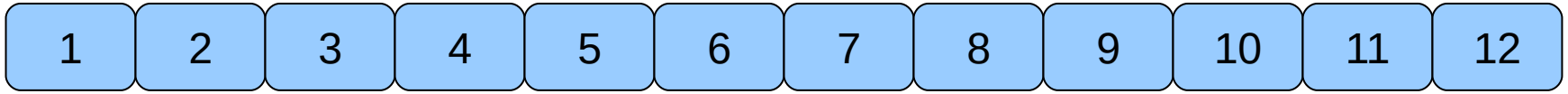
# Introduction to Parallel Computing

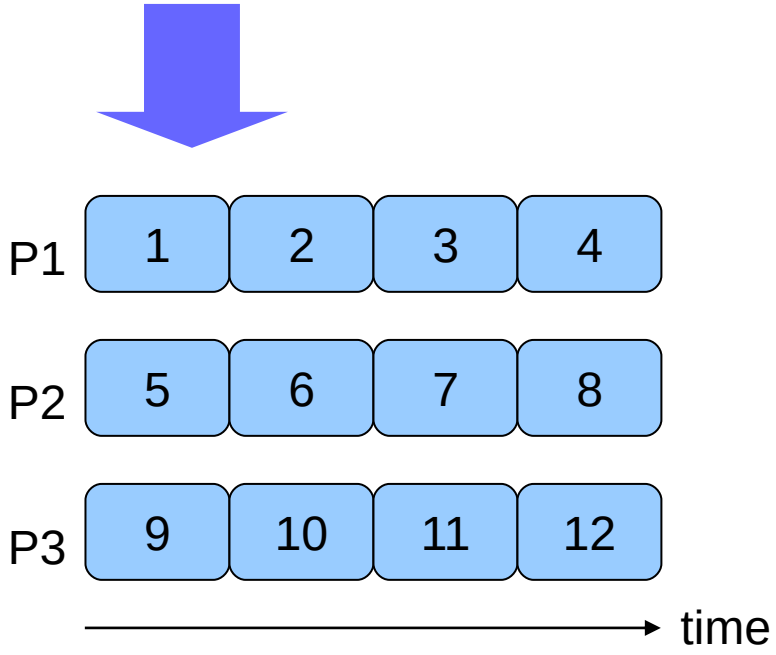
Main Concepts

# How faster can we run?

- 12 tasks, each one requiring 1s
- Total serial time: 12s



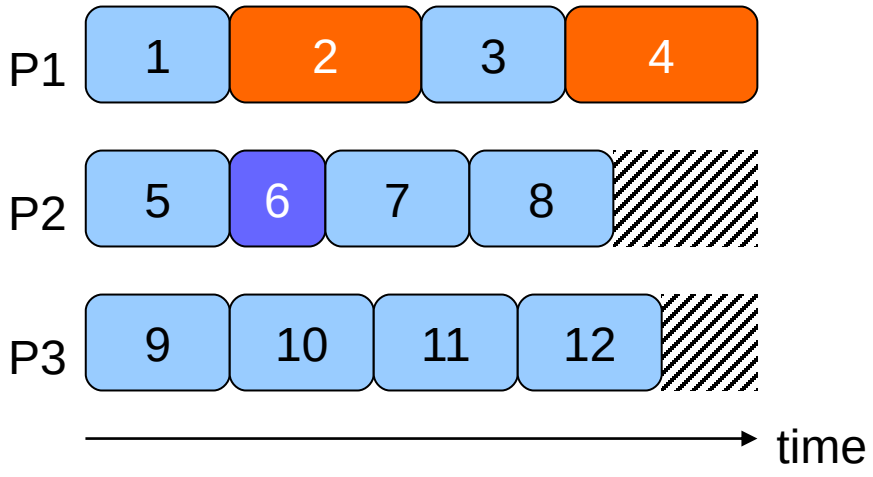

- If we have 3 processors and independent tasks execution time CAN become 4s



# But

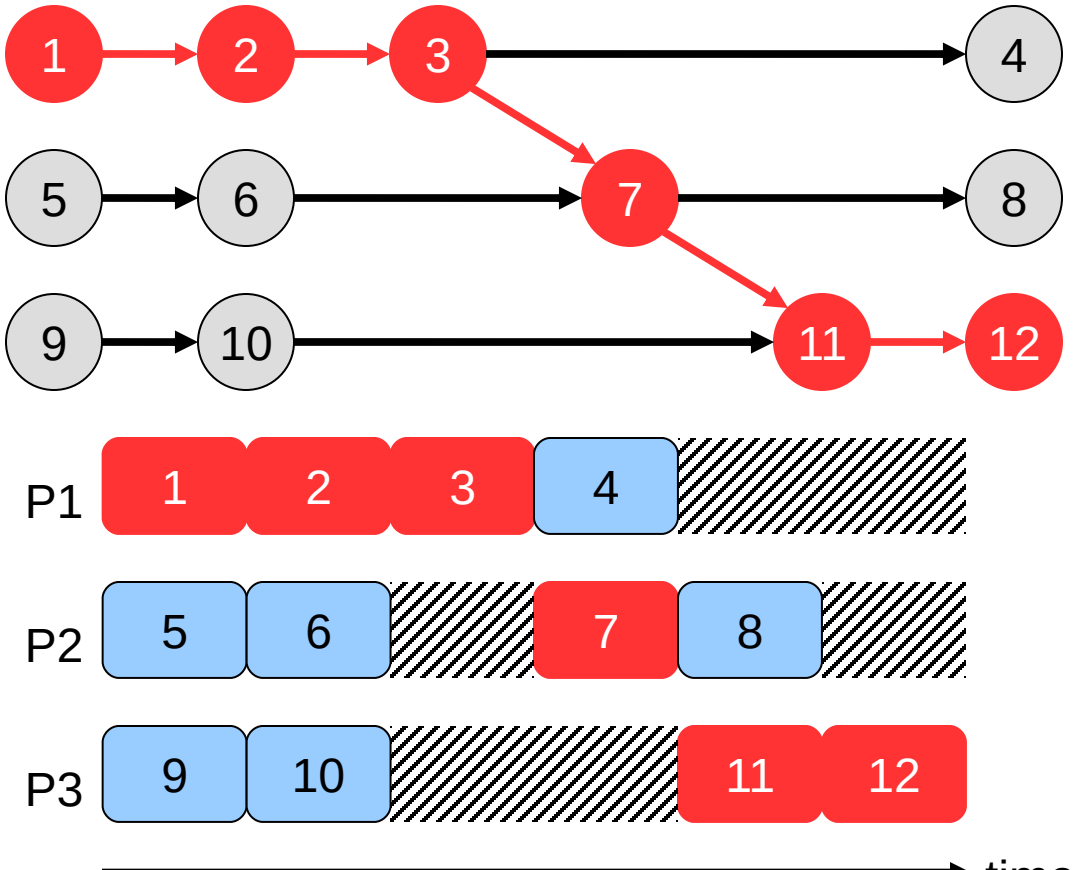
- What if the processors can not execute the tasks with the same speed? I.e., a heterogeneous cluster.
- Or the tasks do not require the same amount of operations?

➢Load imbalance (ending part of P2 and P3)



# And

- What if tasks are dependent?
- Execution time grows from 4s to 6s



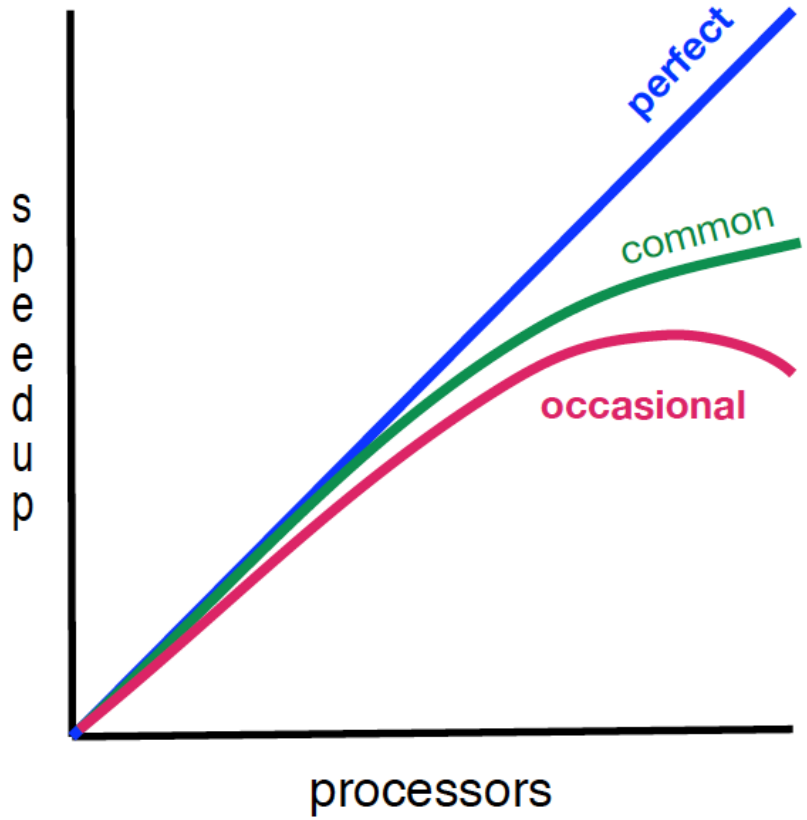

*Dependency graph with highlighted critical path*

# Scalability

- How much faster can a given problem be solved with p parallel processes instead of one?
- How much more work can be done with with p parallel processes instead of 1?
- What impact for the communication requirements of the parallel application have on performance?
- What fraction of the resources is actually used productively for solving the problem?

# Speedup

- Speedup $S_p = T_1/T_p$
- In the ideal case, the parallel program requires 1/p the time of the sequential program, i.e. $T_p = T_1/p$
- $S_p = p$ is the ideal case of linear speedup
- Realistically, $S_p \leq p$ due to the need of coordination the works and to communicate
- Overhead is the difference $T_o = pT_p - T_1$
- Is it possible to observe $S_p > p$ , i.e. a SUPERLINEAR speedup?
  - Yes, mainly for Memory vs Disk

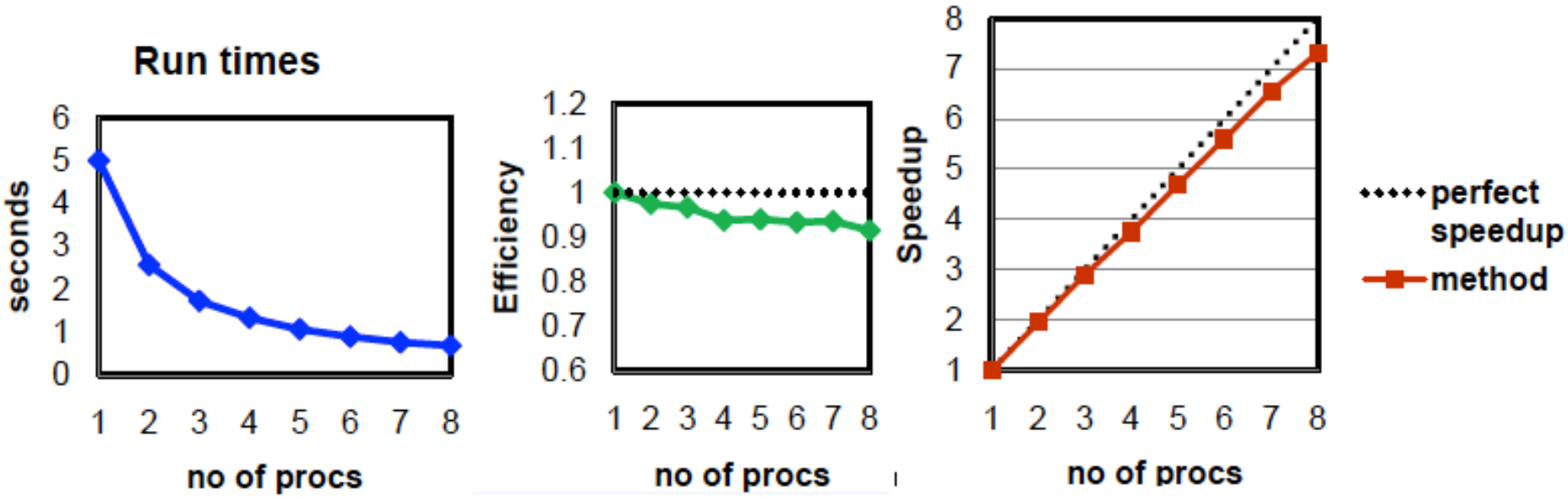


# Performance Metrics

The size of the problem is important too

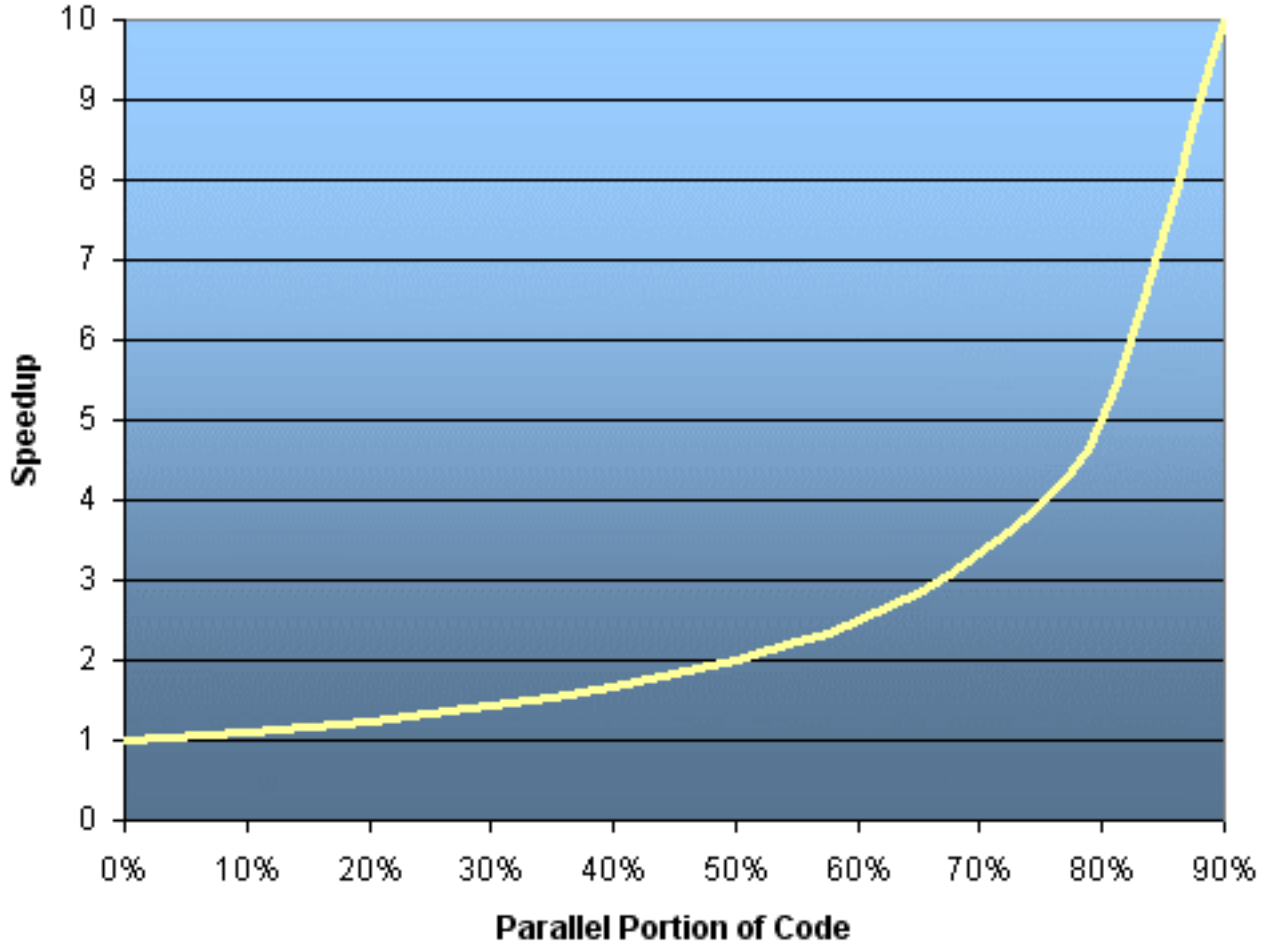
Let T(n,p) be the time to solve a problem of size n using p processors

- Speedup: S(n,p) = T(n,1)/T(n,p)
- Efficiency: E(n,p) = S(n,p)/p



# Amdahl's Law

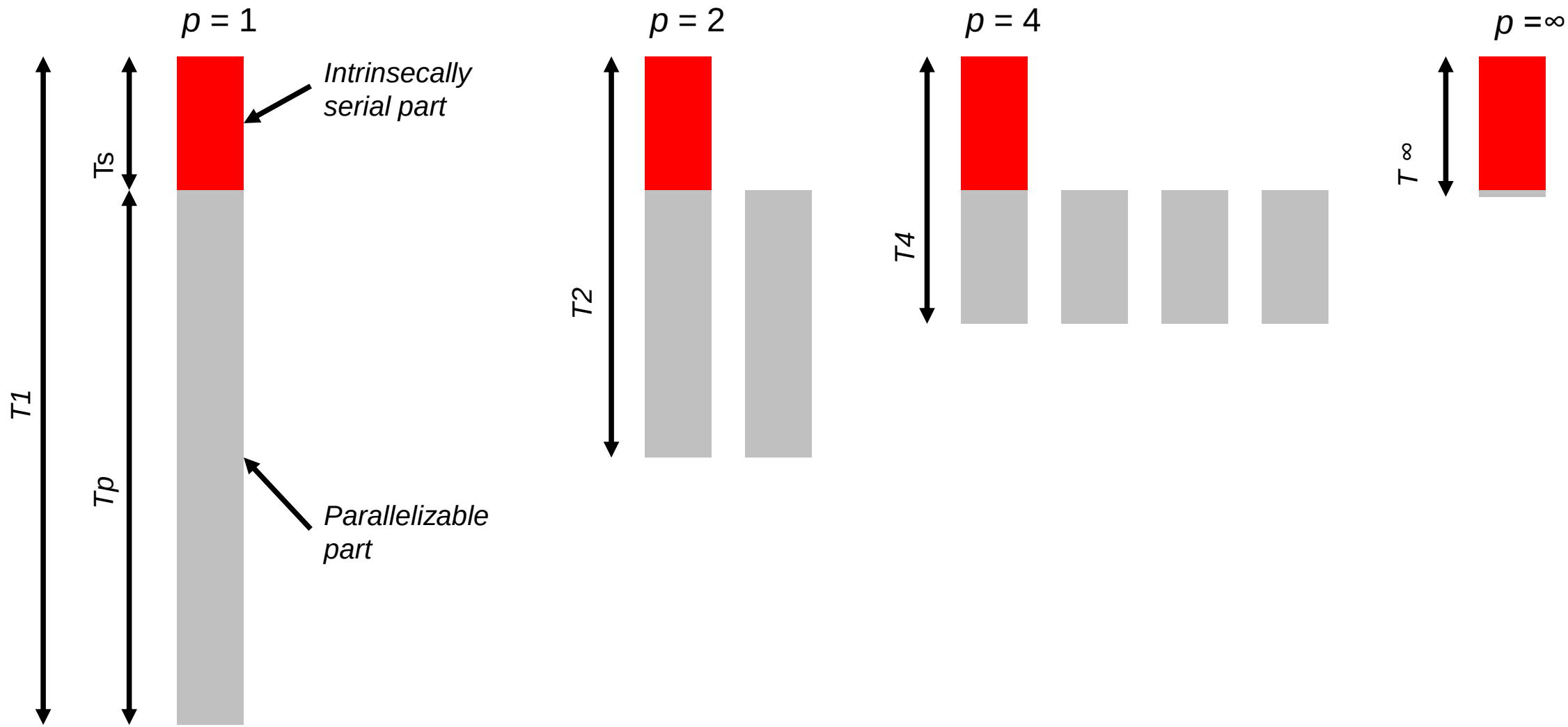
$F_p$ is the parallelizable fraction of the code, with $F_s+F_p = 1$




$$T_p \geq T_1(Fs + \frac{Fp}{p})$$

$$Speedup = \frac{1}{Fs + \frac{Fp}{p}}$$

# Example


p = 1
p = 2
p = 4
p =∞
Intrinsecally serial part
Parallelizable part
Ts
Tp
T1
T2
T4
T ∞

# Example

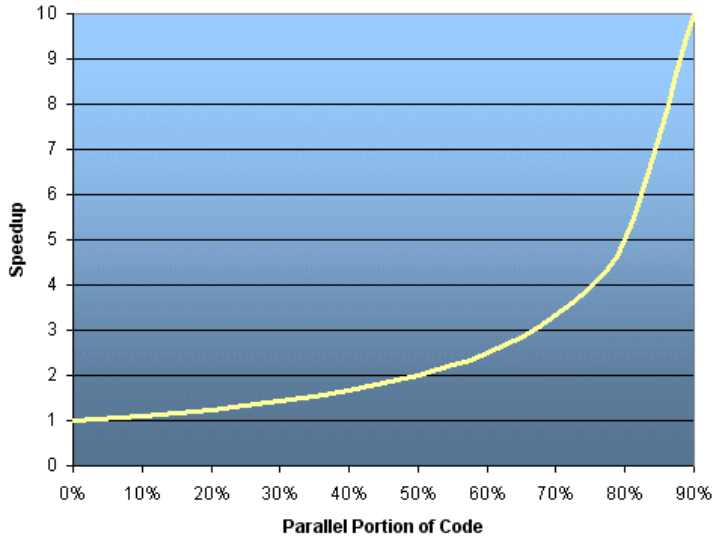
- Suppose that a program has $T_1$ = 20s
- Assume that 10% of the times spent in a serial portion of the program
- Therefore, the execution time of a parallel version with p processors is

$20*(0.1 + 0.9 / p)$

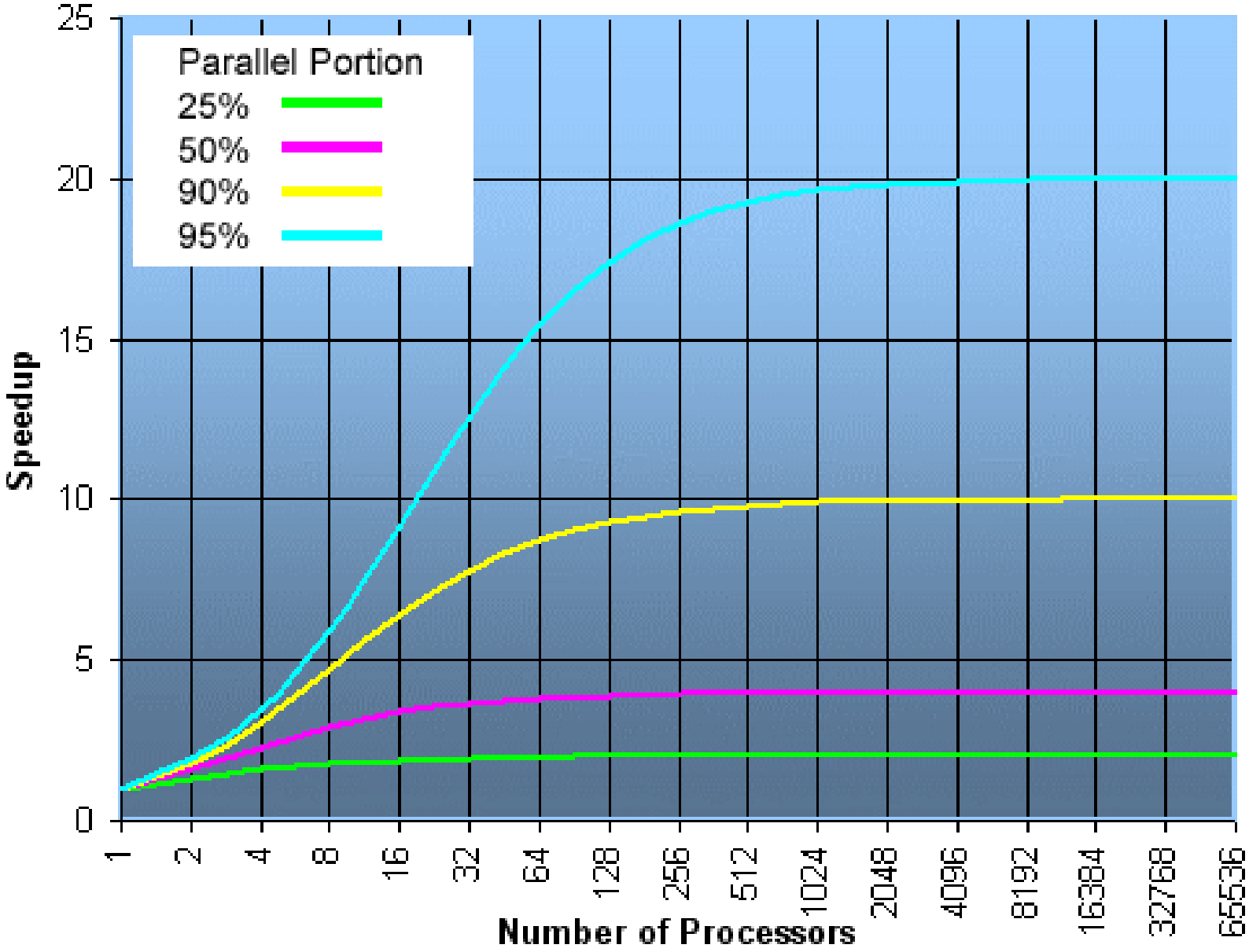
$20*0.1 + (20*0.9 / p) = 2 + 18/p$

$T_p >= 2$

Best possible speedup = 20/2 = 10



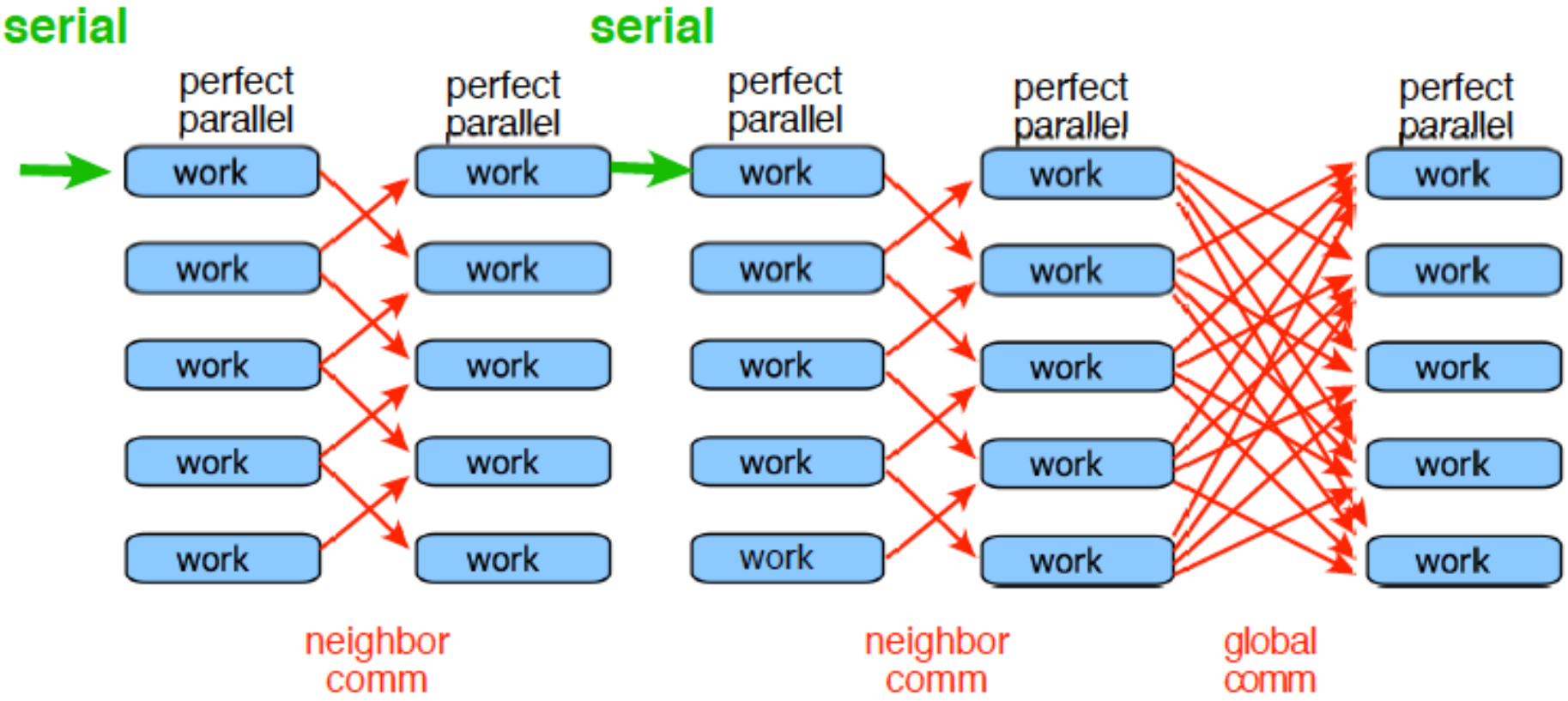
# Amdahl's Law


Parallel Portion
25%
50%
90%
95%
Speedup
0
5
10
15
20
25
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

# Amdahl Was an Optimist

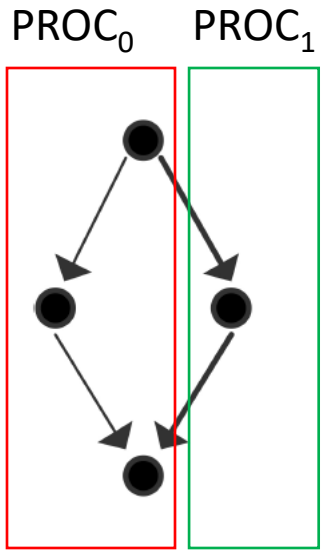
Parallelization usually adds communications/overheads

$$T_p \geq T_1(Fs + \frac{Fp}{p}) + T_{Ovh}$$



# Critical paths

We define the critical path as a (possibly non-unique) chain of dependencies of maximum length. Since the tasks on a critical path need to be executed one after another, the length of the critical path is a lower bound on parallel execution time.

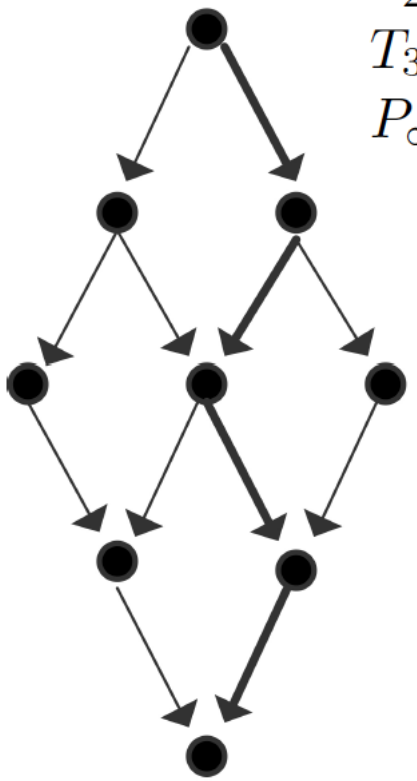



$T_1$ : *the time the computation takes on a single processor*
$T_p$ : *the time the computation takes with* $p$ *processors*
$T_\infty$ : *the time the computation takes if unlimited processors are available*
$P_\infty$ : *the value of* $p$ *for which* $T_p = T_\infty$

$$T_1 = 4, \quad T_\infty = 3 \quad \Rightarrow T_1/T_\infty = 4/3$$
$$T_2 = 3, \quad S_2 = 4/3, \quad E_2 = 2/3$$
$$P_\infty = 2$$

*Actually 1.333/3*

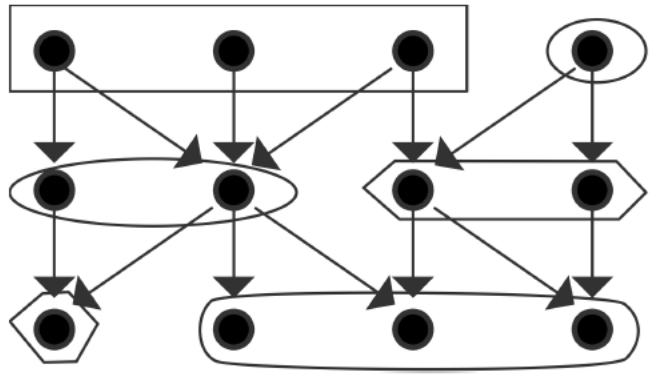
# Critical paths



$$T_1 = 9, \quad T_\infty = 5 \quad \Rightarrow T_1/T_\infty = 9/5$$
$$T_2 = 6, \quad S_2 = 3/2, \quad E_2 = 3/4$$
$$T_3 = 5, \quad S_3 = 9/5, \quad E_3 = 3/5$$
$$P_\infty = 3$$

$T_1$ : *the time the computation takes on a single processor*
$T_p$ : *the time the computation takes with* $p$ *processors*
$T_\infty$ : *the time the computation takes if unlimited processors are available*
$P_\infty$ : *the value of* $p$ *for which* $T_p = T_\infty$

$$T_1 = 12, \quad T_\infty = 4 \quad \Rightarrow T_1/T_\infty = 3$$
$$T_2 = 6, \quad S_2 = 2, \quad E_2 = 1$$
$$T_3 = 4, \quad S_3 = 3, \quad E_3 = 1$$
$$T_4 = 3, \quad S_4 = 4, \quad E_4 = 1$$
$$P_\infty = 4$$



# Amdahl was a Pessimist

Superlinear speedup is very rare but possible

- Parallel computer has p times as much RAM so higher fraction of program memory in RAM instead of disk.

*An important reason for using parallel computers*

- You can use more complex algorithms , where Fs is lower.

*A useful side-effect of parallelization*

- In general, the time spent in serial portion of code is a decreasing fraction of the total time as problem size increases.
- In the previous example Ts = 2, Tp = 18.
  What happens if $T_1$ = (2 + 180)? $F_p$ = 0.99, S = 1/0.01 = 100

# Scaling efficiency

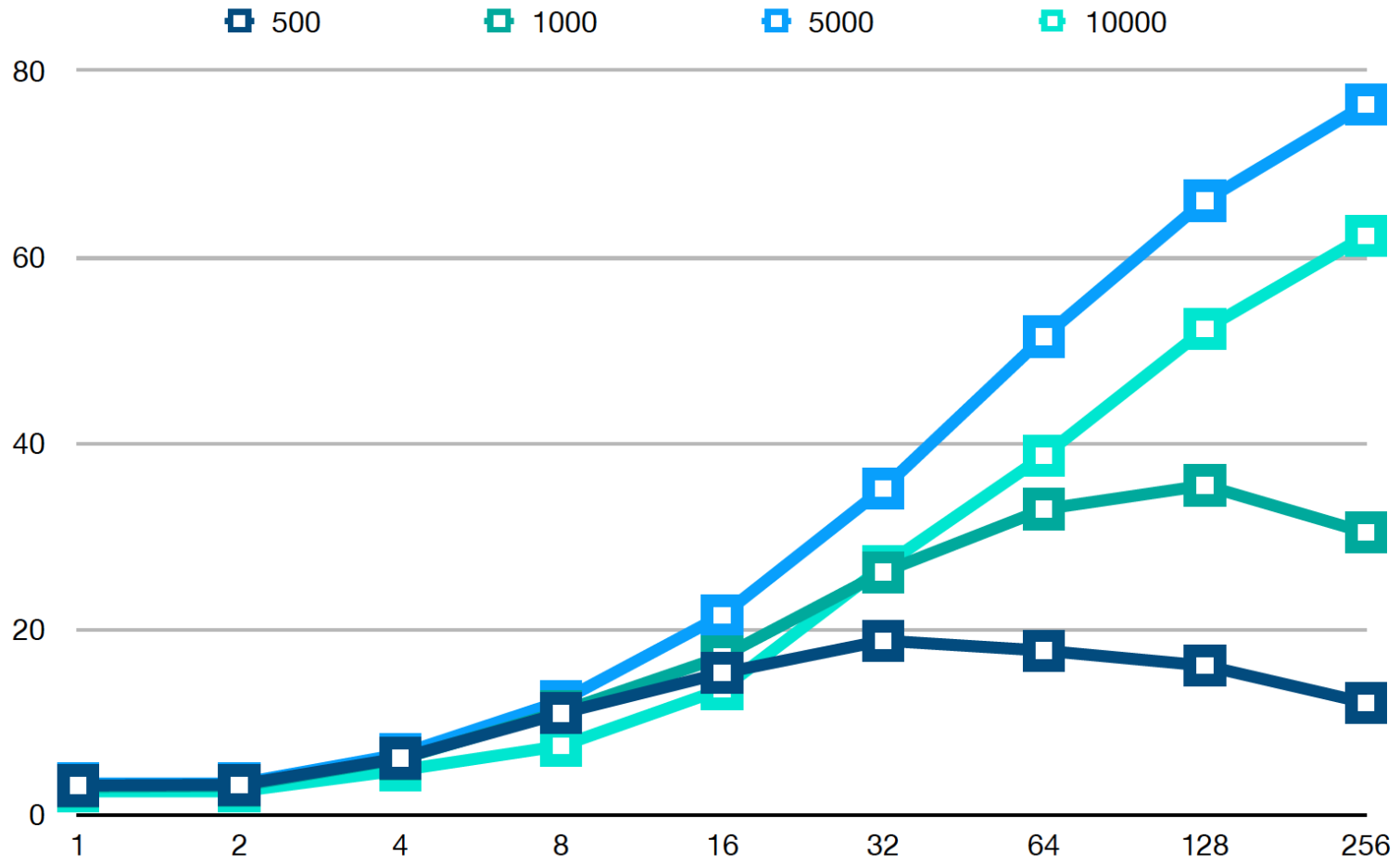
- Strong Scaling: increase the number of processors p keeping the *total problem* size fixed
  - The total amount of work remains constant
  - The amount of work for a single processor decreases as p increases
  - Goal: reduce the total execution time by adding more processors
- Weak Scaling: increase the number of processors p keeping the *per-processor work* fixed
  - The total amount of work grows as p increases
  - Goal: solve larger problems within the same amount of time
- To present the performance of a parallel program you must present the strong scaling with proper p values.
  - Es for a single node of the INFN cluster you should consider at least 256 (4 threads x core), 64 (actual cores) and the sequential time.
  - For the full cluster consider that, in principle, you can use 256 (single node) x 12 (nodes) processors

# A good example

Threads or cpu number

| Data size | seq | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|---|---|---|---|---|
| 500 | 2,450 | 3,125 | 3,194 | 6,050 | 10,889 | 15,218 | 18,703 | 17,627 | 16,014 | 12,010 |
| 1000 | 9,099 | 2,972 | 3,070 | 5,966 | 11,056 | 16,975 | 26,071 | 32,848 | 35,404 | 30,329 |
| 5000 | 251,723 | 3,334 | 3,335 | 6,475 | 12,189 | 21,416 | 35,054 | 51,404 | 66,034 | 76,395 |
| 10000 | 754,078 | 2,491 | 2,490 | 4,761 | 7,370 | 13,483 | 26,680 | 38,599 | 52,221 | 62,274 |

OpenMP, single node



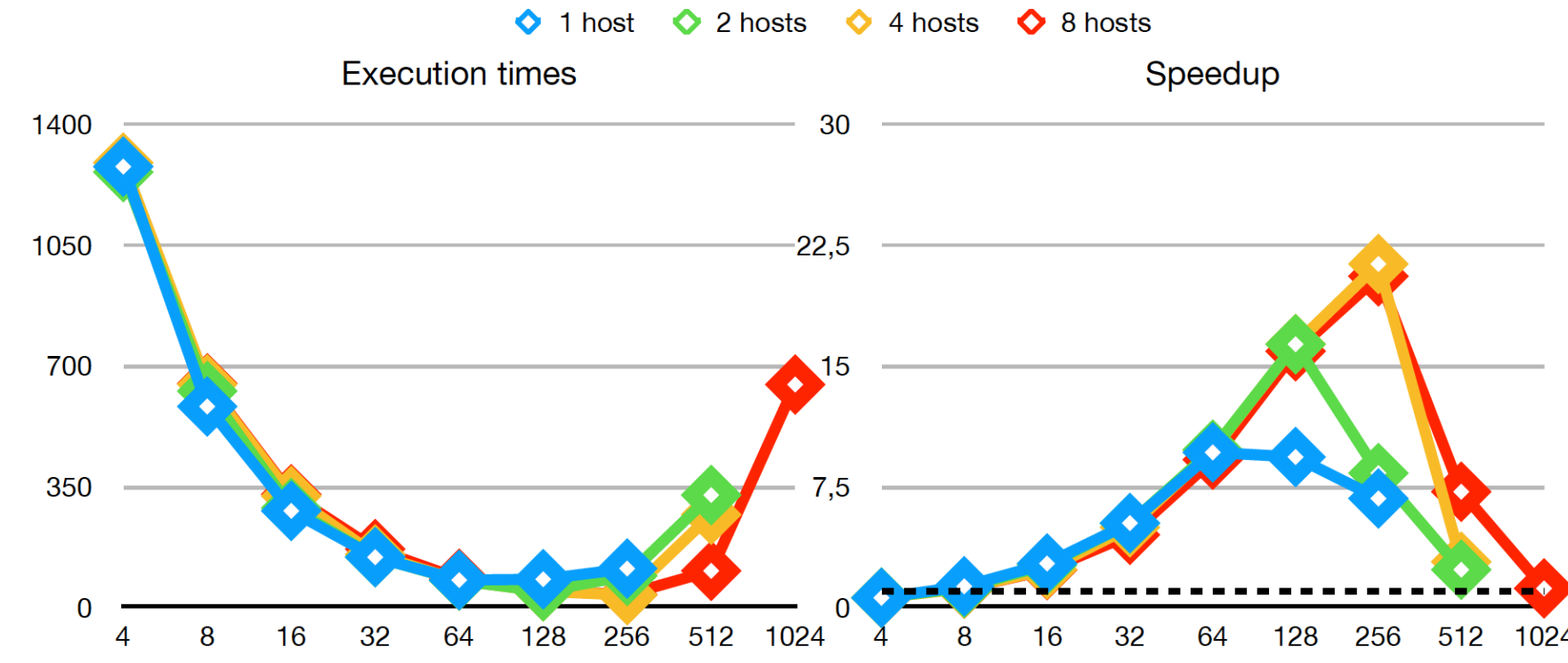
# A good example

MPI, cluster

| 10000 | seq | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 760,510 | - | 0,596 | 1,307 | 2,721 | 5,231 | 9,619 | 9,333 | 6,762 | - | - |
| 2 | 760,510 | - | 0,603 | 1,214 | 2,640 | 5,218 | 9,792 | 16,335 | 8,335 | 2,338 | - |
| 4 | 760,510 | - | 0,590 | 1,177 | 2,356 | 4,969 | 9,740 | 16,342 | 21,317 | 2,832 | |
| 8 | 760,510 | - | - | 1,172 | 2,323 | 4,501 | 9,165 | 15,900 | 20,553 | 7,176 | 1,178 |

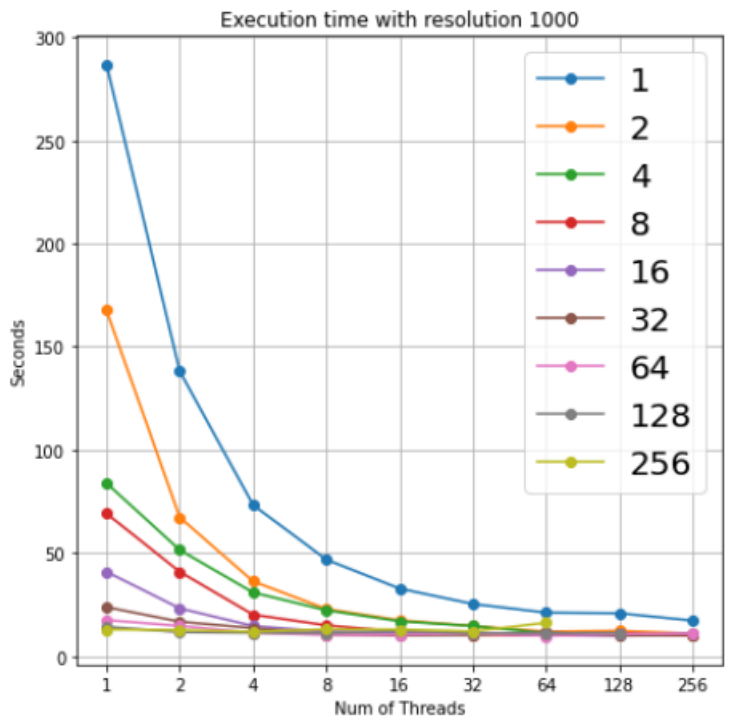
Speedup

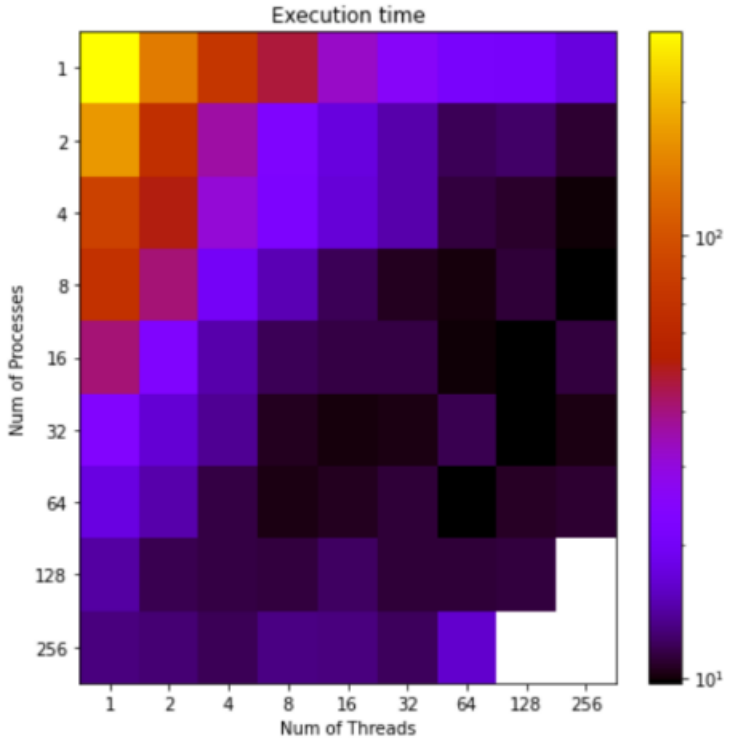
nodes



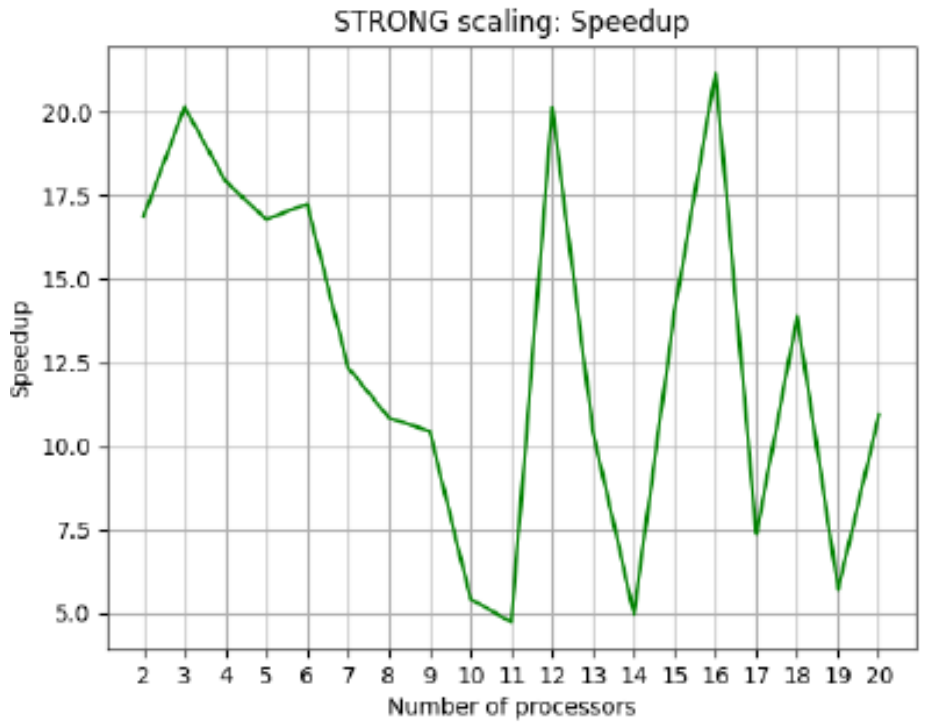
# Another good example: MPI+OMP

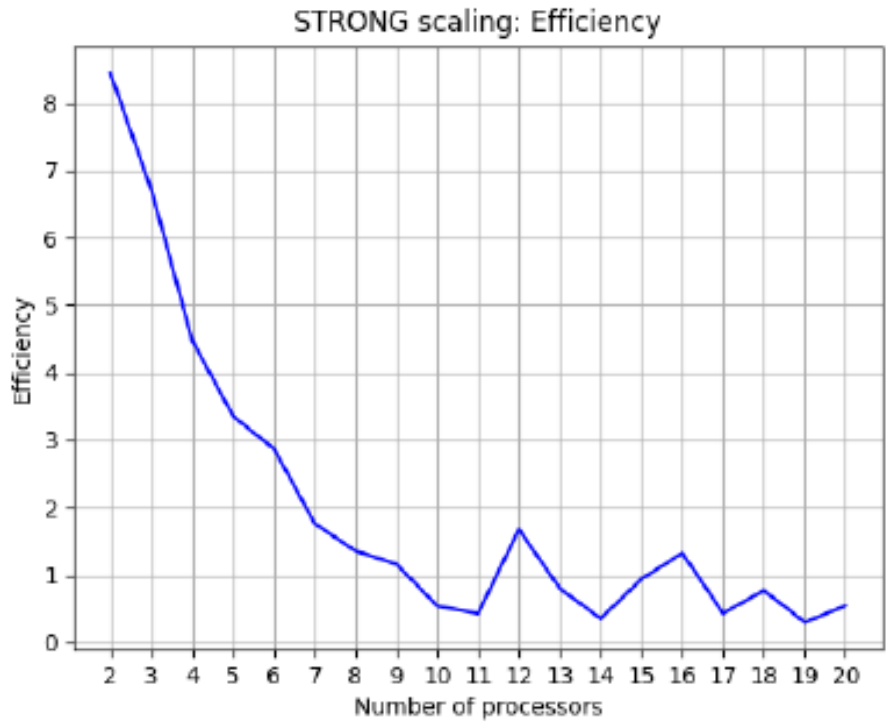
| Resolution 1000: Execution time on 8 Machine | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Processes / Threads | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 |
| 1 | 286.97 | 138.035 | 73.258 | 46.77 | 32.72 | 25.265 | 21.073 | 20.75 | 17.195 |
| 2 | 167.646 | 67.229 | 36.229 | 22.812 | 17.362 | 14.583 | 11.851 | 12.232 | 10.921 |
| 4 | 83.845 | 51.415 | 30.828 | 22.211 | 16.886 | 14.581 | 11.199 | 10.718 | 9.983 |
| 8 | 69.128 | 40.837 | 20.067 | 14.915 | 11.753 | 10.401 | 10.008 | 11.04 | 9.78 |
| 16 | 40.857 | 23.134 | 14.554 | 11.849 | 11.297 | 11.317 | 9.971 | 9.777 | 11.112 |
| 32 | 23.73 | 16.669 | 13.55 | 10.526 | 9.99 | 10.212 | 11.664 | 9.84 | 10.218 |
| 64 | 17.452 | 14.62 | 11.277 | 10.231 | 10.4 | 11.021 | **9.726** | 10.641 | 10.881 |
| 128 | 14.209 | 11.583 | 11.251 | 11.162 | 12.148 | 11.009 | 11.058 | 11.218 | nan |
| 256 | 12.892 | 12.6 | 11.833 | 13.012 | 12.89 | 11.986 | 16.172 | nan | nan |



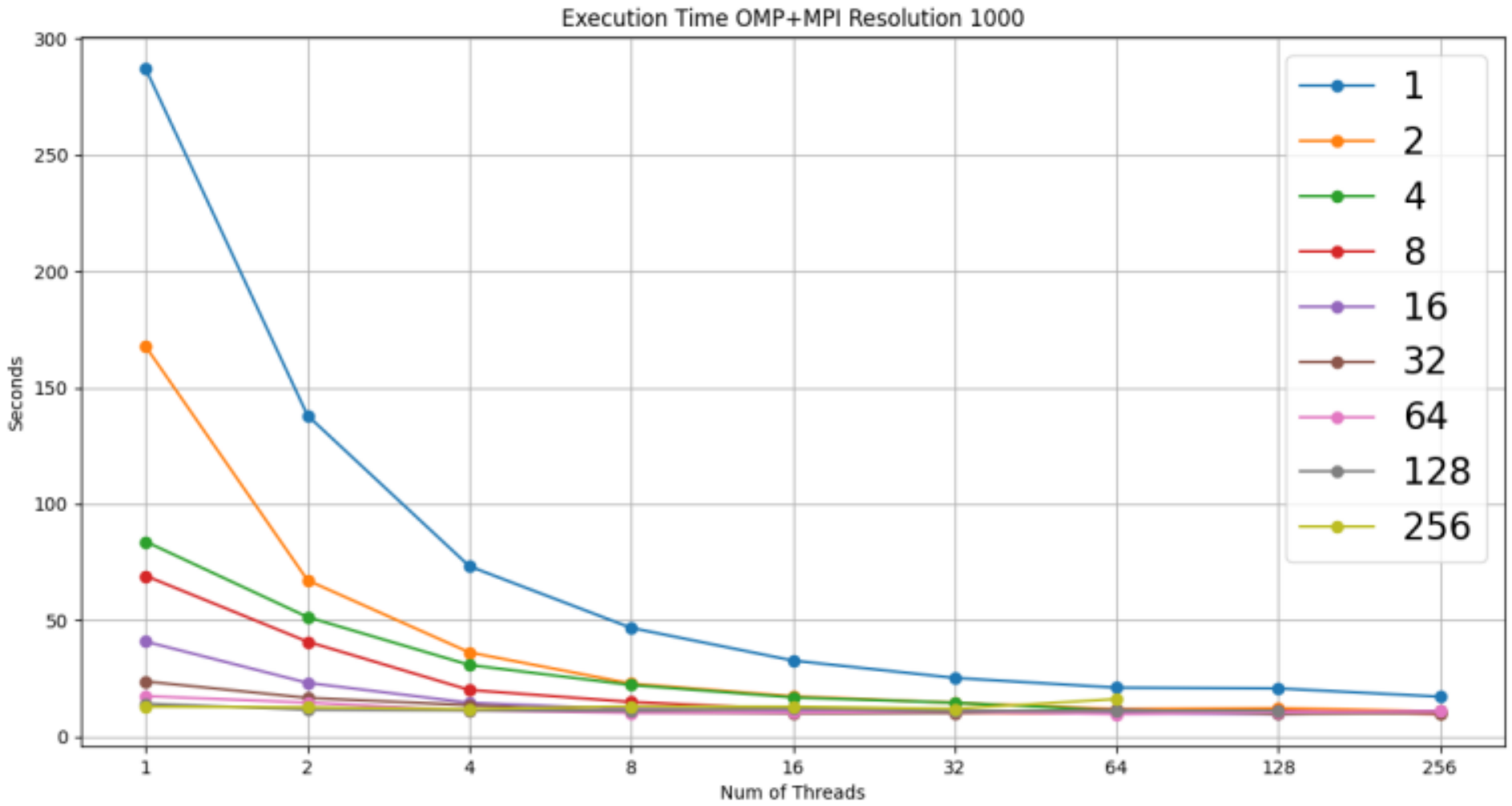
# Poor presentation




At least if you don't want to discuss IN DETAILS why $S_{11} = 5$ and $S_{12} = 20$

# Useless results



# Possible misleading representation

The ideal speedup is
Number of node * threads

i.e. 430 vs 64*8

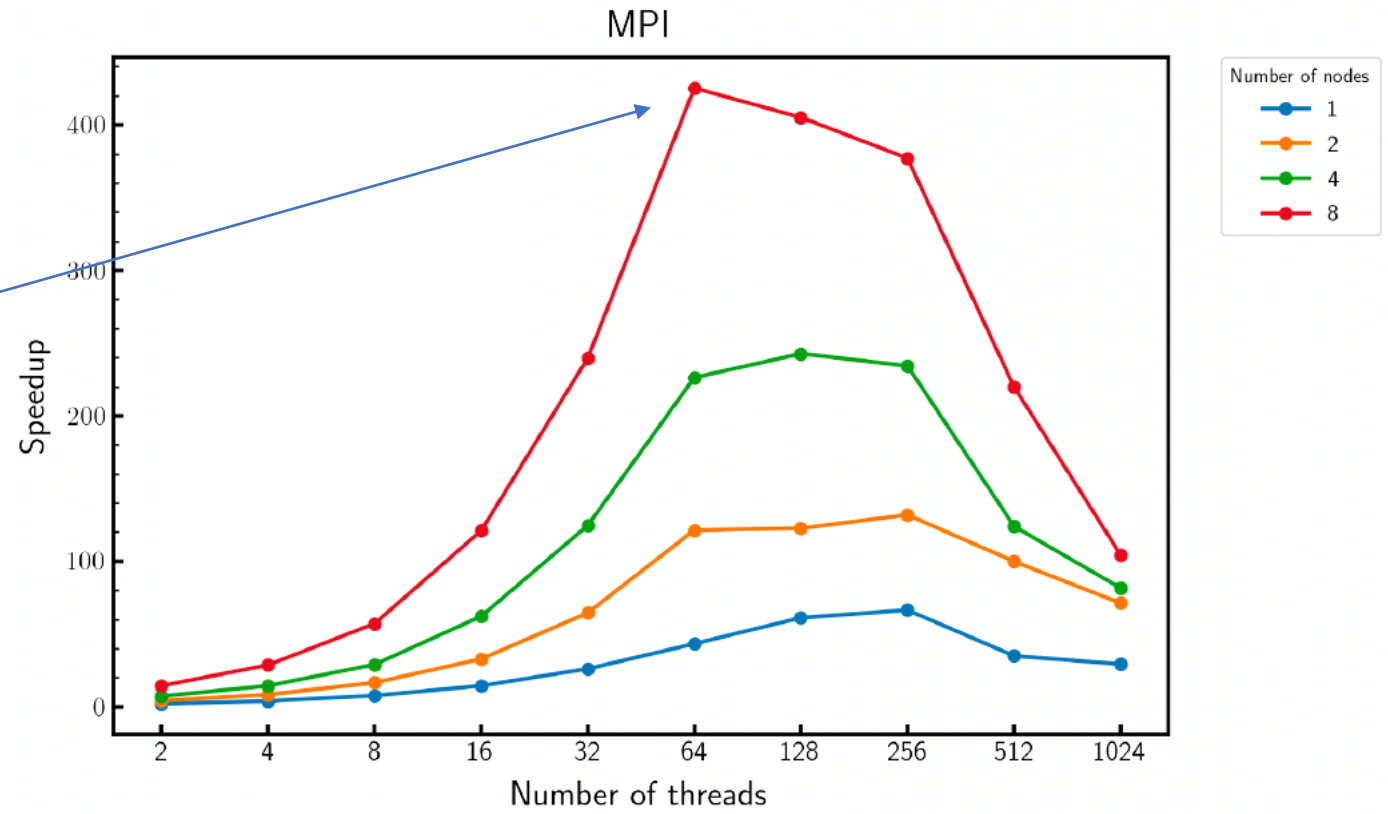



Fig. 11: MPI Speedup, size of the game board 10000x10000

# Remember the main lesson

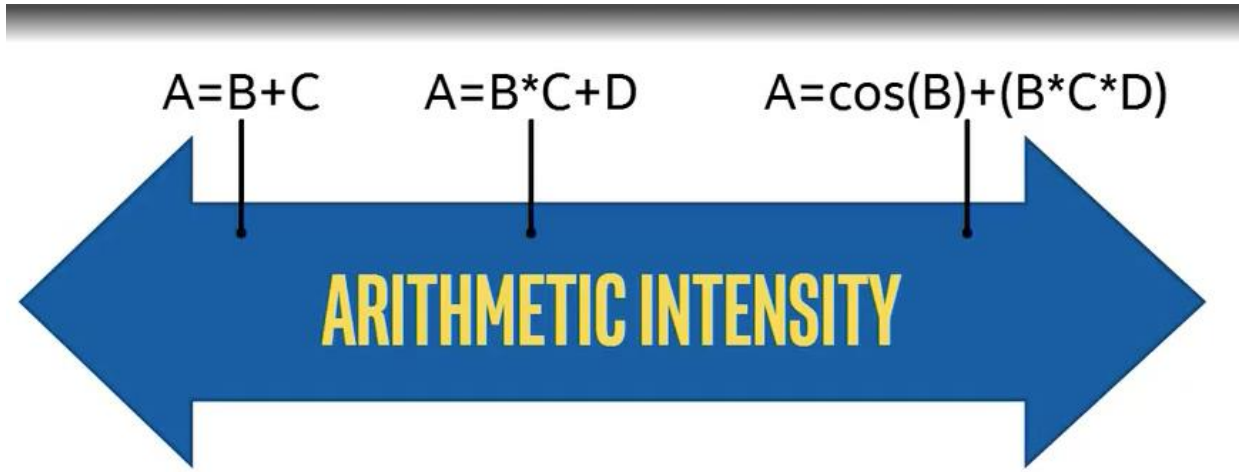
- Linear speedup is rare, due to communication overhead, load imbalance, algorithm/architecture mismatch, etc.
- Further, essentially nothing scales to arbitrarily many processors.
- However, for most users, the important question is:

**Have I achieved acceptable performance on my software for a suitable range of data and the resources I'm using?**

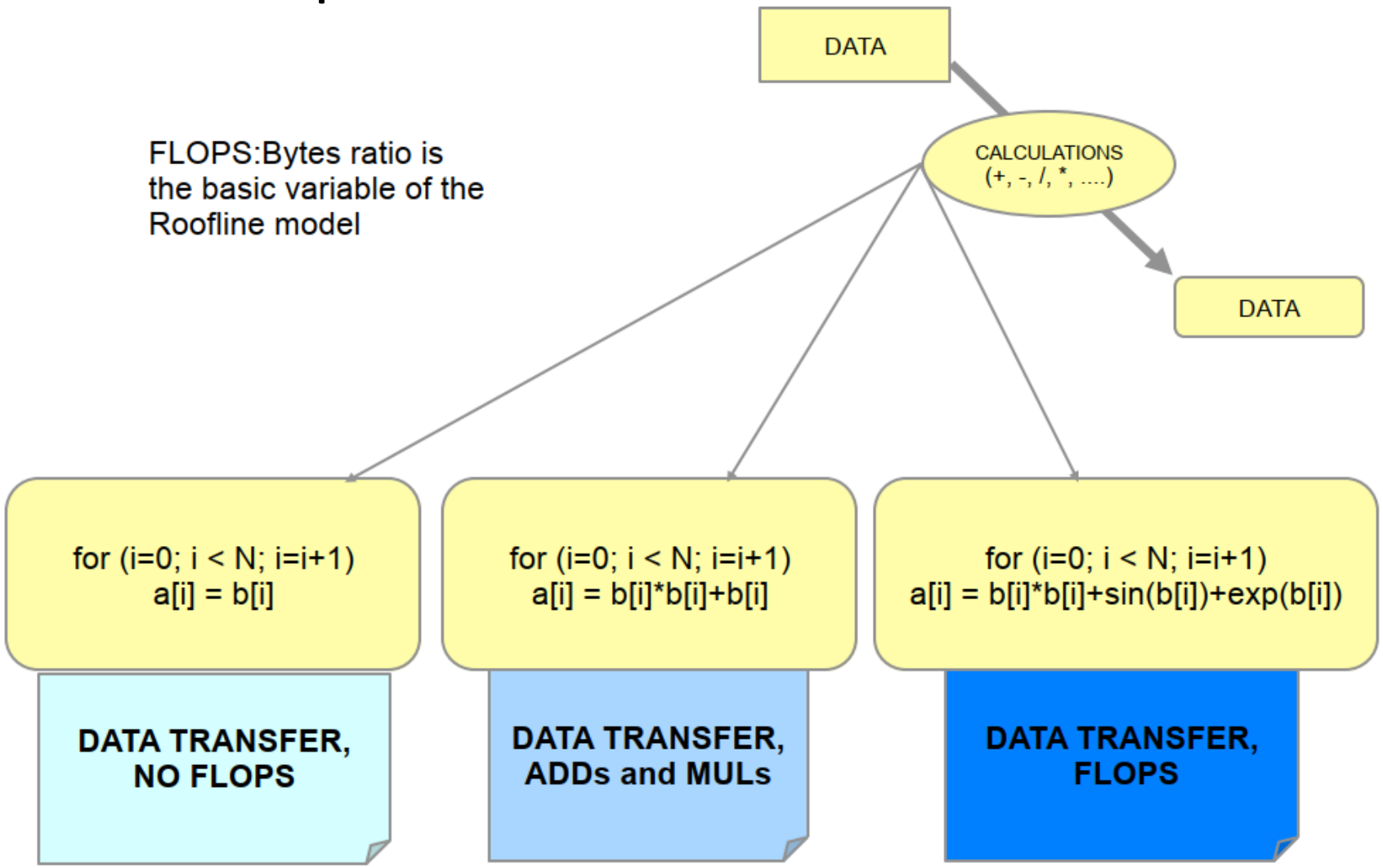
- **In the final project is important not only to achieve acceptable performance but also to be able to present and discuss them. The exam mark is proportional to this…**

# The Roofline performance model

- The Amdahl's Law is an example of bound and  and bottleneck analysis, i.e. rather than try to predict performance, it provides valuable insight into the primary factors affecting the performance of computer systems. In particular, the critical influence of the system bottleneck is highlighted and quantified
- Here we analyse a complementary model that  relates processor performance to off-chip memory traffic
- If n is the number of data items that an algorithm operates on, and f(n) the number of operations it takes, then the **operation/arithmetic intensity** is f(n)/n.



# Basic concepts

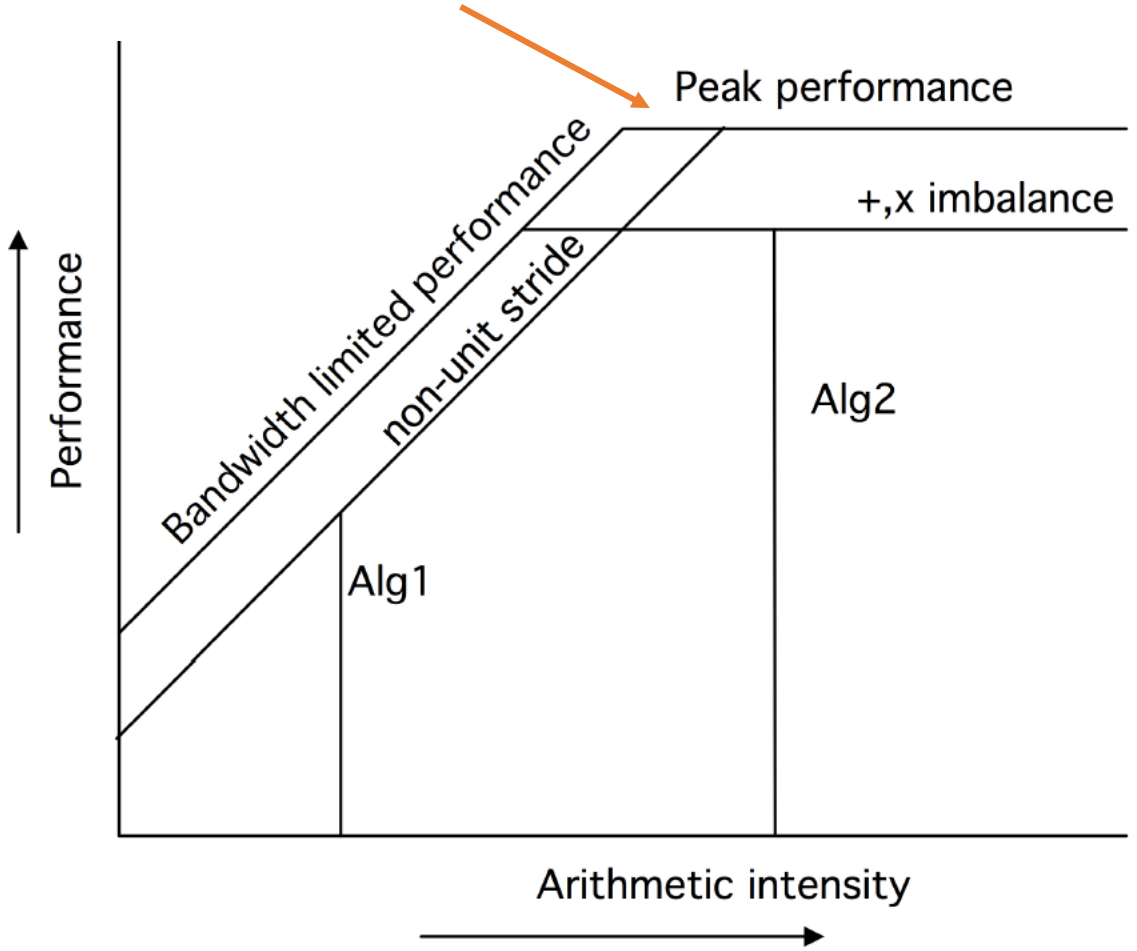



https://www.dam.brown.edu/people/lgrinb/APMA2821/Lectures_2015/

# The roofline performance model

- The peak performance of a system is an **absolute bound** on the achievable performance
- It is  achieved **only if** every aspect of a CPU is **perfectly used**.
- The calculation of this number is purely based on CPU properties and clock cycle
- Limiting factor: load imbalance, bad use of the cache…

$$\frac{operations}{second} = \frac{operations}{data\ item} \cdot \frac{data\ items}{second}$$



# An example

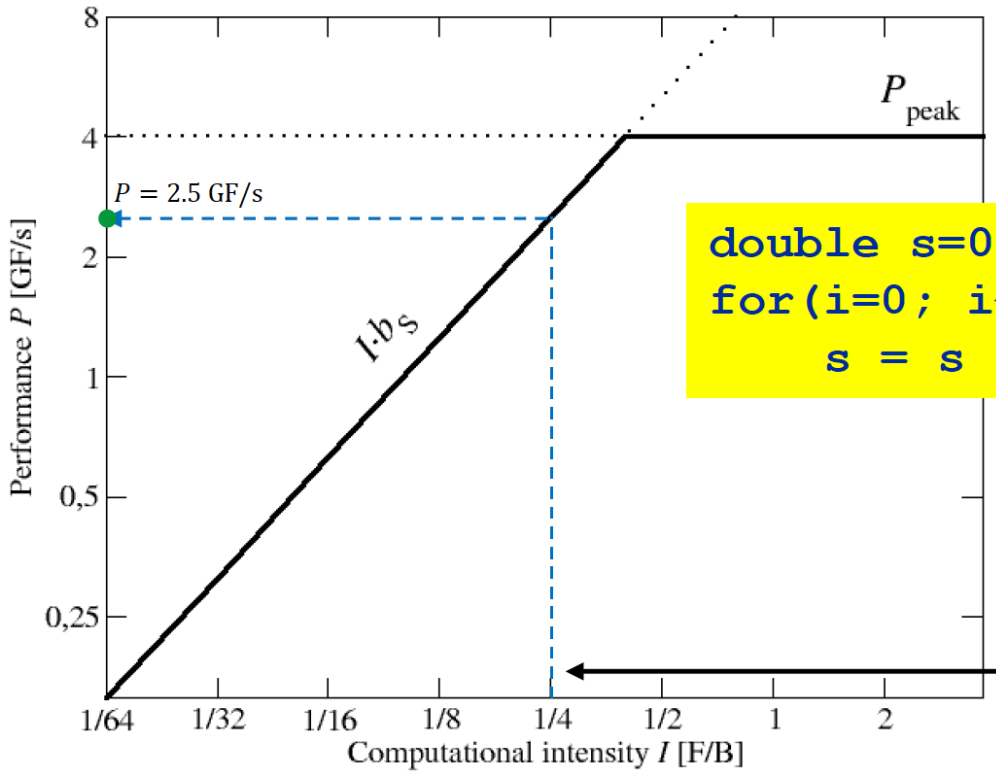
- Machine parameter #1: Peak performance: $P_{peak}$ $\left[\frac{F}{s}\right]$
- Machine parameter #2: Memory bandwidth: $b_S$ $\left[\frac{B}{s}\right]$
- Code characteristic: Computational Intensity: $I$ $\left[\frac{F}{B}\right]$ $\frac{\text{Flops}}{\text{Bytes}}$

Machine properties:

$$P_{peak} = 4\,\frac{\text{GF}}{\text{s}}$$

$$b_S = 10\,\frac{\text{GB}}{\text{s}}$$

Application property: $I$




+,* = 2 Flop
A[i] = 8 bytes
S in a register

```
double s=0, a[];
for(i=0; i<N; ++i) {
    s = s + a[i] * a[i];}
```

$$I = \frac{2\,F}{8\,B} = 0.25\,{}^{F}/_{B}$$

# Arithmetic intensity

```
double a[], b[];
for(i=0; i<N; ++i) {
    a[i] = a[i] + b[i];}
```

$B_C$ = 24B / 1F = 24 B/F

$I$ = 0.042 F/B

```
double a[], b[];
for(i=0; i<N; ++i) {
    a[i] = a[i]+ s * b[i];}
```

$B_C$ = 24B / 2F = 12 B/F

$I$ = 0.083 F/B

Scalar – can be kept in register

```
float s=0, a[];
for(i=0; i<N; ++i) {
    s = s + a[i] * a[i];}
```

$B_C$ = 4B/2F = 2 B/F

$I$ = 0.5 F/B

Scalar – can be kept in register

```
float s=0, a[], b[];
for(i=0; i<N; ++i) {
    s = s + a[i] * b[i];}
```

$B_C$ = 8B / 2F = 4 B/F

$I$ = 0.25 F/B

Scalar – can be kept in register

# Other examples

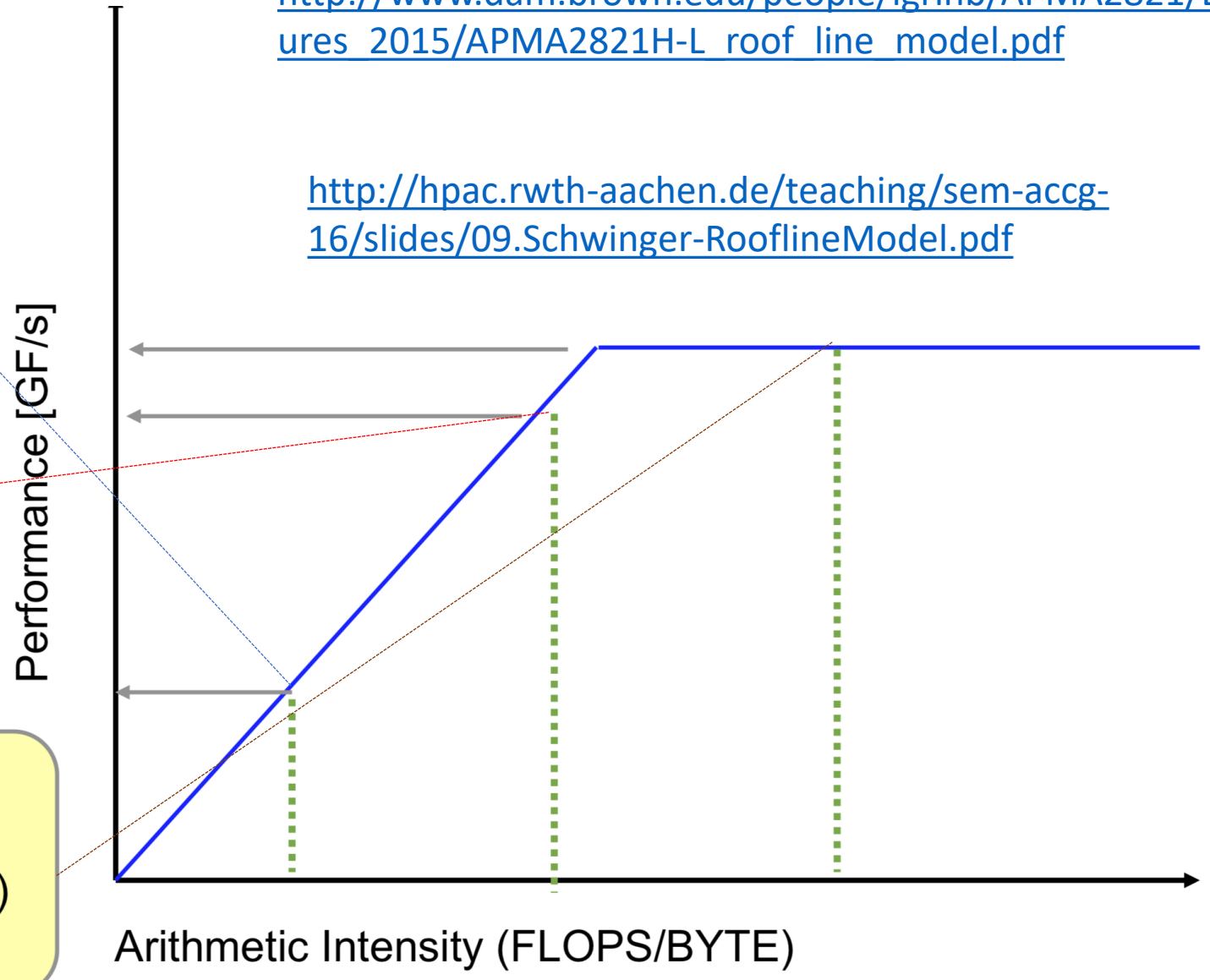
```
for (i=0; i < N; i=i+1)
a[i] = 2.3*b[i]
```

```
for (i=0; i < N; i=i+1)
a[i] = b[i]*b[i]+b[i]
```

```
for (i=0; i < N; i=i+1)
a[i] = b[i]*b[i]+sin(b[i])+exp(b[i])
```

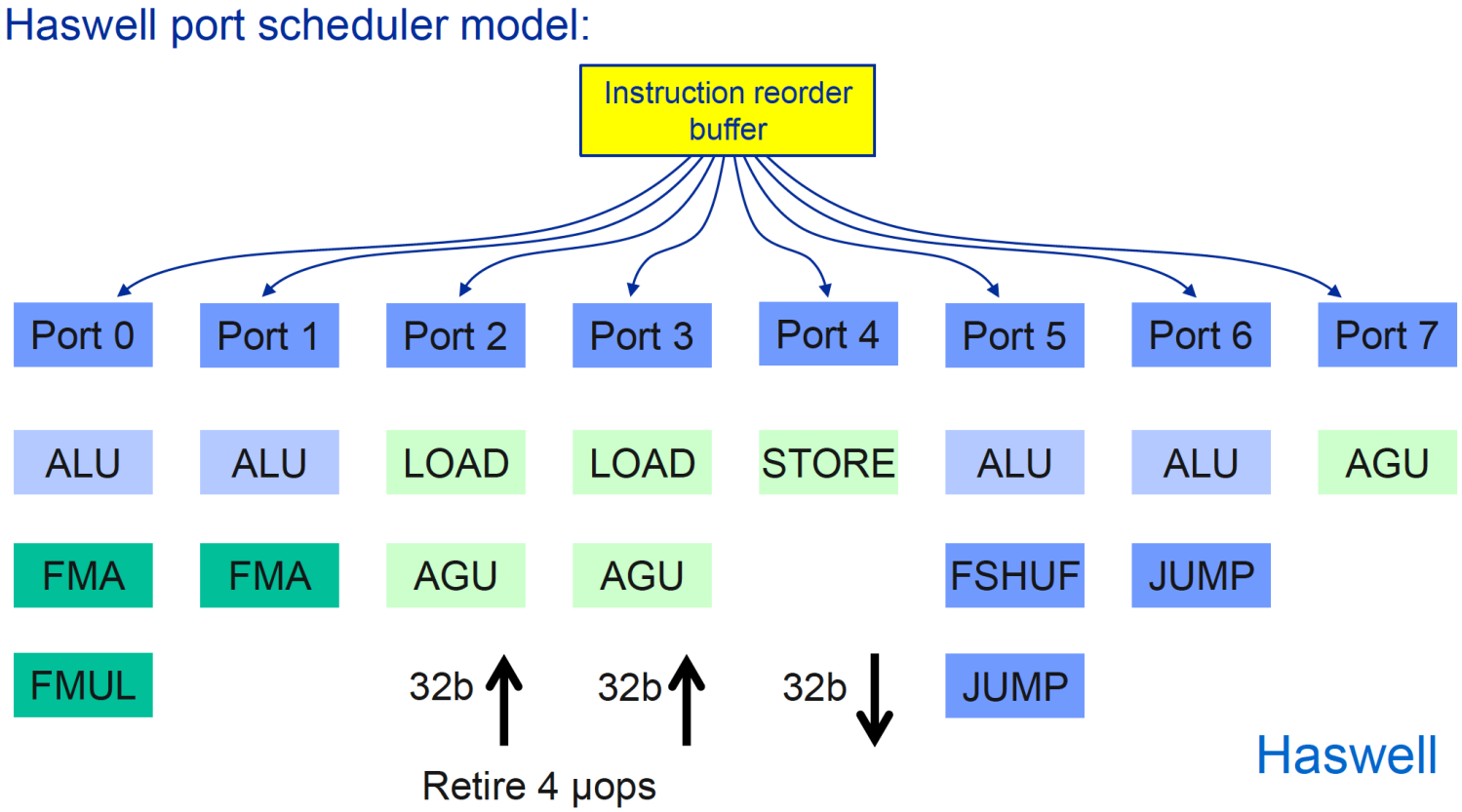
http://www.dam.brown.edu/people/lgrinb/APMA2821/Lectures_2015/APMA2821H-L_roof_line_model.pdf

http://hpac.rwth-aachen.de/teaching/sem-accg-16/slides/09.Schwinger-RooflineModel.pdf



# Estimating $P_{peak}$

- Per cycle with AVX, SSE, or scalar it can execute
  - 2 LOAD AND 1 STORE
  - 2 instructions from 2 FMA, 2 MULT, 1 ADD
- Overall maximum of 4 instructions per cycle
- 1 AVX instruction = 4 DP or 8 SP scalar operations



```
double  *A, *B, *C, *D;
for (int i=0; i<N; i++) {
    A[i] = B[i] + C[i] * D[i];
}
```

One AVX iteration (1.5 cycles) does 4 x 2 = 8 flops

$$2.3 \cdot 10^9 \text{ cy/s} \cdot \frac{8 \text{ flops}}{1.5 \text{ cy}} = 12.27 \frac{\text{Gflops}}{\text{s}}$$ $P_{peak}$

$$12.27 \frac{\text{Gflops}}{\text{s}} \cdot 16 \frac{\text{bytes}}{\text{flop}} = 196 \frac{\text{Gbyte}}{\text{s}}$$ $B_s$

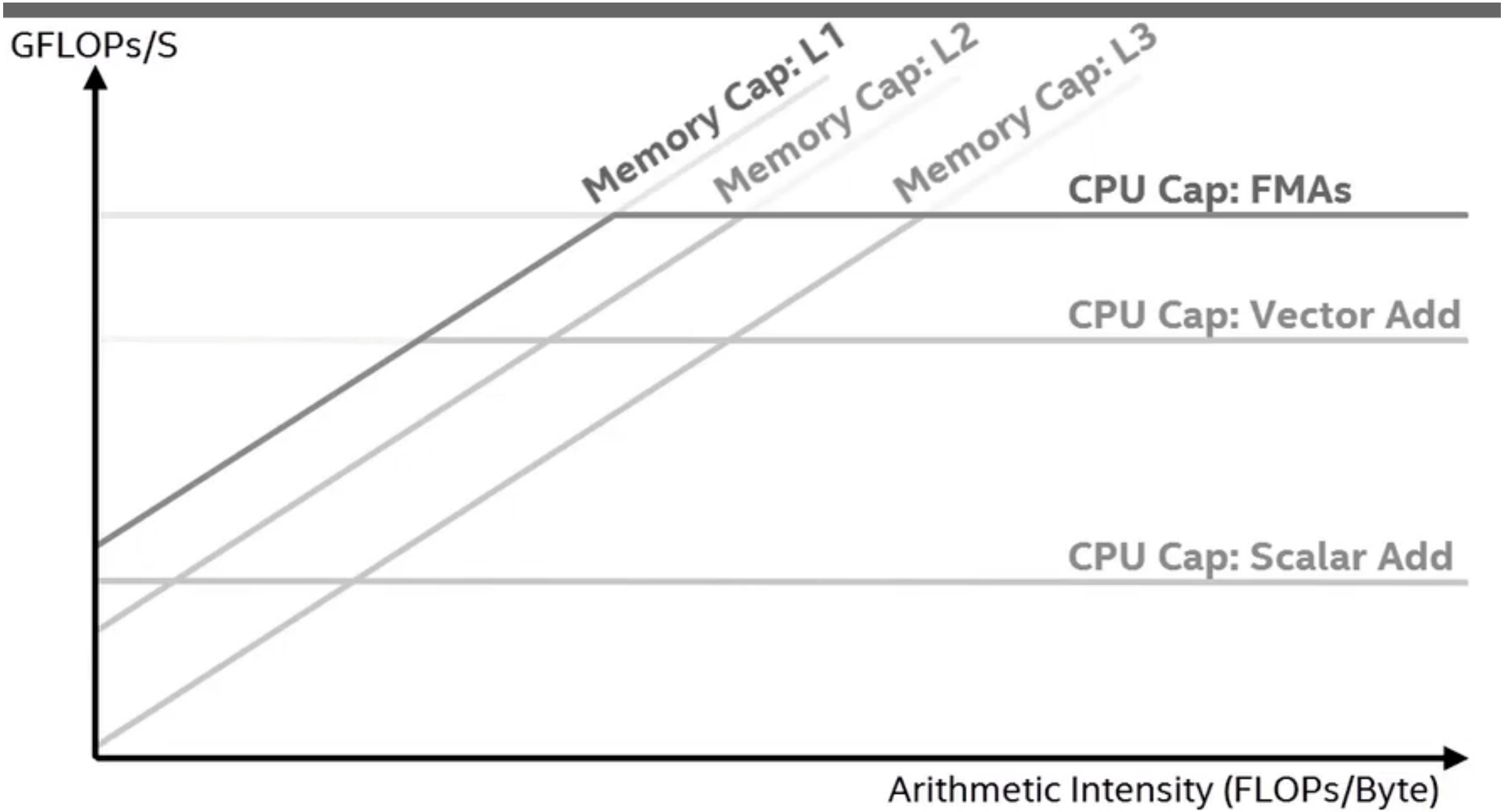
Assembly code (AVX2+FMA, no additional unrolling):

```
..B2.9:
  vmovupd    (%rdx,%rax,8), %ymm2   # LOAD
  vmovupd    (%r12,%rax,8), %ymm1   # LOAD
  vfmadd213pd (%rbx,%rax,8), %ymm1, %ymm2 # LOAD+FMA
  vmovupd    %ymm2, (%rdi,%rax,8)   # STORE
  addq       $4, %rax
  cmpq       %r11, %rax
  jb         ..B2.9
# remainder loop omitted
```

**Minimum number of cycles to process one AVX-vectorized iteration (equivalent to 4 scalar iterations) on one core?**

Cycle 1: LOAD + LOAD + STORE
Cycle 2: LOAD + LOAD + FMA + FMA
Cycle 3: LOAD + LOAD + STORE

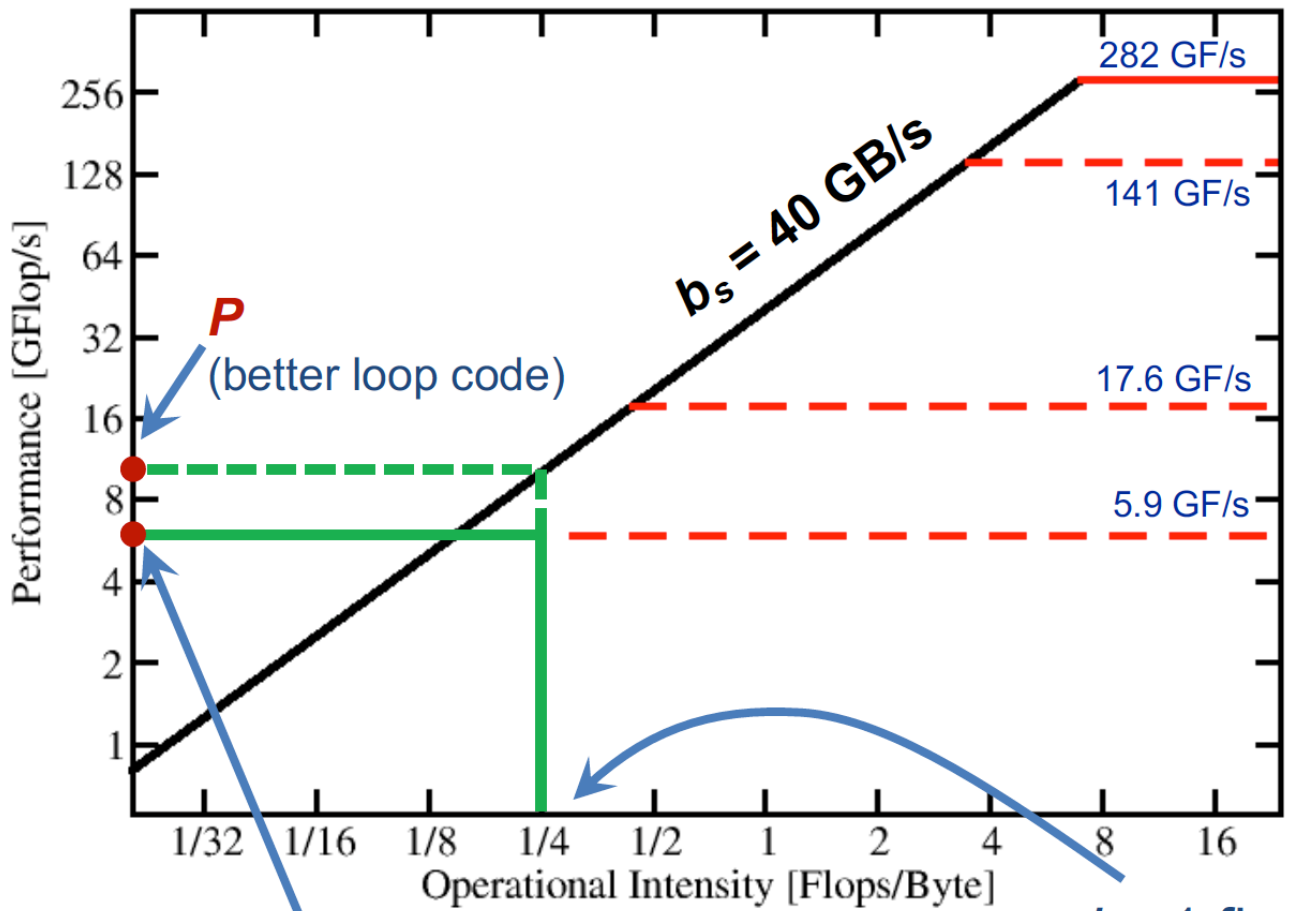
**Answer: 1.5 cycles**


GFLOPs/S
Memory Cap: L1
Memory Cap: L2
Memory Cap: L3
CPU Cap: FMAs
CPU Cap: Vector Add
CPU Cap: Scalar Add
Arithmetic Intensity (FLOPs/Byte)

# Example: `do i=1,N; s=s+a(i); enddo`

in single precision on an 8-core 2.2 GHz Sandy Bridge socket @ "large" N

$$P = \min(P_{\max}, I \cdot b_S)$$




Machine peak (ADD+MULT) Out of reach for this code

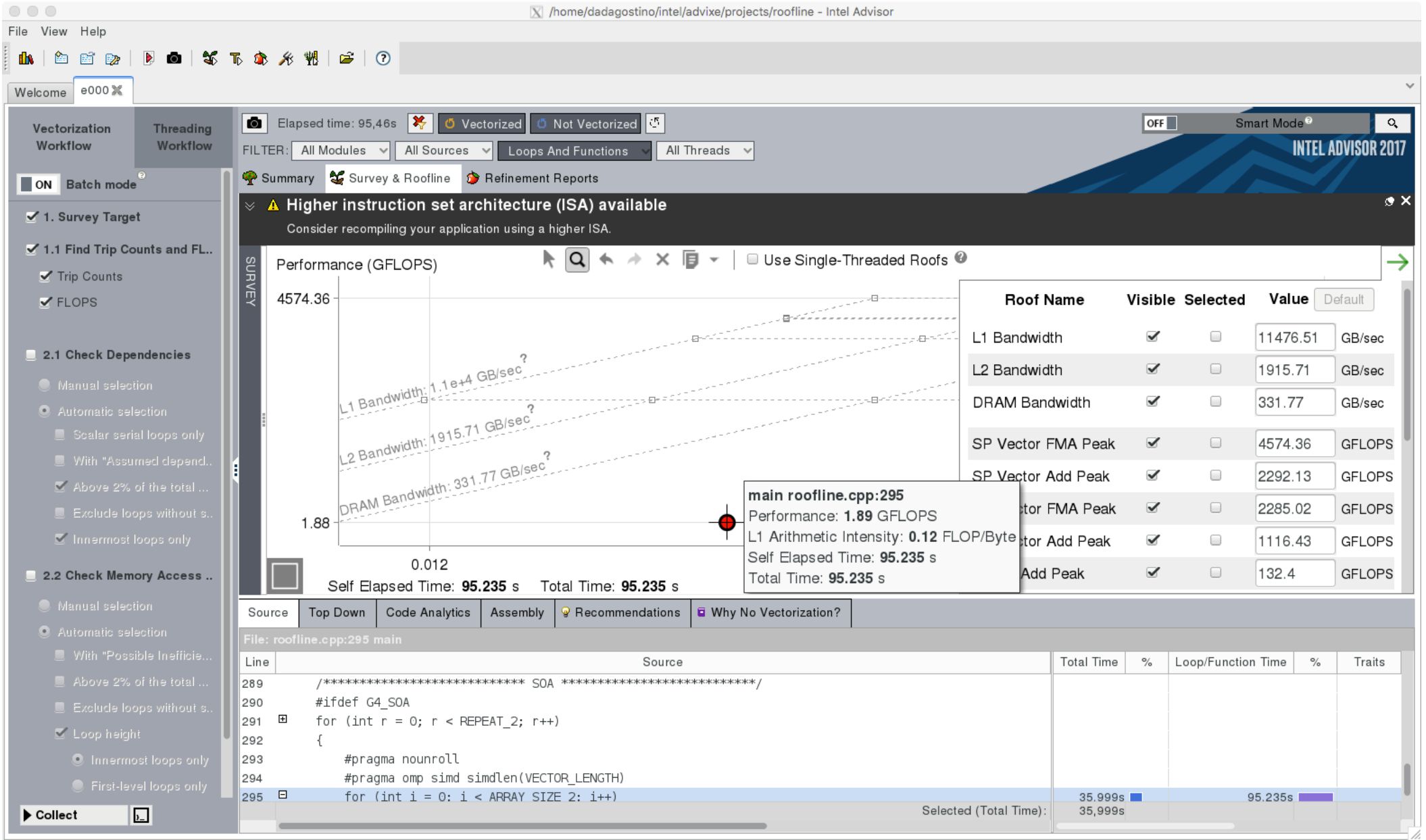
ADD peak (best possible code)

no SIMD

3-cycle latency per ADD if not unrolled

*I* = 1 flop / 4 byte (SP!)

# Roofline model and Intel



# Parallel Programming Models

## Message Passing Interface (MPI).

- Allows parallel processes to communicate via sending “messages” (i.e. data). Most standard way of communication between nodes, but can also be used within a node.

## OpenMP

- Allows parallel processes to communicate via shared memory in a node. Cannot be used between shared memory nodes.

## Hybrid MPI+OpenMP

- Combines both MPI+OpenMP. A situation could be to use OpenMP within a shared memory node and MPI between nodes.

## OpenAcc

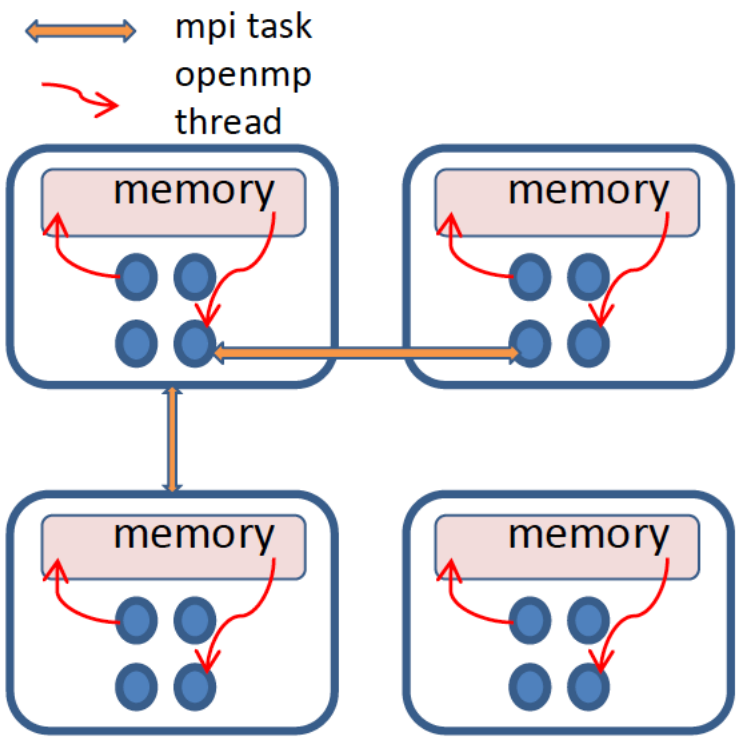
- Similar to OpenMP but used to program devices such as GPUs.

## CUDA

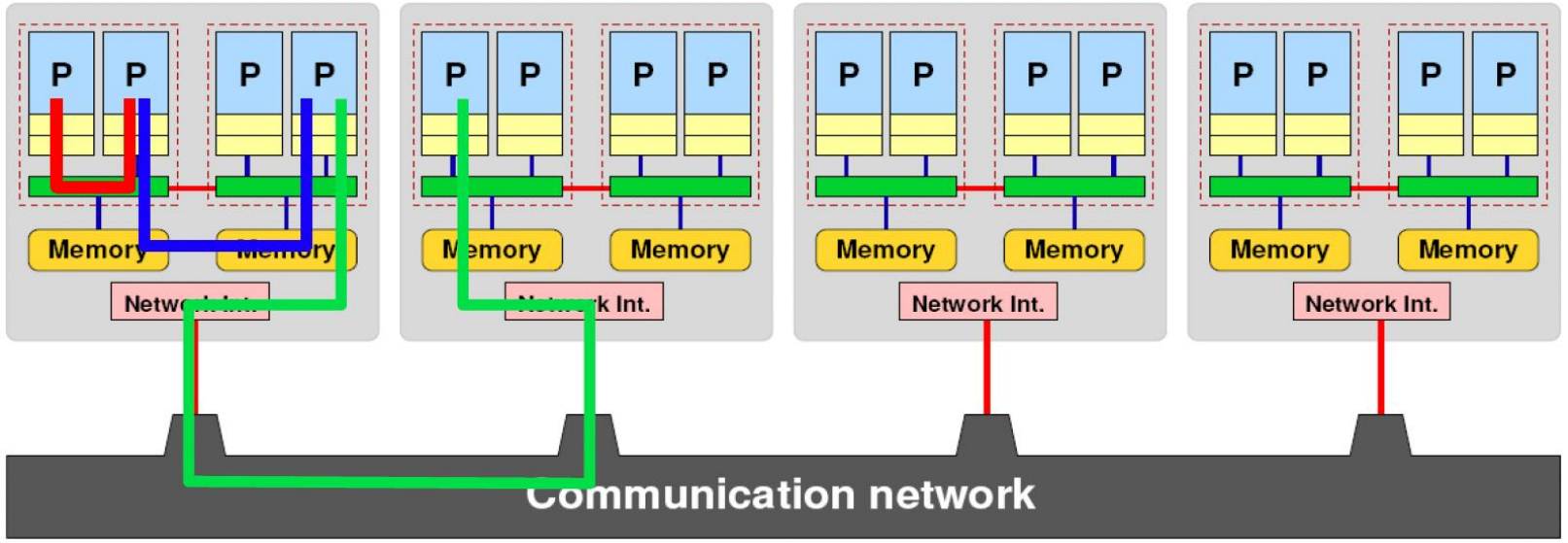
- Nvidia extension to C/C++ for GPU programming.

## OpenCL

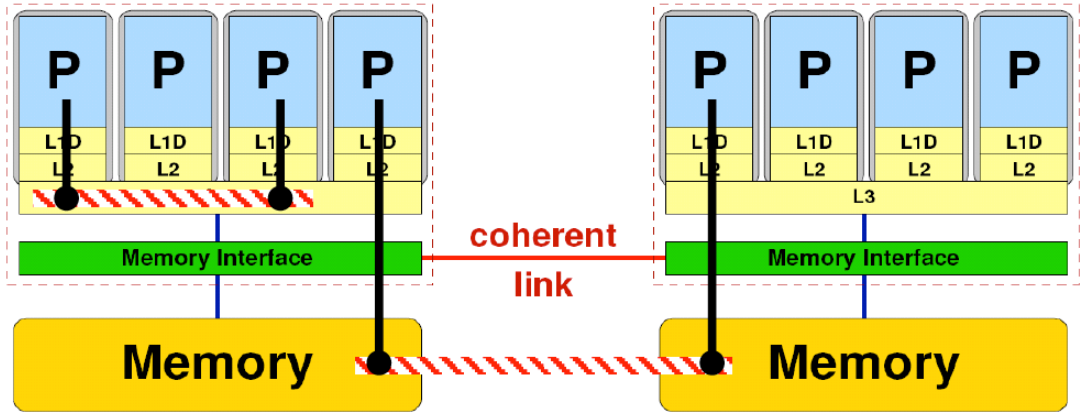
- Non-Nvidia alternative to programming GPUs.

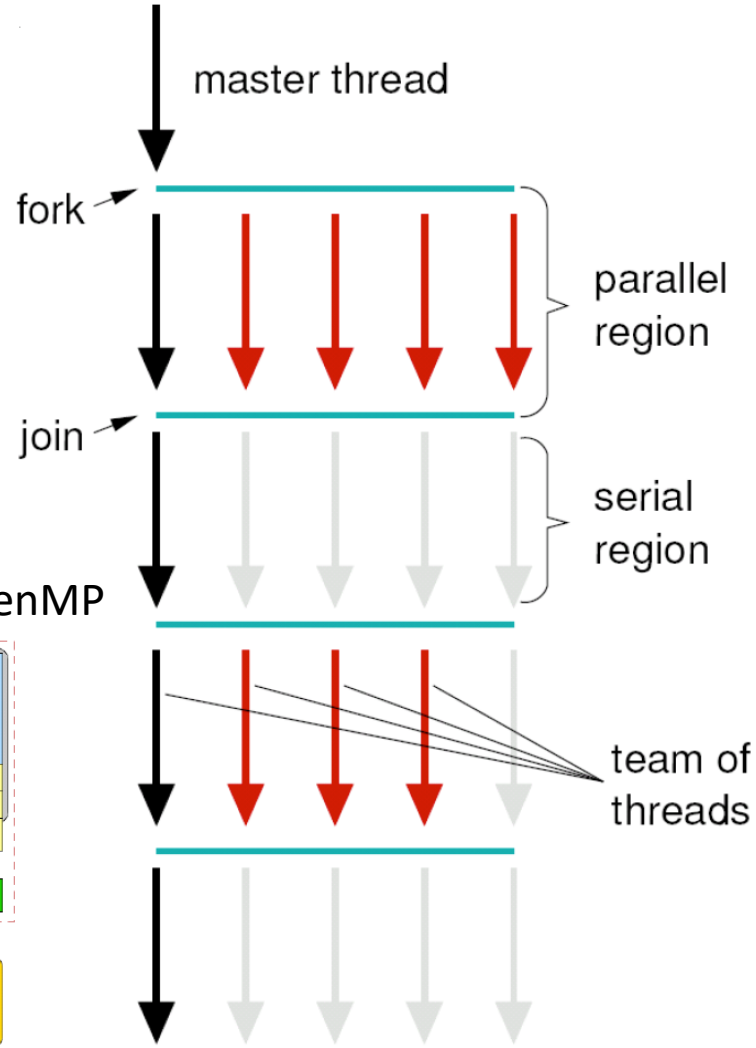


# MPI / OpenMP (OpenACC)




MPI



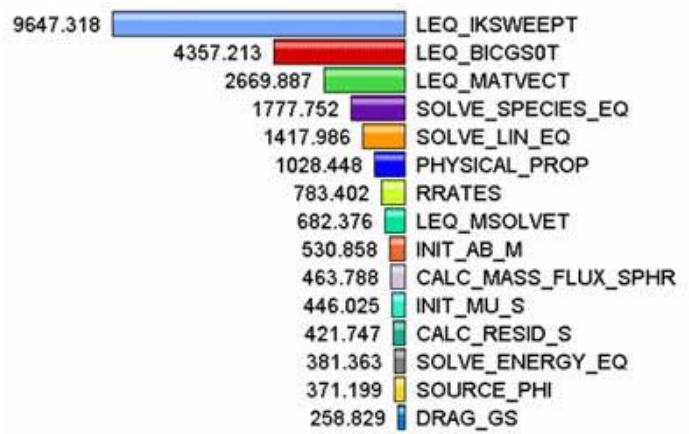

OpenMP

# Design parallel algorithm

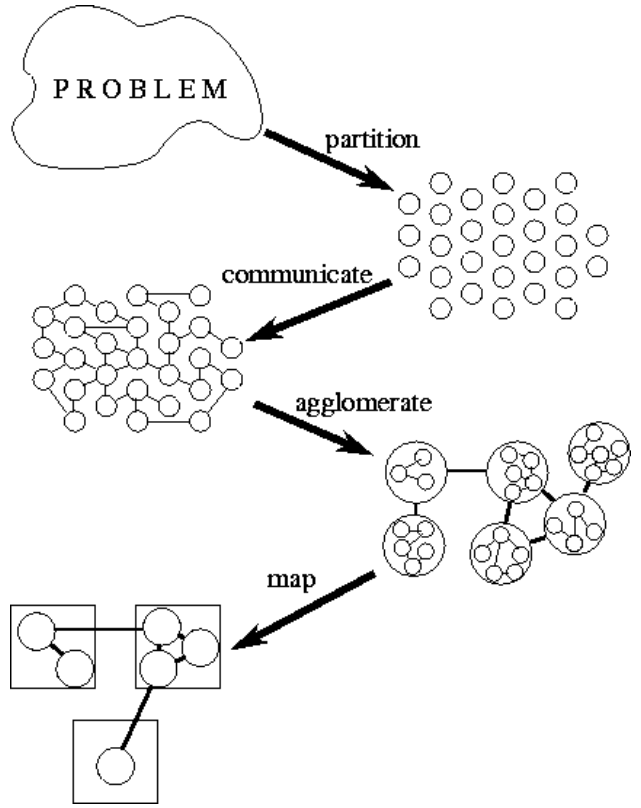
- We have the introduced parallel hardware and software libraries, now we need to map our problem into a parallel algorithm.
- It is a (mostly) manual, time consuming, complex, error-prone and iterative process
- For a number of years now, various tools have been available to assist the programmer with converting serial programs into parallel programs, i.e. parallelizing compiler (loops) or pre-processor (openMP).
- However
  - Works on shared memory architectures
  - Possible wrong results
  - Less flexible than manual parallelization
  - Lower performance

# Designing parallel algorithms

- Identify the program's hotspots
  - Most programs accomplish most of their work in a few places. (profilers and performance analysis tools)
  - Focus on parallelizing the hotspots
- Identify bottlenecks in the program
  - Mainly I/O
- Identify inhibitors to parallelism
  - Data dependence, ...
- Investigate other algorithms if possible
- Then start designing the solution

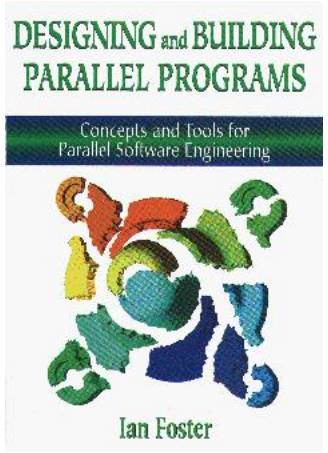



# PCAM Methodology



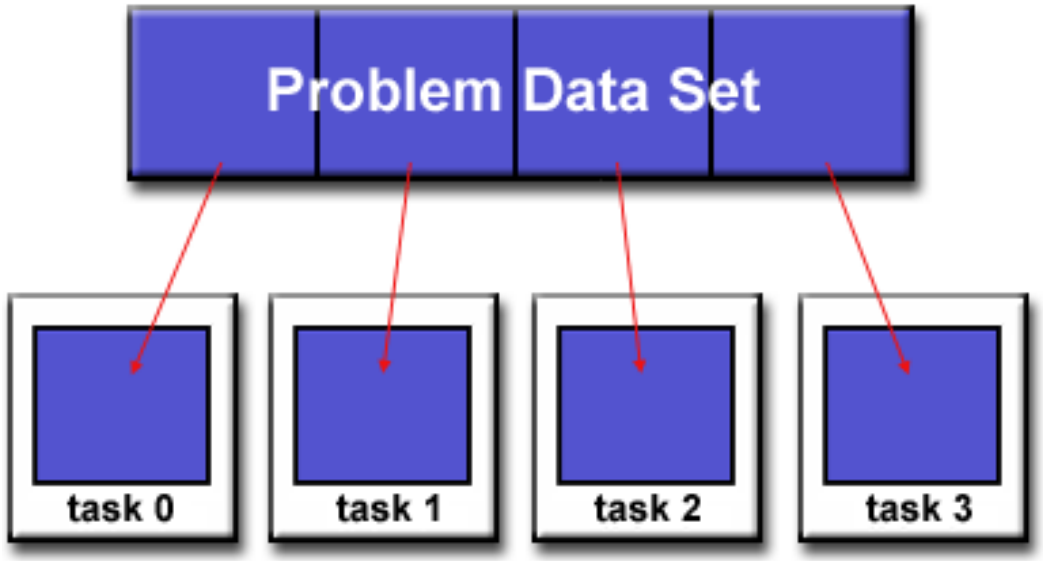

- *Partitioning.* The computation and the data are decomposed into small tasks. Practical issues such as the number of processors in the target computer are ignored, and attention is focused on recognizing opportunities for parallel execution (i.e. concurrency and scalability).
- *Communication.* The communication required to coordinate task execution is determined, and appropriate communication structures and algorithms are defined.
- *Agglomeration.* The task and communication structures defined in the first two stages of a design are evaluated with respect to performance requirements and implementation costs. If necessary, tasks are combined into larger tasks to improve performance or to reduce development costs.
- *Mapping.* Each task is assigned to a processor in a manner that attempts to satisfy the competing goals of maximizing processor utilization and minimizing communication costs. Mapping can be specified statically or determined at runtime by load-balancing algorithms.

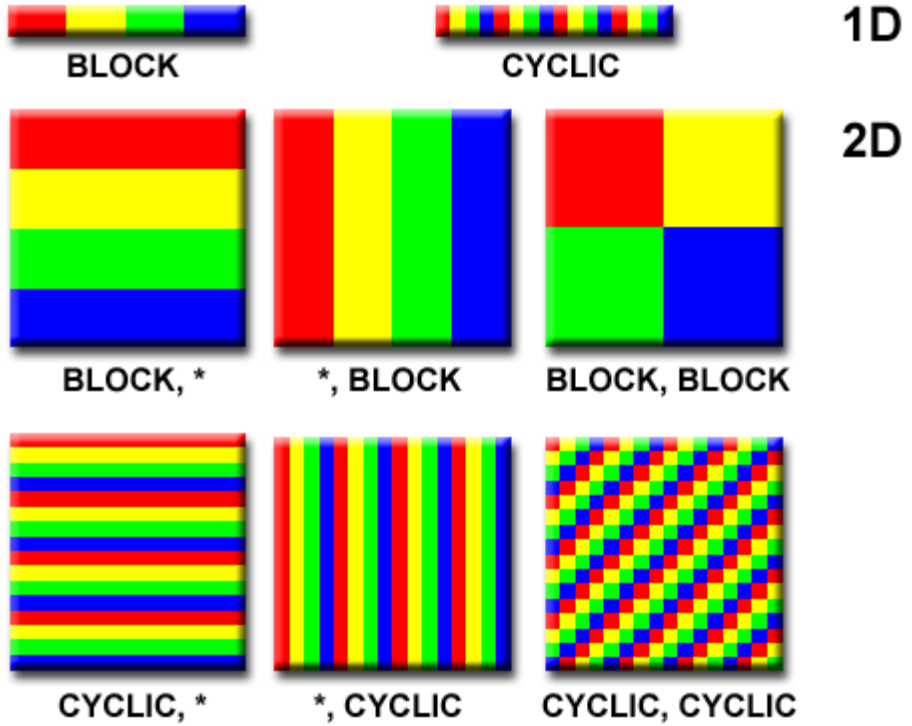
https://www.mcs.anl.gov/~itf/dbpp/



# Partitioning – Domain decomposition

- Break the problem into discrete "chunks" of work that can be distributed to multiple tasks.
- Domain or functional decomposition



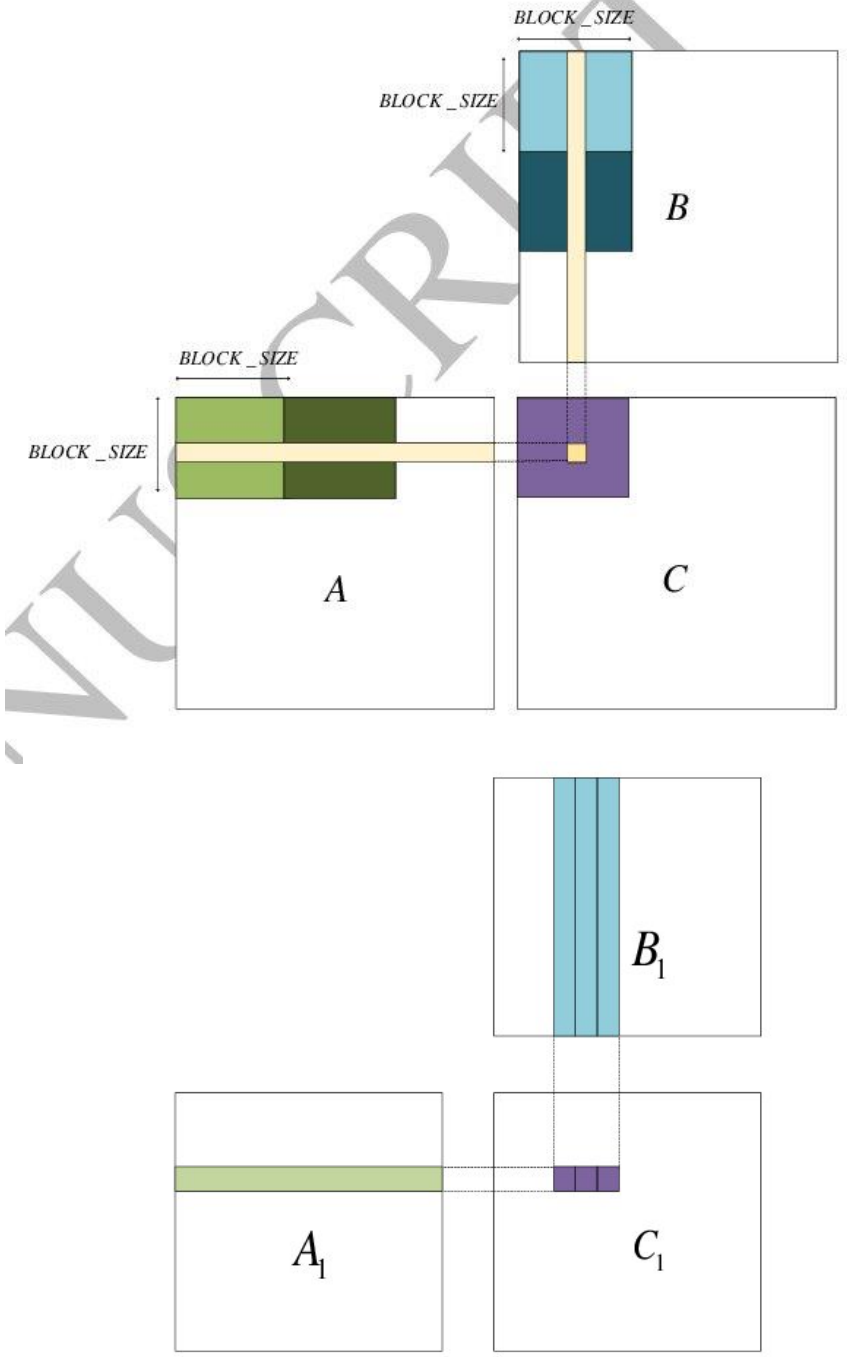
# Example: C = AxB

Different partitioning strategies

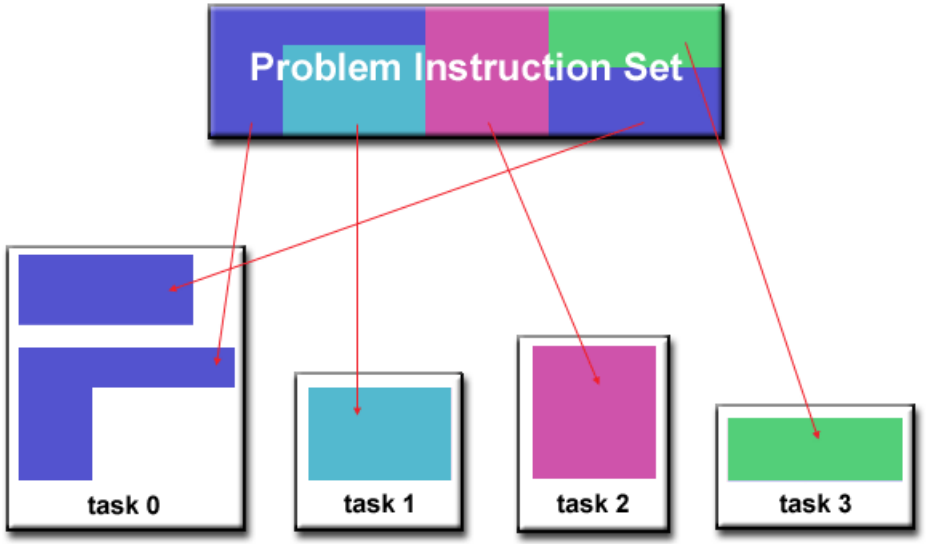
$$\begin{pmatrix} -1 & 2 & 4 \\ 1 & 0 & -1 \\ 2 & -1 & 3 \end{pmatrix} \begin{pmatrix} -2 & 2 \\ 0 & 1 \\ -2 & -1 \end{pmatrix}$$

$$= \left(\begin{array}{c|c} \begin{pmatrix} -1 & 2 & 4 \end{pmatrix} \begin{pmatrix} -2 \\ 0 \\ -2 \end{pmatrix} & \begin{pmatrix} -1 & 2 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} \\ \hline \begin{pmatrix} 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} -2 \\ 0 \\ -2 \end{pmatrix} & \begin{pmatrix} 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} \\ \hline \begin{pmatrix} 2 & -1 & 3 \end{pmatrix} \begin{pmatrix} -2 \\ 0 \\ -2 \end{pmatrix} & \begin{pmatrix} 2 & -1 & 3 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ -1 \end{pmatrix} \end{array}\right)$$

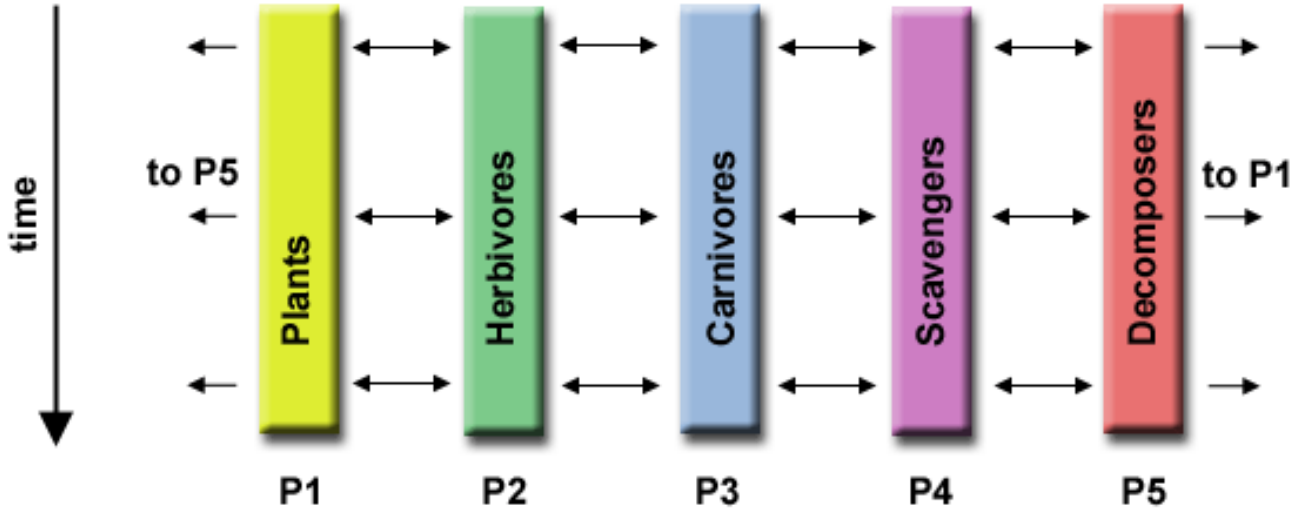
$$= \begin{pmatrix} -6 & -4 \\ 0 & 3 \\ -10 & 0 \end{pmatrix}$$



# Partitioning – Functional decomposition

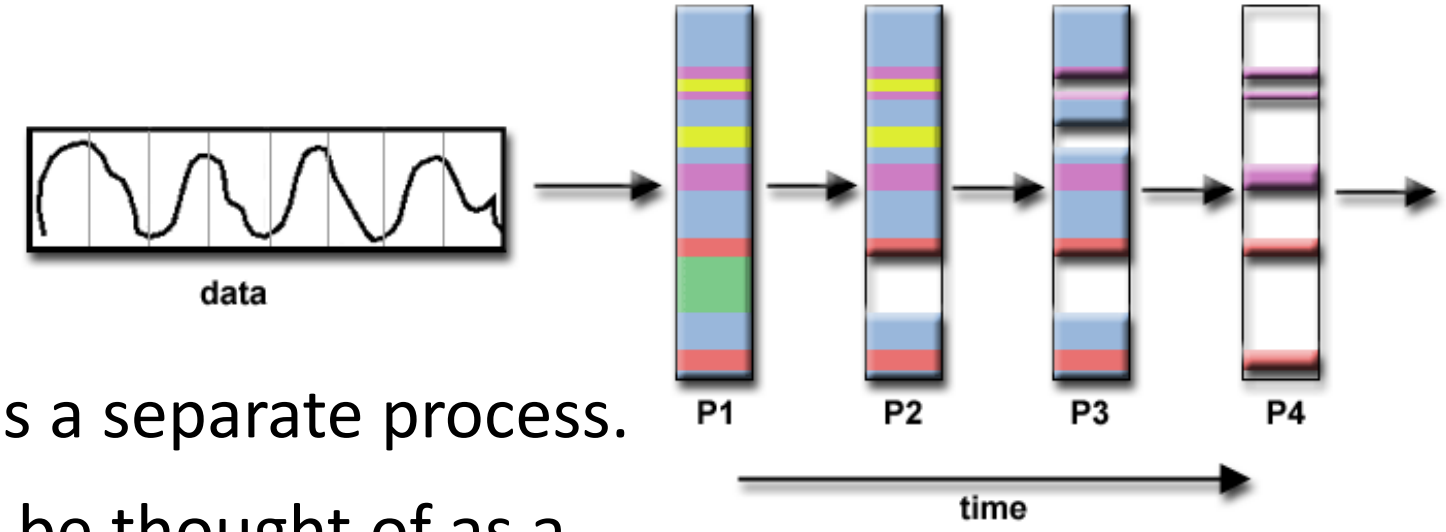



- The focus is on the computation that is to be performed rather than on the data manipulated by the computation.
- Example 1: Each program calculates the population of a given group, where each group's growth depends on that of its neighbors.
  For each time stamp, each process calculates its current state, then exchanges information with the neighbor populations.
- Ex 2: nbody, with each body associated with a process.

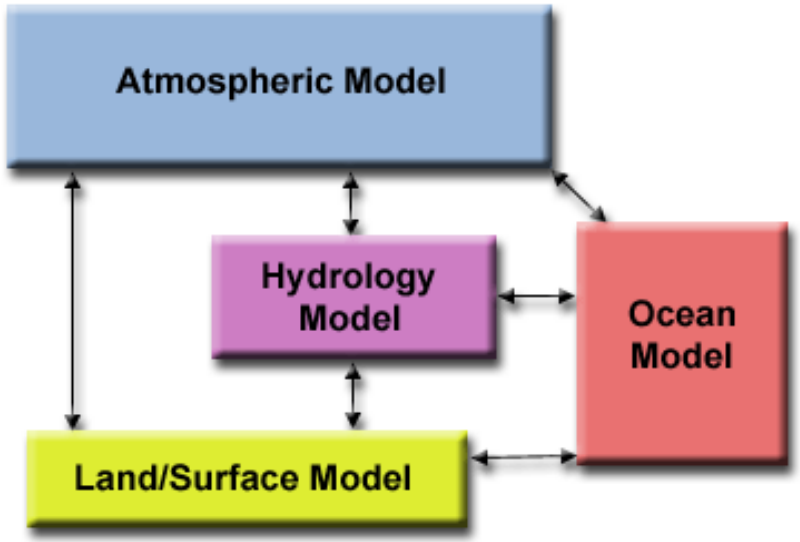


# Partitioning – Functional decomposition

- Ex 3: Signal/image processing
  An audio signal data set is passed through four distinct computational filters. Each filter is a separate process.




- Ex 4: Each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are use by the atmosphere model, and so on.



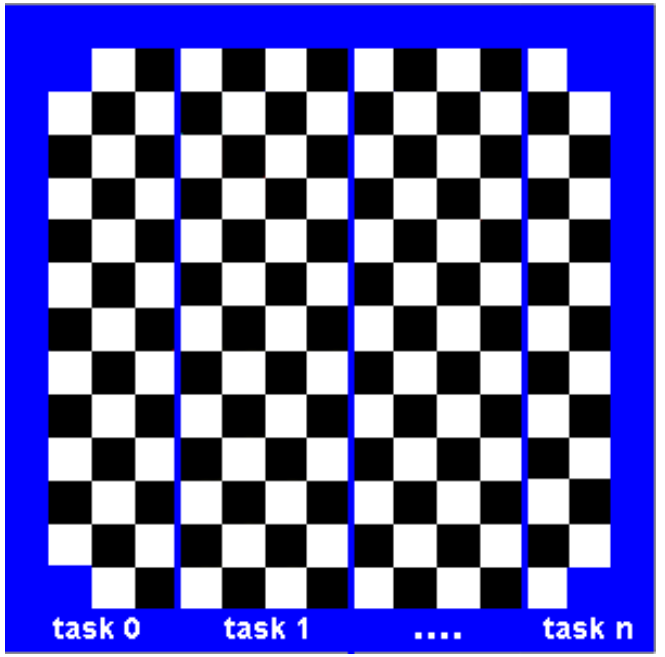
# Partitioning design checklist

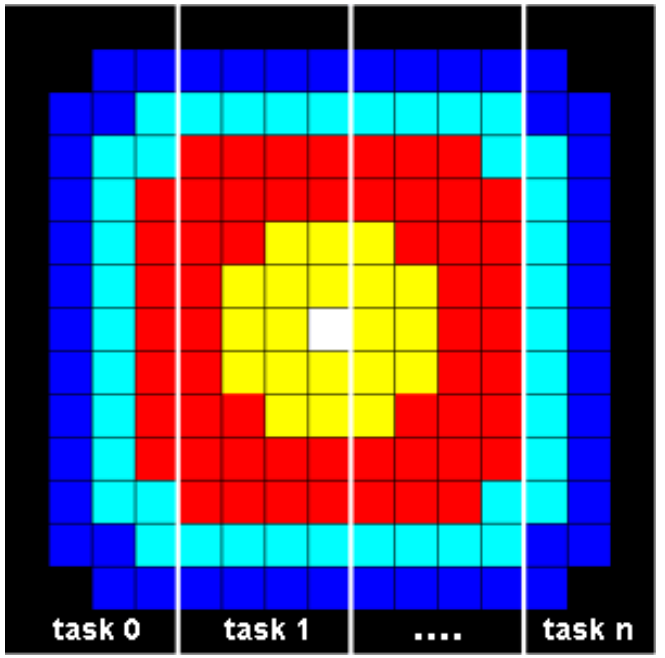
Before proceeding to evaluate communication requirements, you can use the following checklist to ensure that the partitioning you designed has no obvious flaws. Generally, all these questions should be answered in the affirmative.

- Does your partition define at least an order of magnitude more tasks than there are processors in your target computer? If not, you have little flexibility in subsequent design stages.
- Does your partition avoid redundant computation and storage requirements? If not, the resulting algorithm may not be scalable to deal with large problems.
- Are tasks of comparable size? If not, it may be hard to allocate each processor equal amounts of work.
- Does the number of tasks scale with problem size? Ideally, an increase in problem size should increase the number of tasks rather than the size of individual tasks. If this is not the case, your parallel algorithm may not be able to solve larger problems when more processors are available.
- Have you identified several alternative partitions? You should do it now.  And remember to investigate both domain and functional decompositions.

# Communications

- Communications between tasks depends upon the problem
- No Need for communications
  - Problems that can be decomposed and executed in parallel with virtually no need for tasks to share data.
  - Often called embarrassingly parallel because they are so straightforward. E.g. reverse the colors of a chessboard.
- Need for communication
  - Most parallel applications require tasks to share data
  - Example: a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.



# What Factors to Consider?

Cost of Communications

- Inter-task communication always implies overhead
- Resources are used to package/transmit data instead of computation
- Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work
- Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems

Latency vs. Bandwidth

- latency is the time it takes to send a minimal (0 byte) message from point A to point B.
- bandwidth is the amount of data that can be communicated per unit of time (bytes/sec).
- Sending many small messages can cause latency to dominate communication overheads.
- Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth

# What Factors to Consider?

Visibility of communications

- With the Message Passing Model, communications are explicit and under the control of the programmer
- With the Data Parallel Model, communications (e.g. shared data access) often occur transparently to the programmer.

Synchronous vs. asynchronous communications

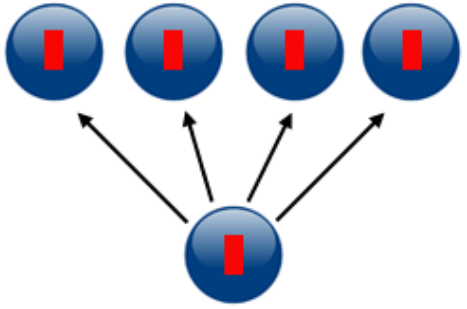
- Synchronous communications require handshaking between tasks that are sharing data.
- Synchronous communications are blocking since other work must wait until the communications have completed. E.g. phione call.
- Asynchronous communications allow tasks to transfer data independently from one another.
- Asynchronous communications are non-blocking since other work can be done while the communications are taking place. E.g. whatsapp
- Interleaving computation with communication is the single greatest benefit for using asynchronous communications

# Scope of the Communication

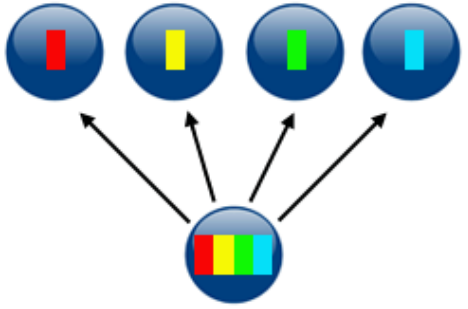
Knowing which tasks must communicate with each other is critical during the design stage of a parallel code.

***Point-to-point*** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.

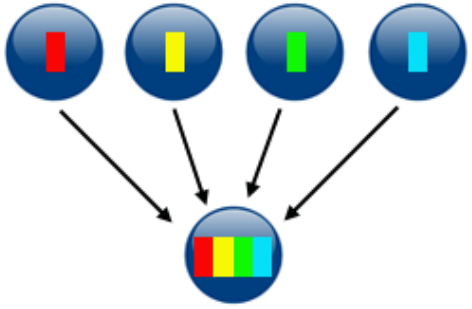
***Collective*** - involves data sharing between more than two tasks, which are often specified as being members in a common group.



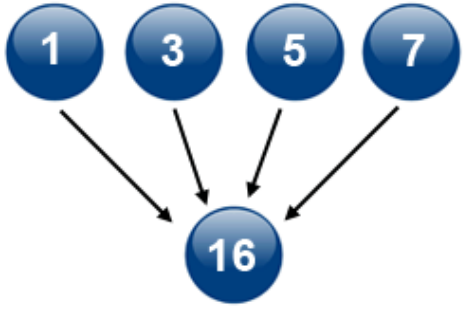
broadcast



scatter



gather




reduction

# Efficiency of communications

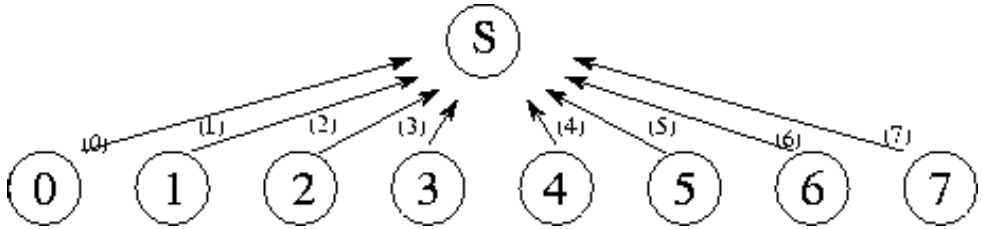
- Oftentimes, the programmer has choices that can affect communications performance. Only a few are mentioned here.
- Which implementation for a given model should be used? Using the Message Passing Model as an example, one MPI implementation may be faster on a given hardware platform than another.
- What type of communication operations should be used? As mentioned previously, asynchronous communication operations can improve overall program performance.
- Network fabric - different platforms use different networks. Some networks perform better than others. Choosing a platform with a faster network may be an option.

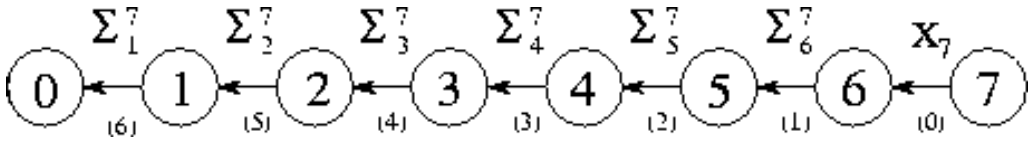
# Communication design checklist

- Do all tasks perform about the same number of communication operations? Unbalanced communication requirements suggest a nonscalable construct. For example, if a frequently accessed data structure is encapsulated in a single task, consider distributing or replicating this data structure.
- Does each task communicate only with a small number of neighbors? If each task must communicate with many other tasks, evaluate the possibility of formulating this global communication in terms of a local communication structure or use optimized, collective oprations (e.g. MPI_Reduce).
- Are communication operations able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable. Try to use divide-and-conquer techniques.
- Is the computation associated with different tasks able to proceed concurrently? If not, your algorithm is likely to be inefficient and nonscalable. Consider whether you can reorder communication and computation operations.

# Example

- A centralized summation algorithm that uses a central manager task (S) to sum N numbers distributed among N tasks.
- We can distribute the summation of the *N* numbers by making each task compute $S_i = X_i + S_{i-1}$

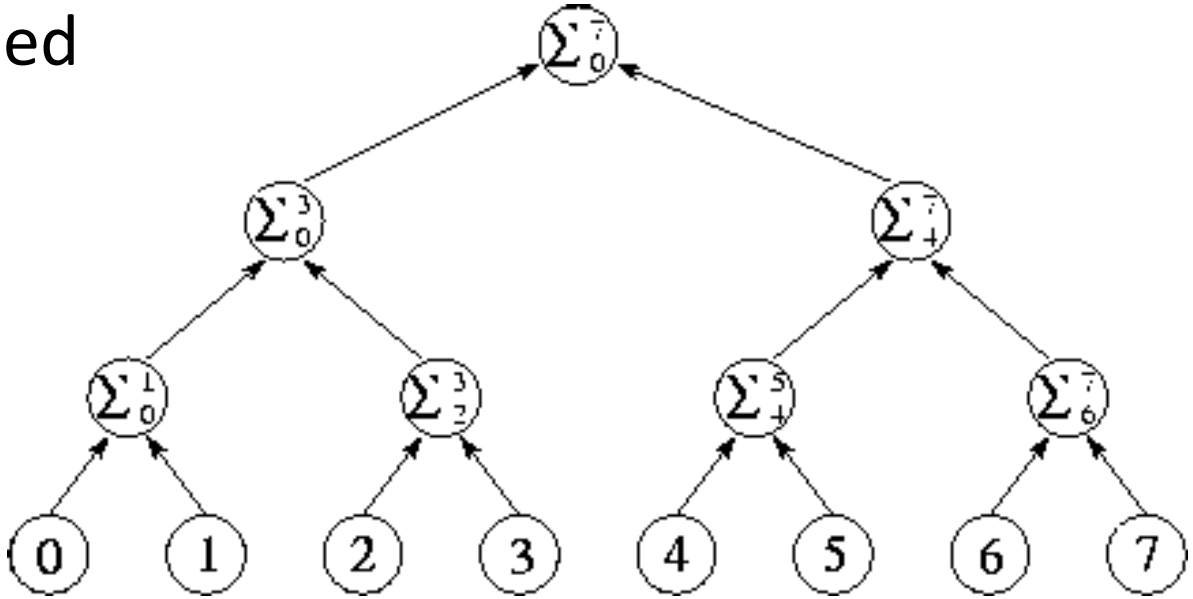


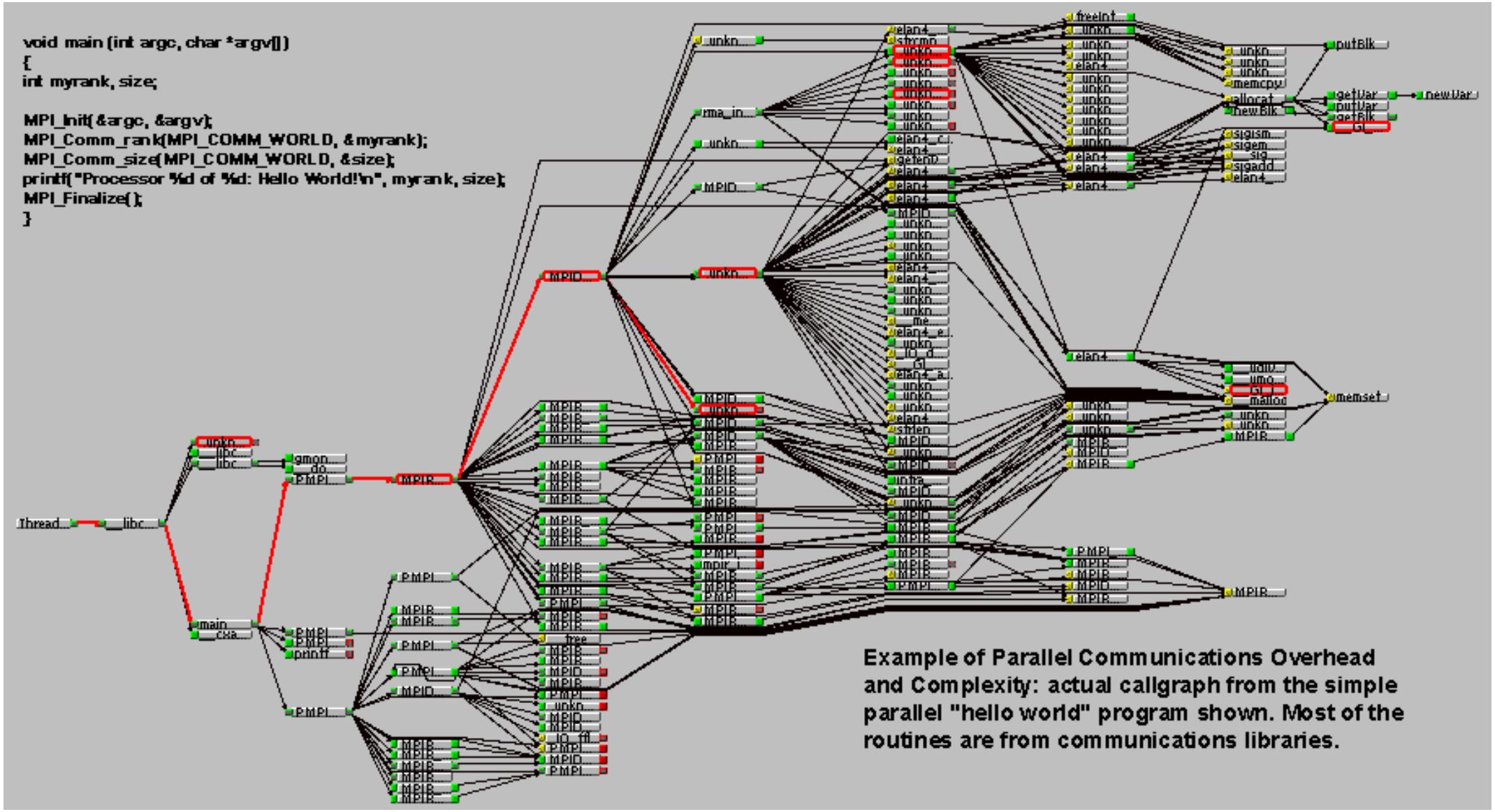

- This algorithm distributes the *N-1* communications and additions, but permits concurrent execution only if multiple summation operations are to be performed. A single summation still takes *N-1* steps.

# Example

- The *divide and conquer* problem-solving strategy: we seek to partition recursively a complex problem into two or more simpler problems of roughly equivalent size
- we have distributed the *N-1* communication and computation operations required to perform the summation and have modified the order in which these operations are performed so that they can proceed concurrently
  - Regular communication structure
  - Each task communicates with a small set of neighbors
  - The solution scales



# Overhead and complexity



# Synchronization

Barrier

- Usually implies that all tasks are involved
- Each task performs its work until it reaches the barrier, then it "blocks"
- When the last task reaches the barrier, all tasks are synchronized before the next operation

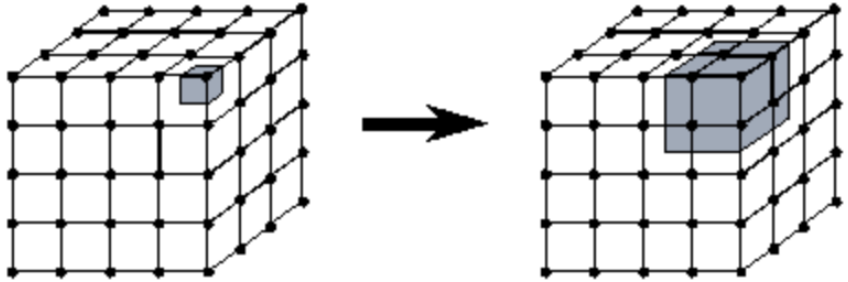
Lock/Semaphore

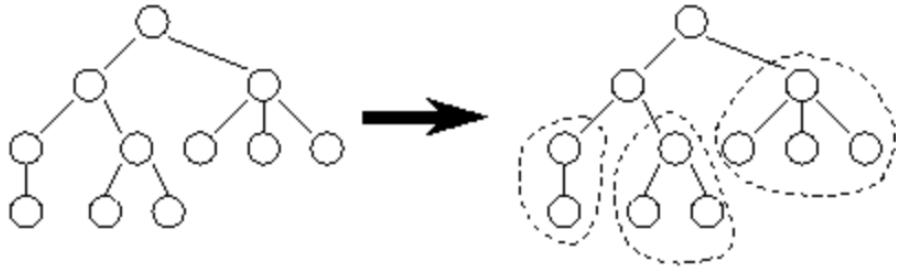
- Can involve any number of tasks
- Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use the lock/semaphore/flag
- Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.

Synchronous communication operations

- Involves only those tasks executing a communication operation
- When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication.

# Agglomeration





- In the first two stages of the design process, we partitioned the computation to be performed into a set of tasks and introduced communication to exchange data required by these tasks.
- The resulting algorithm is still abstract in the sense that it is not specialized for efficient execution on any particular parallel computer.
  - it may be highly inefficient if we create many more tasks than there are processors on the target computer and this computer is not designed for efficient execution of small tasks.
- Here we move from the abstract toward the concrete
- it is useful to combine tasks in a smaller number of greater size tasks?
- It is worthwhile to *replicate* data and/or computation wrt to communications?

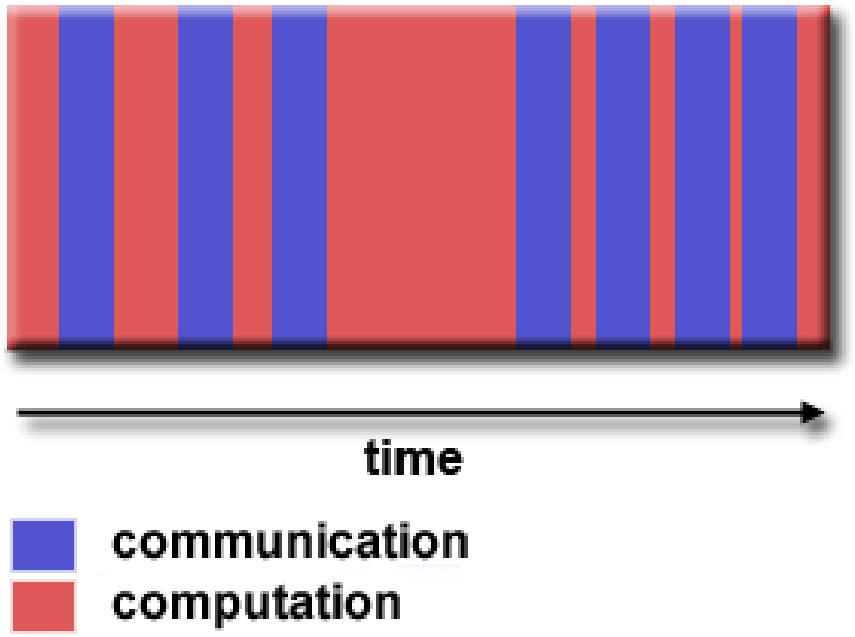
# Increasing Granularity

- In the partitioning phase we focus on defining as many tasks as possible.
- But such a large number of fine-grained tasks does not necessarily produce an efficient parallel algorithm.
  - Communication costs: we have to stop computing in order to communicate
  - Task creation cost
  - Vectorization and cache exploitation
- We can sometimes trade off replicated computation for reduced communication requirements and/or execution time.

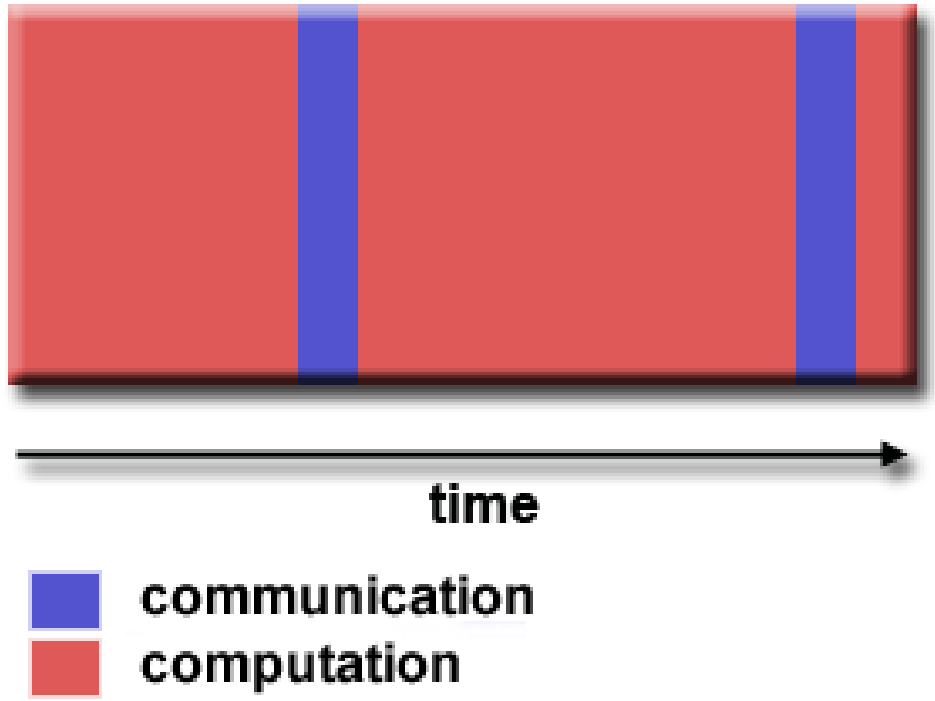
# Granularity

It is a qualitative measure of the ratio of computation to communication.

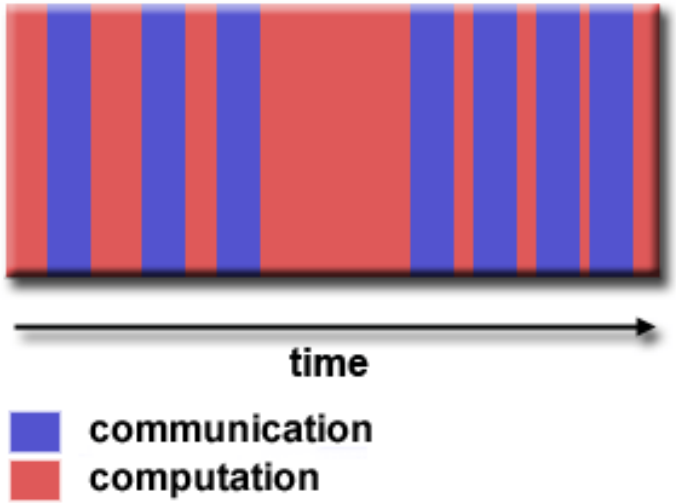
Fine-grain parallelism




Coarse-grain parallelism

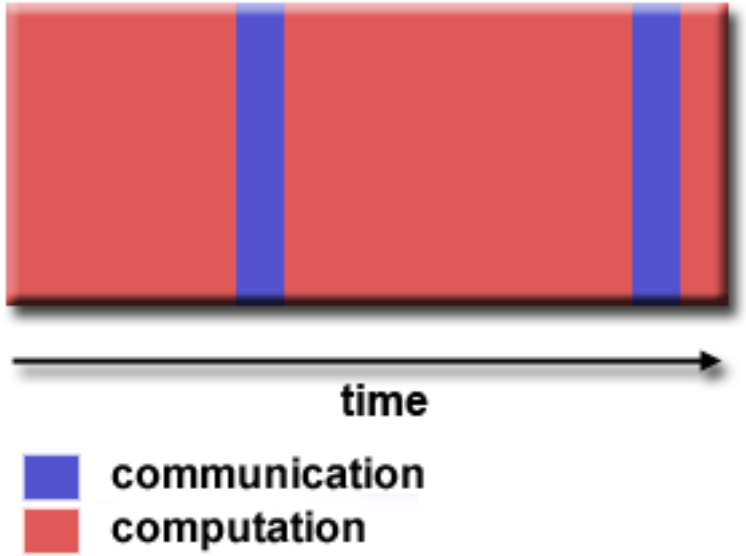


# Fine-grain Parallelism




- Relatively small amounts of computational work are done between communication events
- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.

# Coarse-grain Parallelism




- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently

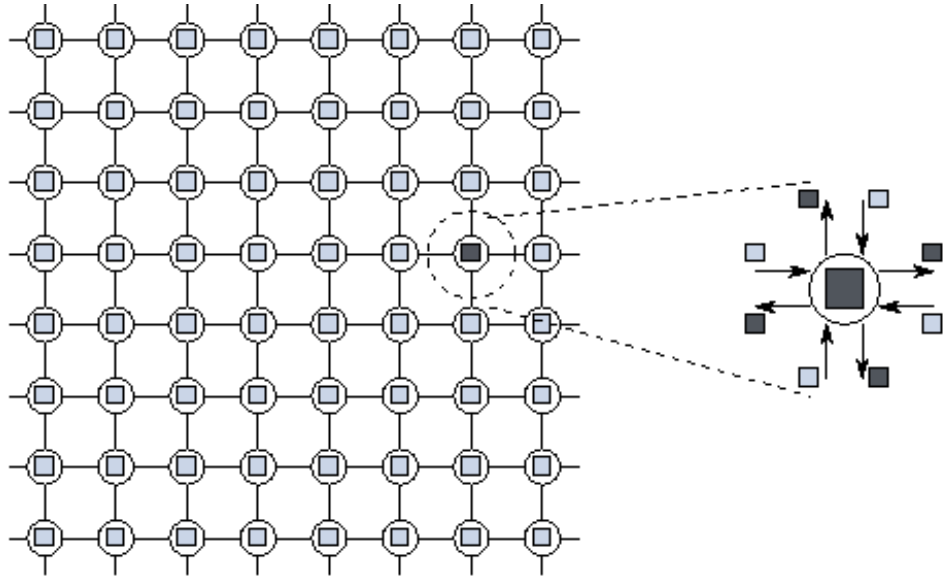
Which is Best?

- It depends on the algorithm and the hardware environment in which it runs.
- In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
- Fine-grain parallelism can help reduce overheads due to load imbalance.

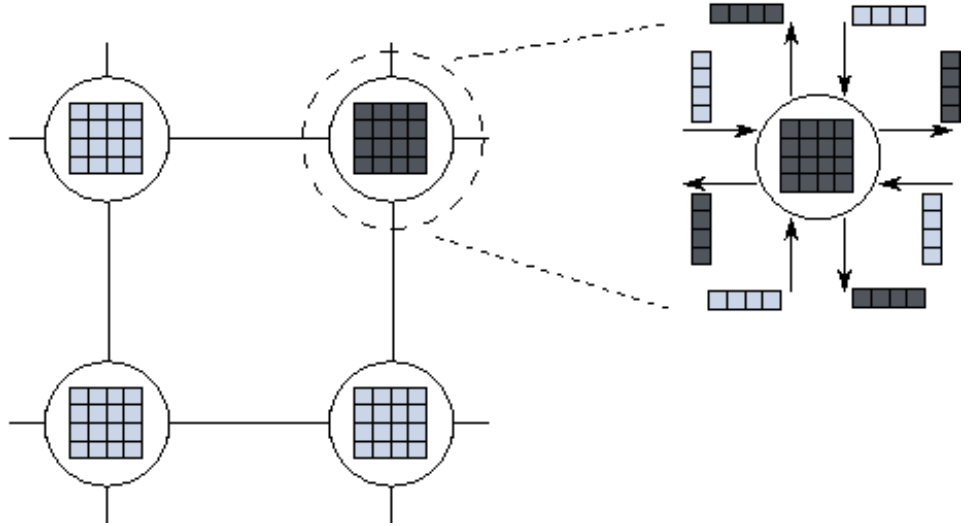
# Examples

- 4 send/rec for updating 1 point (5 point stencil)
- A) 64 task, about 224 communications (36*4 + 24*3 + 4*2)
- B) 4 tasks, 8 communications

(a)



(b)

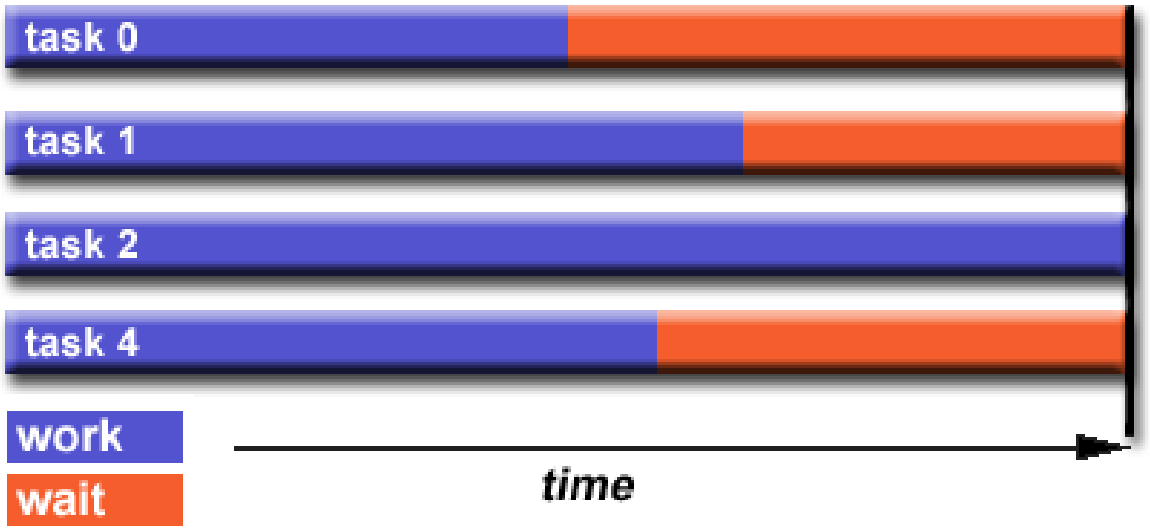


# Agglomeration Design Checklist

- Has agglomeration reduced communication costs by increasing locality? If not, examine your algorithm to determine whether this could be achieved using an alternative agglomeration strategy.
- Has agglomeration yielded **tasks with similar computation and communication costs**? The larger the tasks created by agglomeration, the more important it is that they have similar costs. If we have created just one task per processor, then these tasks should have nearly identical costs.
- Does the number of tasks still scale with problem size? If not, then your algorithm is no longer able to solve larger problems on larger parallel computers.
- Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability? Other things being equal, algorithms that create fewer larger-grained tasks are often simpler and more efficient than those that create many fine-grained tasks.

# Load Balancing

- Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



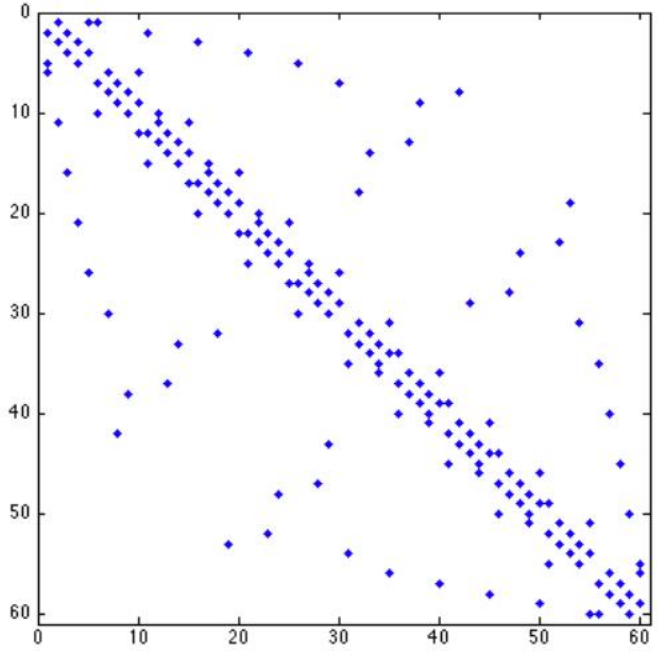
# How to Achieve Load Balance

Equally partition the work each task receives

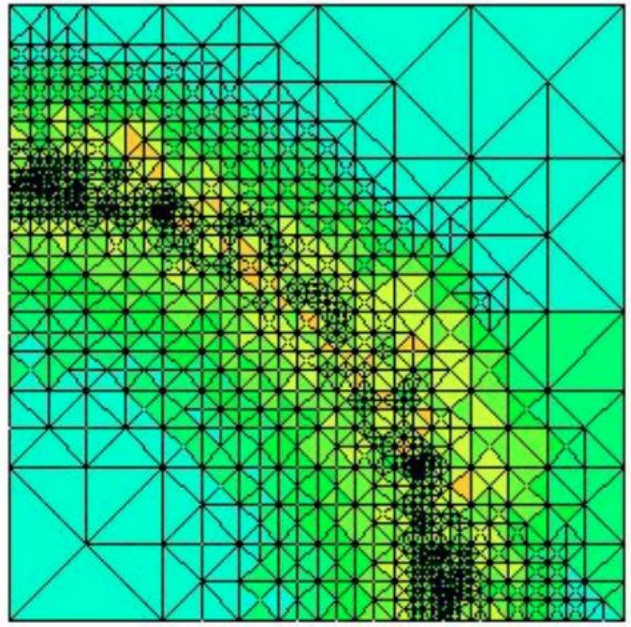
- For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
- For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
- If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool.
  - For example you can perform benchmarks to weight the machines and assign the workload accordingly. It is a static (pre-defined) load balancing strategy.

# How to Achieve Load Balance

Certain classes of problems result in load imbalances even if data is evenly distributed among tasks




Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".

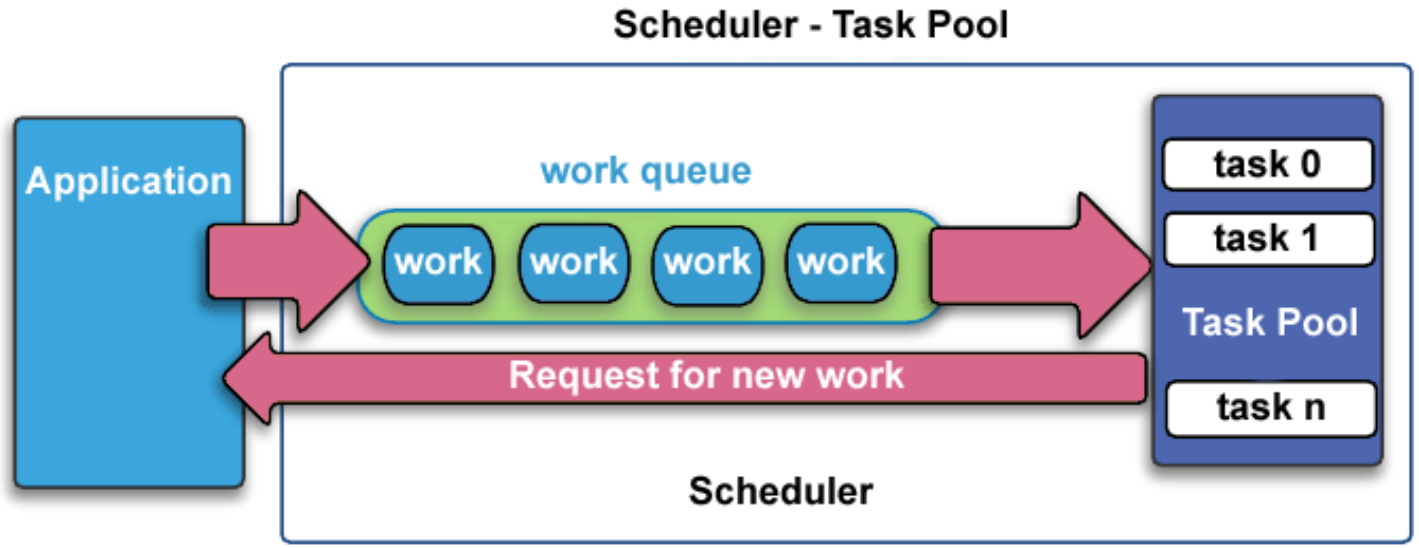


Adaptive grid methods - some tasks may need to refine their mesh while others don't.

# Use Dynamic Work Assignment

When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a scheduler - task pool approach. As each task finishes its work, it queues to get a new piece of work.

- It is the **master-slave/worker paradigm** where the scheduler is a master process (not necessarily process 0)



# Mapping

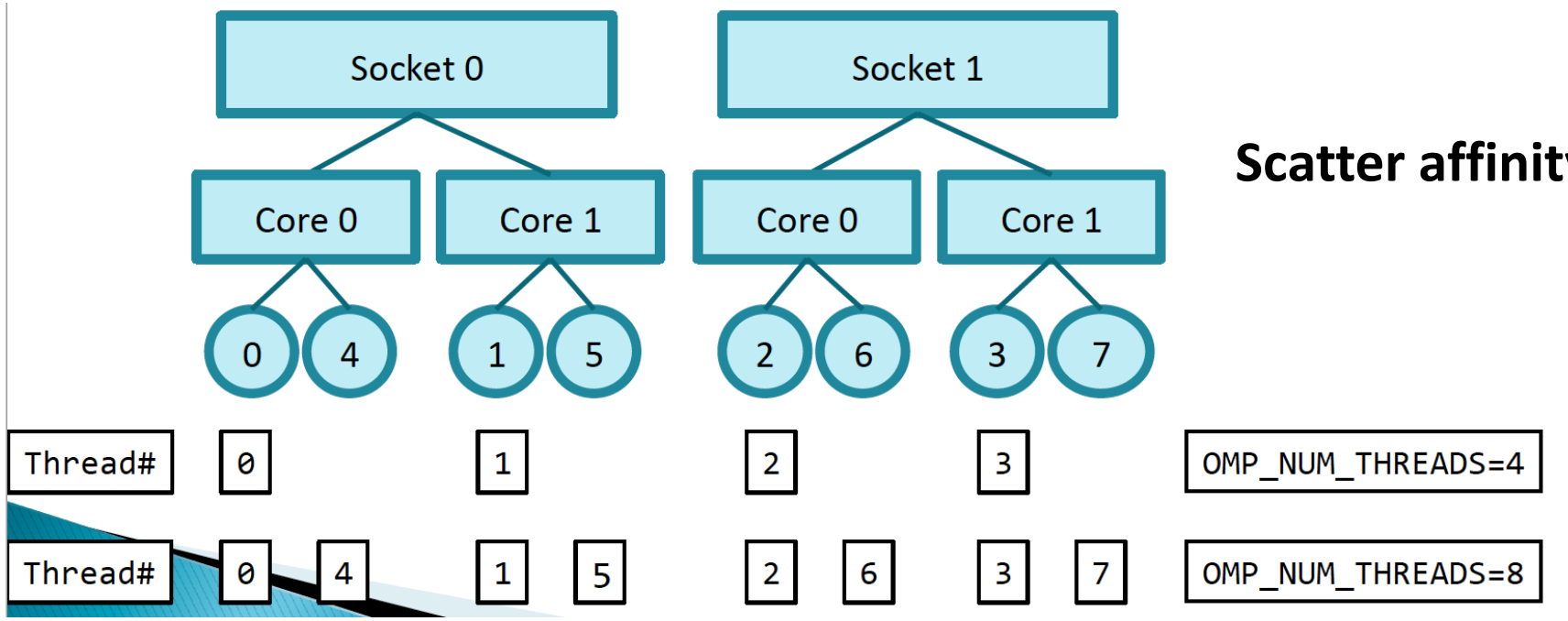
- In the fourth and final stage of the parallel algorithm design process, we specify where each task is to execute.
- Our goal in developing mapping algorithms is normally to minimize total execution time. We use two strategies to achieve this goal:
  - We place tasks that are able to execute concurrently on *different* processors, so as to enhance concurrency.
  - We place tasks that communicate frequently on the *same* processor, so as to increase locality.
- Two important aspects
  - which cores our threads/processes are running on (affinity)
  - where memory has been allocated (NUMA effects)

# NUMA

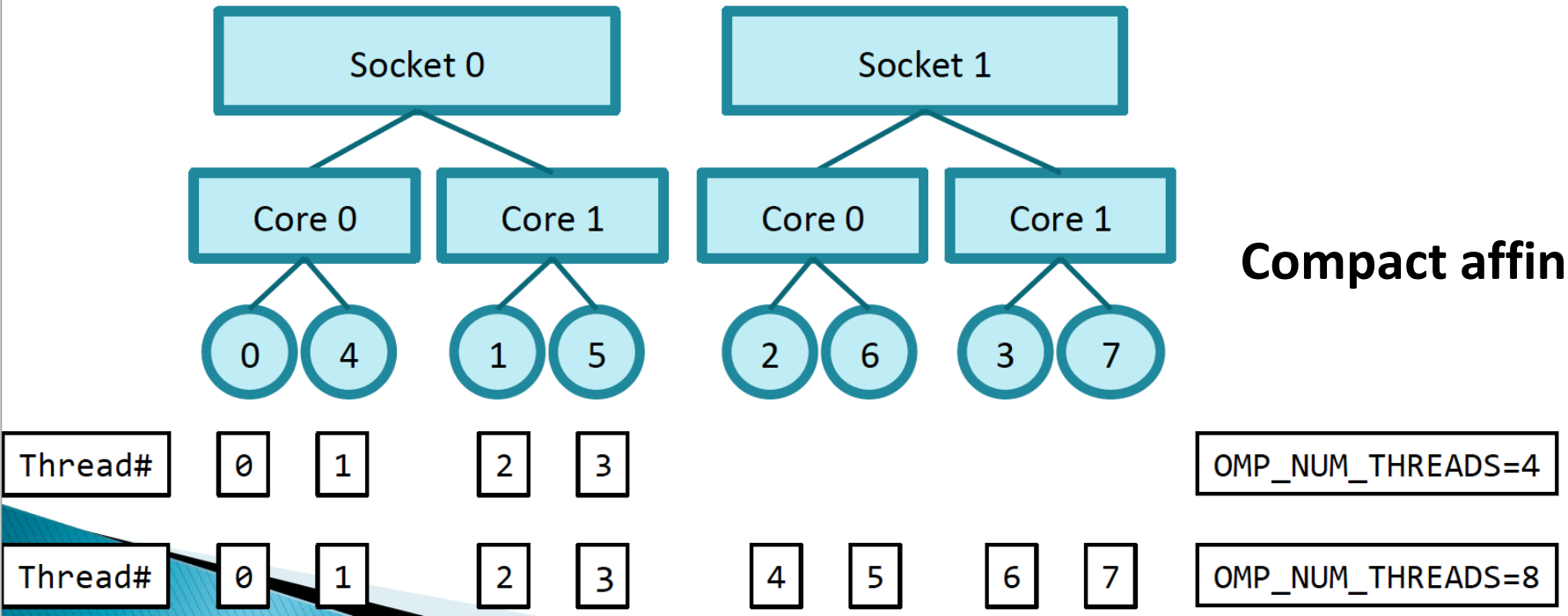
- In an OpenMP application running on a node, the threads running on any socket see one unified memory space, and can read and write to memory that is local to other sockets/dies.
  - The memory is shared between the different sockets on a node.
  - The time taken to access memory on a different socket (non-local access) is slower than time taken to access local memory.
- This memory architecture is called Non-Uniform Memory Access (NUMA)
- "NUMA effects" arise when threads excessively access memory on a different NUMA domain.
- Numa effects are to be avoided!

# Affinity

- The concept of affinity is important to reduce NUMA-effects.
- CPU affinity is the pinning of a process or thread to a particular core
  - If the operating system interrupts the task, it doesn't migrate it to another core, but waits until the core is free again
  - For most HPC scenarios where only one application is running on a node, these interruptions are short
- Memory affinity is the allocation of memory as close as possible to the core on which the task that requested the memory is running
- Both CPU affinity and memory affinity are important if we are to maximise memory bandwidth on NUMA nodes
  - If memory affinity is not enabled then bandwidth will be reduced as we go off-socket to access remote memory
  - If CPU affinity is not enabled then allocating memory locally is of no use when the task that requested the memory might no longer be running on the same socket




**Scatter affinity**




**Compact affinity**

# Parallel examples

- See here https://computing.llnl.gov/tutorials/parallel_comp/#Examples
- My PhD Thesis – see the paper on Aulaweb