UiT

THE ARCTIC
UNIVERSITY
OF NORWAY

Parallelization: process and architectures

Inf-2202 Concurrent and Data-intensive Programming
Fall 2017
Lars Tiede (lars.tiede@uit.no)

Outline

- The parallelization process
  - “How to parallelize programs”
  - Based mostly on book chapters 2 and 3 in *Parallel Computer Architecture: A Hardware/Software Approach*. David Culler, J.P. Singh, Anoop Gupta. Morgan Kaufmann. 1998
  - Also a few of Lars Ailo Bongo’s slides from last year (lectures 5-6)

- Parallel (hardware) architectures
  - If we have time! Just for an overview of what’s out there
  - CPUs with special instructions, multicore CPUs, GPUs, clusters, clouds…

Parallelization process: goals

- High performance
  - solve larger problems, solve them faster
- Efficient resource utilization
  - waste no time, energy, money, on processors being *idle* or busy with *overhead* (work)
- Low developer effort
  - parallel program should be reasonably simple, little overhead (code) compared to sequential program

- Goals are sometimes at odds with each other
- Different hardware architectures favor different solutions

Performance: (Maximum) speedup

- Speedup factor:

$$S(p) = \frac{Execution\ time\ on\ one\ processor\ (best\ seq\ algorithm)}{Execution\ time\ using\ p\ processors\ (parallel\ algorithm)} = \frac{t_s}{t_p}$$

- Maximum speedup? -> Amdahl’s law

Amdahl's law

- Observation: Programs contain sections that can be parallelized, and sections that are serial
- Let f be fraction of program spent in serial sections (0..1). Assume we can parallelize uniformly over p processors (ideal). Assume the parallel program doesn't have overhead compared to the serial program (ideal). Then:

$$t_p \;=\; f t_s + (1-f)t_s/p$$

$$S(p) \;=\; \frac{t_s}{t_p} \;=\; \frac{t_s}{f t_s + (1-f)t_s/p} \;=\; \frac{p}{1+(p-1)f}$$

Maximum speedup (Amdahl’s law)

- For p processors, speedup is:

$$S(p) = \frac{p}{1 + (p - 1)f}$$

- Maximum speedup: f=0 (i.e. no serial sections in program)

$$S(p) = p$$

Speedup against number of processors

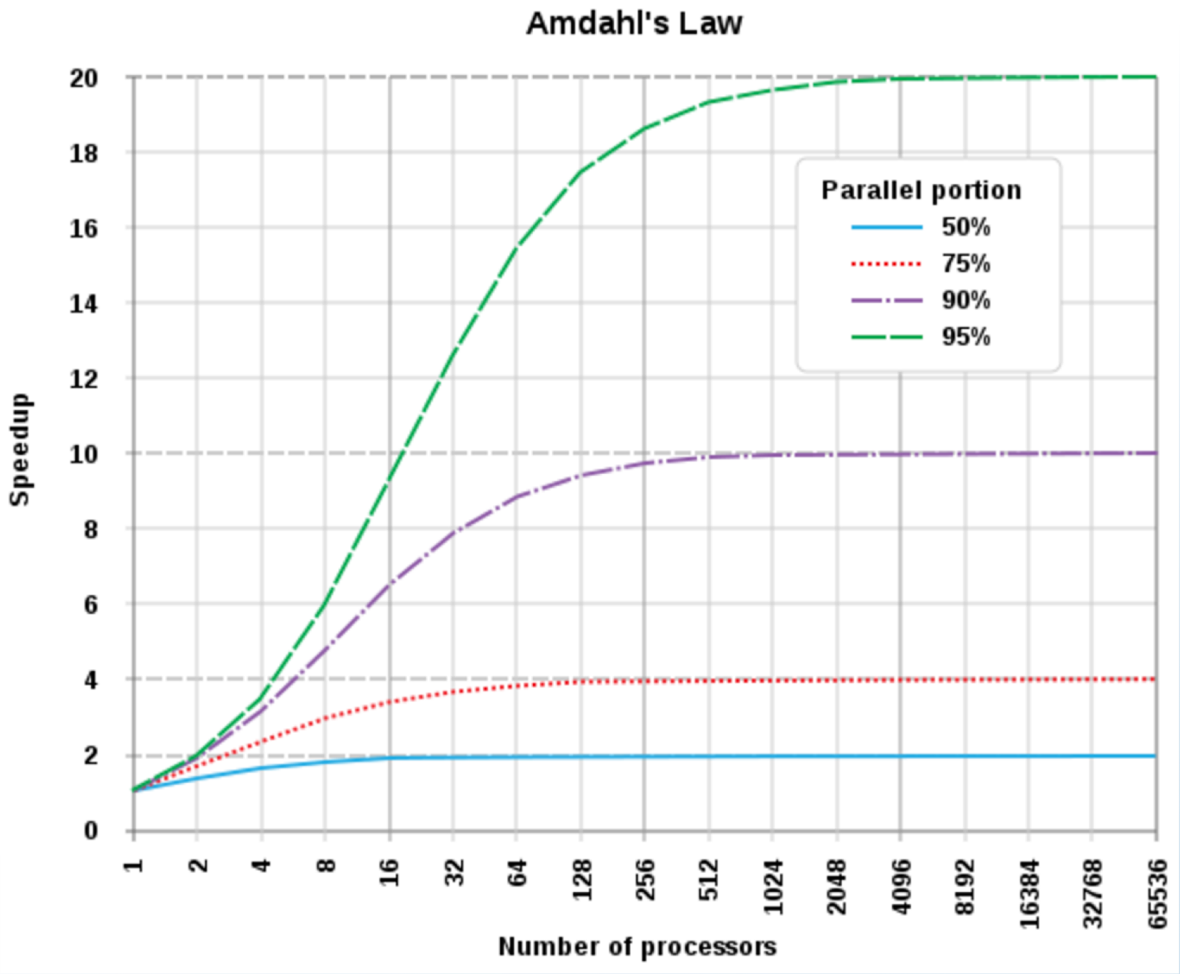


Image CC-BY-SA Daniels220 on wikipedia

Superlinear speedup

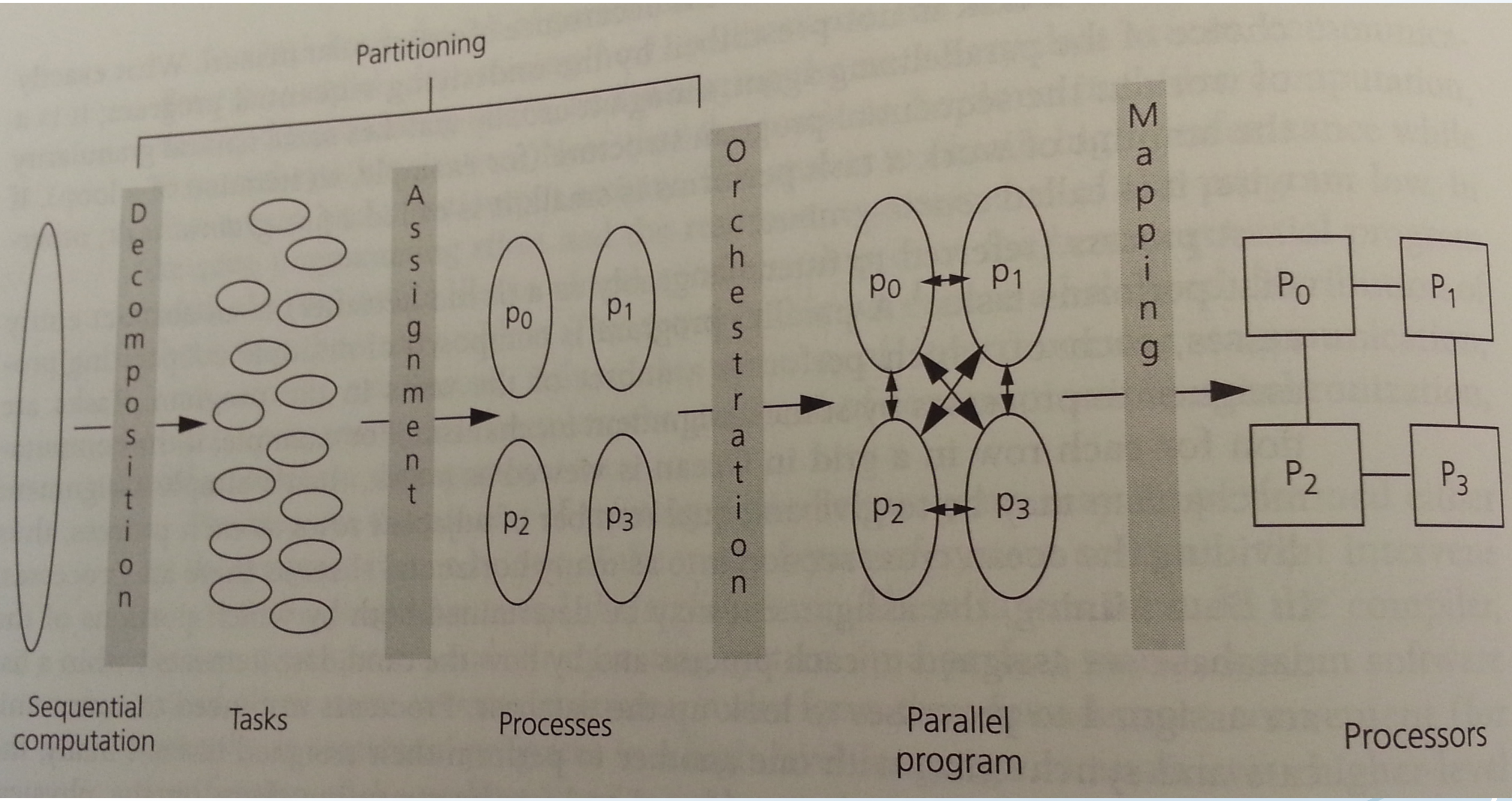
- In practice, we sometimes measure speedups greater than p (“superlinear speedup”)

- This due to:
  - Hardware, for example extra memory in multiprocessor system
  - Nondeterministic algorithms
  - Search algorithms (e.g. when finding result in p-th partition)

Parallelization process: nomenclature

- Task: piece of work
- Process/thread: entity that performs the work
- Processor/core: physical processor cores

Parallelization process: steps (overview)



Parallelization process: decomposition

- Split computation into a collection of tasks
- Goal: expose opportunities for parallelism
- Task granularity (-> # tasks) limits parallelism
- Deals mostly with algorithm, less with hardware architecture

Slightly abstract example: pizza preparation

- Task decomposition from simple sequential algorithm:
  - For each of the many pizzas (each with a spec) you want to make:
  - Prepare dough
  - Let dough rise
  - Roll out dough
  - Sauce
  - Toppings (several tasks, one for each? Tasks for preparing toppings?)
  - Bake
  - Cut to slices

Very concrete example: word count in Python

```
f = open(“huge_text_file.txt”, “r”)
wordcount = {}    # { word: count }

for w in f.read().split():
    wordcount[w] = wordcount.get(w,0) + 1

for item in wordcount.items():
    print(“{}\t{}”.format(*item))
```

- Possible decomposition into tasks that expose opportunities for parallelism?
  - Read the file into text. Perhaps not whole but in chunks? (Many tasks then)
  - Split text(s) at word boundaries, yield word after word
  - Count word(s) (could go crazy and say it’s one task per word)
  - Print result
- Is the above the only decomposition?

Common decomposition tactics (given a sequential program)

- Look at loops in the sequential program - can we decompose a loop into its iterations?
  - Works well if an iteration does not depend on the result of a previous iteration
  - If an iteration uses results of earlier iterations, we have a *data dependency* that will at least cost us later, maybe make parallelization outright impossible
  - If the sequence of iterations is critical wrt correctness, we call the loop a "sequential loop". Can't parallelize this.

- Maybe rewrite loops?
  - We may get rid of data dependencies by using private instead of shared data structures (but this necessitates merging those later on)

- Modify algorithm or use another one
  - Requires good understanding of the underlying problem

Parallelization process: assignment

- Goal: load balancing
    - All processes should do equal amount of work
    - Important for performance and resource efficiency

- Goal: reduce communication volume
    - Communication is not free (might be very expensive), so send around minimum amount of data, and minimum amount of messages

- Deals mostly with algorithm, less with hardware architecture

- Two types: static and dynamic (next slides)

Static and dynamic assignment

- Static assignment of tasks to processes
  - Algorithmic mapping
  - Example: if we have n tasks and m processes, assign task $i \in (1, \dots, n)$ to process $\lfloor {}^{i}/_{m} \rfloor$
  - Low overhead
  - Works well if workload is uniform across tasks. If not, will lead to load imbalance.
- Dynamic assignment of tasks to processes
  - Pool of available tasks
  - Typically balances load better than static assignment
  - More overhead
- In our examples?

Assignment in our examples: pizza prep

- Static example: each cook does the whole process for a predetermined set of n/p pizzas.
  - Works if every cook operates at same speed, every pizza takes equally long to prepare
  - Otherwise: load imbalance. The slowest cook who got the most complex pizzas to make will cause overall runtime to go up
- Dynamic example: each cook does the whole process for a pizza, then picks another pizza spec to make from a pool
  - Balances workload better among
  - Overhead: need a pool of pizza specs to make, communication and synchronization for pool's operations

Assignment in our examples: word count

- Static example: text is cut into p equally sized chunks (size given in bytes), each processor does one chunk
  - Works well if word length is uniform over the whole text
  - If not: some processes have many words to count, others fewer. Load imbalance.
- Dynamic example: text is cut into 100*p equally sized chunks, chunks are placed into a work pool, processors pick chunks from pool
  - Balances work better
  - Overhead: need pool, need communication and synchronization for pool’s operation

Kinds of concurrency in to seek out in partitioning

- (Partitioning = decomposition + assignment)

- Data parallelism
  - Processes do same computation on different parts of the data
  - Opportunity for parallelism grows with data size
  - Most often used

- Functional parallelism
  - Processes do different computations, often in the form of pipelined computation
  - Typically used in combination with data parallelism
  - Often modest amount

Concurrency in our examples: pizza prep

- Data parallelism
  - Many cooks can prepare pizza in parallel (from a-z), assuming plenty resources and place

- Functional parallelism
  - Cooks specialize on one (or short sequence of) tasks
  - Pass intermediate results between cooks
  - Pizza prep pipeline.

- Best solution might use both functional and data parallelism

Concurrency in our examples: word count

- Data parallelism
  - if we split the text into p smaller chunks, we can let p processes count words in the individual chunks
  - Do we need/want chunk-local word count that must be merged at the end? Or rather global word count that all processes write into?

- Functional parallelism: maybe pipelined processes for
  - text loading
  - splitting into chunks
  - count words in chunks
  - merge and print results

- Best solution might use both functional and data parallelism (but sketched functionally parallel partitioning probably not good)

Parallelization process: orchestration

- Goals:
  - Reduce communication cost
  - Reduce synchronization cost
  - Locality of data
  - Efficient scheduling
  - Reduce overhead

- Specific to computer architecture, programming model, and programming language

Orchestration in our examples: pizza prep

- Pizza prep:
  - Determine “communication lanes”. Pass intermediate results directly from one cook to another? Or use a big central table in the kitchen to stash them?
  - Determine when and how to pass around intermediate results between cooks
  - Determine where to store, perhaps cache, supplies
  - …
- Word count:
  - Shared memory? Or message passing? Or does the language/library we use have other comm/sync primitives?

Parallelization process: mapping

- Specific to system or programming environment
  - Parallel system resource allocator
  - Queuing systems
  - OS scheduler

Summary: goals of the parallelization process

Step	Architecture dependent?	Major performance goals
Decomposition	Mostly no	• Expose enough concurrency but not too much
Assignment	Mostly no	• Balance workload • Reduce communication volume
Orchestration	Yes	• Reduce noninherent communication via data locality • Reduce communication and synchronization cost as seen by the processor • Reduce serialization to shared resources • Schedule tasks to satisfy dependencies early
Mapping	Yes	• Put related threads on the same core if necessary • Exploit locality in chip and network topology

Parallel hardware architectures

- Subset of chapter 6 from “Computer Organization and Design”
  - Google knows this book, the library probably too
- These slides cannot be published on a publically accessible web site. Distribution through other channel (will be announced on slack)

Common parallel hardware architectures - overview

- CPUs
  - Vector and multimedia instructions
  - Hyperthreading
  - Multicore
- GPUs
  - Plenty cores
  - plenty*plenty threads, switching between them super fast
  - But: groups of threads run in lockstep (if/then/else possible, but threads that don’t enter some branch will be idle)
  - Double-But: rabbit hole
- Clusters
  - Plenty of computers connected through a network
  - Requires programming with message passing (at least on low abstraction level)