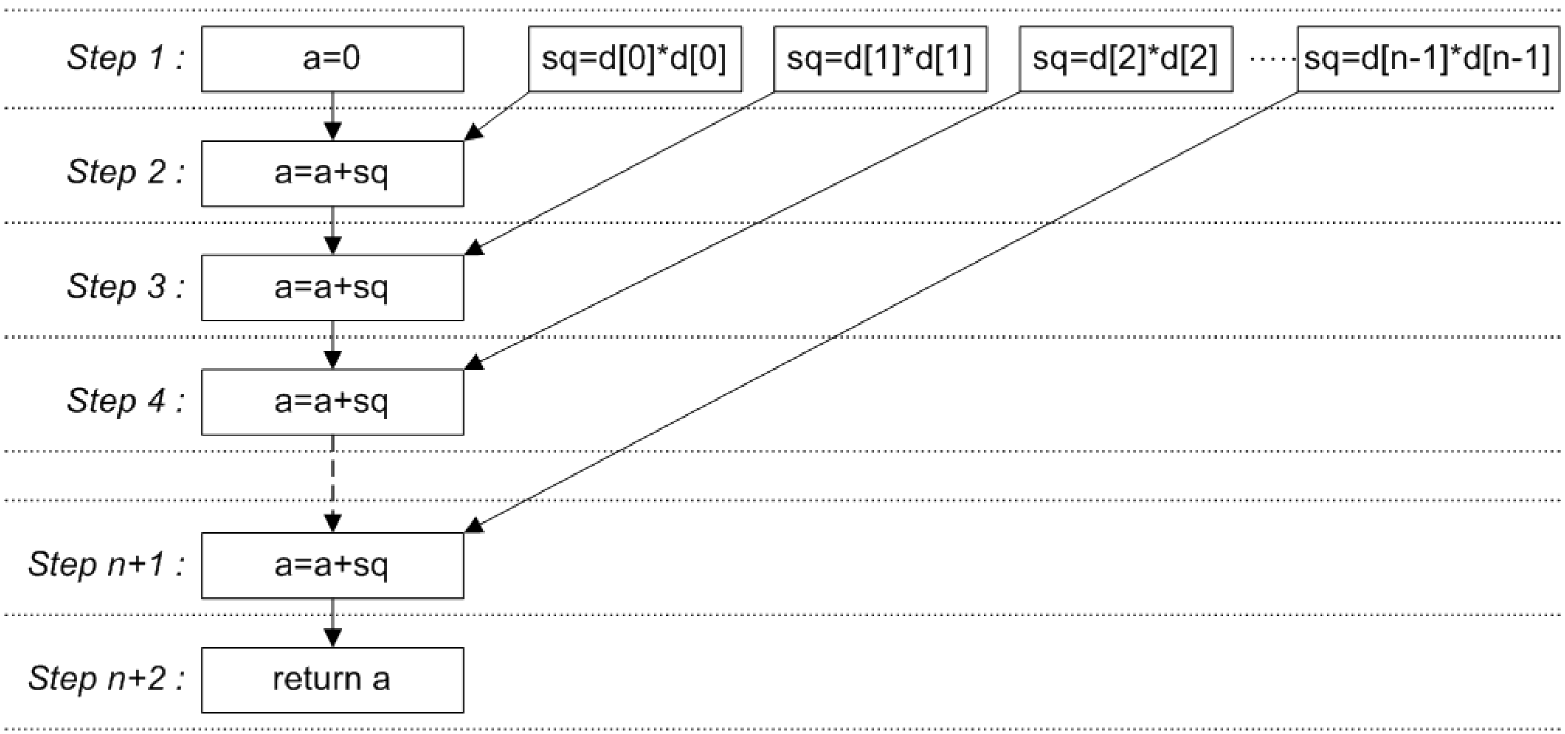


- Re-express dependencies in terms of ***true*** dependencies
  - ***Data dependencies*** – one task consumes data produced by another
  - ***Control dependencies*** – loop or branch depending on a tasks output data
- Reduce ***false dependencies*** which are an artefact of imperative programming
  - Step 2.[i+1].**1** does not depend on 2.i.2
  - But, step 2.[i+1].**2** does depend on 2.i.2



- Scheduling: choose an execution order for each task
  - Must still obey true dependencies
  - Try to exploit parallelism

- Many types of scheduling methods:
- **ASAP** (As Soon As Possible)
  - Start tasks as soon as all dependents have completed
- **ALAP**: As Late As Possible
  - Schedule tasks just before they are needed, ***without*** increasing total time



Scheduling for compute systems

- Given a graph of tasks we need to schedule execution
  - Co-ordinated by many systems: OS, libraries, compiler, device
- ***Static scheduling***: decide on task order ***before*** execution
  - *Programmer*: use imperative structures (loops) to impose order
  - *Compiler*: convert statements to sequence of instructions
  - *Processor*: increment program-counter, execute next instruction
- ***Dynamic scheduling***: schedule tasks ***while*** executing
  - *Programmer*: use parallel libraries to expose dependencies
  - *Parallel Libraries*: Keep track of which tasks are ready to run
  - *Operating System*: Choose which of multiple threads to execute
  - *Processor*: Select which of multiple instructions to execute

- ***General Problem***: maximise parallelism; minimise execution time

The Cilk Language

- Two fundamental operations in Cilk
  - **spawn** : indicate a function call that *may* operate in parallel
  - **sync** : wait until all spawned functions have completed

```
cilk int Fib(int n)
{
   if(n<2) return n;


   int x=spawn Fib(n-1);
   int y=spawn Fib(n-2);
   sync;
   return x+y;
}
```

```
int Fib(int n)
{
   if(n<2) return n;
   int x, y;
   tbb::task_group g;
   g.run( [&](){ x=Fib(n-1); } );
   g.run( [&](){ y=Fib(n-2); } );
   g.wait();
   return x+y;
}
```

The Cilk Language

- Two fundamental operations in Cilk
  - **spawn** : indicate a function call that *may* operate in parallel
  - **sync** : wait until all spawned functions have completed
- Cilk is a faithful extension of C
  - If you delete Cilk keywords from a program it will still execute as C
  - ***Serial Elision principle***: remove the keywords, it becomes serial

```
cilk int Fib(int n)
{
    if(n<2)
        return n;

    int x=spawn Fib(n-1);
    int y=spawn Fib(n-2);
    sync;
    return x+y;
}
```

```
int Fib(int n)
{
    if(n<2)
        return n;

    int x=Fib(n-1);
    int y=Fib(n-2);

    return x+y;
}
```

Cilk programs as a DAG

- The pattern of spawn and sync commands defines a graph
  - The graph contains dependencies between different functions
  - **spawn** command creates a new task with an out-bound link
  - **sync** command creates inbound link from spawned tasks

```
cilk int Fib(n=3)
{
   if(n<2)
       return n;

   int x=spawn Fib(n-1);
   int y=spawn Fib(n-2);
   sync;
   return x+y;
}
```

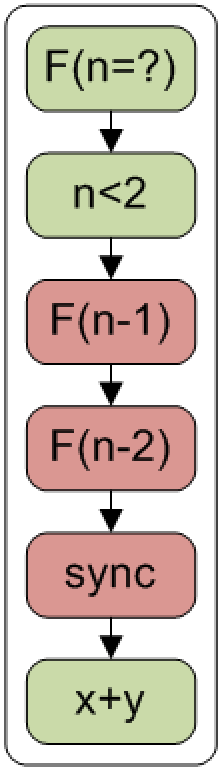
```
cilk int Fib(n=2)
{
   if(n<2)
       return n;
   int x=spawn Fib(n-1);
   int y=spawn Fib(n-2);
   sync;
   return x+y;
}
```

```
cilk int Fib(n=1)
{
   if(n<2)
       return n;
   int x=spawn Fib(n-1);
   int y=spawn Fib(n-2);
   sync;
   return x+y;
}
```

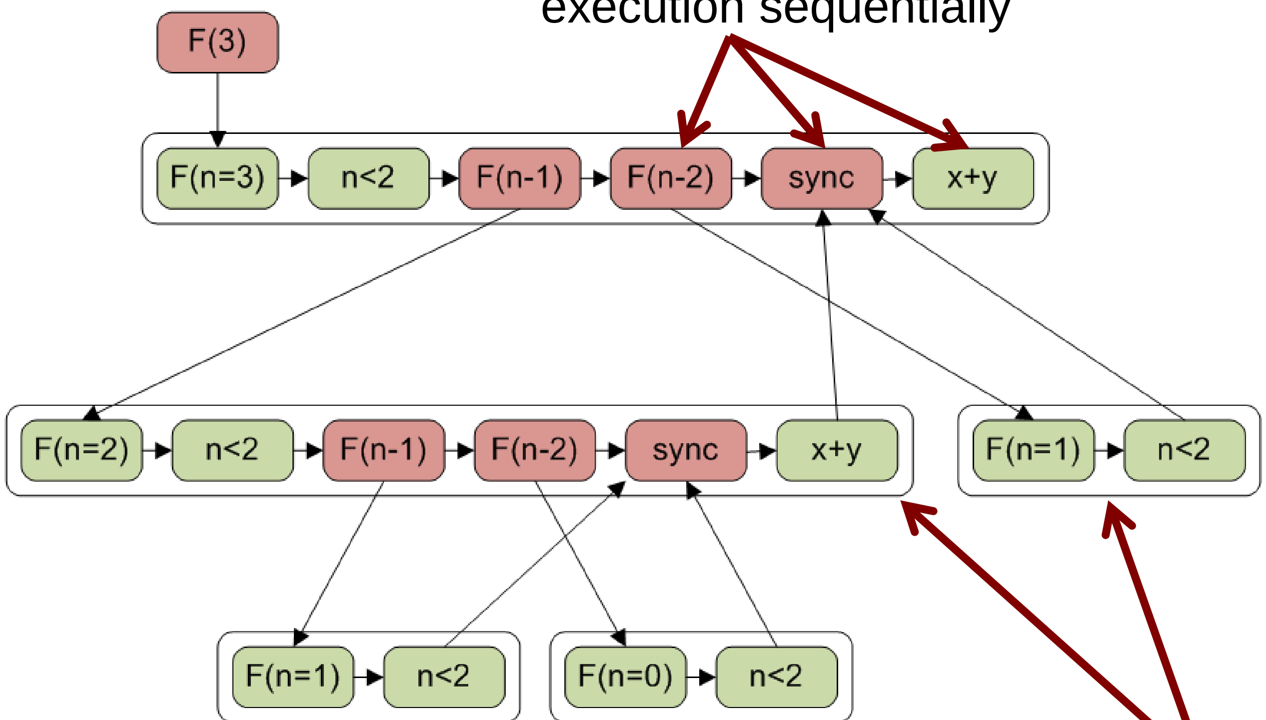
```
cilk int Fib(n=1)
{
   if(n<2)
       return n;
   ...
}
```

```
cilk int Fib(n=0)
{
   if(n<2)
       return n;
   ...
}
```

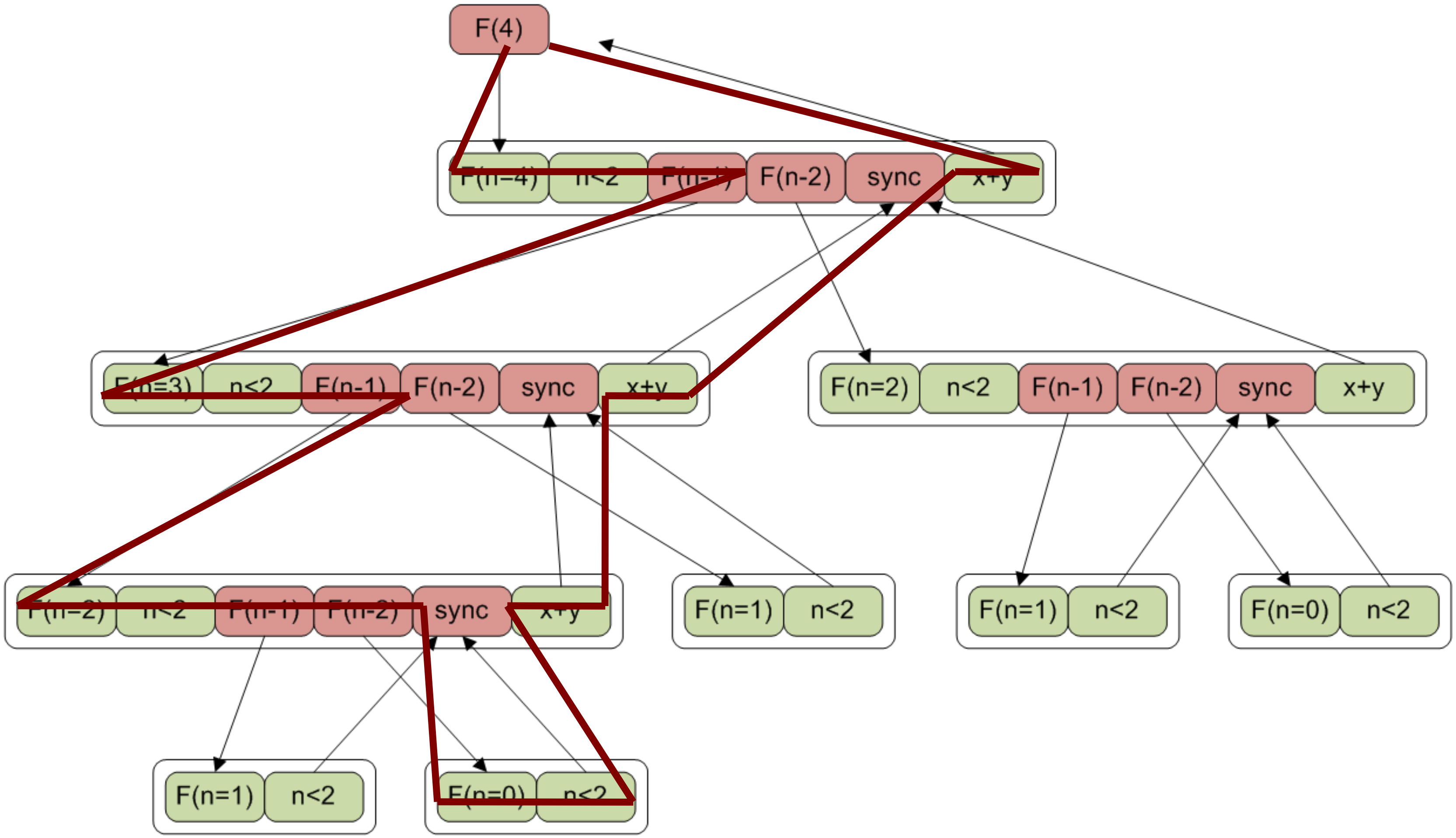
```
cilk int Fib(int n)
{
   if(n<2)
       return n;

   int x=spawn Fib(n-1);
   int y=spawn Fib(n-2);
   sync;
   return x+y;
}
```



Steps within a function execution sequentially



Independent functions may execute in parallel



Total Work : T_1 - total time required to execute all tasks

Critical path : T_∞ - longest path through all tasks

assume each step takes unit time : total work = 35; critical path = 16

Best case and worst-case times

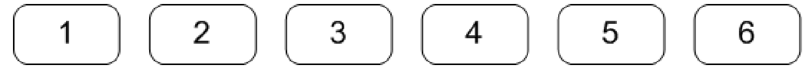
- Define three times: T_1, T_P, T_∞
  - T_1 : Time to execute on one processor (***Total Work***)
  - T_P : Time to execute on P processors
  - T_∞ : Time to execute on infinite processors (***Critical Path***)
  - T_1 / T_P : Speedup with P processors
- Can establish an ordering on the times
  - $T_1 / P \leq T_P$ - *Maximum speedup with P processors is P*
  - $T_P \geq T_\infty$ - *Finite processors are no faster than infinite*
- Can talk about scalability
  - if $T_1 / T_P \sim P$ then ***Linear speedup*** (perfect scaling)
  - We always want linear speedup – can we achieve it?

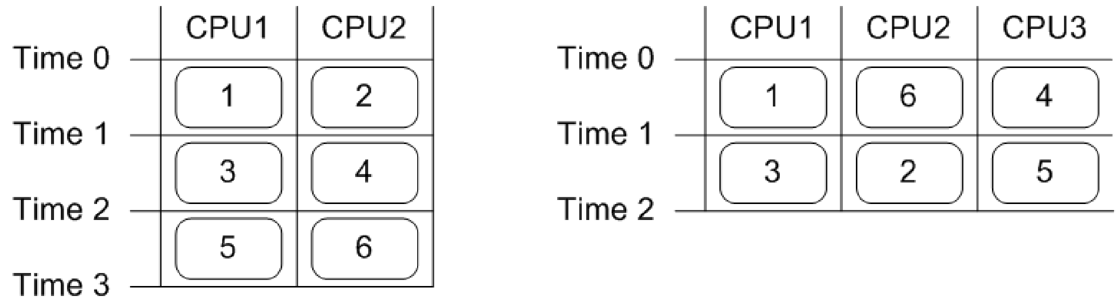
Greedy Schedulers

- A Greedy Scheduler executes work using an ASAP approach
  - Each “time step” launch all tasks with no dependencies
  - *The notion of a time-step is deliberately context dependent*
- When executing with P processors we have two types of step
  - **complete step** : There are P or more tasks ready to execute
  - **incomplete step** : There are less than P tasks ready to execute

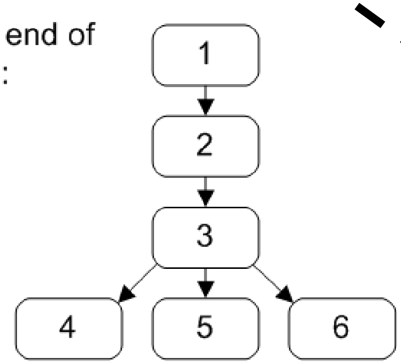
- A greedy scheduler always achieves $T_P \leq T_1 / P + T_\infty$
  - Best case is easy to visualise
    - we do all work in T_P *complete* steps
  - Worst case is a bit more difficult
    - Steps on critical path execute in *incomplete* steps
    - Last step on critical path frees up all remaining work for *complete* steps

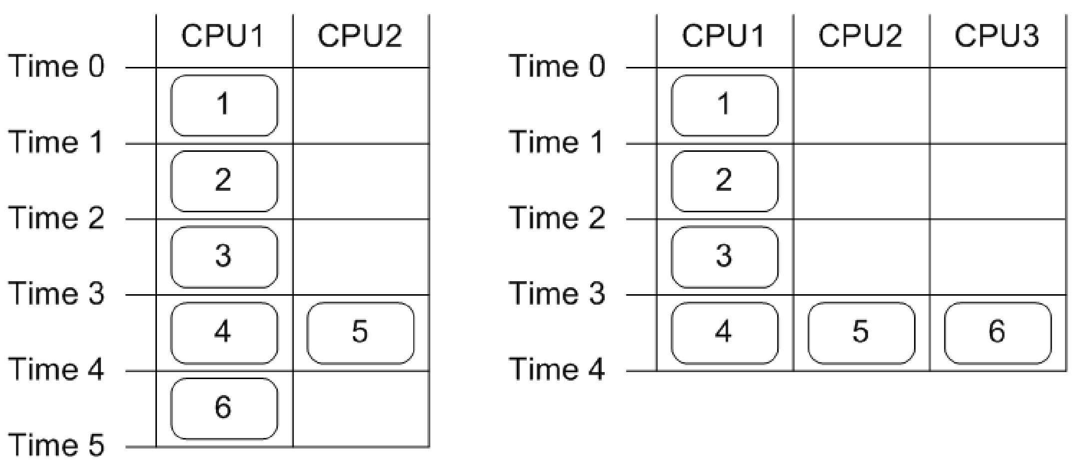
All tasks ready to run, no dependencies at all





All tasks occur at end of critical path:





Linear Scaling and Greedy Schedulers

- Previous equations assume zero-cost scheduling
  - Some overhead involved in tracking tasks that can be run
  - Some overhead in scheduling ready tasks to a processor
- Define ***critical overhead*** : c_∞
  - Smallest c_∞ such that $T_P \leq T_1 / P + c_\infty \times T_\infty$
  - Covers the cost of tracking dependencies on critical path
- Linear scaling if there is usually much more work than CPUs
  - **Average parallelism** : $\underline{P} = T_1 / T_\infty$
  - **Assumption of parallel slackness** : $\underline{P} / P >> c_\infty$
  - Therefore: $T_1 / P >> c_\infty \times T_\infty$
  - And so: $T_P \approx T_1 / P$ (linear speedup)
- *Assumption of parallel slackness implies linear speedup*

Is that a reasonable assumption?

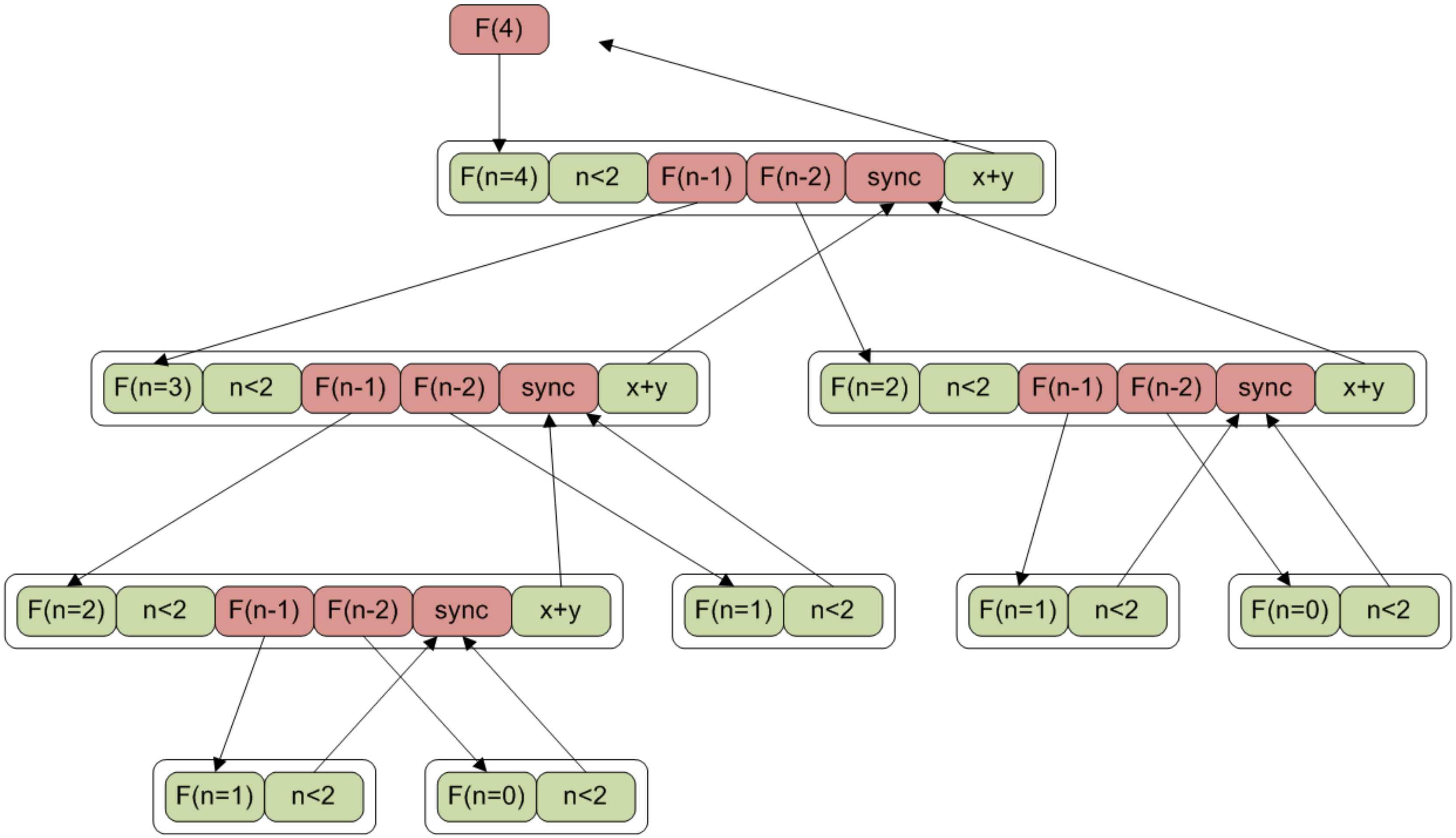
- Central idea is that most steps are complete
  - All processors are occupied most of the time
  - Does computation look like that?
- Recall Gustafson's law and the finite-difference example
  - $T_1 = O(n^2)$; $T_\infty = O(n)$
  - $\underline{P} = T_1 / T_\infty = O(n)$
  - Assuming c_∞ is not too high we should get linear scaling

- For lots of stuff the assumption is broadly true

Work-first rule

parallel_for(...) for(...)

- Define **work overhead** : $c_1 = T_1 / T_S$
  - T_S : Time to run serial version of program (*serial elision*)
  - Cost of dynamic scheduling vs static scheduling on one CPU
- What is the importance of c_1 vs c_∞ ?
  - Substitute into previous defn ($T_P \leq T_1 / P + c_\infty \times T_\infty$)
  - $T_P \leq c_1 T_s / P + c_\infty \times T_\infty$
  - Now re-introduce assumption of parallel slackness ($\underline{P} / P >> c_\infty$)
    - $T_1 / P \quad >> c_\infty T_\infty$
    - $c_1 T_S / P \quad >> c_\infty T_\infty$
  - Therefore: $T_P \approx c_1 T_s / P$
- **Work-first rule**: *minimise c_1 rather than c_∞*



Total Work : T_1 - total time required for Cilk on one processor (red+green)

Serial Work : T_S - total time required for serial-elisions (green only)

assume each step takes unit time : total work = 35; serial work = 22

Interpreting the work-first rule

- The work-first rule appears in many guises
  - What are c_1 and c_∞ in practise?
- Multi-core CPUs and OSs support traditional threads
  - c_1 : How much time to swap between two threads on a CPU?
  - c_∞ : How much time to create a new thread?
- GPUs support hundreds of parallel threads
  - c_1 : Nano-second scheduling of threads in a kernel
  - c_∞ : Milli-second cost to manage kernels from the CPU
- Intel TBB supports thousands of tasks
  - c_1 : Agglomeration of loop iterations to reduce overheads
  - c_∞ : Hierarchical task based scheduler (based on Cilk)
- *Bear this principle in mind when looking at real systems*