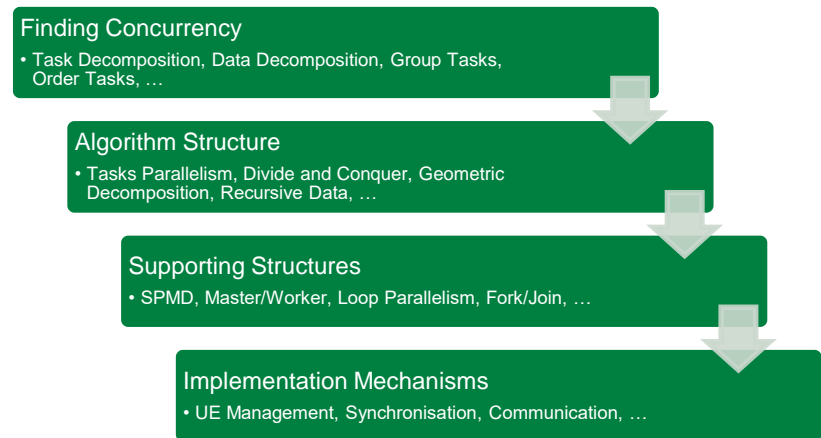


# Parallel Design Patterns-L02

Parallel algorithm analysis,  
Finding concurrency

---

Course Organiser: Dr Nick Brown  
nick.brown@ed.ac.uk  
Bayes room 2.13



# Parallel algorithm analysis

- Predict the behaviour of a planned parallelisation
  - Evaluate existing parallelisations
  - Explain benchmarking results

- Speed up
  - typically  $S(N,P) < P$

$$S(N, P) = \frac{T(N,1)}{T(N,P)}$$

- Parallel efficiency
  - typically  $E(N,P) < 1$

$$E(N, P) = \frac{S(N,P)}{P} = \frac{T(N,1)}{PT(N,P)}$$

- Serial efficiency
  - typically  $E(N) \leq 1$

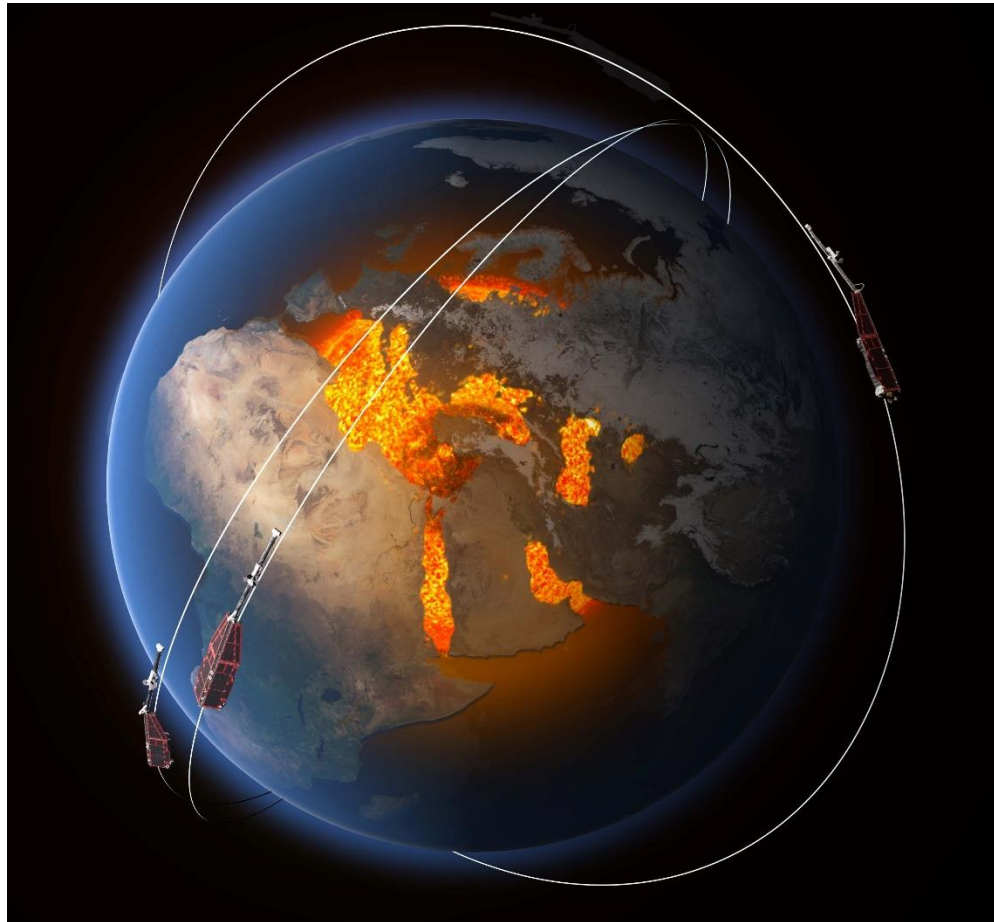
$$E(N) = \frac{T_{best}(N)}{T(N,1)}$$

*Where  $N$  is the size of the problem and  $P$  the number of processors*

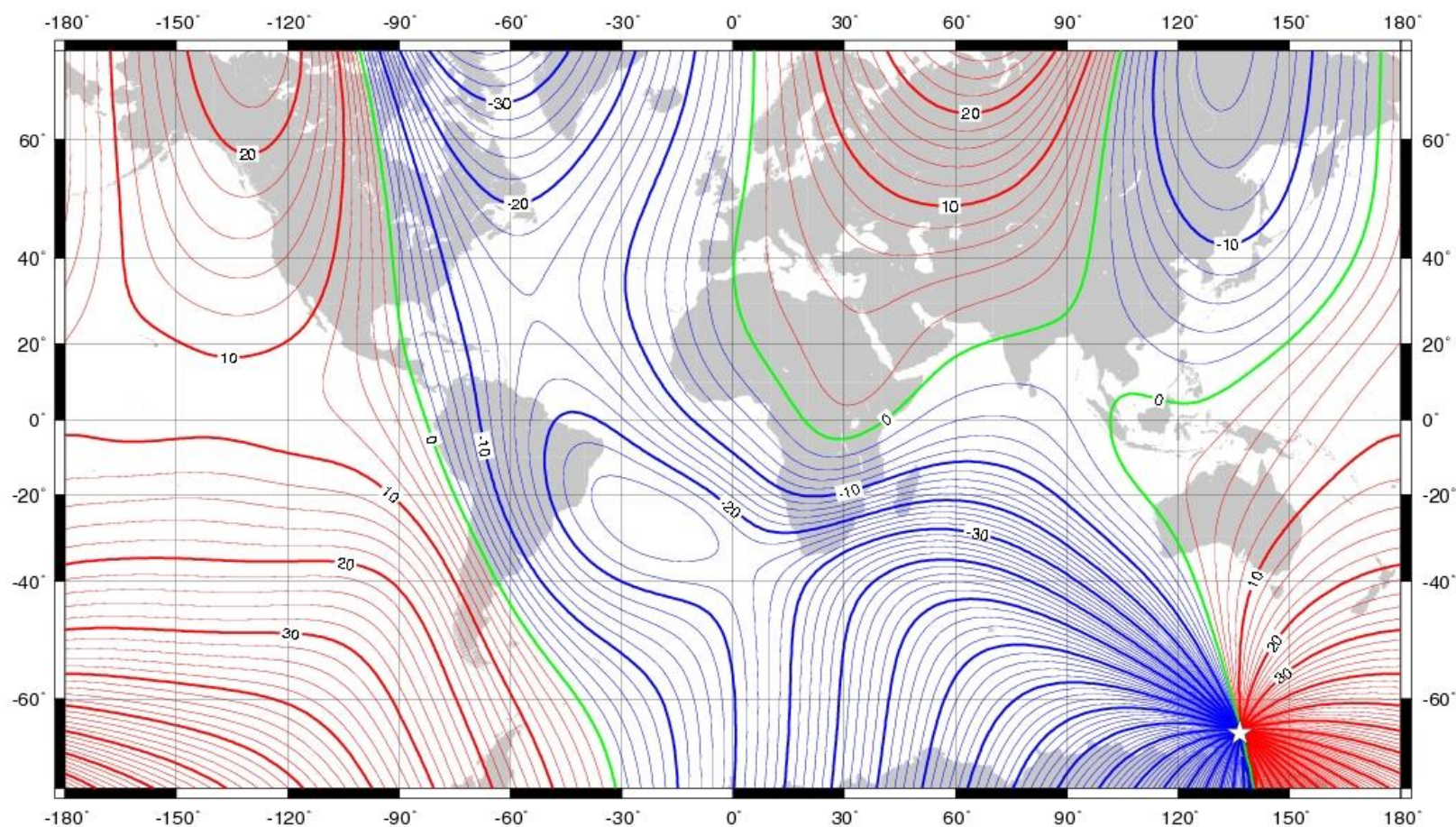


- This can give some very useful information, but requires an existing parallelised code to benchmark & measure
- Be careful what you claim, how many data points is enough to make certain claims?
- How can we predict performance and scalability at higher core counts or with certain modifications made to the algorithm?

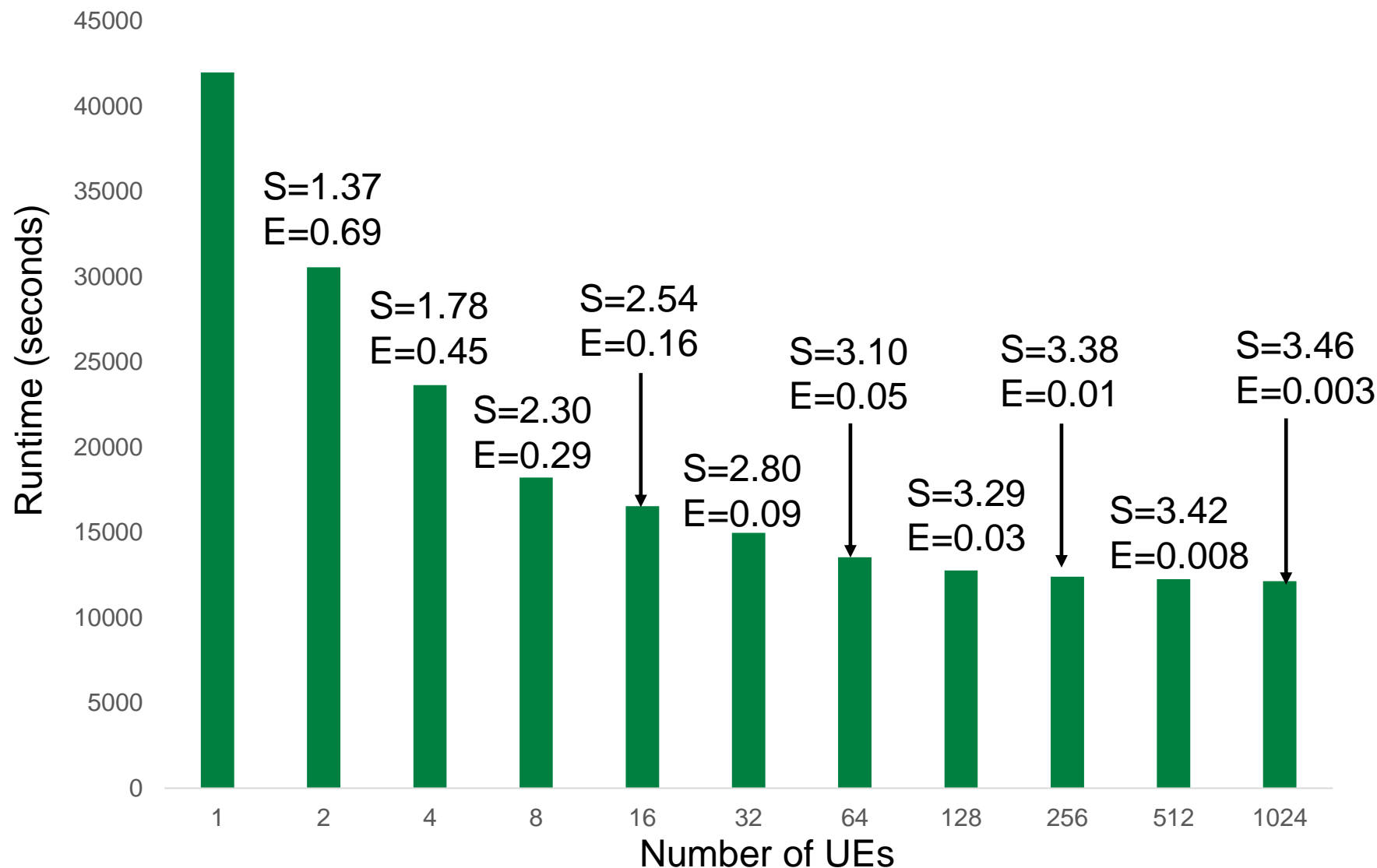
- Model for predicting the geomagnetic field lines of the earth
  - Ran in parallel but very limited scalability and wanted to do more science with this model



- Model for predicting the geomagnetic field lines of the earth
  - Ran in parallel but very limited scalability and wanted to do more science with this model







- A fraction,  $\alpha$ , is completely serial

- Parallel runtime
$$T(N, P) = \alpha T(N, 1) + \frac{(1-\alpha)T(N, 1)}{P}$$
  - Assuming parallel part is 100% efficient

- Parallel speedup
$$S(N, P) = \frac{T(N, 1)}{T(N, P)} = \frac{P}{\alpha P + (1 - \alpha)}$$

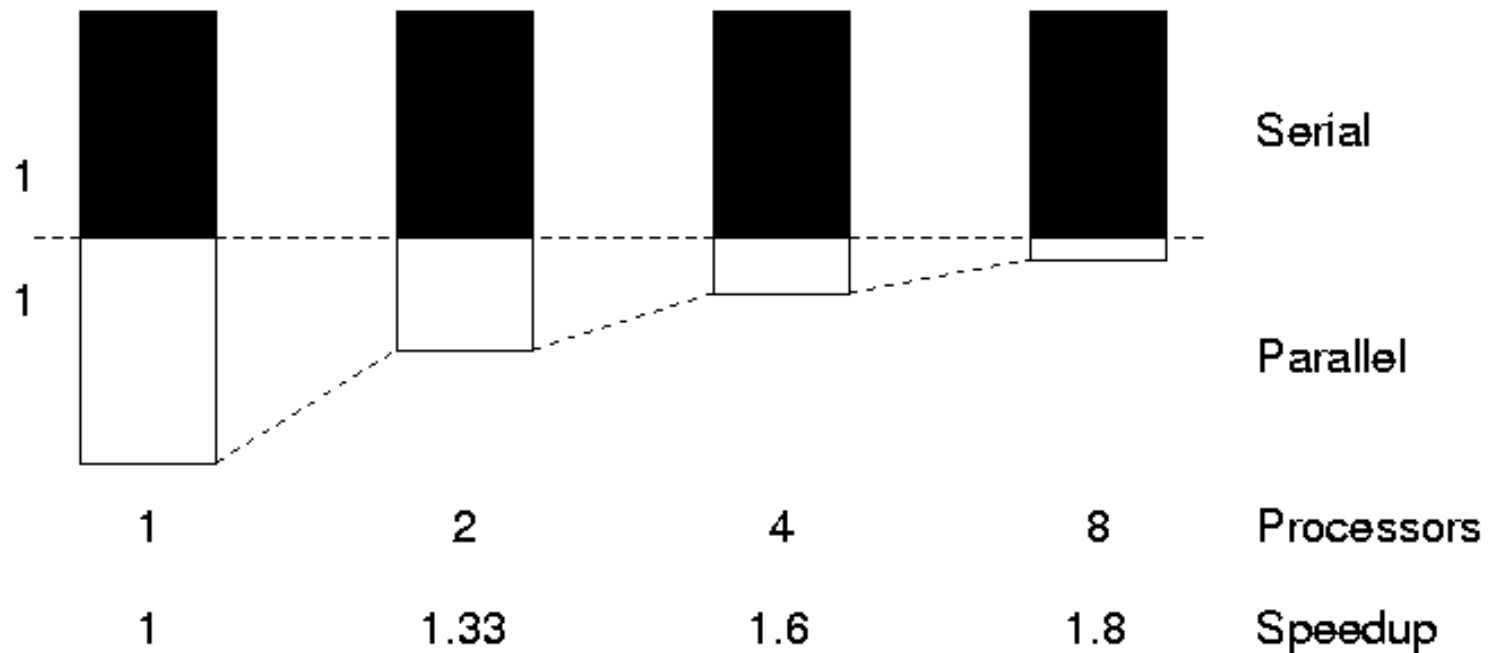
- We are fundamentally limited by the serial fraction
  - For  $\alpha = 0$ ,  $S = P$  as expected (i.e. *efficiency* = 100%)
  - Otherwise, speedup limited by  $1/\alpha$  for any  $P$ 
    - For  $\alpha = 0.1$ ;  $1/0.1 = 10$  therefore 10 times maximum speed up
    - For  $\alpha = 0.1$ ;  $S(N, 16) = 6.4$ ,  $S(N, 1024) = 9.9$



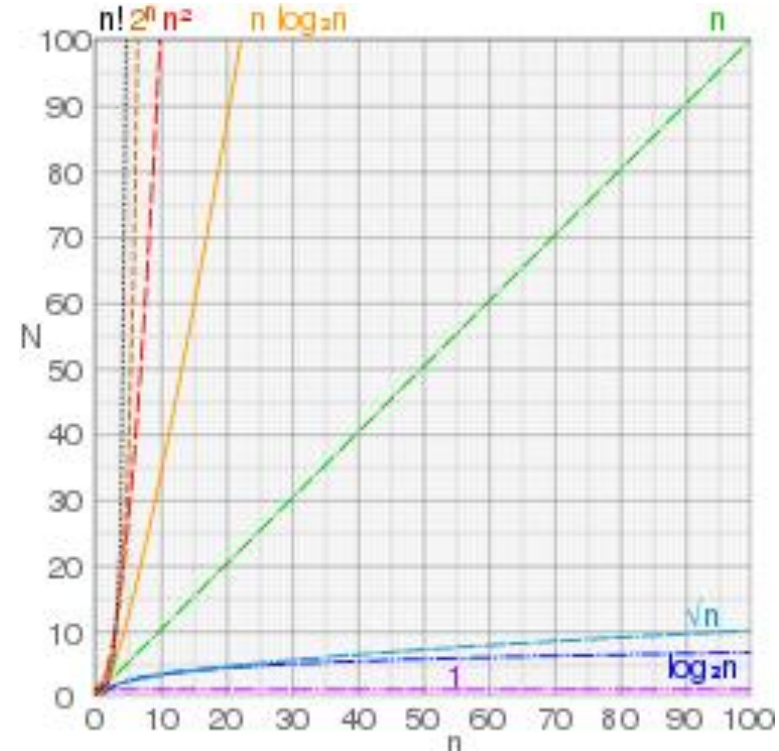
# The serial section of code

*“The performance improvement to be gained by parallelisation is limited by the proportion of the code which is serial”*

*Gene Amdahl, 1967*



- The rough growth rate of resources (specifically runtime) with respect to the input size
- Estimated by counting the number of elementary operations the algorithm is required to perform
  - i.e.  $8n + 12n^2$  where  $n$  is the number of input elements
  - The worst case time complexity is most commonly used and here it would be  $O(n^2)$
- Provides a way to evaluate and compare sequential algorithms



# Examples

```
for (i=0;i<50;i++) {  
    result=result+a[0]  
}
```

$$50 * (2 + 3 + 1) = O(1)$$

```
for (i=0;i<n;i++) {  
    result=result+a[i]  
}
```

$$n * (2 + 3 + 1) = O(n)$$

```
for (i=0;i<n;i++) {  
    for (j=0;j<n;j++) {  
        result=result+a[i]  
    }  
}
```

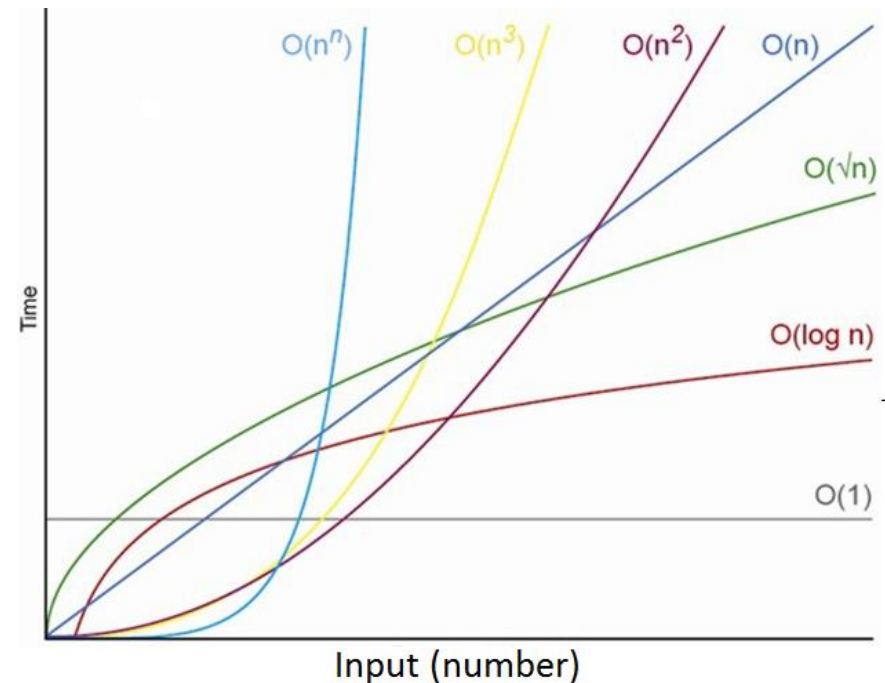
$$n * (2 + n * (2 + 3 + 1)) = O(n*n) = O(n^2)$$

- Remember, we are concerned with how the runtime grows as a function of the input size ( $n$ )



# But theory != practice

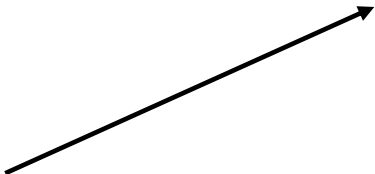
- This can hide important aspects
  - $10N + N \log N = O(N \log N)$  but the  $10N$  term is most important for small  $N$
  - $1000 * N \log N = O(N \log N)$  is “better” than  $10 * N^2 = O(N^2)$  but the later is more performant for small  $N$
- The hidden constant can be a significant factor in the algorithm performance
- But it does provide a general starting point for evaluating algorithms
  - Which can be combined with profiling




- In parallel we have both the computation and parallelism to consider, for the parallel worst case running time:

$$t(n,p)=C(n,p) + R(n,p)$$

Worst case growth of local computational runtime based on input size & number of UEs



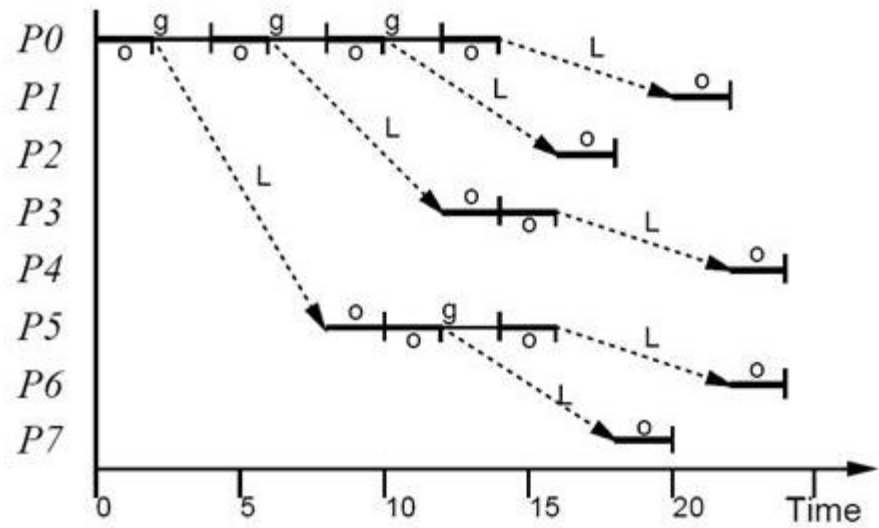
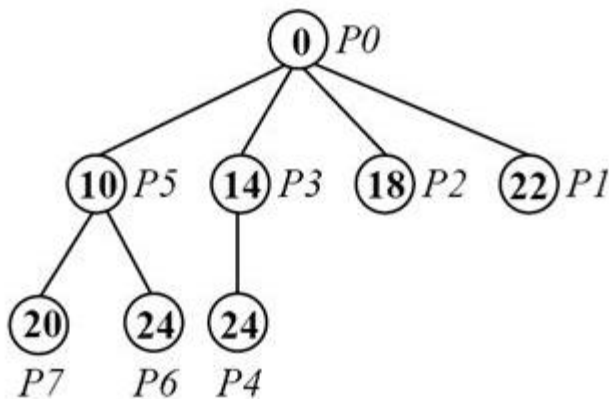
Worst case growth of local routing (parallel overhead) runtime based on input size & number of UEs.



- As we scale the data size,  $n$  and number of UEs,  $p$ , what impact will this likely have on the overall runtime?
- These can trade off against each other, i.e. higher routing complexity for lower computational complexity

# Routing time complexity

- A number of different ways of modelling this
- Log P is one common approach in the literature
  - L is the latency of the communication medium (cycles)
  - o is the overhead of sending and receiving messages (cycles)
  - g is the gap required between messages due to bandwidth limitations (cycles)
  - P is the number of UEs



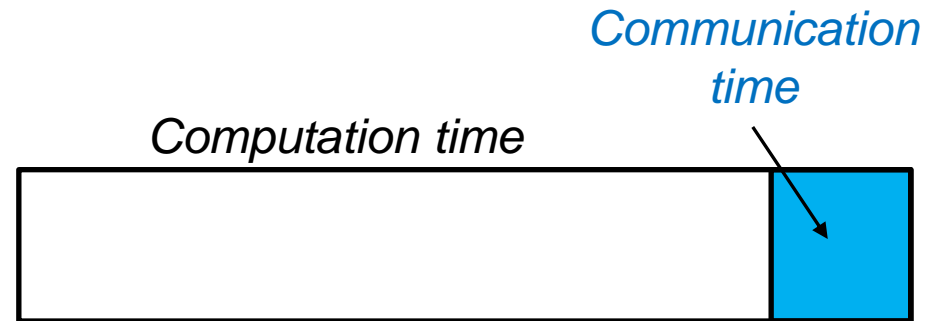
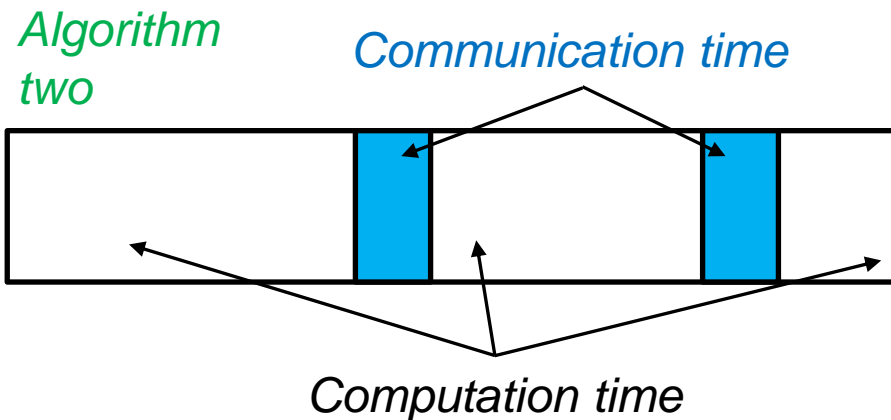
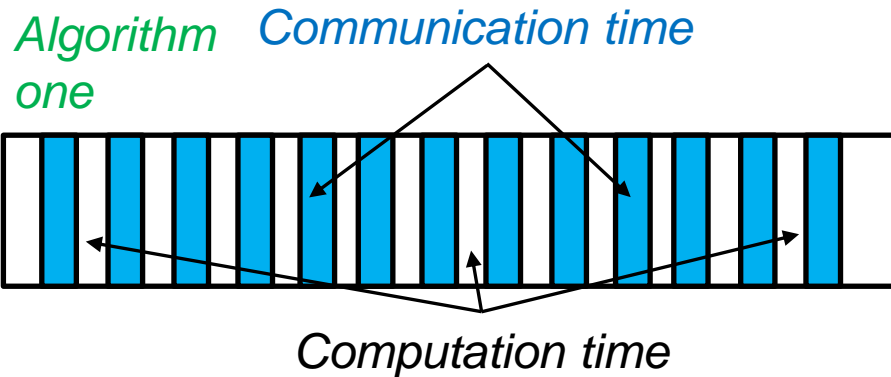
Example taken from <http://slideplayer.com/slide/8123828/> where  $P=8$ ,  $L=6$ ,  $g=4$ ,  $o=2$



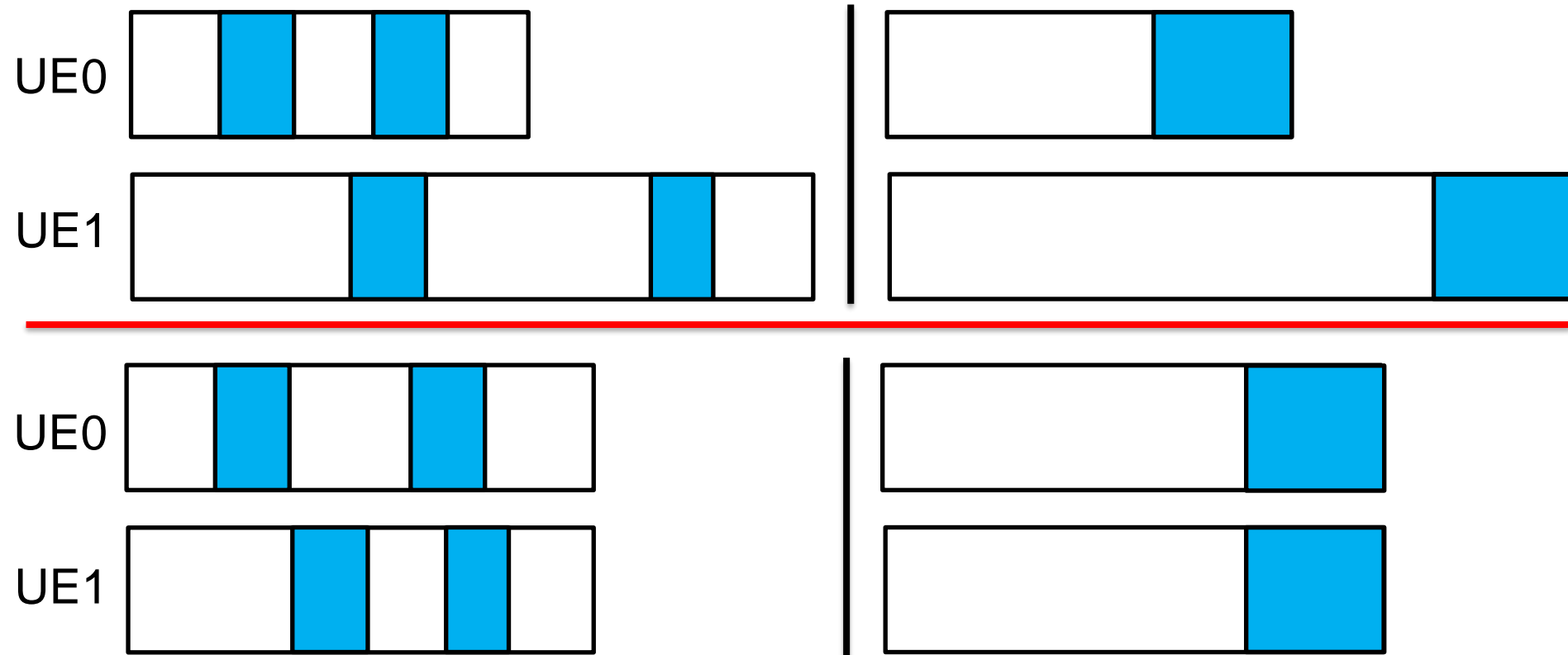
# And many other options...

- Communicating Sequential Processes (CSP)
  - <http://www.usingcsp.com/>
- BSP
  - [http://www.staff.science.uu.nl/~bisse101/Book/PSC/psc1\\_2.pdf](http://www.staff.science.uu.nl/~bisse101/Book/PSC/psc1_2.pdf)
- PRAM
  - <https://www.cs.fsu.edu/~engelen/courses/HPC-adv-2008/PRAM.pdf>
- LogP
  - [https://hlor.inf.ethz.ch/teaching/CS498/hoeftler\\_cs498\\_lecture\\_4.pdf](https://hlor.inf.ethz.ch/teaching/CS498/hoeftler_cs498_lecture_4.pdf)
- Circuits
- Functional models

## 1. Communication to computation ratio



## 2. Load balance between processes

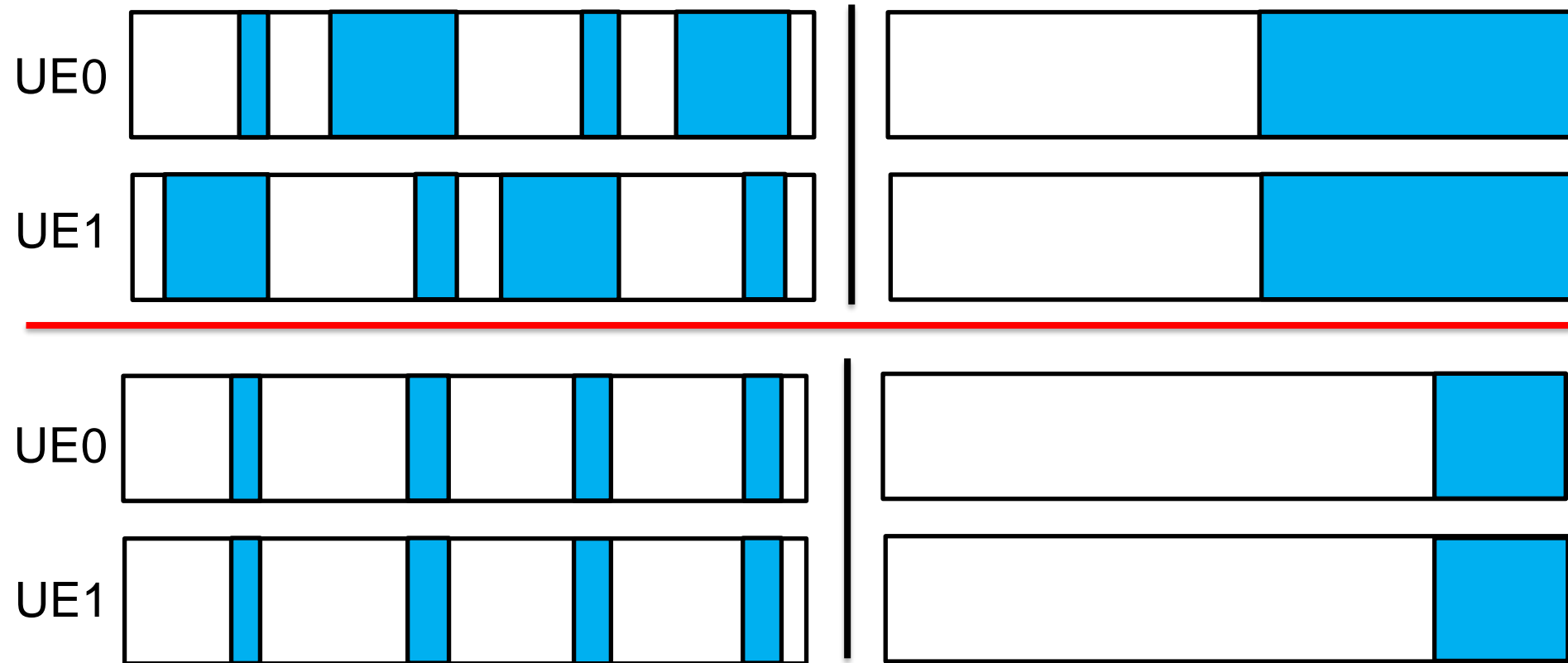


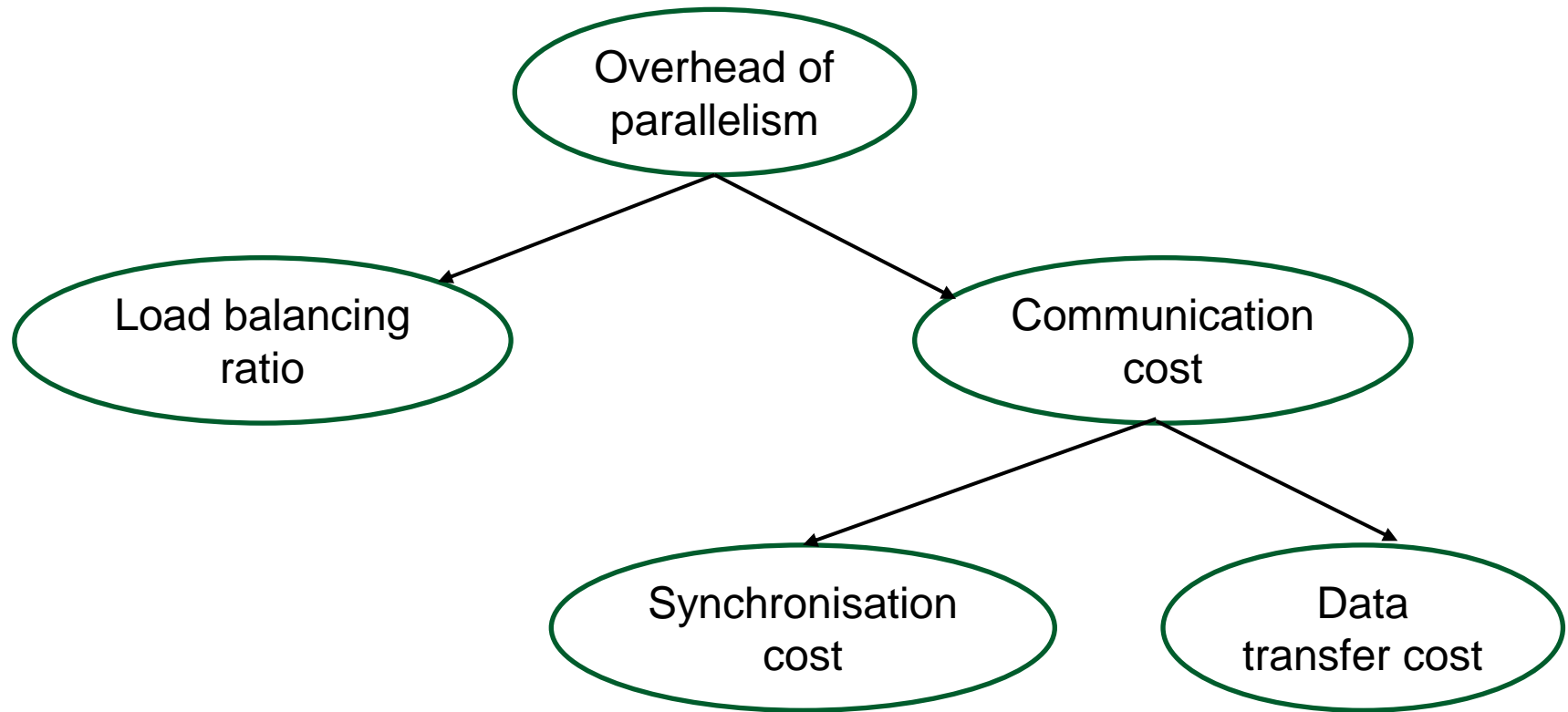
*Load Imbalance Factor (LIF):*  
*maximum load / average load*

*Where 1 is ideal*



## 3. Synchronisation costs





- Needs to be balanced against the computational complexity
- Need to consider code maintainability

# Can be obvious from code

```
if (rank == 0) {  
    for (i=0;i<1000;i++) {  
        a[i]=i;  
    }  
    send a to rank 1  
    b=recv from rank 1  
}
```

```
if (rank == 1) {  
    b=recv from rank 0  
    for (i=0;i<100;i++) {  
        a[i]=i+1000;  
    }  
    send a to rank 0  
}
```

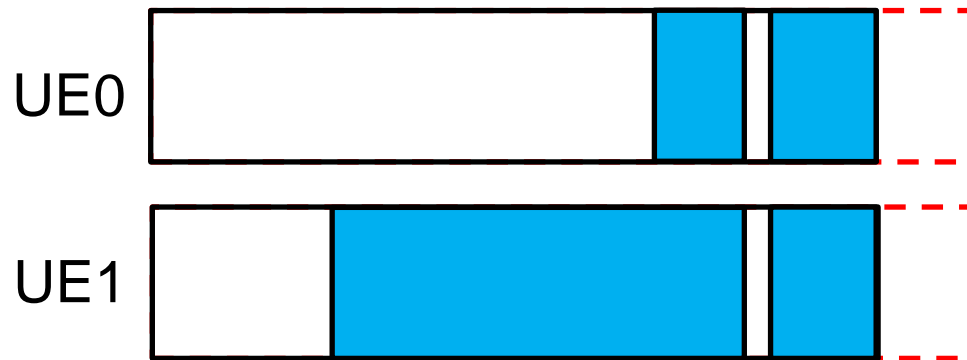




# A slight improvement....

```
if (rank == 0) {  
    for (i=0;i<1000;i++) {  
        a[i]=i;  
    }  
    send a to rank 1  
    b=recv from rank 1  
}
```

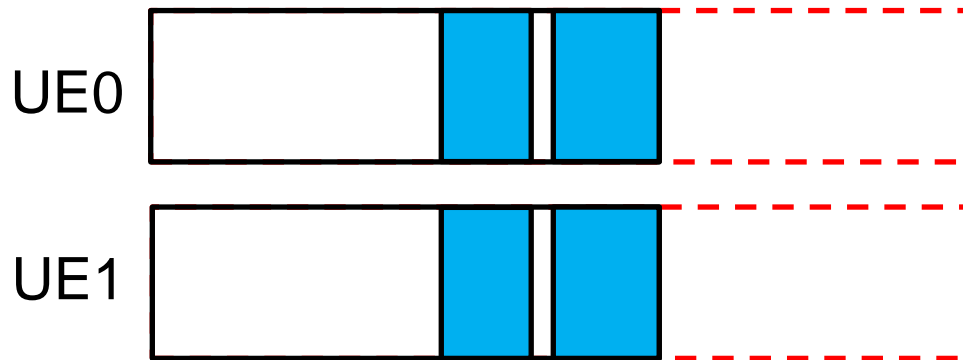
```
if (rank == 1) {  
    for (i=0;i<100;i++) {  
        a[i]=i+1000;  
    }  
    b=recv from rank 0  
    send a to rank 0  
}
```



# More of an improvement....

```
if (rank == 0) {  
    for (i=0;i<550;i++) {  
        a[i]=i;  
    }  
    send a to rank 1  
    b=recv from rank 1  
}
```

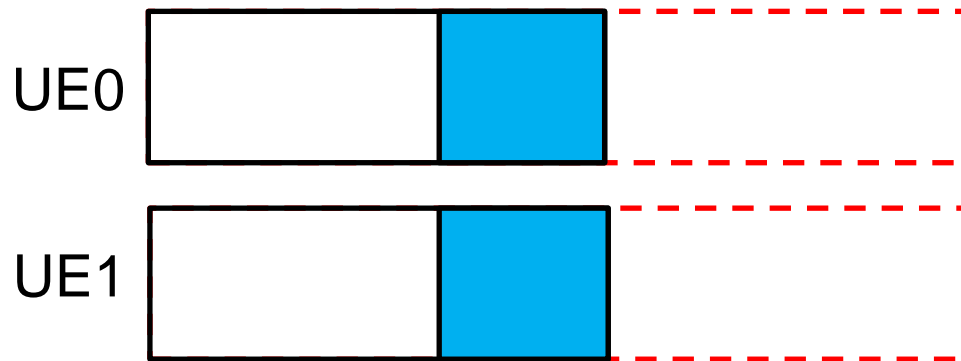
```
if (rank == 1) {  
    for (i=0;i<550;i++) {  
        a[i]=i+550;  
    }  
    b=recv from rank 0  
    send a to rank 0  
}
```



# Potentially even better

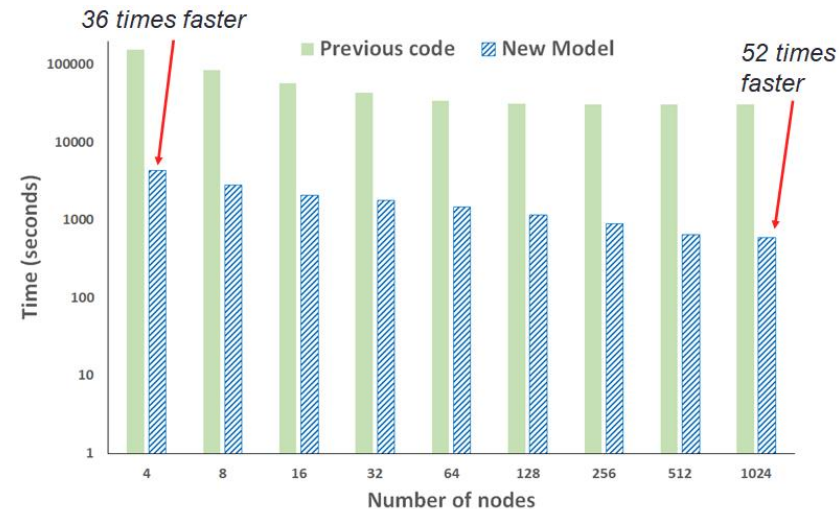
```
if (rank == 0) {  
    b=nonblocking recv from rank 1  
    for (i=0;i<550;i++) {  
        a[i]=i;  
    }  
    nonblocking send a to rank 1  
    wait on all comms  
}
```

```
if (rank == 1) {  
    b=nonblocking recv from rank 0  
    for (i=0;i<550;i++) {  
        a[i]=i+550;  
    }  
    nonblocking send a to rank 0  
    wait on all comms  
}
```



# Still requires expertise.....

- Algorithm analysis is not simple and requires programmer insight & reasoning
  - There are very many contributing factors
    - Machine specific factors
    - Compiler optimisations
    - Underlying libraries and runtime
    - Current network traffic
    - The time of day and year
- Theoretical ways can be unwieldy in practice, so often intuition is needed
  - Consider what the overhead of parallelism is and how to reduce it





#### Finding Concurrency

- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, ...

#### Algorithm Structure

- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, ...

#### Supporting Structures

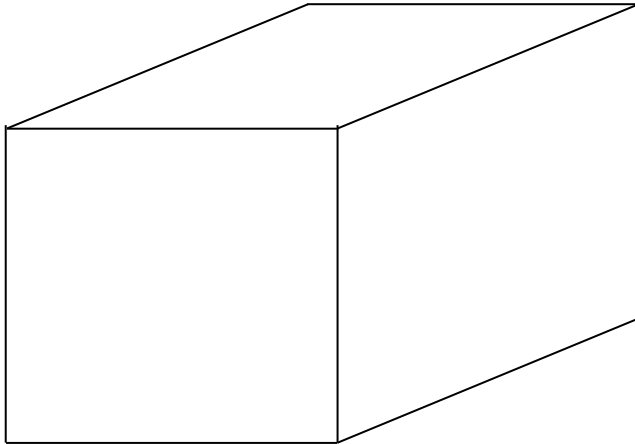
- SPMD, Master/Worker, Loop Parallelism, Fork/Join, ...

#### Implementation Mechanisms

- UE Management, Synchronisation, Communication, ...

Strategy:

# FINDING CONCURRENCY



- Your task is to parallelise a problem
  - You know about HPC but little about the problem domain
  - Those who know about the problem domain know little about HPC
  - Where do we start?
- At this point we are miles away from the implementation
  - Is it even possible or worth parallelising the problem?
  - If so we are trying to understand how we might split the problem up
- What bits are independent, can we group any of these together and how well will this work?

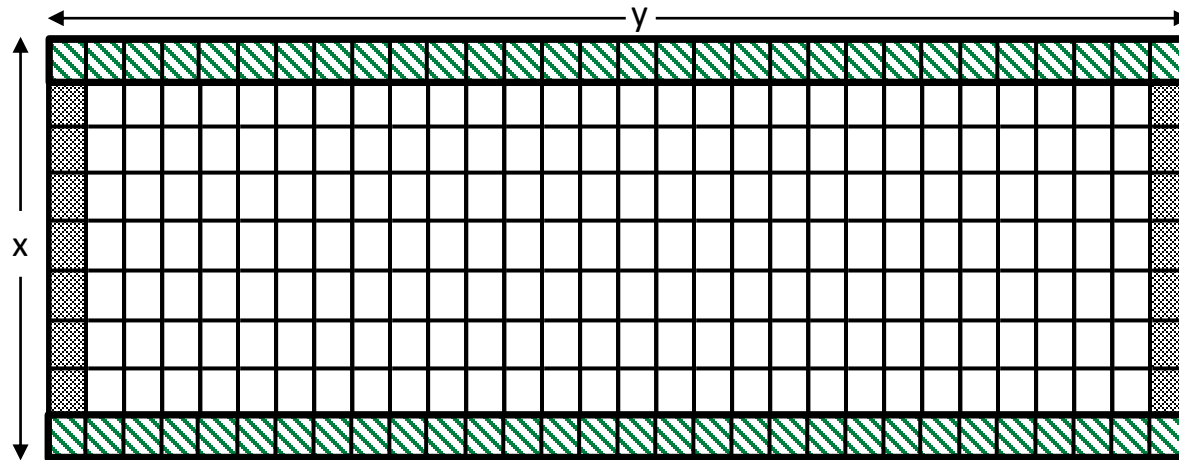
- How can we break the problem into distinct tasks that operate independently?

Tightly  
coupled  
problem

Embarrassingly  
parallel  
problem

- Split the problem up based upon its functionality
  - Define the tasks and data decomposition implied by these
  - Sometimes independent tasks are easily identified
    - Calls to a function
    - Independent iteration of a loop
    - A number of independent activities being performed
- *cat datafile | grep “energy” | awk ‘{print \$2, \$3}’*

- Split the problem up based upon the data it is operating on
  - If it is difficult to split the problem into distinct tasks then instead concentrate on the data is manipulates - especially if this is the most computationally intensive part.
  - E.g. **Arrays**: Concurrency can be defined in terms of updates to different segments of an array which might be decomposed in a variety of different ways.



- $4(y-2*x-2)+2(x-2)$  data oriented tasks: Jacobi iteration for  $y-2*x-2$ , residual of solution for  $y-2*x-2$ , initial residual calculation for  $y-2*x-2$ , boundary condition at left  $(x-2)$  and right  $(x-2)$ , initial guess for location  $y-2*x-2$



# Our starting point: serial practical code

```
double * u_k = malloc(sizeof(double) * x * y);
```

```
.....
```

```
initialise(u_k, u_kp1);
```

```
double rnorm=0.0, bnorm=0.0, norm;
```

```
int i, j, k;
```

```
for (i=1;i<x-1;i++) {
```

```
    for (j=1;j<y-1;j++) {
```

```
        bnorm=bnorm+.....
```

```
    }
```

```
}
```

```
bnorm=sqrt(bnorm);
```

```
for (k=0;k<MAX_ITERATIONS;k++) {
```

```
    for (i=1;i<x-1;i++) {
```

```
        for (j=1;j<y-1;j++) {
```

```
            rnorm=rnorm+.....
```

```
        }
```

```
    }
```

```
norm=sqrt(rnorm)/bnorm;
```

```
if (norm < CONVERGENCE_ACCURACY) break;
```

```
for (i=1;i<x-1;i++) {
```

```
    for (j=1;j<y-1;j++) {
```

```
        u_kp1[i]=0.25 * .....
```

```
    }
```

```
}
```

```
temp=u_kp1; u_kp1=u_k; u_k=temp;
```

```
rnorm=0.0;
```

```
}
```

*Compute the  
initial absolute  
residual*

*Compute the  
absolute residual  
of the current  
solution*

*Jacobi  
iteration to  
progress  
the solution*

```
void initialise(double * u_k, double * u_kp1) {
```

```
    int i,j;
```

```
    for (i=1;i<x-1;i++) {
```

```
        u_k[i*y]=LEFT_VALUE;
```

```
        u_kp1[i*y]=LEFT_VALUE;
```

```
    }
```

```
    for (i=1;i<x-1;i++) {
```

```
        u_k[y-1+(i*y)]=RIGHT_VALUE;
```

```
        u_kp1[y-1+(i*y)]=RIGHT_VALUE;
```

```
    }
```

```
    for (i=1;i<x-1;i++) {
```

```
        for (j=1;j<y-1;j++) {
```

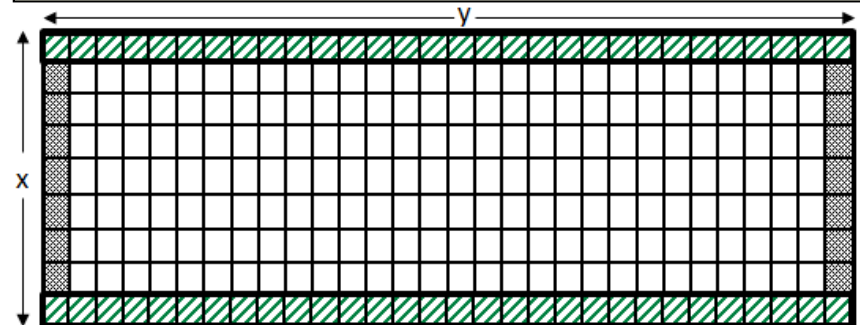
```
            u_k[j+(i*y)]=0.0;
```

```
            u_kp1[j+(i*y)]=0.0;
```

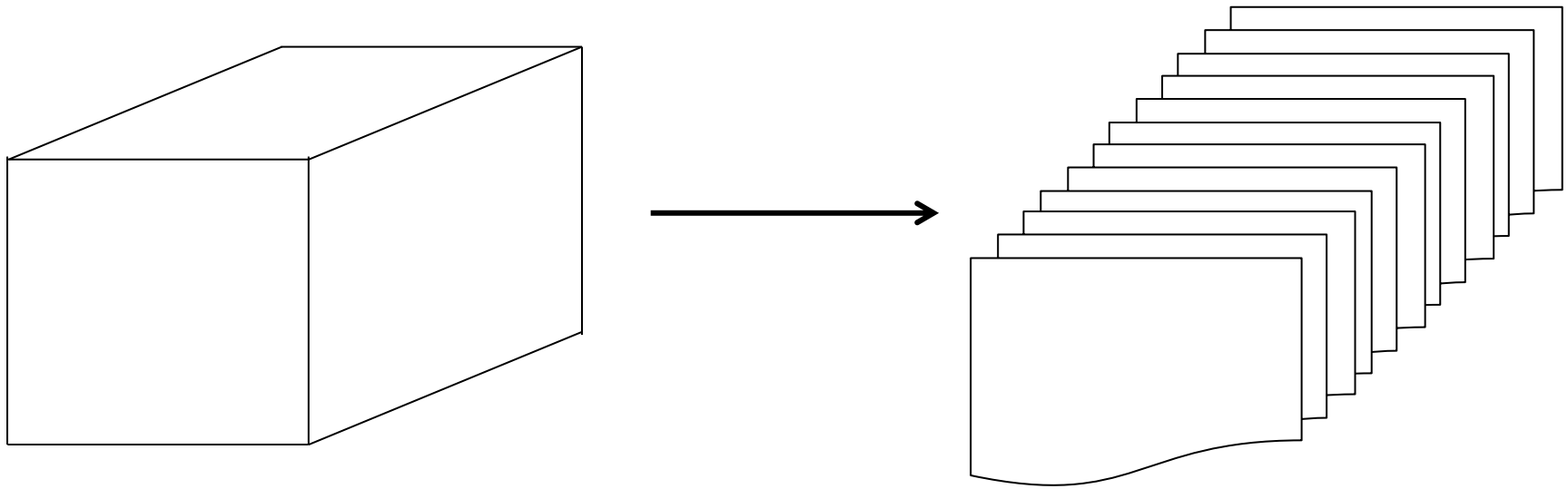
```
        }
```

```
    }
```

*Sets the pollution values at each end of the  
pipe and the rest to be zero (initial guess)*



**$4(y-2)x-2+2(x-2)$  data oriented tasks:** Jacobi iteration for  $y-2 \times x-2$ , residual of solution for  $y-2 \times x-2$ , initial residual calculation for  $y-2 \times x-2$ , boundary condition at left ( $x-2$ ) and right ( $x-2$ ), initial guess for location  $y-2 \times x-2$



- We have now split the problem into lots of primitive tasks
  - How can we group these to simplify managing dependencies
  - Order the groups to ensure constraints are met
  - Share any necessary data between the groups of tasks

- Gives structure to the set of tasks based on constraints
  - In terms of correctness, every task in a group can run concurrently
  
- 1. How did we decompose the original problem?
  - A high level operation or structure (e.g. a loop) often results in a group of tasks
  
- 2. Do tasks share a constraint?
  - Such as a file to be in a specific state before operation?
  
- 3. Can we merge groups?
  - The larger the task group the better (provide more concurrency)

## $4(y-2 \cdot x-2)+2(x-2)$ tasks

Jacobi iteration for 1,1

....

Jacobi iteration for y-1,x-1

Residual of solution for 1,1

....

Residual of solution for y-1,x-1

Initial residual calculation for 1,1

....

Initial residual calculation for y-1,x-1

Boundary condition left 1 to x-1

Boundary condition right 1 to x-1

Initial guess for 1,1

....

Initial guess for y-1,x-1



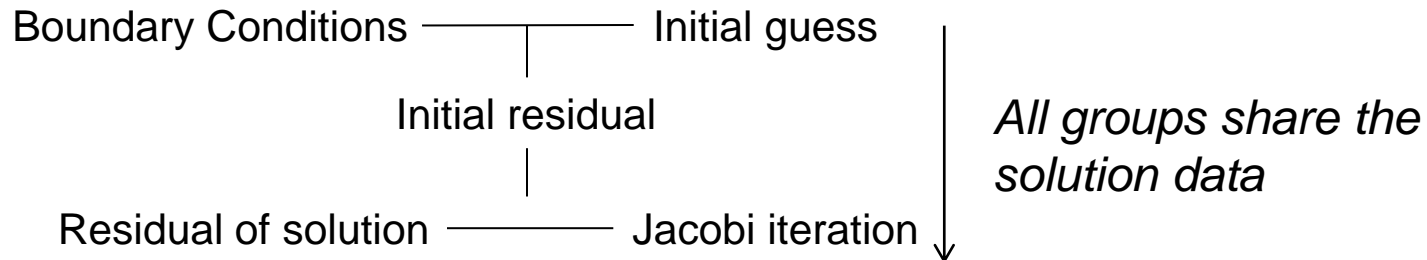
## 5 groups

Jacobi iteration (contains  $y-2 \cdot x-2$  tasks)  
Residual of solution (contains  $y-2 \cdot x-2$  tasks)  
Initial residual (contains  $y-2 \cdot x-2$  tasks)  
Boundary conditions (contains  $2 \cdot x-2$  tasks)  
Initial guess (contains  $y-2 \cdot x-2$  tasks)

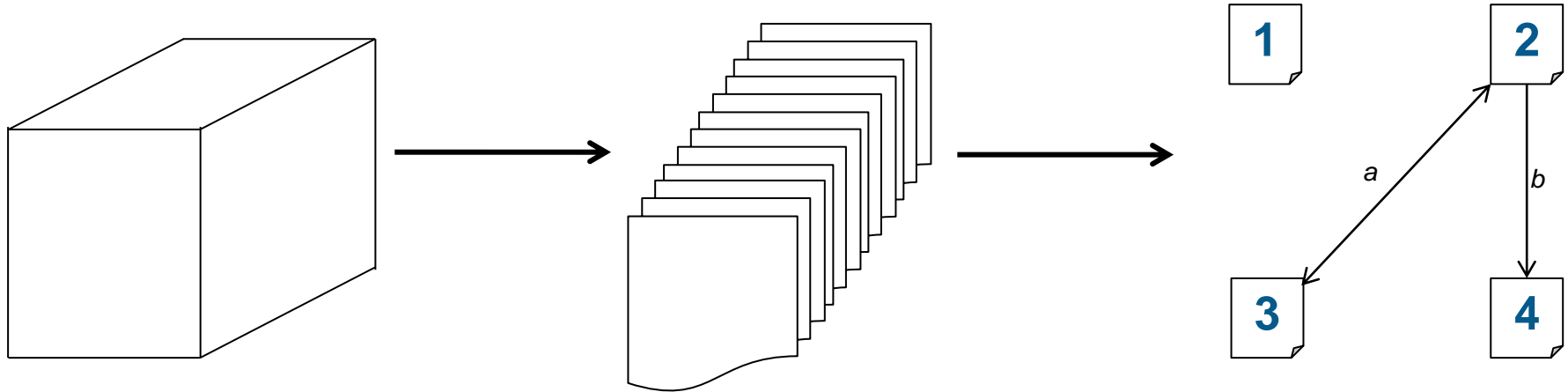


- Find and account for dependencies resulting from the constraints on the order of execution of groups of tasks
  - Needs to be restrictive enough to satisfy all constraints but no more.
- 1. Look at the data required by a group before it can execute
  - Find the task group that creates this to form a constraint
- 2. Can external services impose constraints
  - For instance if a program must write to a file in a specific order
- 3. Note when an ordering does not exist
  - This is equally important as if groups of tasks can execute independently then there is an opportunity for increased parallelism.

- Find and account for dependencies resulting from the constraints on the order of execution of groups of tasks
  - Needs to be restrictive enough to satisfy all constraints but no more.



- Distinctly identify task local and shared (global) data
  - Task groups might define some global data that must be shared
  - Some task might need access to a portion of another task's data
  - How we deal with shared data impacts the correctness (whether it produces the correct result) and performance (not waiting excessively in synchronisation calls and/or reducing communication overhead.)
- Broadly falls into categories of
  - *Read only*: Because there is no modification no protection is needed
  - *Effectively local*: Partitioned into subsets, each accessed by 1 task
  - *Read-write*: The general case and most difficult to deal with
  - *Accumulate*: Updated by many tasks with some operation (eg. Sum)
- Jacobi makes life easier as it separates the previous iteration values ( $u_k$ ) and the current values being computed ( $u_{k+1}$ )



- Now we have our ordered groups of tasks we need to evaluate these
  - You can think of the steps so far as refining the problem to guide your work in the next stage.
  - But are these ordered groups good enough to move onto the next overall strategy (Algorithm strategy)? Will they give us enough information to work with?

- Often there are multiple approaches possible
  - Evaluate the emerging design and ensure that it is appropriate
  - This strategy is an iterative process
- Design quality
  - Simplicity
  - Flexibility, efficiency
- Suitability for target platform
  - How many PEs are available, how is data shared, will the time spent doing useful work be significantly greater than managing the parallelism
- Preparation for the next stage
  - Are interactions between tasks synchronous or asynchronous, are tasks grouped in the best way possible?



- The analysis of parallel algorithms
  - Requires insight on behalf of the programmer
  - Combine theoretical measures with practical performance & scalability testing as early on as possible
  - Consider “**what is parallelism giving me here**” and “**what is the cost of parallelism**”?
  - Trading off  $C(n,p)$  vs  $R(n,p)$
- Finding concurrency
  - In the problem domain, working with the scientists and engineers who know their code
  - Rarely one obviously correct way, instead a more subtle and iterative approach to split the problem up and then evaluate.
  - Output are ordered groups of tasks that can be used as the basis for the next stage when we think about how the parallel algorithm should be organised to best take advantage of these tasks.