Parallel Systems Course: Chapter V

Performance Analysis

See Chapter 6 of Jan's PhD

Kumar Chapter 5

Jan Lemeire
Parallel Systems lab
November 2017

Vrije Universiteit Brussel

Performance Metrics

We assume sequential version is run on the same processor/core as the parallel version.

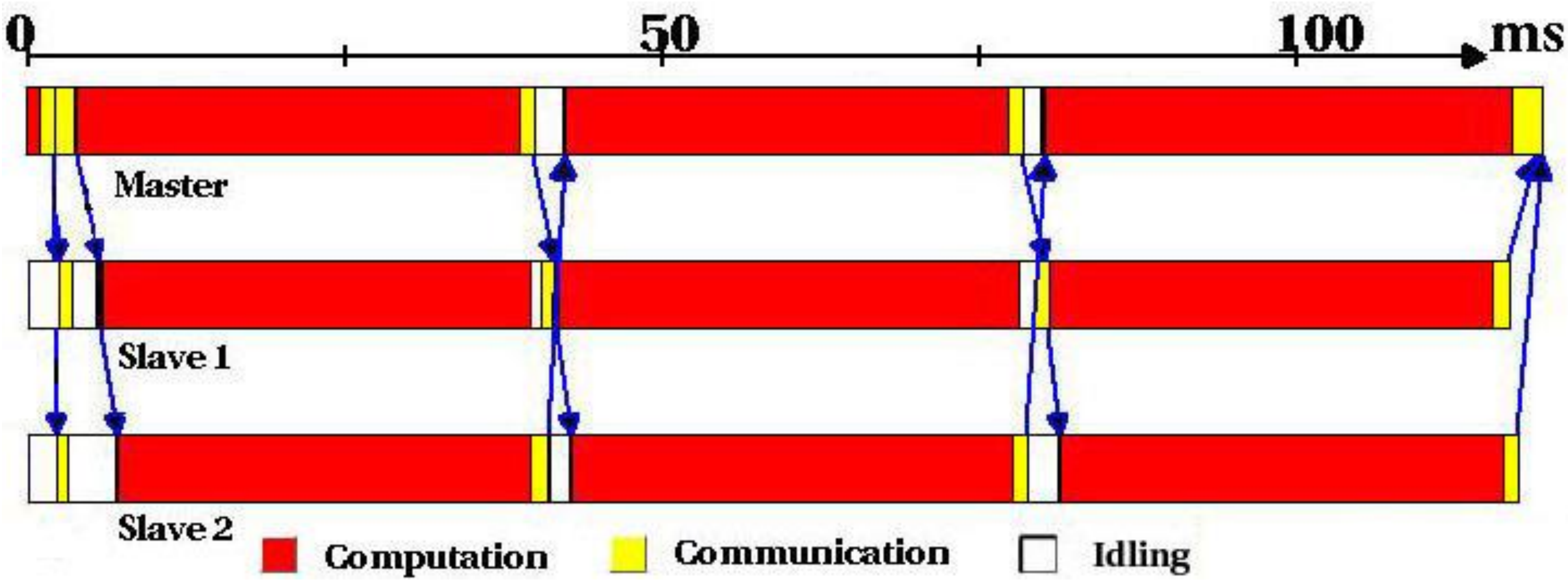
$$Speedup = \frac{T_{seq}}{T_{par}}$$

$$Efficiency = \frac{Speedup}{p} = \frac{T_{seq}}{p.T_{par}}$$

Other metrics we could try to optimize: energy consumption, cost, …

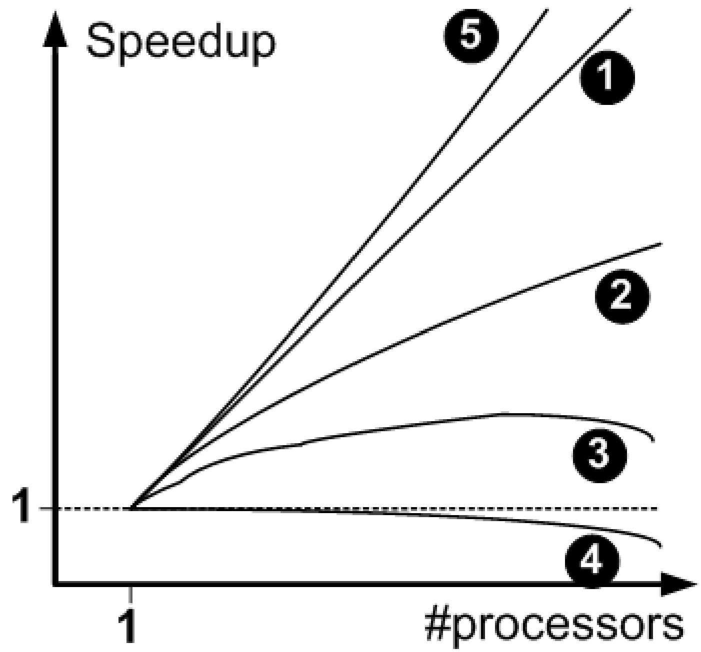
Parallel Matrix Multiplication: Execution Profile

On cluster of 3 computers - MPI



Speedup=2.55 Efficiency = 85%

Speedup i.f.o. processors



1) Ideal, linear speedup
2) Increasing, sub-linear speedup
3) Speedup with an optimal number of processors
4) No speedup
5) Super-linear speedup

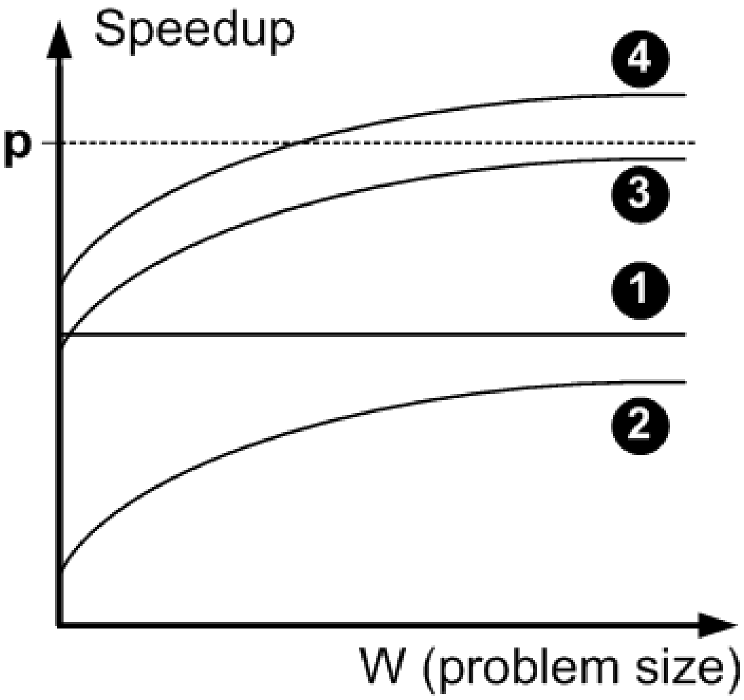
PPP p. 92

Super-linear speedup

- The parallel execution works with data that fits in lower-level memory, while this is not the case for the sequential execution

- The work in parallel is less than that of the sequential program, called *parallel anomaly*. See Chapter DOP.

Speedup i.f.o. problem size



1) Constant speedup
2) Increasing, asymptotically, towards value sublinear speedup (< *p*)
3) Increasing towards *p*
4) Increasing towards super-linear speedup

W is a problem-specific parameter which is related to the amount of computational work (most often linearly-related)

Performance Analysis

Goals:

- Understanding of the computational process in terms of resource consumption
- Identification of inefficient patterns
- Performance prediction
- Performance characterization of program and system

Overhead or *Lost Cycles*

Ideally $T_{par} = T_{seq}/p$ ➡ $Speedup = p$

In practice $T_{par} > T_{seq}/p$

$$overhead = p.T_{par} - T_{seq}$$ ➡ $\boxed{lost\ processor\ cycles}$

= all cycles with T_{par} that are not utilized for **useful work**

For all processes:

$$T^{i}_{par} = T^{i}_{work} + \sum_{j}^{O} T^{i,j}_{ovh}$$

i: index of process *j*: index of overhead

Impact of Overhead on Speedup?

$$T_{seq} + T_{anomaly} = \sum_{i}^{p} T_{work}^{i}$$

$$T_{par} = T_{par}^{1} = \ldots = T_{par}^{p} \Rightarrow T_{par} = \frac{\sum_{i}^{p} T_{par}^{i}}{p}$$

$$T_{par} = \frac{\sum_{i}^{p} T_{work}^{i} + \sum_{i}^{p} \sum_{j}^{O} T_{ovh}^{i,j}}{p}$$

$$= \frac{T_{seq} + T_{anomaly} + \sum_{j}^{O} T_{ovh}^{j}}{p}$$

We sum overheads of type *j* over all processes *i*

$$Speedup = \frac{T_{seq}}{\frac{T_{seq} + \sum_{j}^{O+1} T_{ovh}^{j}}{p}} = \frac{p}{\frac{T_{seq}}{T_{seq}} + \frac{\sum_{j}^{O+1} T_{ovh}^{j}}{T_{seq}}}$$

Speedup & Overhead Ratios

$$\boxed{Speedup = \frac{p}{1 + \sum_j \frac{T^j_{ovh}}{T_{seq}}}}$$

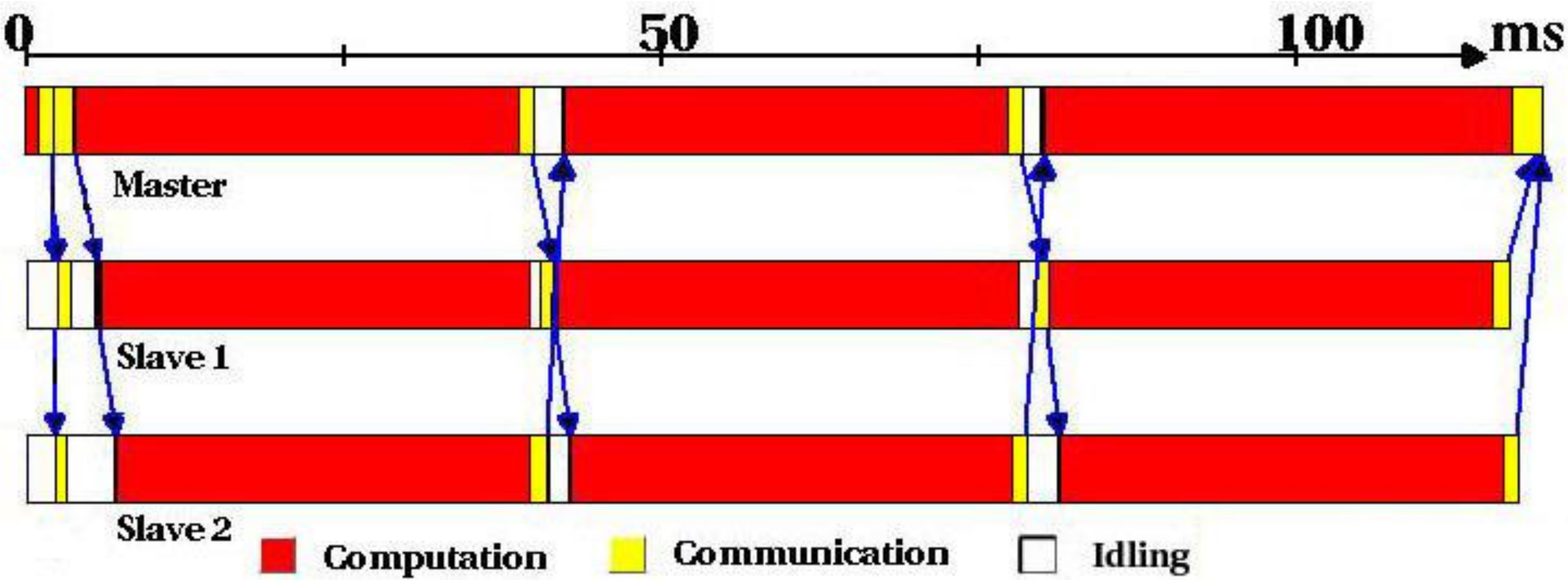
$$Ovh_j = \frac{T^j_{ovh}}{T_{seq}}$$

$$Speedup = \frac{p}{1 + \sum_j Ovh_j}$$

$$Efficiency = \frac{1}{1 + \sum_j Ovh_j}$$

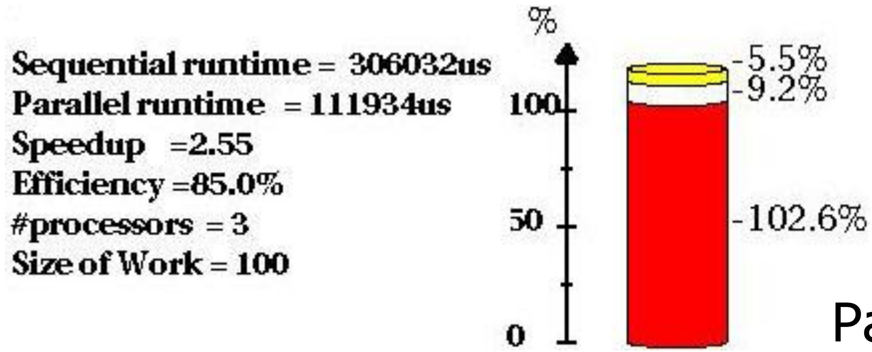
Total overhead (sum of per process overhead)!

Example 1: Execution Profile of Parallel Matrix Multiplication



Speedup=2.55 Efficiency = 85%

Parallel Matrix Multiplication



Parallel anomaly = 2.6%

$$\boxed{Efficiency = \frac{1}{1+(5,5+9,2+2,6)/100} = \frac{1}{1,173} = 0,85}$$

Analysis per process

If you assume that each process has $\boxed{\frac{T_{seq}}{p}}$ work,

We can calculate the overhead ratio per process: $$\boxed{Ovh_j^i = \frac{T_{ovh}^{i,j}}{T_{seq}/p}}$$

KUMAR p195

Overhead Classification

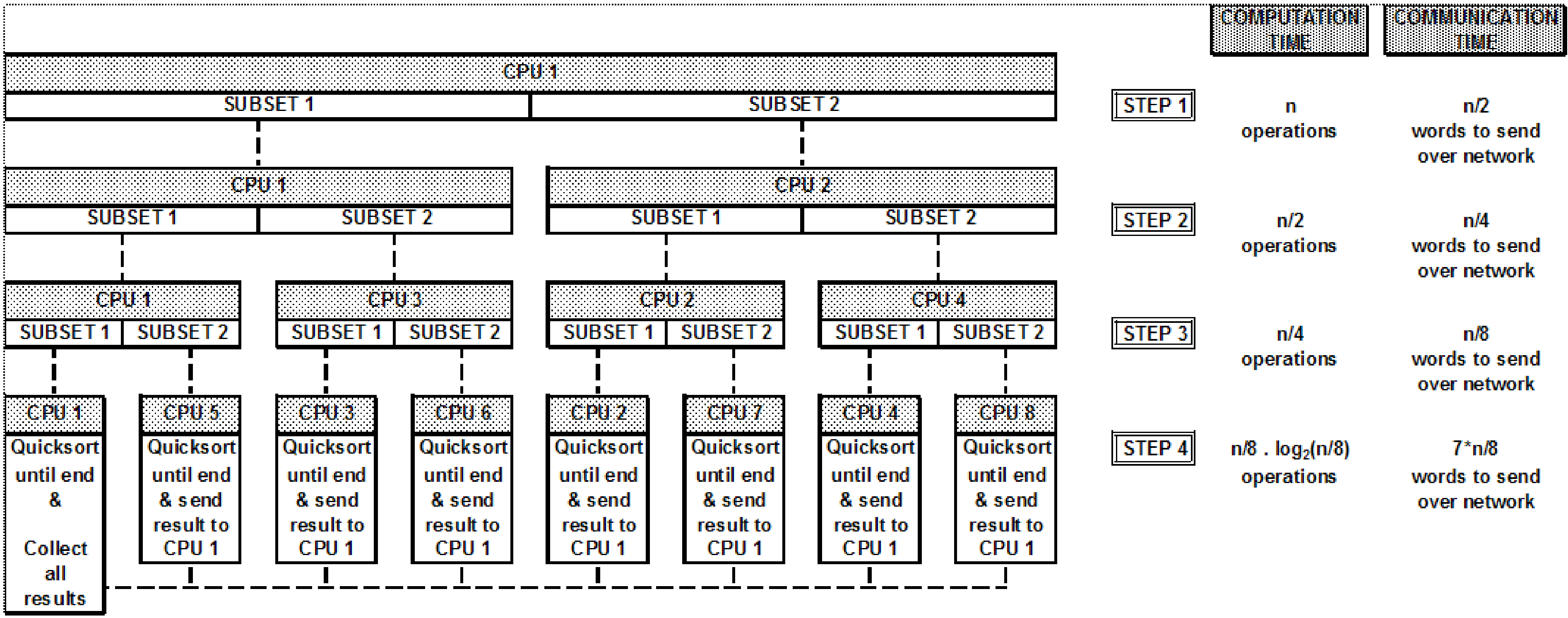
- ***Control of parallelism***: extra functionality necessary for parallelization (like partitioning)
  - Extra computations required
  - Part of computational phases are not for useful work!
- ***Communication***: overhead time not overlapping with computation
- ***Idling***: processor has to wait for further information
- ***Parallel anomaly*** : useful work differs for sequential and parallel execution

$$T_{seq} + T_{anomaly} = \sum_{i}^{p} T^{i}_{work}$$

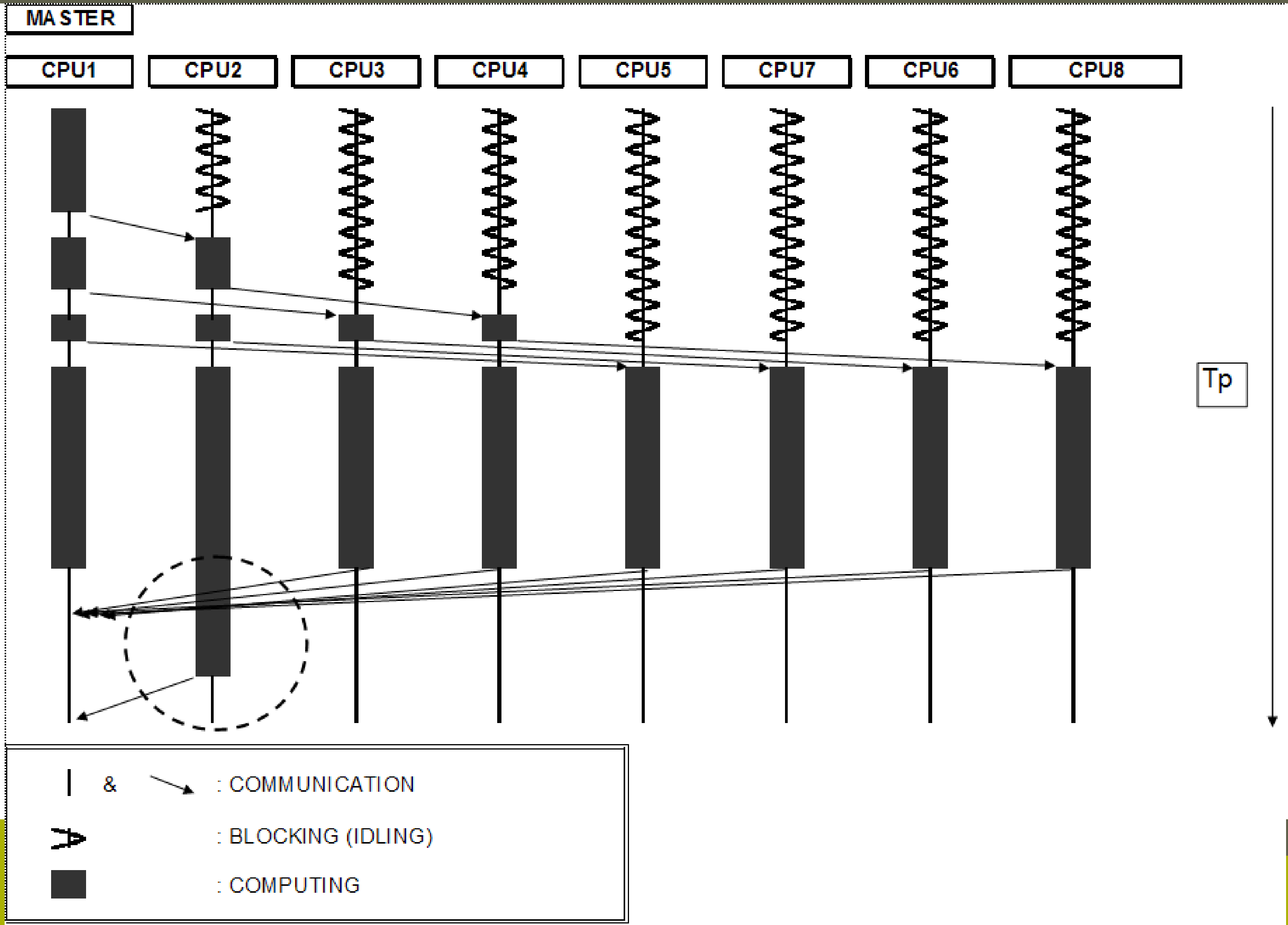
Causes of Idling

- Limitations of parallelism
  - Cf Amdahl's law (see further)
- Load imbalances
- Waiting for incoming messages, due to
  - Message latency
  - Limited bandwidth
  - Congestion in interconnection network

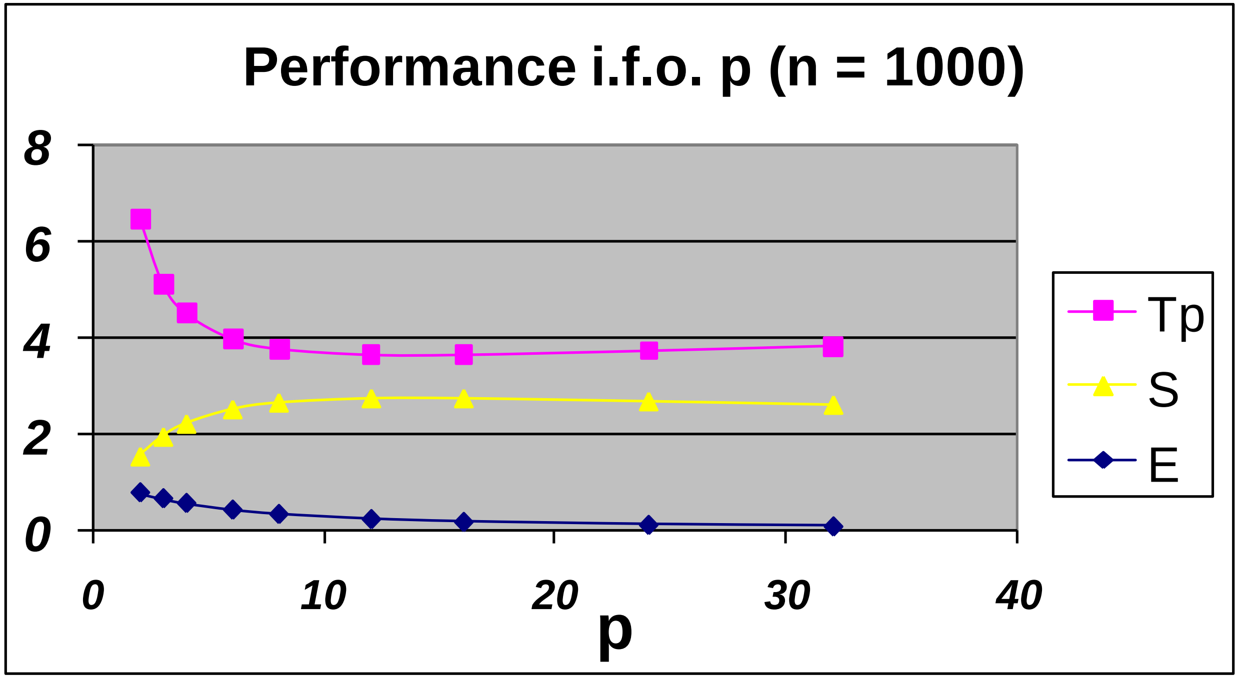
Example 2: Parallel Quicksort



Execution Profile of Parallel Quicksort



Quicksort's performance



Without considering load imbalances

Speedup growth is limited!

Reason?

PPP p. 80

Amdahl's Law

- Limitations of inherent parallelism: a part *s* of the algorithm is not parallelizable

$$\boxed{T_{seq} = \underbrace{(1-s).T_{seq}}_{\text{parallelizable}} + \underbrace{s.T_{seq}}_{\text{not parallelizable}}}$$

$$\boxed{T_{par} = \frac{(1-s).T_{seq}}{p} + s.T_{seq}}$$

$$\boxed{Speedup_{\max} = \frac{T_{seq}}{T_{par}} = \frac{T_{seq}}{\frac{(1-s).T_{seq}}{p} + s.T_{seq}} = \frac{p}{1+(p-1).s}}$$

Assume no other overhead

Amdahl's Law

$$Speedup < \frac{p}{1+(p-1).s}$$

$$Efficiency < \frac{1}{1+(p-1).s}$$

If *p* is big enough:

$$Speedup < \frac{1}{s}$$

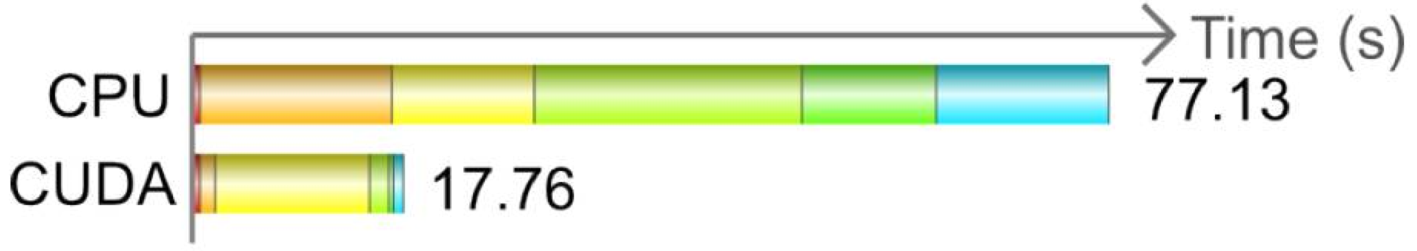
s	Speedup$_{max}$
10%	10
25%	4
50%	2
75%	1.33

Amdahl example: video decoding

Thanks to Wladimir van der Laan, University of Groningen

Decoding 1080p video sequence

Stage	CPU (s)	CUDA (s)	
1 MOTION_DECODE	0.64	0.64	
2 MOTION_RENDER	16.16	1.33	12 ×
3 RESIDUAL_DECODE	12.00	12.94	
4 WAVELET_TRANSFORM	22.52	1.63	14 ×
5 COMBINE	11.27	0.39	29 ×
6 UPSAMPLE	14.53	0.85	17 ×
Total	77.13	17.76	4.3 ×

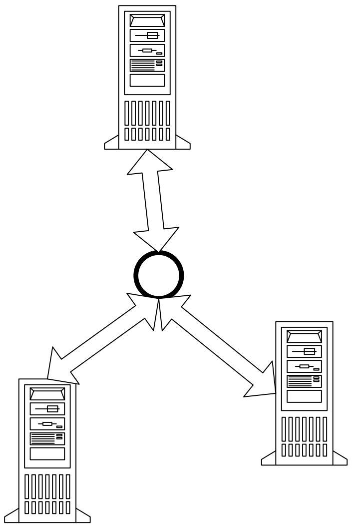


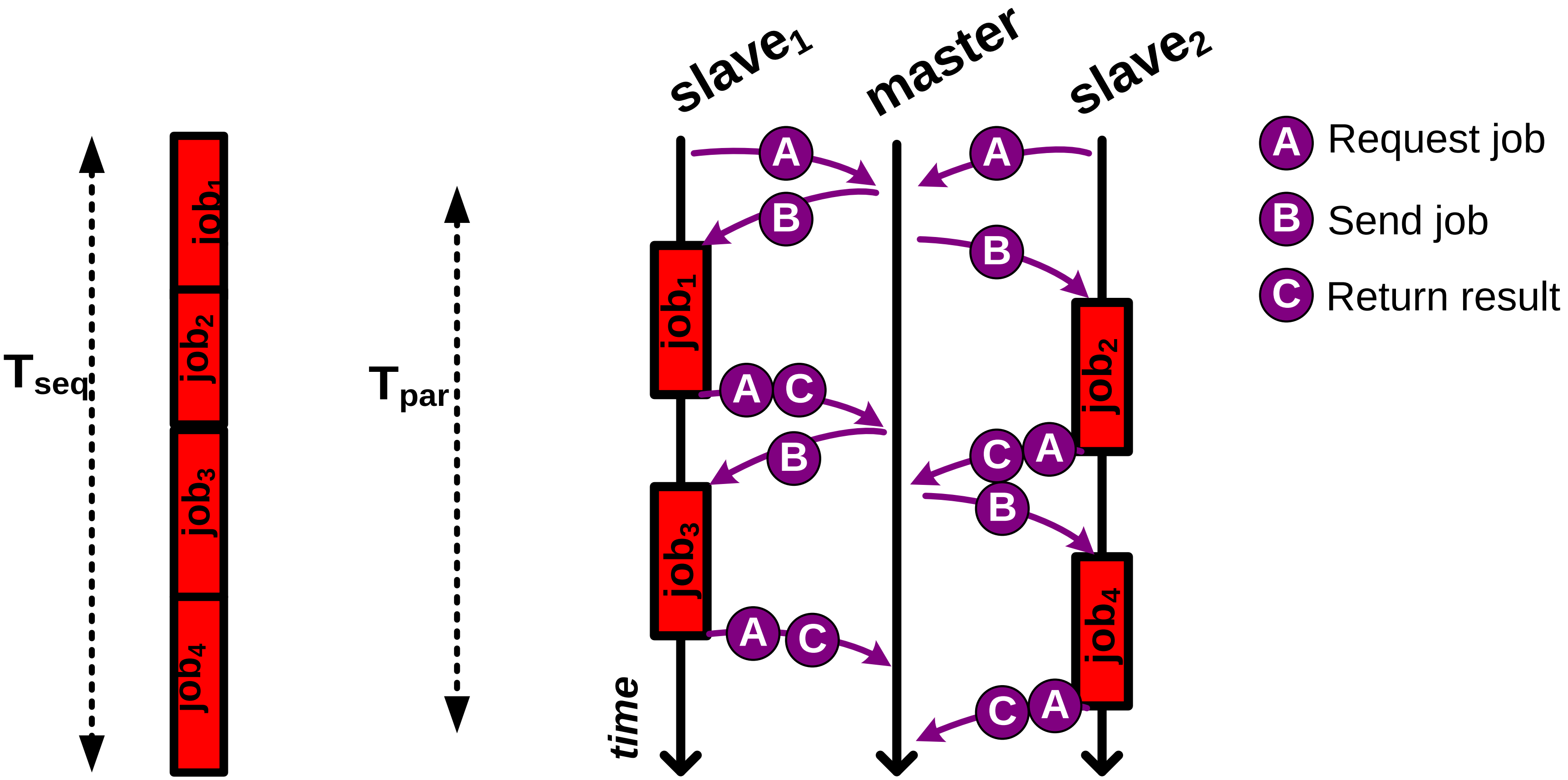
Example 3: Job Farming

Set of jobs & cluster of computers

= *Independent task parallelism*

{job1, job2, job3, job4}





T_{seq}
T_{par}
job_1
job_2
job_3
job_4
$slave_1$
master
$slave_2$
time
A
B
C
A Request job
B Send job
C Return result

Speedup = ±1.2

Performance of Job Farming?

Overheads? Bottlenecks?

1. Communication overhead
   - *Impact on speedup ~ T_{seq}/T_{comm} ~ granularity*
   - *Granularity* = computation/communication
   - ➡ overlap communication with computation
2. Bottleneck at master => idling of slaves
   - ➡ use several masters (‘tree’-structure)

Scalability

Can we keep efficiency constant while simultaneously increasing W and p?

Table 5.1 Efficiency as a function of n and p for adding n numbers on p processing elements.

n	$p = 1$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
64	1.0	***0.80***	0.57	0.33	0.17
192	1.0	0.92	***0.80***	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	***0.80***	0.62

Scalability

KUMAR p. 211

PPP p. 95

Gustafson's law

- Runtime remains constant if efficiency remains constant and increasing *p* and *W* at the same rate:

$$T_{par} = \frac{T_{seq}}{speedup} = \frac{\alpha . W}{efficiency(W,p).p}$$
$$= \frac{\alpha}{efficiency(W,p)} . \frac{W}{p} = \text{constant}$$

- *Problem doubles? Double processing power! Same time!*

- Program is ***scalable***: the ability to maintain efficiency at a fixed value by simultaneously increasing the number of processors and the size of the problem.
- It reflects a parallel system's ability to utilize increasing processing resources effectively.

Iso-efficiency

$$Efficiency = \frac{1}{1 + \frac{T_{ovh}}{T_{seq}}}$$

iso-efficiency curve:
When is efficiency constant

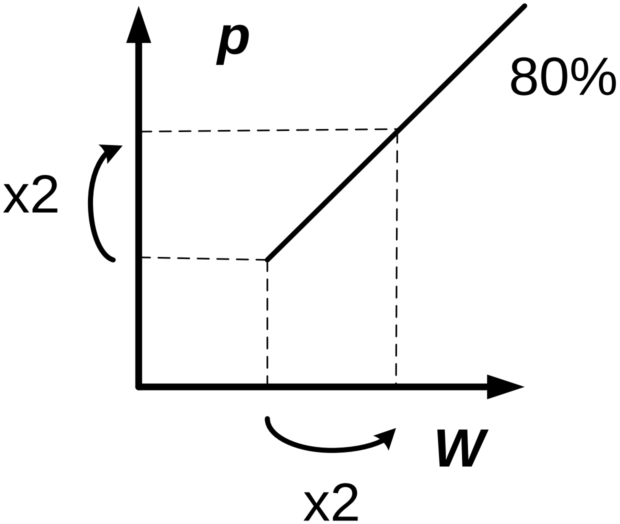
$$\Rightarrow \frac{T_{ovh}(W, p)}{T_{seq}} = \text{constant} = \frac{T_{ovh}(W, p)}{\alpha.W}$$

If sequential runtime~W

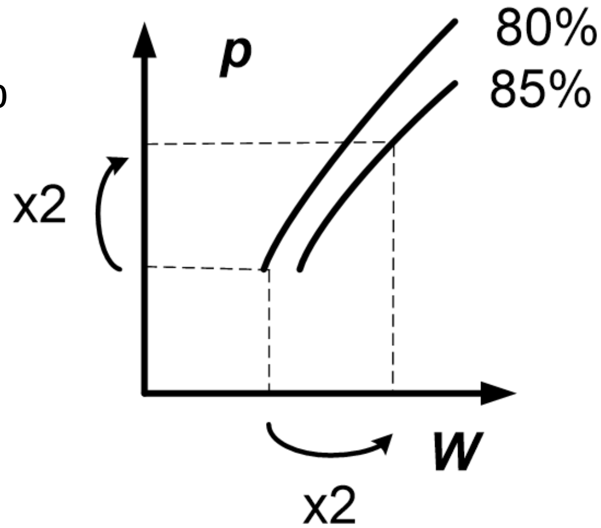
Function tells us **how *W* must increase with an increasing *p* for maintaining efficiency**

- *If perfectly scalable* (T_{ovh} linear or sub-linear in *p)*:
  - Increase *W* linearly with increasing *p*
  - Parallel run time stays the same
  - Workload per processor remains constant (see next slide)
- *If fairly/poorly scalable* (T_{ovh} super-linear in *p)*:
  - Problem size should be increased more than *p* to keep the efficiency
  - Bigger work load per processor (see next slide)
    - More memory needed!!

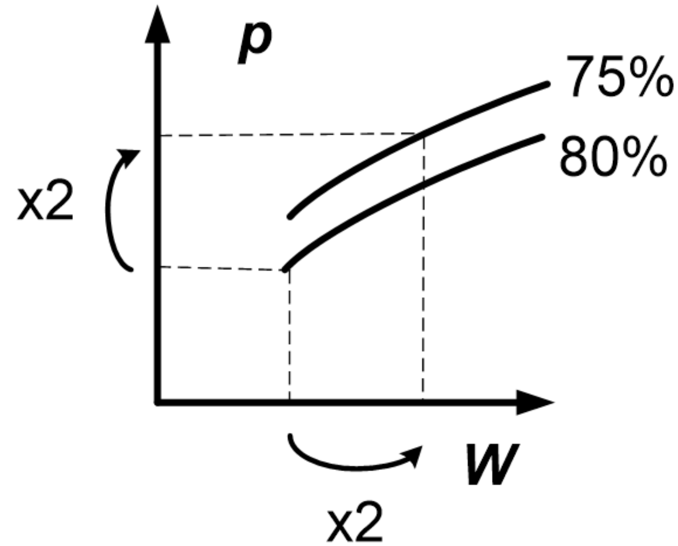
Iso-efficiency curves



scalable



highly scalable



poorly scalable

Thanks to Noah Van Es (2016)

See Link

Gustafson’s law

- **Amdahl’s law**: pessimistic view
  - parallelization is limited
  - Amdahl only changes *p*, keeps *W* constant
- **Gustafson**: more optimistic
  - the problems we run in parallel will be bigger and have more parallelism: for higher *p*, higher *W*
    - ➢ *Iso-efficiency curve*
  - Bigger problems: smaller serial fraction, less overhead

Approach to follow

I. Generate/draw execution profile

II. Identify lost cycles

III. Determine causes of overhead

IV. Plot performance in function of p and W

V. Study impact of overheads on speedup

VI. Study scalability

VII. Determine optimization possibilities

Performance analysis of your GPU program

- Measure computational performance (Gflops) and memory bandwidth (Bytes/sec)
  - Estimate number of instructions
  - Count data access
  - If applicable: memory bandwidth for each memory level:
    - CPU ⇔ GPU: PCIexpress bus (*this you can measure separately*)
    - Global memory access
    - Local memory access
- Compare with peak performance (see next slide)
  - Try to explain non-ideal performance
- Compare results for different versions of your program
  - From a naïve version to a highly-optimized version. *Are you coming close to peak performance?*
  - You can make idealized versions to measure impact of a certain aspect

Measure peak performance

- *Microbenchmarks*: small programs that measure a specific performance characteristic in isolation
  - E.g. Flops, bandwidth, cost of special functions such as cos, ...
- www.gpuperformance.org
  - Java app with microbenchmarks
    - Write them to database
  - Consult database

Theoretical performance analysis

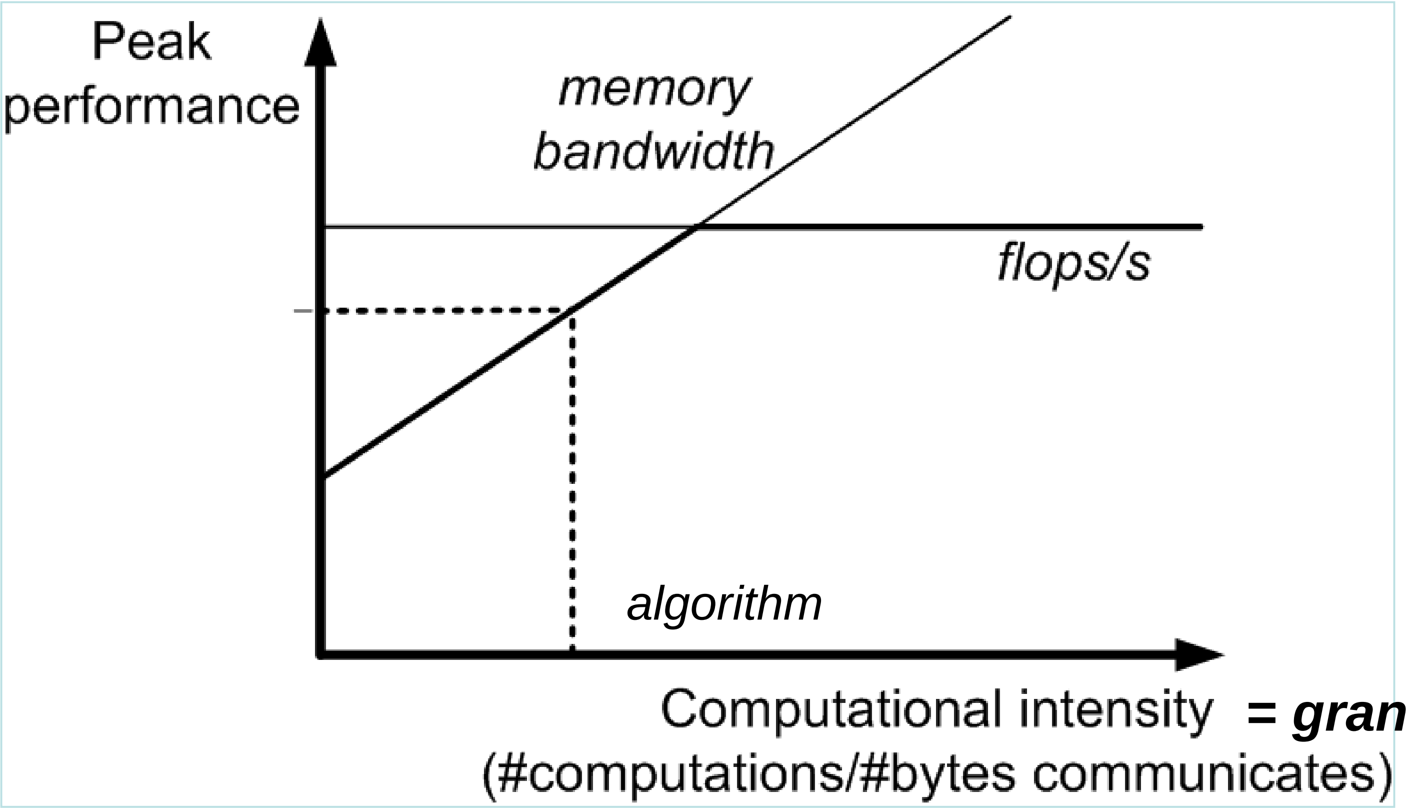
- Estimate a performance bound for your kernel
  - Count #instructions (in kernel, multiplied with the number of threads)
  - Count #memory transfer in bytes
  - ***Compute bound***: t_1 = #instructions / #instructions per second (theoretical computational peak performance of GPU)
  - ***Data bound***: t_2 = # memory accesses / bandwidth
  - Minimal runtime $t_{min} = \max(t_1, t_2)$

    expressed by roofline model (compute intensity = granularity)

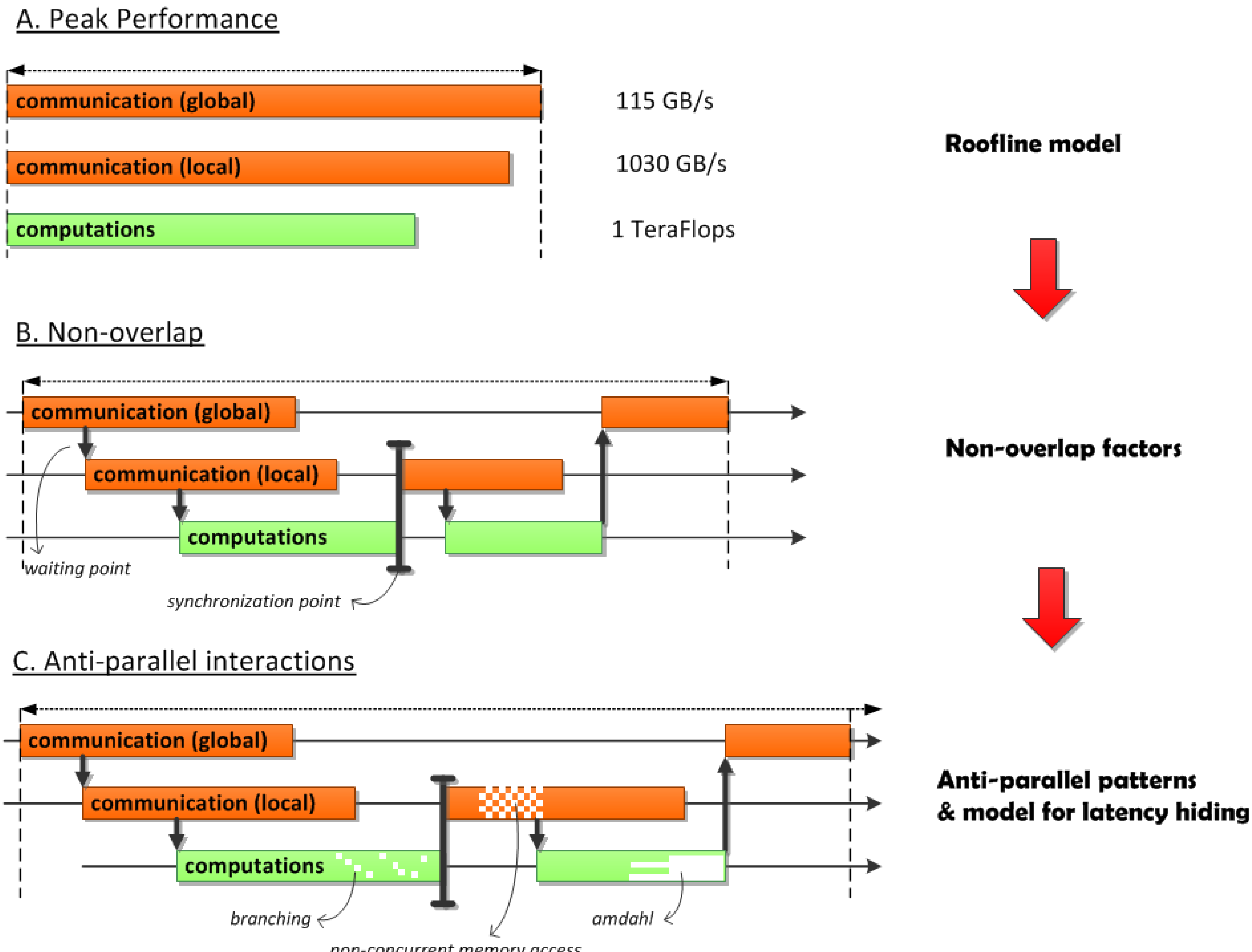
- Measure the actual runtime
  - $t_{actual} = t_{min} + t_{delta}$

- Try to account for and minimize t_{delta}
  - Due to non-overlap of computation and communication
  - Due to overheads caused by *anti-parallel patterns (APPs)*
  - Consult remedies for the overheads

Roofline model



= *granularity*



A. Peak Performance
communication (global)
115 GB/s
communication (local)
1030 GB/s
computations
1 TeraFlops
Roofline model
B. Non-overlap
communication (global)
communication (local)
computations
waiting point
synchronization point
Non-overlap factors
C. Anti-parallel interactions
communication (global)
communication (local)
computations
branching
non-concurrent memory access
amdahl
Anti-parallel patterns
& model for latency hiding