KOÇ UNIVERSITY

COMP429/529
Parallel Programming

Didem Unat

Lecture 2: Speedup and Scalability

Class Projects

- First project is done individually, the rest can be done with pairs
- Each uses a different parallel environment
- Each involves measurement, tuning, analysis and scalability studies

Project 1: SIMD parallelism and Parallel Programming via Shared-memory Model

Project 2: CUDA programming on NVIDIA GPUs

Project 3: Parallel Programming via MPI

Reading Assingments

- Will assign technical papers before class and discuss those during the lecture
- You will have 1-2 weeks to read the paper
- This will affect your participation grade
- Will post the first paper by Monday

Programmer's Perspective

Question: How do you make your program run faster?

Answer *before* 2004:

- Just wait 6 months, and buy a new machine!
- *(Or if you're really obsessed, you can learn about parallelism.)*

Answer *after* 2004:

- You need to write parallel software.

Concurrent, Distributed vs Parallel

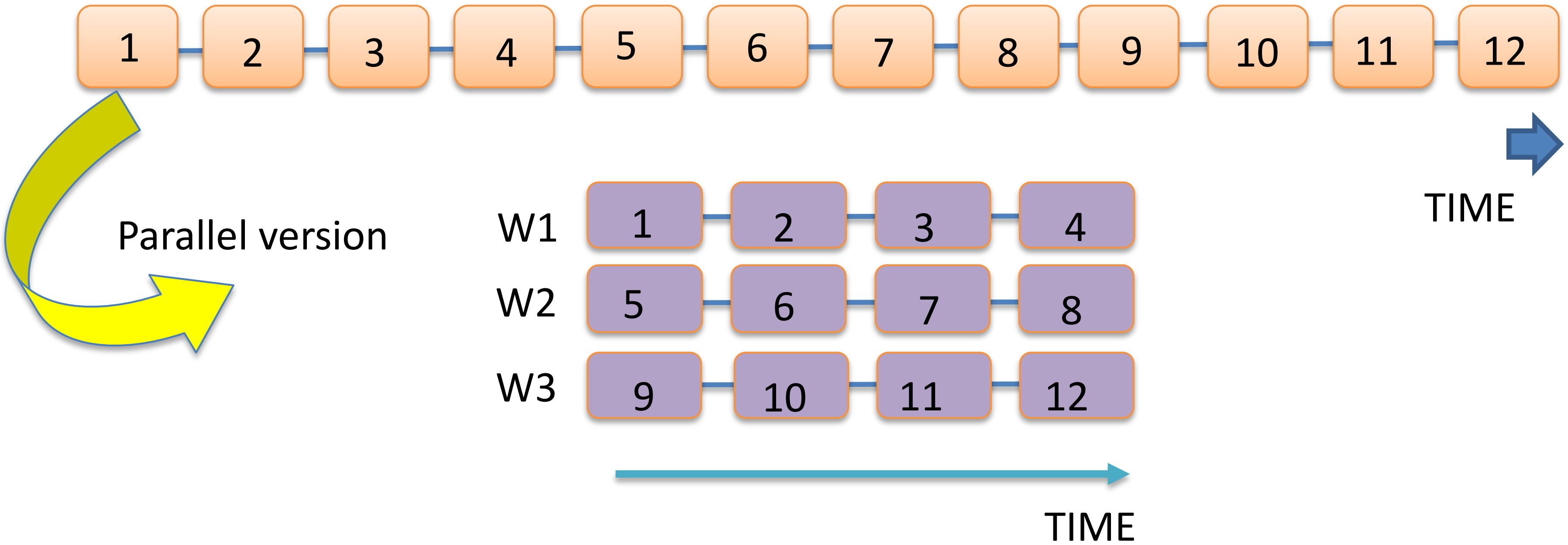
- **Concurrent:** a program with multiple tasks *in progress* at any instant
  - OS executing multiple tasks on a single CPU
- **Distributed:** a program which may be cooperating with other programs to solve a problem
  - Loosely coupled, geographically far away
  - Independently created programs
  - Cloud computing, web search, bank transactions etc.
- **Parallel:** a program with multiple tasks *cooperating closely* to solve a problem
  - Fast network connection, tightly coupled system
  - Single application or program

Speedup

- Number of workers or cores or processors= p
- Serial runtime = T_s or T_1 (runtime on 1 core)
- Parallel runtime = T_p (runtime on p cores)

- **Ideal case**: $T_p = T_1/p$
- Called **perfect** or **linear** speedup (a speedup of p)

Perfect Speedup



- **Ideal case**: Tp = T1/p
- Called **perfect** or **linear** speedup (a speedup of p)
- But this is rarely the case
  - Why?

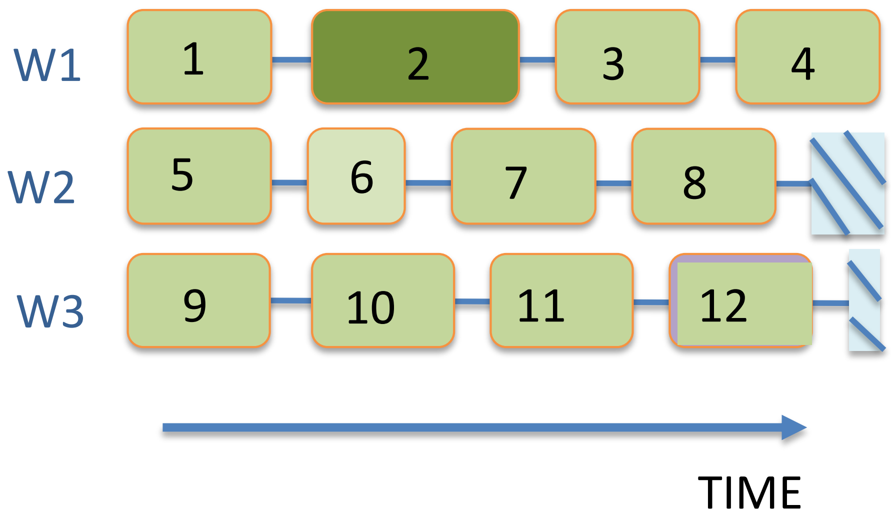
Challenges of Parallel Programming

- Serial vs Parallel
  - What kind of problems you may encounter when writing parallel programs?

1. Communication and data movement
   - Participating tasks need to communicate and share data
2. Synchronization
   - Tasks need to coordinate
3. Load imbalance
   - Some tasks may do more work than others, need to balance
4. Parallelization overhead
   - Above features lead to overhead

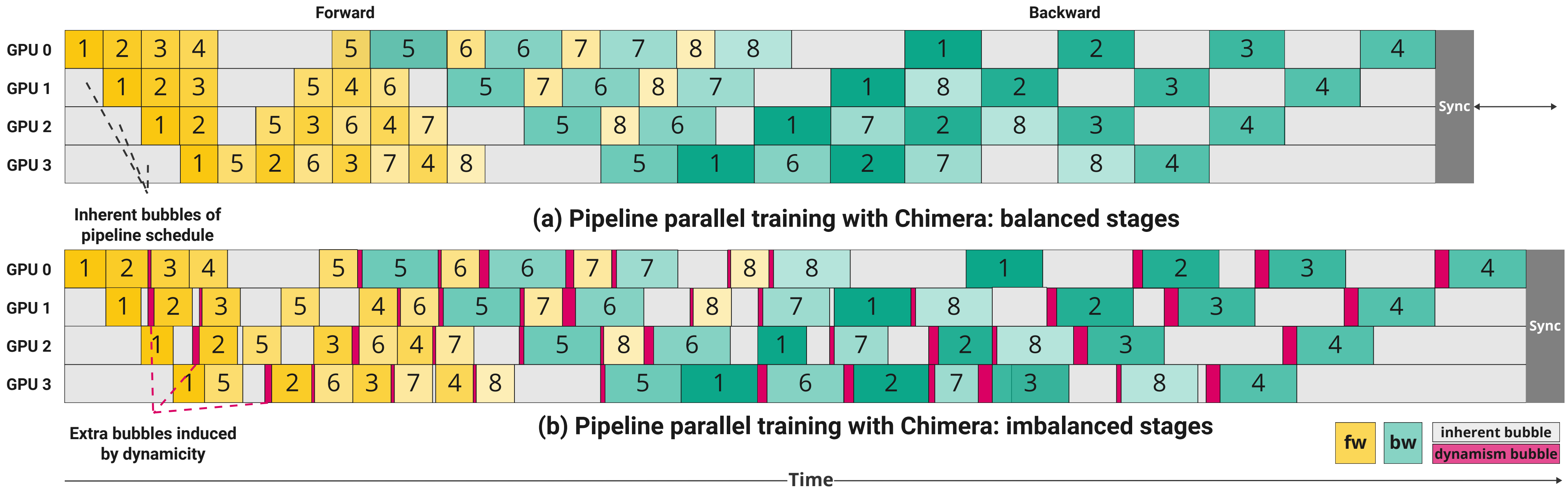
Perfect Speedup is Rarely the Case

- Because of Load Imbalance
  - Some work items may require more computation than others
  - Some tasks might be executed by different cores at different speeds
  - Some cores/processors stay idle while waiting for others to finish



Example from a Real Workload

- From a recent paper we submitted to ICLR*

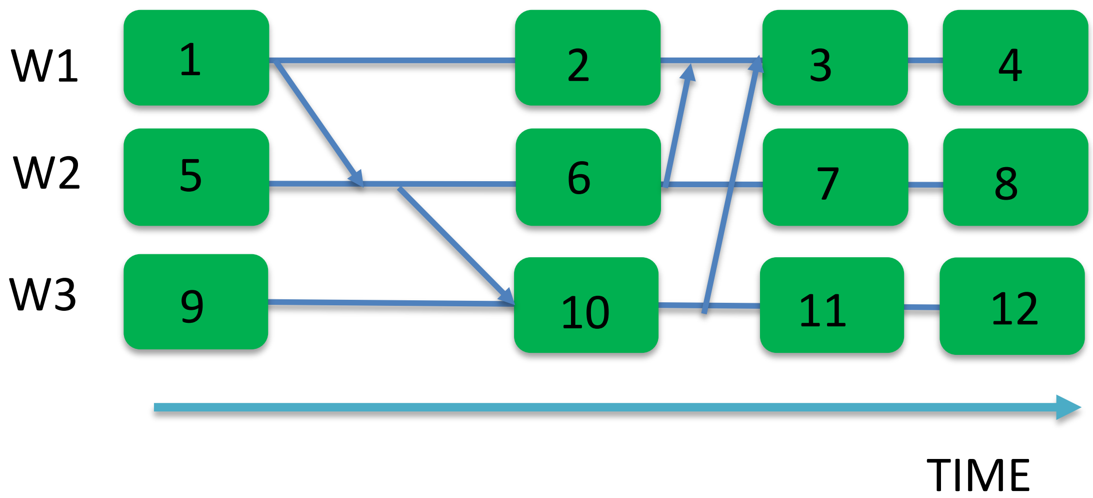


- You are as fast as your slowest worker

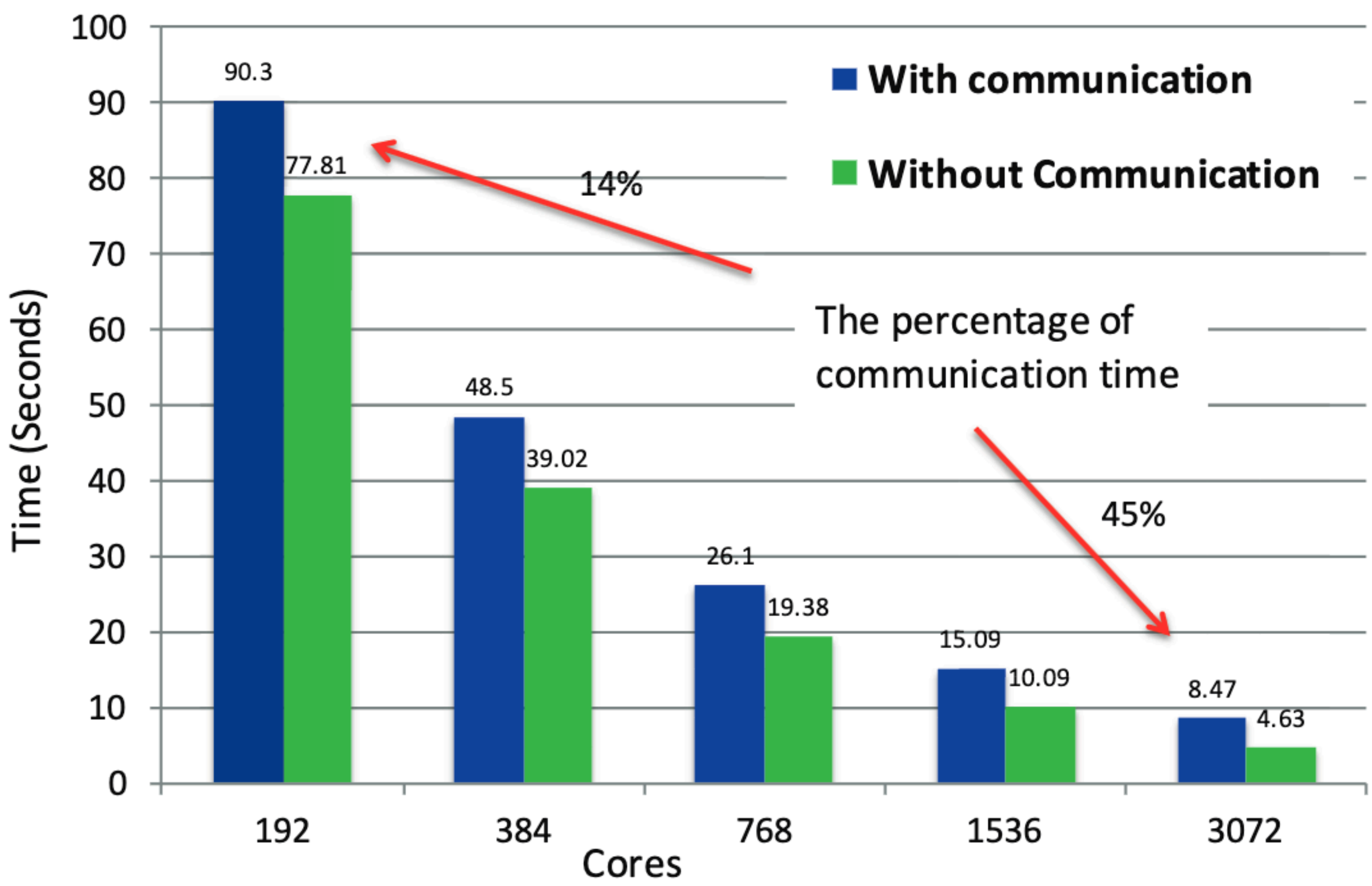
*together with Mohammad Attia (RIKEN)

Perfect Speedup is Rarely the Case

- Because of Communication
  - Communications between processes or threads can limit scalability if these operations cannot be **overlapped** with computation.
  - Arrows represent the necessary data transfers between workers



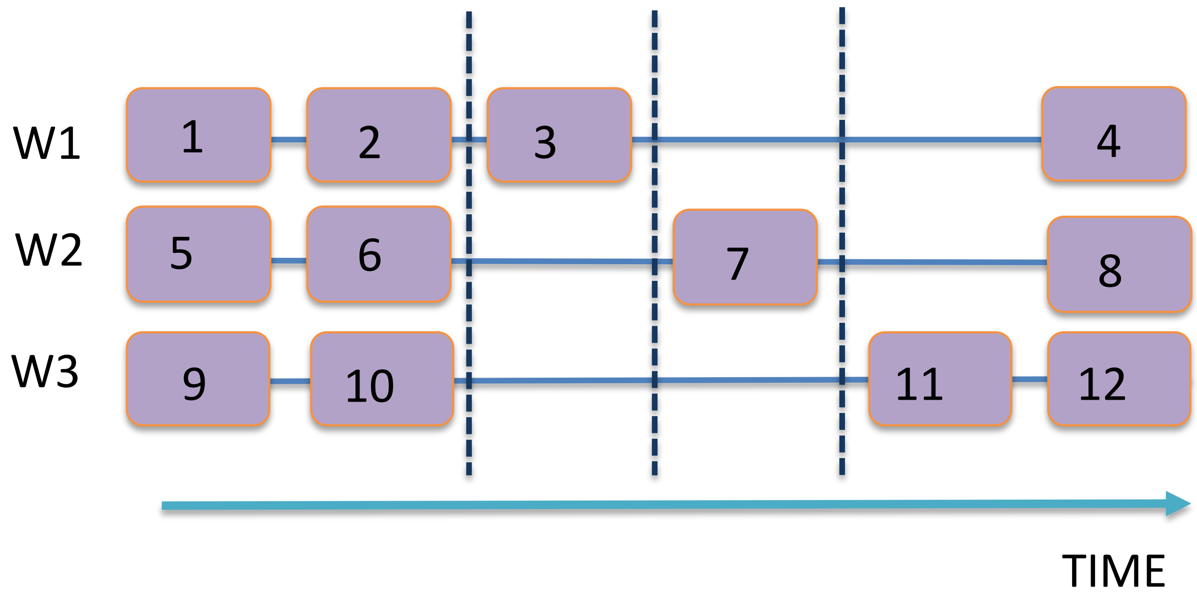
An example from a Real Workload



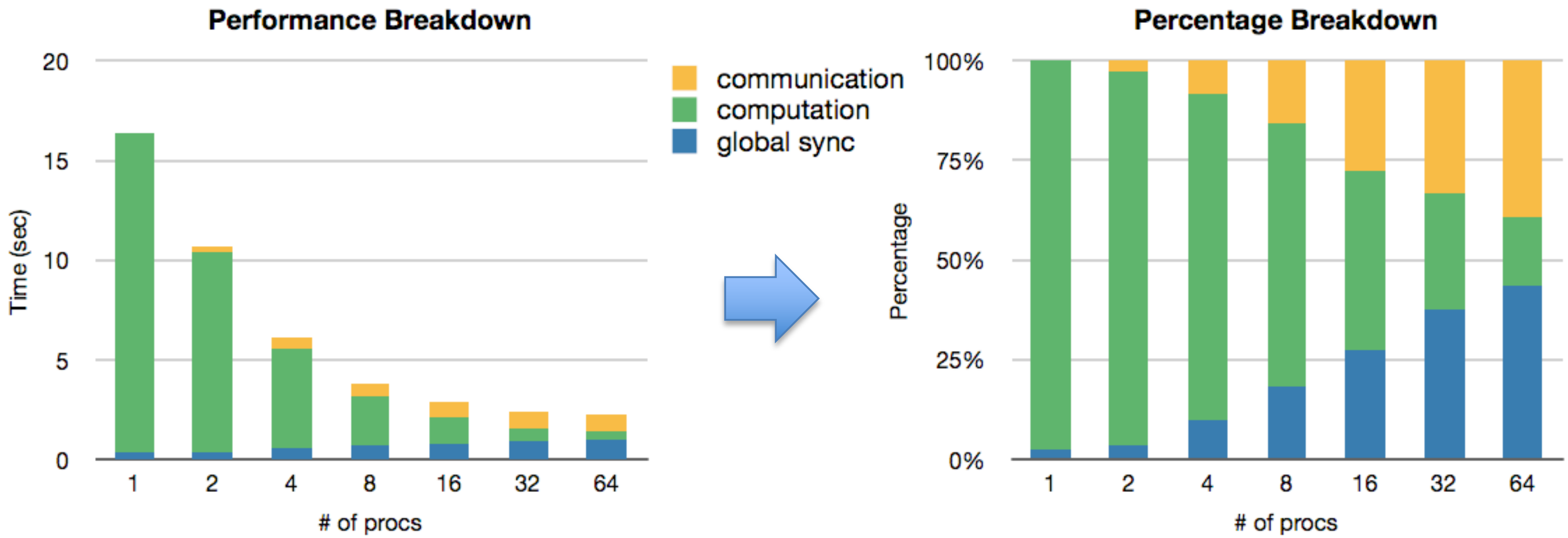
T. Nguyen, D. Unat, W. Zhang, A. Almgren, N. Farooqi and J. Shalf, "Perilla: Metadata-Based Optimizations of an Asynchronous Runtime for Adaptive Mesh Refinement," *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, USA, 2016, pp. 945-956, doi: 10.1109/SC.2016.80.

Perfect Speedup is Rarely the Case

- Because of Algorithmic Limitations or Non-parallelizable Sections
  - Operations cannot be done in parallel because of mutual **dependencies**, can only be performed one after another.
  - Shared resources **serializes** execution
    - For example, I/O or shared paths to memory in multicore chips



Fake Example



Speedup is affected by communication and synchronization

https://web.eecs.utk.edu/~huangj/hpc/hpc_intro.php

Perfect Speedup is Rarely the Case

- Because of Parallelization Overhead
  - Creating threads or processes is not free, there is always a startup overhead, which cannot be eliminated

- **Superlinear** speedup: is possible but it is rare:
  - Superlinear speedup means "*experiencing speedup more than p when using p workers*"
  - Typically due to having more cache space and better cache utilization

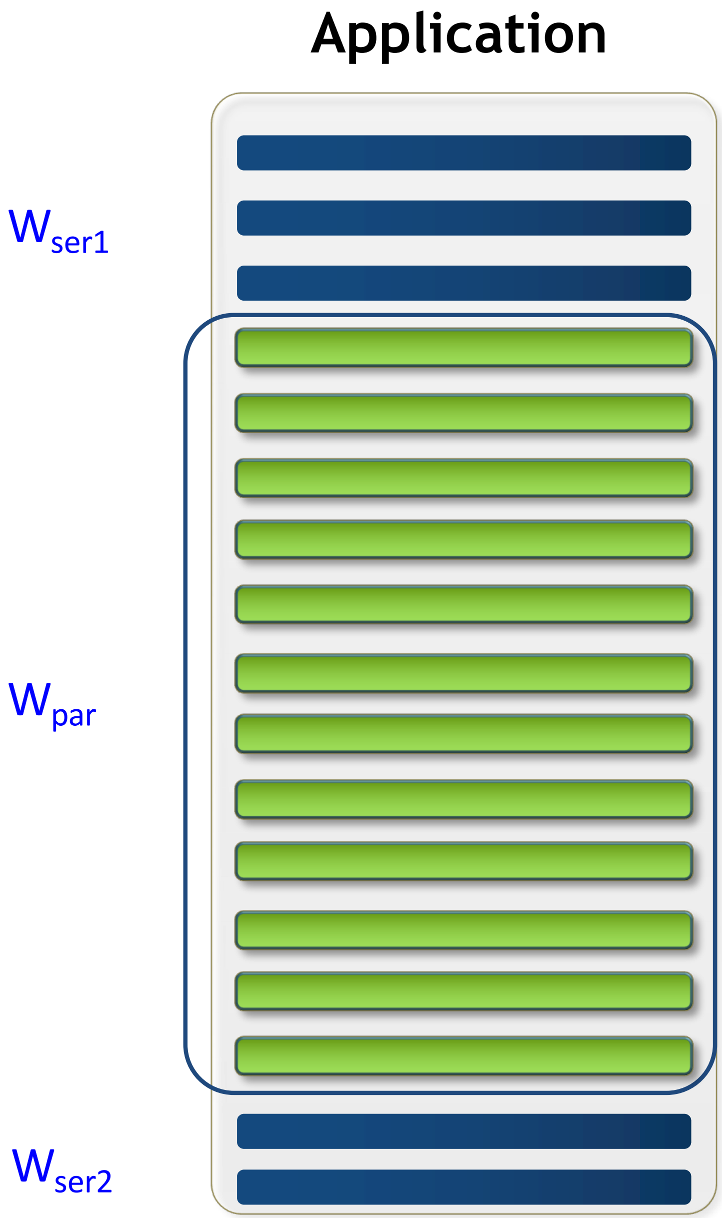
Speedup & Efficiency

- Speedup on p cores: $S_p = T_1 / T_p$
- Efficiency on p cores = Real speedup/ Expected Speedup
  - $E_p = S_p / p$ (note that p represents perfect speedup)
  - $E_p = T_1 / (pT_p)$
- For p cores:
  - $S_{ideal} = p$, $E_{ideal} = 1$ or 100%
- For advertisements, use speedup, not efficiency ☺

Which serial version to use?

- When computing speedup, the dilemma is what the baseline should be.
  - Should you use the most optimal serial version?
  - Should you use the serial code that you have started to develop a parallel version?
  - Should you use the single thread version of your parallel version?
  - **Because the optimal serial version may not be easily parallelizable, you may start with a different serial version for the same algorithm.**
  - What is important is **to explicitly mention what your baseline is**

Amdahl's Law



- Captures one aspect of the potential overheads:
  - **non-parallelizable serial** computations

$$T_1 = W_{ser} + W_{par}$$

W_{ser} : Time spent on non-parallelizable serial work

W_{par} : Time spent on parallelizable work

$$T_p \geq W_{ser} + W_{par} / p$$

Amdahl's Law

- Amdahl's Law: $$S_P \leq \frac{W_{ser} + W_{par}}{W_{ser} + W_{par}/P}$$

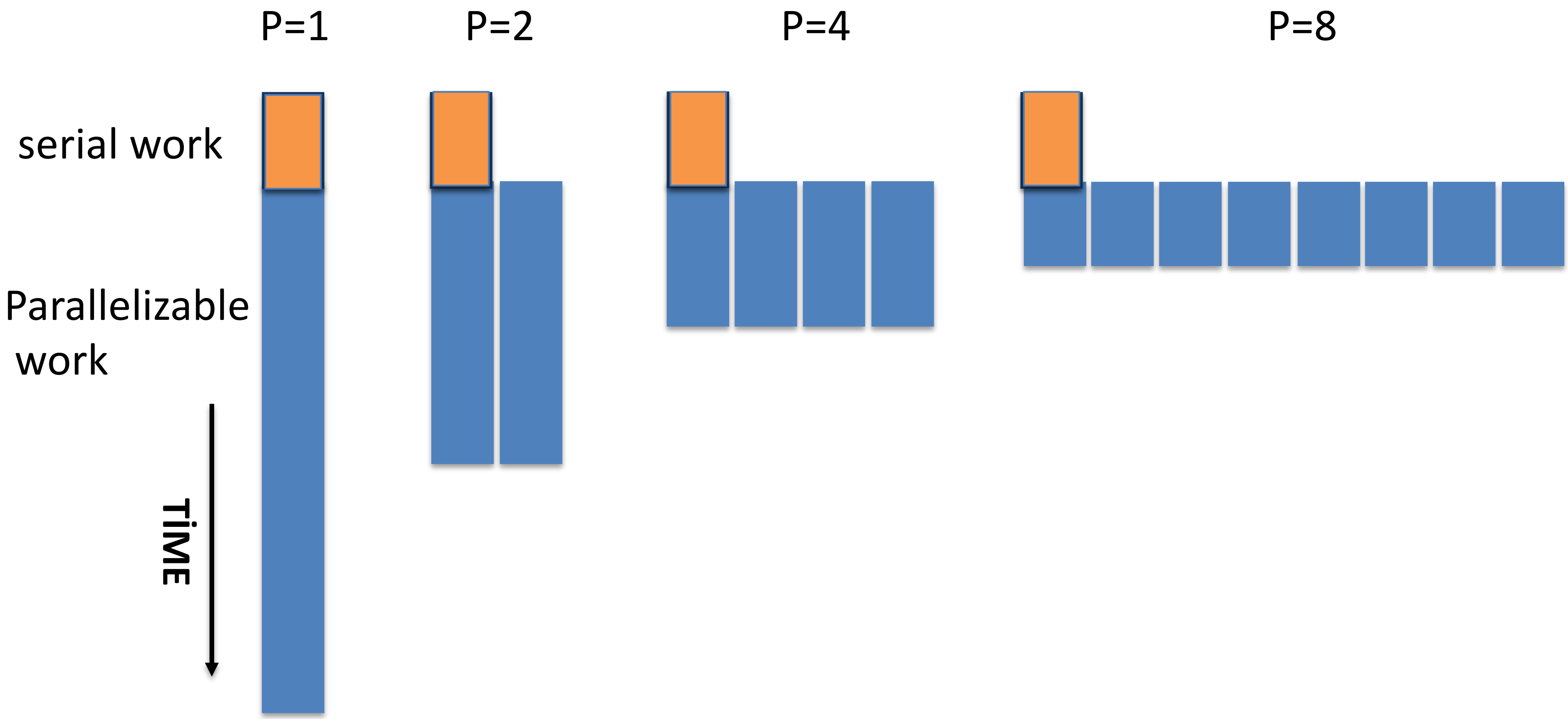
- or let f be the serial fraction of the total of the work,

$$\begin{aligned} W_{ser} &= fT_1 \\ W_{par} &= (1-f)T_1 \\ S_p &\leq \frac{1}{f + (1-f)/P} \end{aligned}$$

- An important corollary: Even when there are infinite resources, the speedup can not be greater then 1/ f!

$$S_\infty \leq \frac{1}{f}$$

Amdahl's Law Pictorially

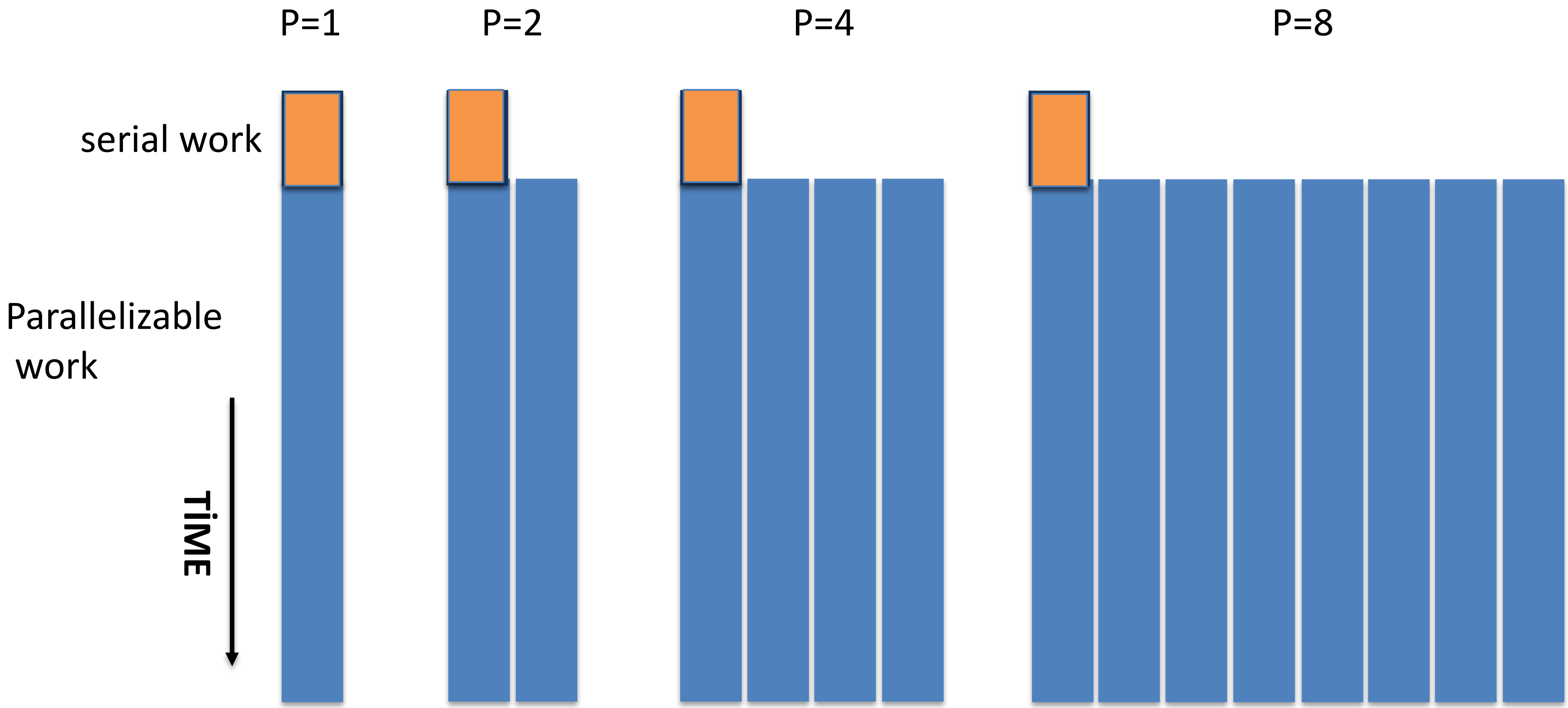


Source: McCool, Robinson, Reinders

Amdahl’s Law & Strong scalability

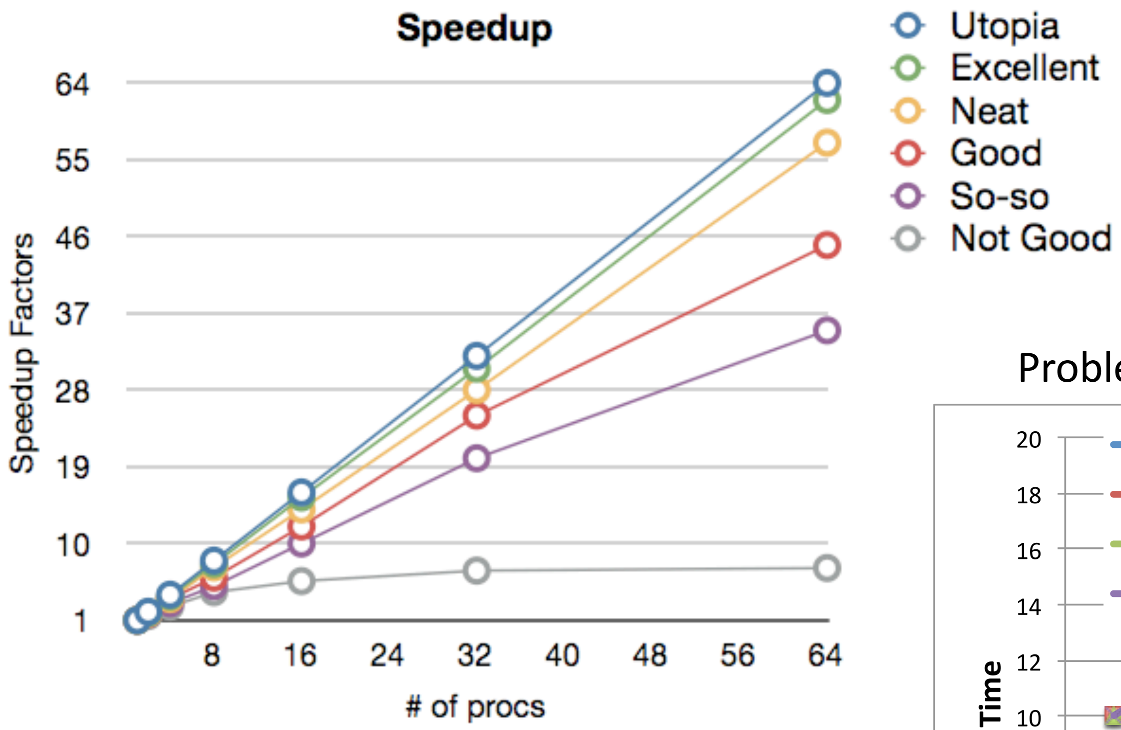
- Amdahl’s law is closely related to a commonly used notion of scalability: strong scaling
- **Strong scaling:** how does the execution time vary as we increase the number of processors for a fixed total problem size?
- But, the problem sizes that we want to solve often grow as the machines that we work with grow, too!
- **Weak scaling:** how does the execution time vary as we increase the number of processors and the total problem size at the same rate? (attributed to Gustafon & Barsis)

Weak scaling (Gustafson-Barsis)



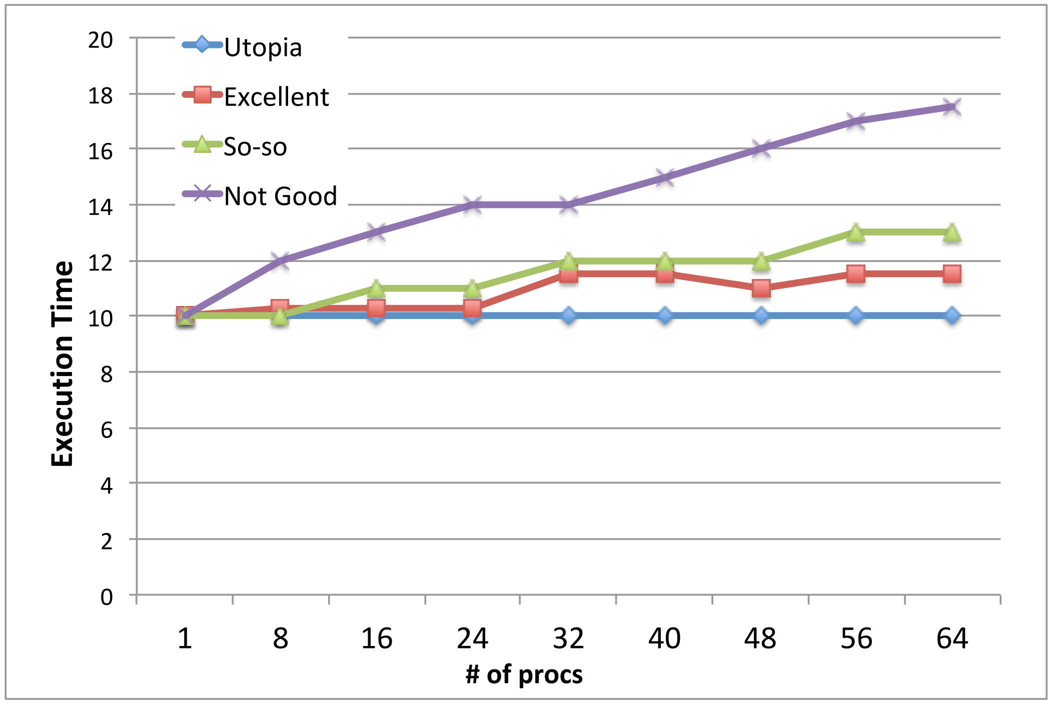
Source: McCool, Robinson, Reinders

Strong vs Weak Scaling (Fake Example)

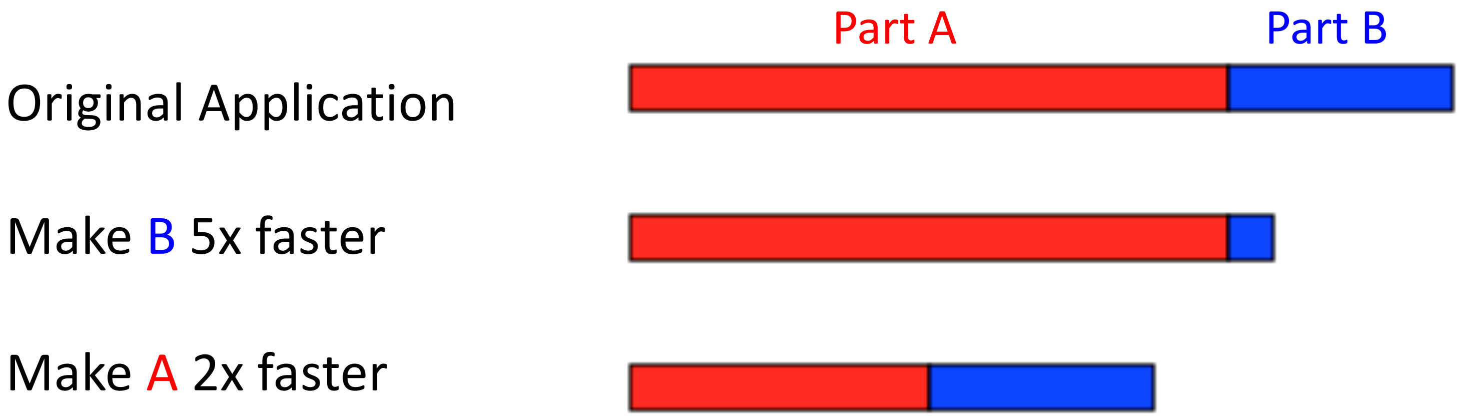


Strong scaling
Problem size is fixed, increase number of processors

Weak Scaling,
Problem size increases as # of processors increases



Focusing on the bottleneck



- Assume that a task has two independent parts, *A* and *B*.
- Part *B* takes roughly 25% of the time of the whole computation.
- Case 1: By working very hard, one may be able to make part B 5 times faster
- Case 2: In contrast, one may need to perform less work to make part *A* 2 times faster
- Compare the benefit of case 1 and case 2?

Acknowledgments

- These slides are inspired and partly adapted from
  - Randy Bryant and Nathan Beckmann (CMU)
  - Mary Hall (Univ. of Utah)
  - John Shalf (LBNL)
  - Rich Vuduc (Georgia Tech)
  - Metin Aktulga (Michigan State Univ.)
  - Scott Baden (UCSD)
  - The course book (Pacheco)