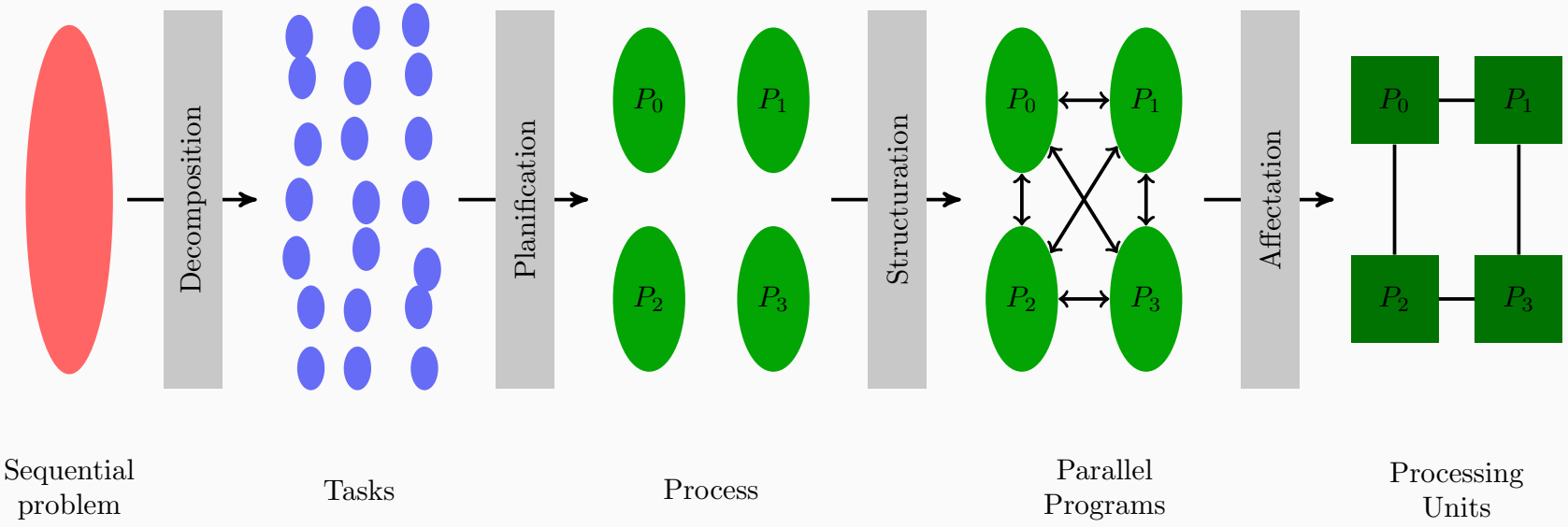
GP-GPU and High Performances Computing

Lecture 09 – Performance Evaluation

December 18, 2023

Previsouly


Decomposition
Planification
P_0
P_1
P_2
P_3
Structuration
P_0
P_1
P_2
P_3
Affectation
P_0
P_1
P_2
P_3
Sequential problem
Tasks
Process
Parallel Programs
Processing Units

Computing performance

Performance is defined by 2 factors

- Computational requirements (what needs to be done)
- Computing resources (what it costs to do it)

Computational problems translate to requirements

Factors: hardware, time, energy, money

Why is it important

- Performance itself is a measure of how well the computational requirements can be satisfied
- We evaluate performance to understand the relationships between requirements and resources $\Longrightarrow$ Decide how to change methodology to target objectives
- Performance measures reflect decisions about how and how well approaches are able to satisfy the computational requirements

Some definition of parallelism

Performance issues when using a parallel computing environment: Performance with respect to parallel computation

- Performance is why we do parallelism
- Parallel performance versus sequential performance
- If the “performance” is not better, parallelism is not necessary

Parallel processing includes techniques and technologies necessary to compute in parallel: Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools, …

Parallelism must deliver performance: How? How well?

What can we expect?

If each processor is rated at $k - GFlops$ and there are p processors, should we see $kpGFlops$ performance?

If it takes 100 seconds on 1 processor, shouldn't it take 10 seconds on 10 processors?

Several causes affect performance

- Each must be understood separately
- But they interact with each other in complex ways: solution to one problem may create another or one problem may mask another

Scaling (system, problem size) can change conditions

Need to understand performance space

Analytical measure

Embarrassingly Parallelism

An embarrassingly parallel computation is one that can be obviously divided into completely independent parts that can be executed simultaneously

- In a truly embarrassingly parallel computation there is no interaction between separate processes
- In a nearly embarrassingly parallel computation results must be distributed and collected/combined in some way

Embarrassingly parallel computations have potential to achieve maximal speedup on parallel platforms

- If it takes T time sequentially, there is the potential to achieve T/P time running in parallel with P processors
- What would cause this not to be the case always?

Scalability

- A program can scale up to use many processors: What does that mean?
- How do you evaluate scalability?
- How do you evaluate scalability goodness?
- Comparative evaluation: if double the number of processors, what to expect? Is scalability linear?
- Use parallel efficiency measure: is efficiency retained as problem size increases?
- Apply performance metrics

A measure of performance

Evaluation

- Sequential runtime T_{seq} is a function of problem size and architecture
- Parallel runtime T_{par} is a function of problem size, parallel architecture and number of processors used in the execution
- Parallel performance affected by algorithm and architecture

Definition (Scalability)
Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

Performance Metrics and Formulas

- T_1 is the execution time on a single processor
- T_p is the execution time on a p processor system
- $S(p)$ is the speedup

$$S(p) = T_1/T_p$$

- $E(p)$ is the efficiency

$$Efficiency = S_p/p$$

- $Cost(p)$ is the cost

$$Cost = pT_p$$

- Parallel algorithm is cost-optimal
  Parallel time = sequential time ($C(p) = T_1$, $E(p) = 100\%$)

Performance metrics

Time execution

Optimizing an application of fixed size is difficult

- If parallel management cost are high (synchronization and communication) parallel acceleration low.
- If complexity (spatial or temporal) of parallel approach is much greater than the best sequential algorithm, final gain is weak.

Time representation

If parallel algorithm has

- decreasing execution time
- limited overhead on one resource

final gain might be important

but analysis of should be continued to consider Execution time vs Ideal execution time, speedup, efficiency, size-up and scalability,...

how to represent time execution

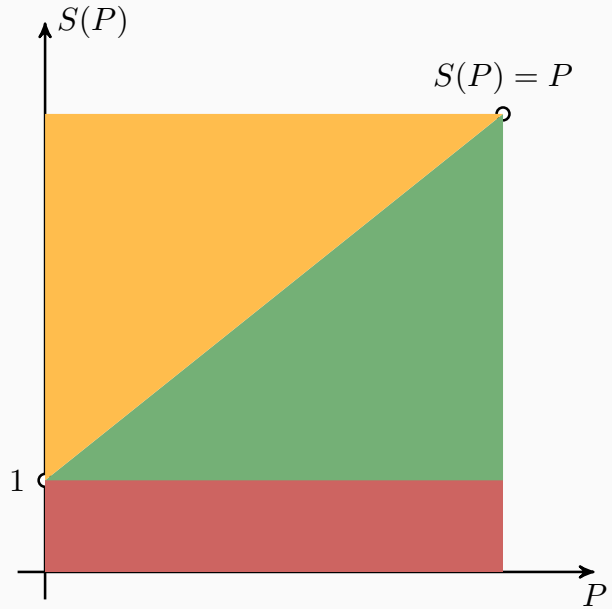
It is best to have an idea of the plot and choose a representation accordingly to identify simple shapes : N^3, N^2, $\log(N)$, with the appropriate N

For parallel execution, the ideal curve is $T(P) = T(1)/P$ but it is hard to identify simply, hence one should choose an other representation.

We will prefer to have the representation $\log(T(P)) = \log(T(1)) - \log(p)$ or use a log scale on the graph $\longrightarrow$ easier to detect distance to theory $\longrightarrow$ easier to detect distance to other laws

Speedup

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



- $S(P) < 1$: parallelism is erroneous
- $1 < S(P) < p$: normal behavior
- $S(P) > p$: super linear speedup

Super linear speedup

This observation in the measurement is an anomaly:

- It should be analyzed to provide clear explanation
- A correction should be proposed or an optimization should be operated.

Some sample explanation

- The operation performed are not correct: the result is wrong.
- Data fit in the total cache of P processors.
- The starting algorithm has been modified and convergence is faster (e.g. optimized genetic algorithm)
- Computation is based on tree exploration and the program is stopped

Efficiency metrics

Usage rate of available resources or percentage of usage from perfect speedup

$$e(P) = \frac{S(P)}{P}$$

- $e(P) \in [0; 1]$
- $e(P) > 100\%$: super-linear speedup.

How to choose sequential reference?

Algorithm

- Same program on one processor.
- Same algorithm than in sequential.
- Best sequential algorithm.

Compilation

- Sequential compilation with the same compiler?
- Compilation with the best sequential compiler?

Optimizations

- Sequential optimization allowed by parallelism.
- Best sequential optimizations

Execution

- Execution on one processor/core/unit of the parallel computer?
- Execution on the best sequential computer?

Sequential reference: algorithm

All choices are valid. Each choice depends on

- A different point of view
- A different concern
- A different objective in the analysis

One should choose accordingly to its problematic The choice should be clearly state

Examples

- Final user : his sequential program on his sequential computer
- Developer: his parallel program on one processor of his parallel computer.

Sequential reference:

The sequential reference may be obtain

- Same algorithm, same language, same sequential optimization, same processor. => Good performances, easy to get
- Best algorithm, best langage, best sequential optimization, best processor => Good performances, very hard to get

Loosing performances

- Under usage of each core
  - Sequential sub-optimization
  - No possibility to vectorize loops
- Under usage of nodes
  - Sequential fractionning
  - Overhead of synchronization
  - Overhead of communication
  - Unbalanced between tasks and loads
- Weak platform
  - Sequential IO
  - Interconnection network undersized

Size up metrics

Objective 1: work on larger problem on more resources

An application that replicate most of the data on all compute unit will always be memory bound. It won't be able to scale up.

An application that distribute most of the data on all compute unit will be able to store more data per compute unit. It will be able to scale up.

Size up metrics

Design an application with an initial distribution of data and a communication scheme with minimal volume.

- Need to have move initial data from compute unit to compute unit to allow continuation of computation on data different than its own.
- Need to have intermediate data move from compute unit to compute unit to allow continuation of computation from an other compute unit, with its own data.

Execution time

Objective 2 Maintain constant execution time

$$T(1 \times n_1, p_1) = T(2 \times n_1, p_2) = T(k \times n_1, p_k) = C$$

Where n_1 is the size of the problem, k is size scale up and p_k is the number of compute unit needed to solve the problem at constant time.

Execution time and ressources

Objective 3 Maintain constant execution time with the minimal number of ressources

$$T(1 \times n_1, p_1) = T(2 \times n_1, p_2) = T(k \times n_1, p_k) = C$$
$$\mathcal{O}(p_k) = k \times n_1$$

Criteria for scale up

Metric 1: based on T

When the size of the problem increase such that we are not able to perform sequential execution:

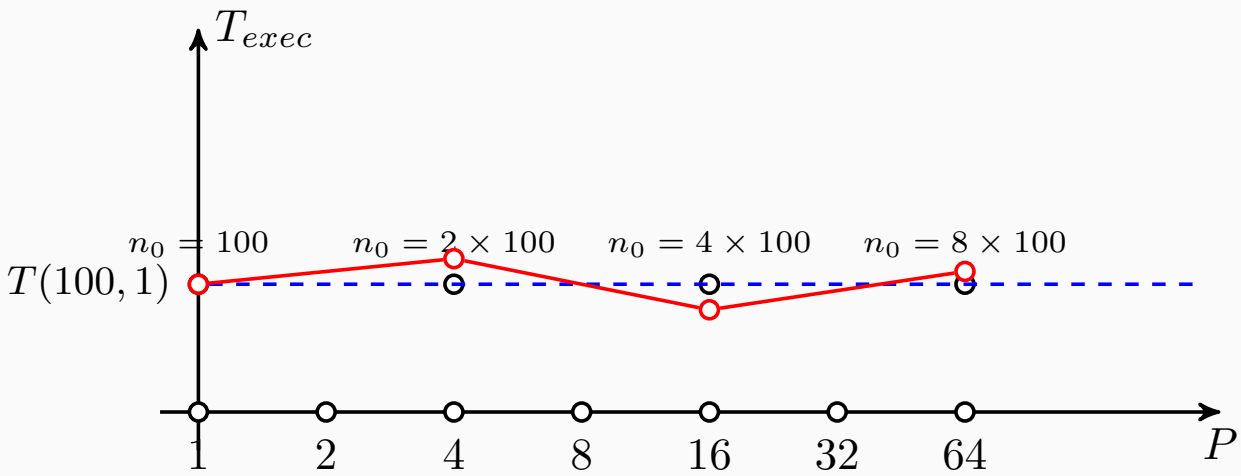
- Not enough memory, not enough storage
- Execution time too high, compute unit not available.

Simple size-up

Definition and criteria for simple up scaling

- Allow for a size up to deal with larger problem on more compute unit
- Beware of energy consomption and other costs

Size up with constant time and minimal compute unit for complexity $\mathcal{O}n^2$

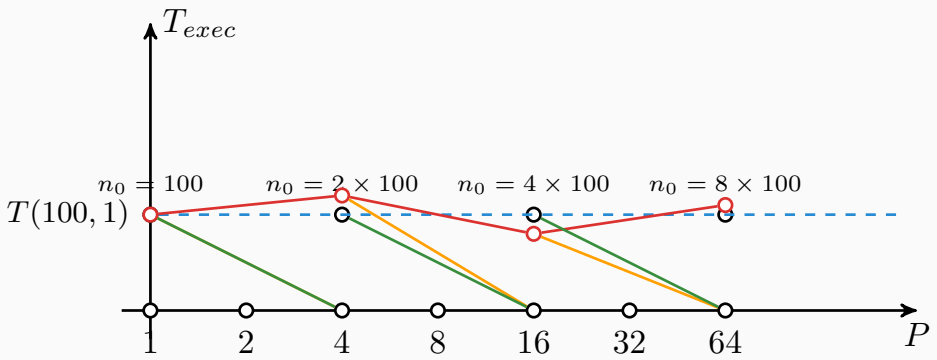


Complete size-up

Definition and criteria for complete up-scaling

- Allow for a size up to deal with larger problem on more compute unit
- Beware of energy consomption and other costs
- Allow for a speedup for all sizes of problem
- Keep the same profile of time evolution

Size up with constant time and minimal compute unit for complexity $\mathcal{O}n^2$



Plotting scale-up

How to build and use a graph for scale up

- Measure $t(n, p)$ for different size of problem n and number of compute unit p.
- Plot $T(n, p)$ using $\log$ scale.

Ideal case: for each size of problem, we obtain a line parallel to others.

Validation of scale up: comparing measures to expected curves, slop and positions.

Use as an abacus: for a given problem size, we may identify a number of compute unit to use in order to respect a maximum T_{exec}

Scale up cheklist

- meet the needs of computation.
- required the least possible number of resources.
- quantify the number of required resources.
- plan for associated expanses.

Laws on performances

Amdahl's law

Let f be the fraction of a program that is sequential. $1 - f$ is the fraction that can be parallelized

Let T_1 be the execution time on 1 processor

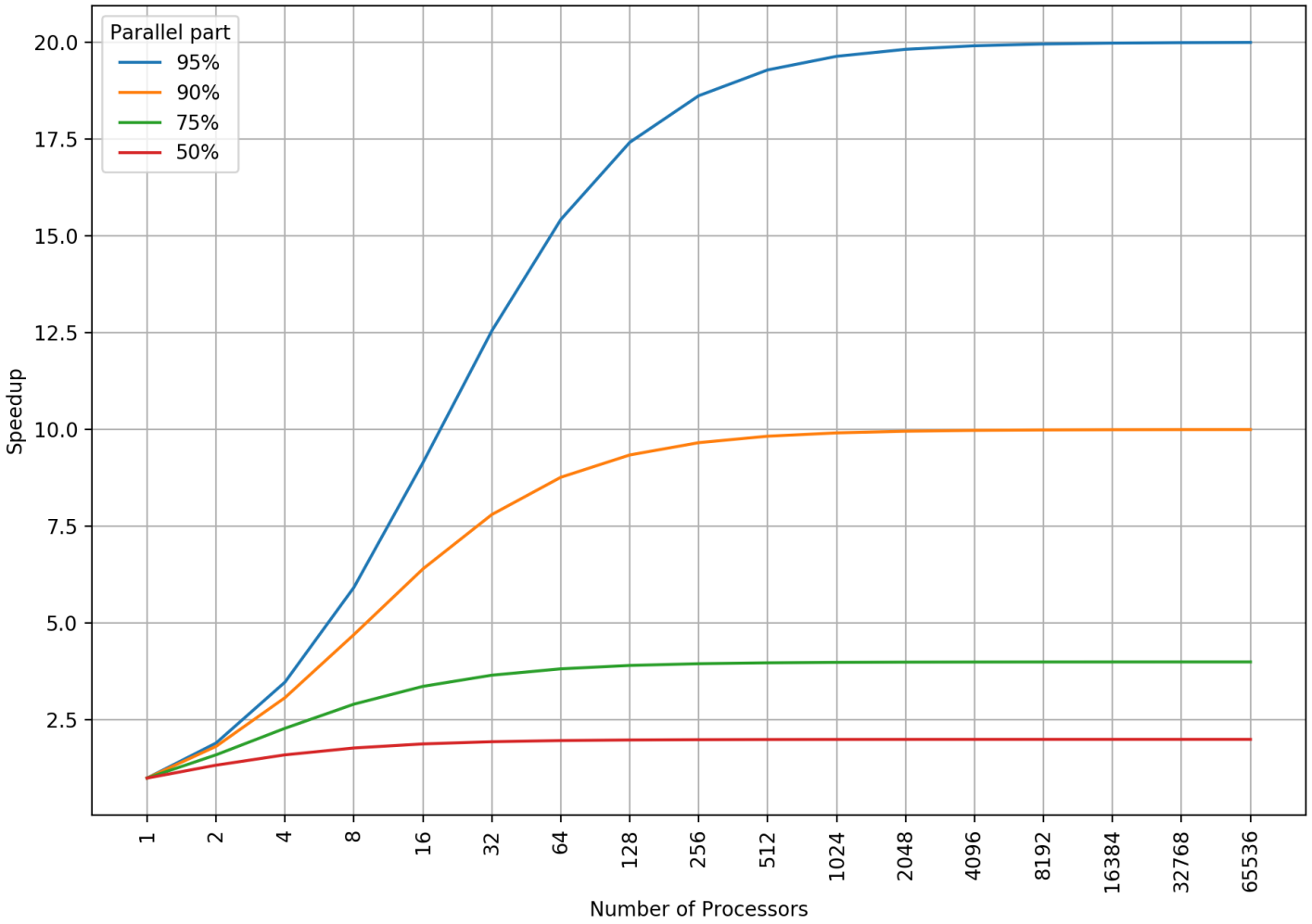
Let T_p be the execution time on p processors

S_p is the speedup

$$\begin{aligned} S_p &= T_1/T_p \\ &= T_1/(fT_1 + (1-f)T_1/p) \\ &= 1/(f + (1-f)/p) \end{aligned}$$

As $p \to \infty$, $S_p = 1/f$

Amdahl's law


Parallel part
95%
90%
75%
50%
Speedup
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors

Gustafson-Barsis' Law

Definition (Scalability)
Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

When does Amdahl's Law apply?

- When the problem size is fixed
- Strong scaling ($p \to \infty$, $S_p = S_\infty \to 1/f$)
- Speedup bound is determined by the degree of sequential execution time in the computation, not number of processors!!!
- Perfect efficiency is hard to achieve

Gustafson-Barsis' Law

- Often interested in larger problems when scaling
  - How big of a problem can be run
  - Constrain problem size by parallel time
- Assume parallel time is kept constant

$$T(p) = C = (f + (1 - f)) * C$$

- What is the execution time on one processor? Let $C = 1$, then $T(s) = f_{seq} + p(1-f_{seq}) = 1 + (p-1)f_{par}$
- What is the speedup in this case?

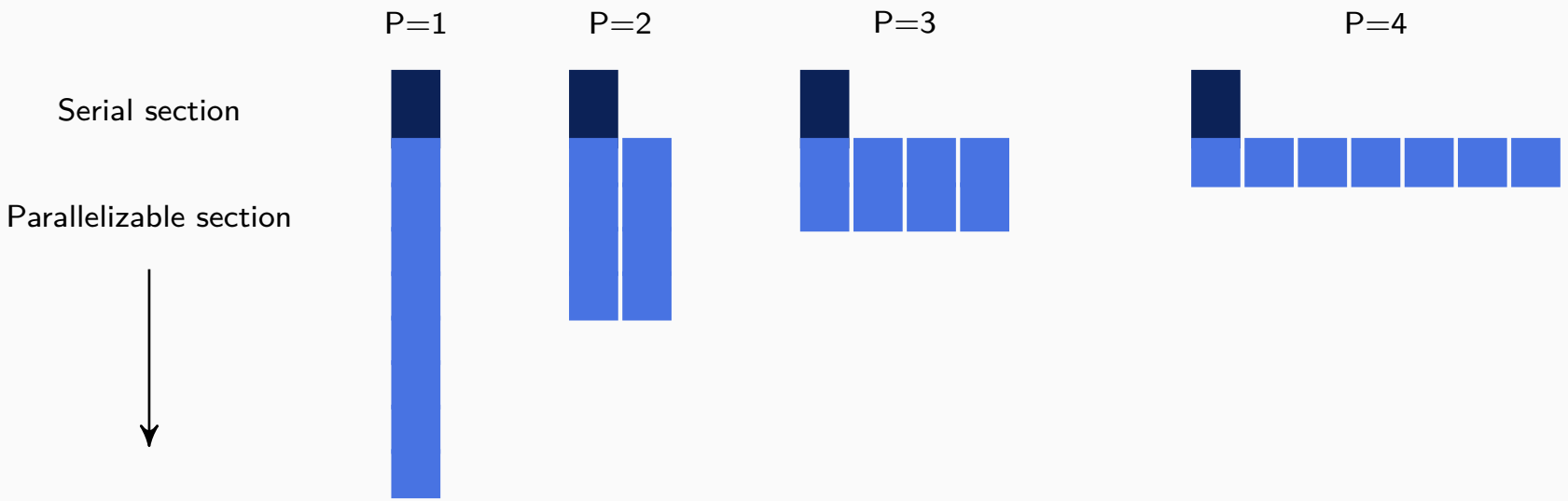
$$S(p) = T_s/T_p = T_s/1 = f_{seq} + p(1-f_{seq}) = 1 + (p-1)f_{par}$$

Definition (Scalability)
Ability of parallel algorithm to achieve performance gains proportional to the number of processors and the size of the problem

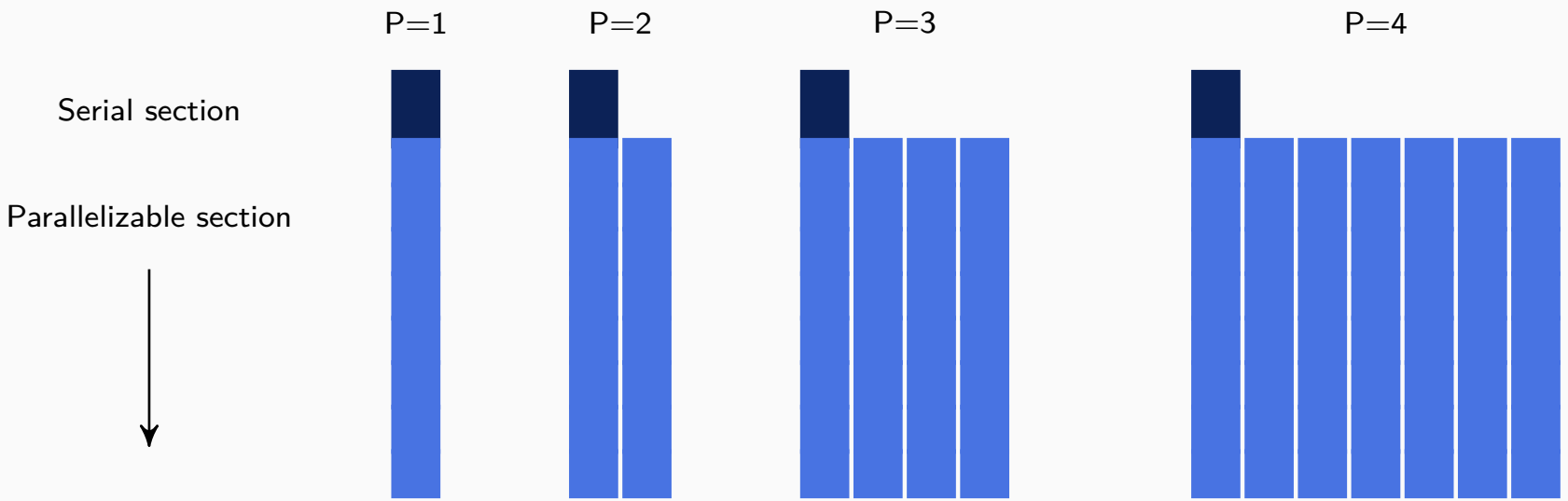
When does Gustafson's Law apply:

- When the problem size can increase as the number of processors increases
- Weak scaling ($S_p = 1 + (p-1)f_{par}$)
- Speedup function includes the number of processors!!!
- Can maintain or increase parallel efficiency as the problem scales

Amdahl's law vs Gustafson



Amdahl's law vs Gustafson



DAG Model of Computation

A program seen as directed acyclic graph (DAG) of tasks

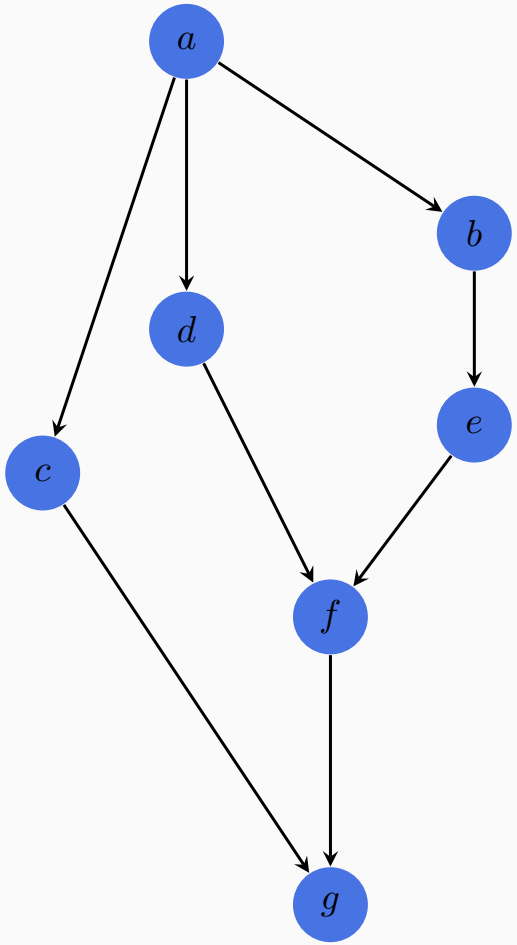
- A task can not execute until all the inputs to the tasks are available
- These come from outputs of earlier executing tasks
- DAG shows explicitly the task dependencies

Hardware consists of workers (processing units)

We consider a greedy scheduler of the DAG tasks to workers

$\Longrightarrow$ No worker is idle while there are tasks still to execute

Example of DAG



Execution time

T_P is time to execute with P workers

T_1 is time for serial execution. It is the sum of all tasks

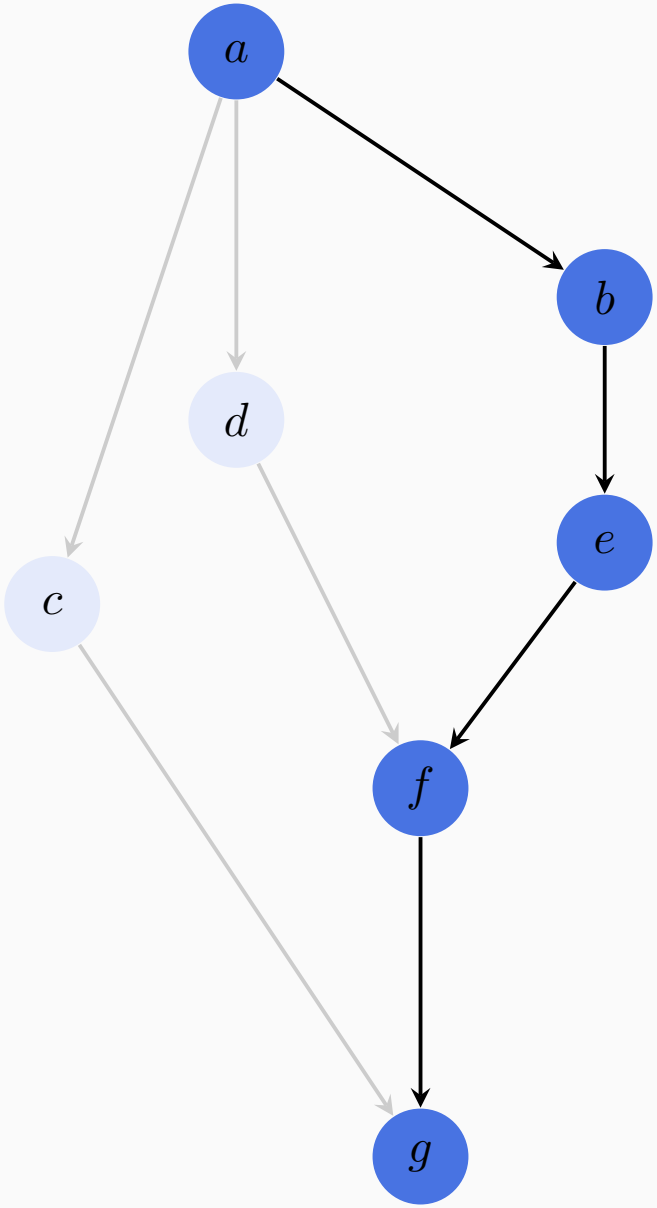
T_∞ is time along the critical path

- Sequence of task execution (path) through DAG that takes the longest time to execute
- Assumes an infinite number workers available

Example

Let each task take 1 unit of time

- $T_1 = 7$: All tasks have to be executed and in serial order
- $T_\infty = 5$: Time along the critical path
- In this case, it is the longest path length of any task order that maintains necessary dependencies



Bounds

Suppose we only have P workers. We can write a work-span formula to derive a lower bound on T_P

$$\max(T_1/P, T_\infty) \leq T_P$$

T_∞ is the best possible execution time

Theorem (Brent)
Capture the additional cost executing the other tasks not on the critical path.

Assume can do so without overhead. Then

$$T_P \leq (T_1 - T_\infty)/P + T_\infty$$

Application

- $T_1 = 7$
- $T_\infty = 5$
- For $P = 2$

$$\begin{aligned} T_2 &\leq (T_1 - T_\infty)/P + T_\infty \\ &\leq (7 - 5)/2 + T_\infty \\ &\leq 6 \end{aligned}$$

Estimation of running time

- Scalability requires that T_∞ be dominated by T_1

$$T_P \simeq T_1/P + T_\infty \text{ if } T_\infty \ll T_1$$

- Increasing work hurts parallel execution proportionately
- The span impacts scalability, even for finite P

Available parallelism

Sufficient parallelism implies linear speedup

$$T_p \sim T_1/P \text{ if } T_1/T_\infty \gg P$$

Roofline model

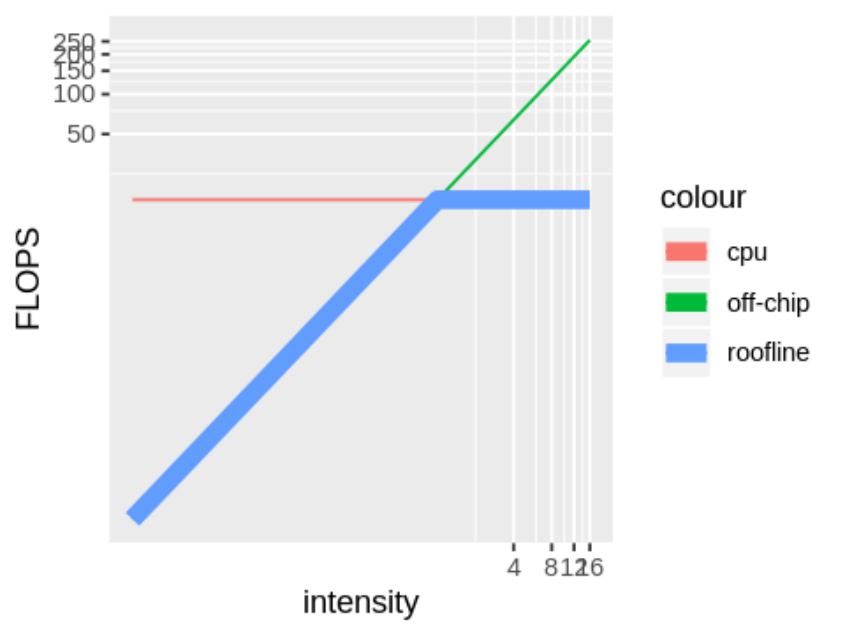
Concepts

The total number of FLOPS/s realized is bounded by either:

- the amount of data delivered to the processor times the operational intensity (GB/s ·FLOPS/GB)
- the peak processing throughput of the processor

Operational intensity is a measure of how many times you use each byte.

Limits



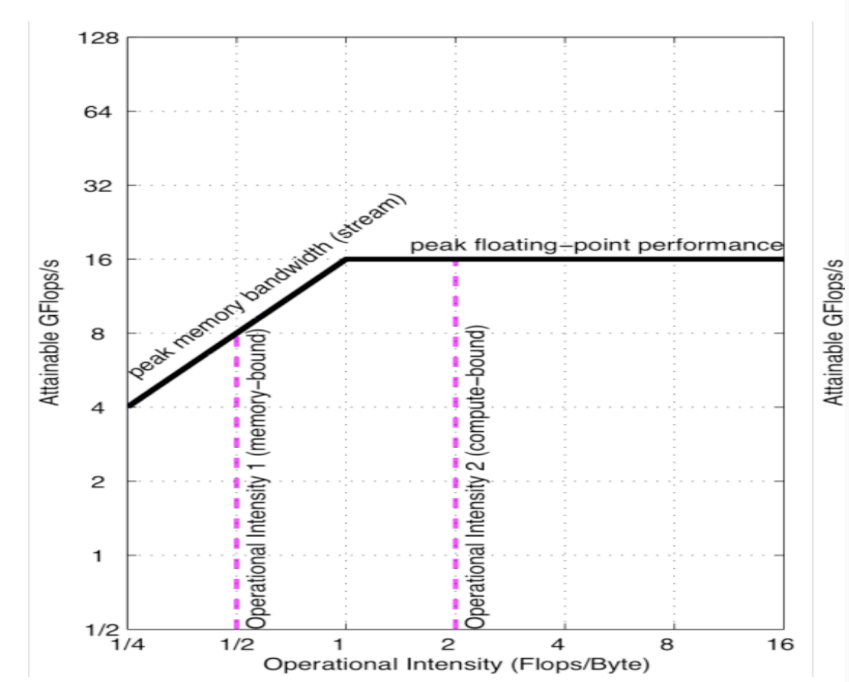
Taking the minimum of these functions describes the roofline. This is the feasible region for a computation:

> no possible program can exceed the roofline
> inefficiencies in programs may be less than the roofline

Kernels

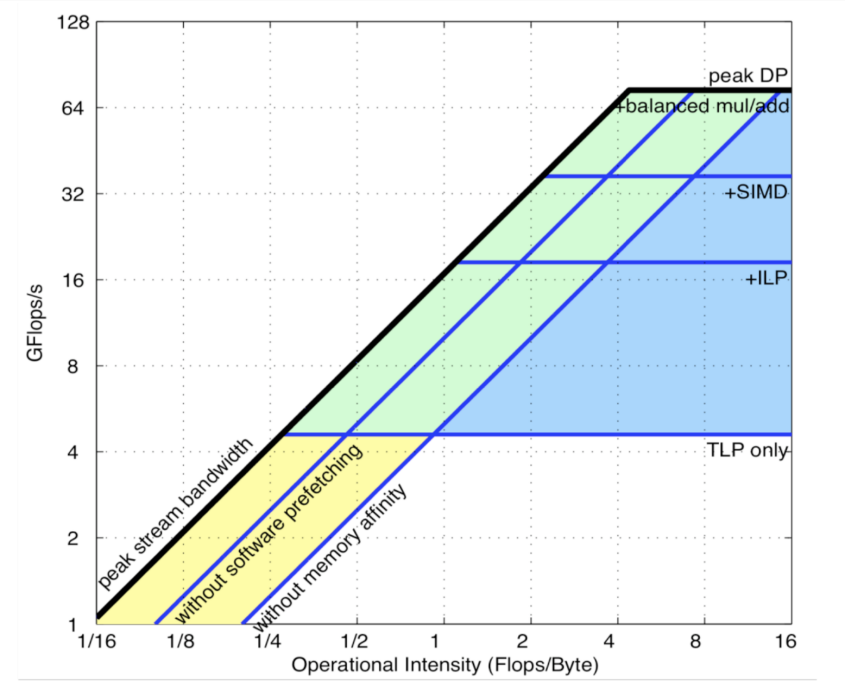
The idea is to measure a given programs operational intensity, which will correspond to a vertical line on the plot. The peak of the roofline divides the world into:

- memory-bound kernels
- CPU bound kernels



Roofline to Optimize Code

- Enhance CPU performance to raise peak.
  - unroll loops (increase compute density)
  - SIMD (vector instructions)
- Improve data movement to shift memory bound
  - sequential data access
  - coherent data access
  - software prefetching



Conclusion

Conclusion

- Different metrics for optimization.
- Metrics are associated with different objectives.
- Comparing with the correct sequential approach is critical.