Programmation Concurrente

CM1 - Introduction to Parallelism

Florian Brandner & Laurent Pautet
LTCI, Télécom ParisTech, Université Paris-Saclay

Outline

Course Outline

- **CM1: Introduction**
  - Forms of parallelism
  - Computer architectures exploiting parallelism
  - Parallel programming paradigms
  - Amdahl's law and dependencies
- CM2: The *shared-memory* model (me)
- CM3-6: Concurrent programming POSIX/Java (L. Pautet)
  Patterns and Algorithms (L. Pautet)
- CM7: Actor-based programming (me)
- CM8: Transactional memory (me)

Organization

- Control continue: 25%
  - Two multiple choice tests (QCMs)
    (first 10 minutes of TD2 & TD7, be on time!)
  - Graded TP later on by LP
- Exam: 75%
  - All course material allowed
- Course material:

  https://se205.wp.imt.fr/
- I like group exercises ...

Introduction

Group Exercise: Forms of parallelism

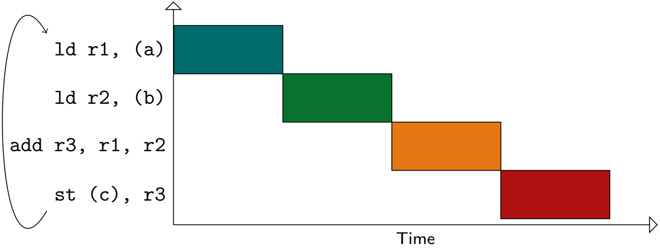
What is parallelism?

- Form groups of 2-3 persons
- Discuss what forms of parallelism exist in computer science
- Think about:
  - Computer architectures/hardware
  - Programming languages/paradigms
  - Granularity/level of parallelism (what is parallelized?)
- You have 5 minutes

Forms of parallelism

A view on computer architectures

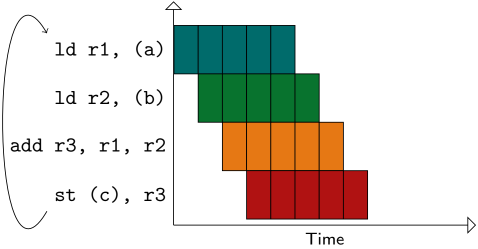
- Computers perform computations
- *Parallel* computers perform (some) computations in parallel
- Capabilities may vary between computer architectures (which developed of course over time)



A sequential, non-parallel processor (SISD)

Pipelining (80s)

- Computers execute instructions to perform a computation
- Decompose instructions into steps
    - Read the instruction from memory
    - Read the instruction's operands
    - Perform the computation
    - Write the result
- Perform these *steps* in parallel for different instructions

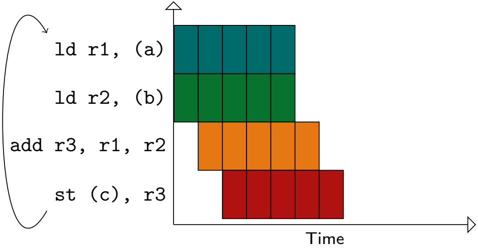


A simple pipelined processor (still SISD)

Instruction-level parallelism (ILP, 90s)

- Execute (independent) instructions in parallel

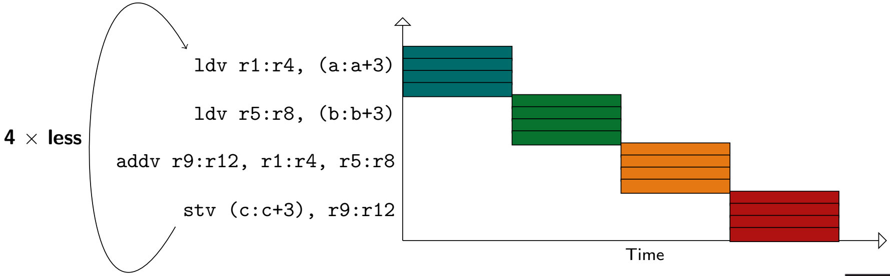
  We will come back to *dependencies* later today.
- Implementations:
  - Super-scalar or Very Long Instruction Word (VLIW)
  - Hardware/software detects independence online/offline
  - Usually combined with pipelining



A processor exploiting instruction-level parallelism (MIMD)

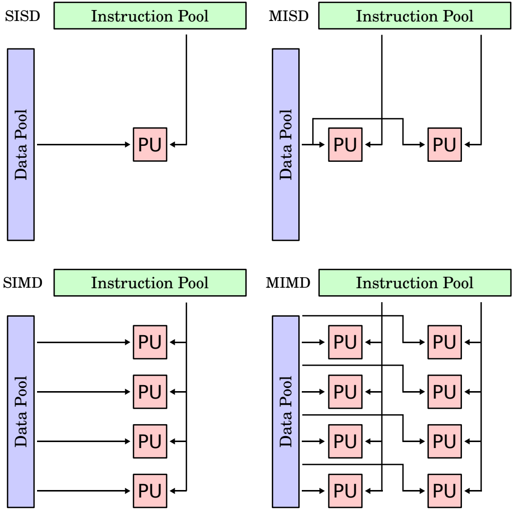
Data-level parallelism (70s and again 90s)

- Computers operate on data
- Data-parallel machines operate on many data items at once
- Implementations:
  - Vector machines (super-computers of 70s)
  - SIMD-extensions (PCs since 90s)
  - GPGPUs (still developing)



A processor with vector support (SIMD)

Flynn’s taxonomy



PU . . . Processing Unit

Image source: Wikipedia

Flynn's taxonomy (2)

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

SISD Single Instruction Single Data, a non-parallel computer

MISD Multiple Instruction Single Data, rather exotic model, sometimes found in safety-critical systems (airplanes)

SIMD Single Instruction Multiple Data, a vector computer

MIMD Multiple Instruction Multiple Data, a computer exploiting instruction-level parallelism (ILP).

Most modern computers (PCs) are a mix of the SIMD/MIMD model.

Beyond Instructions

Thread-level parallelism (60s)

- Also called task-level parallelism
- Execution of several threads (or programs) in parallel
  - Each thread represents its own stream of instructions (which may of course be executed in parallel themselves)
  - Threads may have private data
  - Threads may share data
  - Threads may need to coordinate among each other

⇒ Requires much more involvement of the programmer
⇒ Interaction with programming languages and models
⇒ Heavily researched even today after 50 years!

Implementations

- Multiprocessor computers (60s)
  A computer with multiple processors interconnected by a bus or a simple network. The processor may or may not access the same main memory.
- Multicore processors (2000s)
  Several processors on a single chip. Processors on the same chip usually share caches and the connection to main memory.
- Clusters/grids/distributed computers (70s)
  Several (often thousands) of computers interconnected by a network. Each computer has its on memory.

Models of parallel programming

Shared-memory model

Typically multicore or -processor computers where all processors access the same main memory. Threads coordinate by accessing shared data in the shared main memory space.

Subject of CM2-4 & 7-8

Message-passing model

Typically used for parallel systems using a network (e.g., clusters, grids). Threads coordinate by exchanging messages.

Subject of CM5-6

The programming model is somewhat implied by the computer architecture.

Group Exercise: Why parallel programming?

Discuss in groups of 2-3:

- Why should we care about parallel programming?
- What does it bring us?
- Why should we teach it?
- What issues might be relevant with regard to teaching parallel programming?

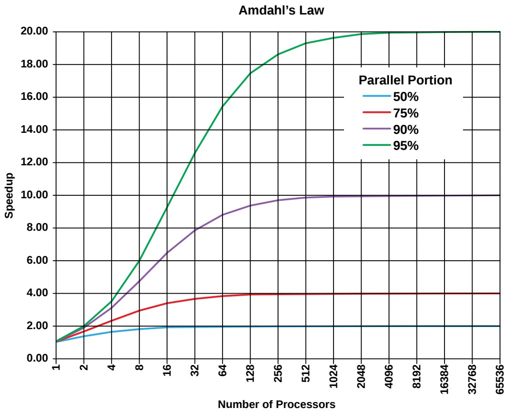
Dependencies

Amdahl's Law

Has to be mentioned in every lecture on parallelism:

Speedup (S) through parallelization by n is governed by the amount of *strictly sequential code* (B):

$$S(n) = \frac{1}{B + \frac{1}{n}(1 - B)}$$



Amdahl's Law: What does it mean?

- Nice speedups are possible (that's the good news)
- Throwing processors at a problem is not always helpful
- Even with a modest amount of sequential code the speedup levels off quickly

$\Rightarrow$ Algorithms have to be *designed* to be parallel
$\Rightarrow$ Programs have to be *written* to be parallel
$\Rightarrow$ Programmers need to know how to do this
(and tools have to be available to help them)

Amdahl's Law: What does it mean?

- Nice speedups are possible (that's the good news)
- Throwing processors at a problem is not always helpful
- Even with a modest amount of sequential code the speedup levels off quickly

$\Rightarrow$ Algorithms have to be *designed* to be parallel
$\Rightarrow$ Programs have to be *written* to be parallel
$\Rightarrow$ Programmers need to know how to do this
(and tools have to be available to help them)

$\Rightarrow$ You have to know what dependencies are

Dependencies

A somewhat simplified definition:

Data Dependence: The result obtained from one computation is needed in order to perform another computation.

Control Dependence: The outcome of one computation determines whether another computation is performed or not.

Example: Dependencies

```
int foo(int x)
{
  int a = x >> 3;
  int b = x & 7;
  int c = a + b;
  return c;
}
```

- `c` cannot be computed before `a` or `b`
  ⇒ *true* data dependence
- `a` and `b` are independent
  ⇒ no data dependence
  ⇒ they can be computed in parallel

Dependence Graphs

Representation of dependencies as a graph $G = (V, E)$

V Nodes in the graph, representing a computation step

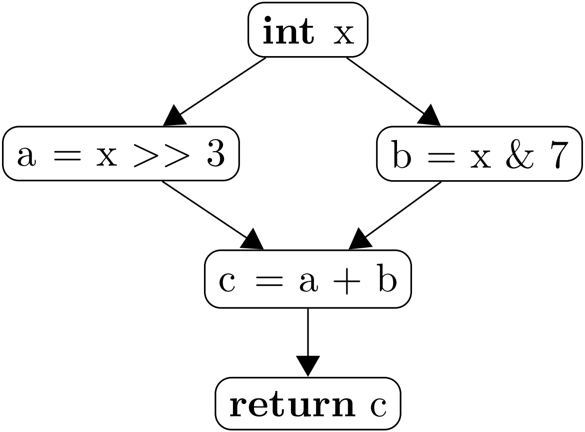
E Pairs of nodes $(a, b) \in V \times V$

Example from before:

$N = \{\mathtt{x}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{ret}\}$

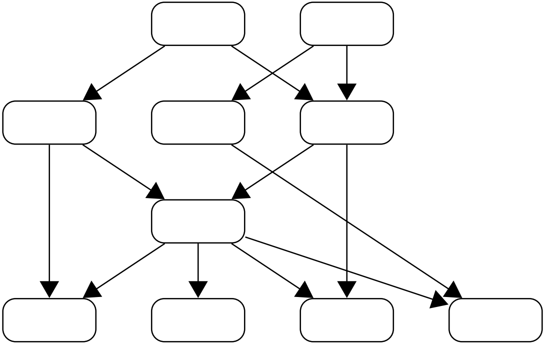
$V = \{(\mathtt{x}, \mathtt{a}), (\mathtt{x}, \mathtt{b}), (\mathtt{a}, \mathtt{c}), (\mathtt{b}, \mathtt{c}), (\mathtt{c}, \mathtt{ret})\}$

Drawing Dependence Graphs



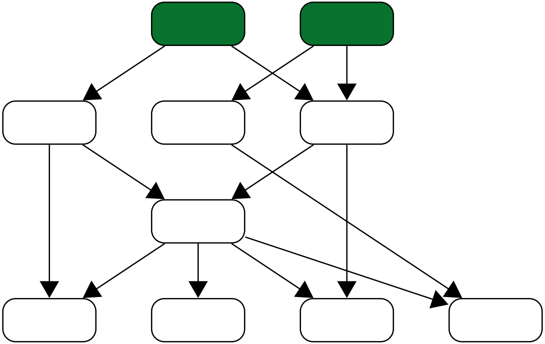
Dependencies and Parallelism

How do dependencies constrain parallelism?



Dependencies and Parallelism

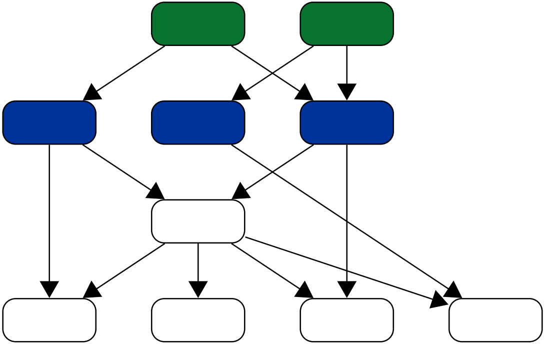
How do dependencies constrain parallelism?



Execute the two operations without dependencies in parallel

Dependencies and Parallelism

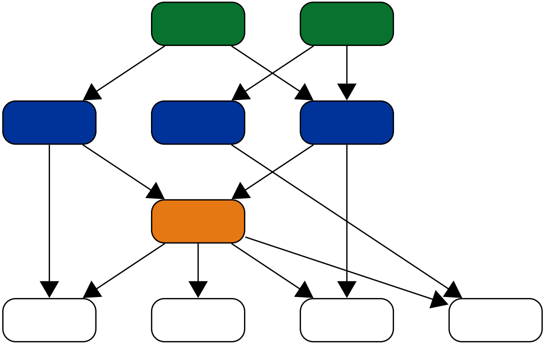
How do dependencies constrain parallelism?



The next three operations can be executed in parallel

Dependencies and Parallelism

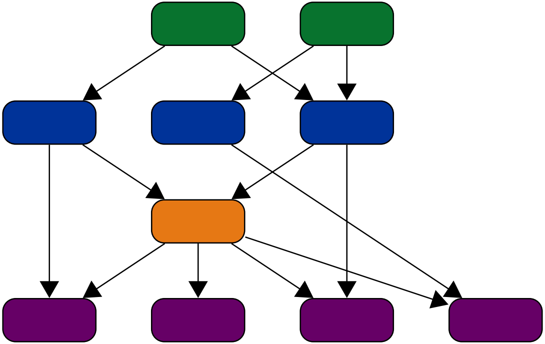
How do dependencies constrain parallelism?



The next operations can be executed (no parallelism)

Dependencies and Parallelism

How do dependencies constrain parallelism?



The next four operations can be executed in parallel

Forms of Data Dependencies

True Dependence: (aka read-after-write dep., or RAW, →)
The value produced by the source a of the dependence is used by the sink b (information flows from a to b).

Anti Dependence: (aka write-after-read dep., or WAR, ↔)
Arise from reusing names (e.g., variables or memory locations), a value used by the source a of the dependence is *overwritten* by the target b (information does not flow from a to b).

Output Dependence: (aka write-after-write dep., or WAW, ↠)
Also arise when names are reused, a value written by the source of the dependence is *overwritten* by the target.

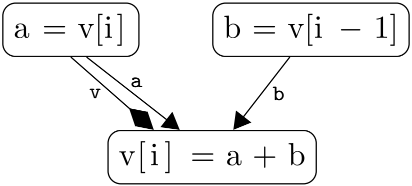
Loop Carried Dependencies

Dependencies across loop iterations (**`for`**, **`while`**, **`do`**):

- **Distance vectors**:
  Attached to dependencies, expressing the relation between loop iterations reading and/or writing a value.
- Typically involve array accesses or pointers
- Example:

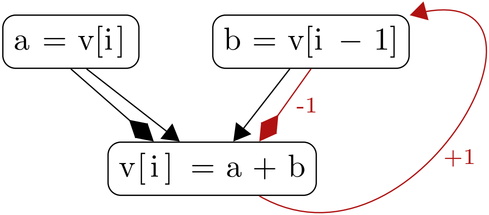
```
void foo(int *v, int n)
{
  for(int i = 1; i < n; i++) {
    a = v[i];
    b = v[i - 1];  // from previous iteration!
    v[i] = a + b;
  }
}
```

Dependence Graph with Distance Vectors



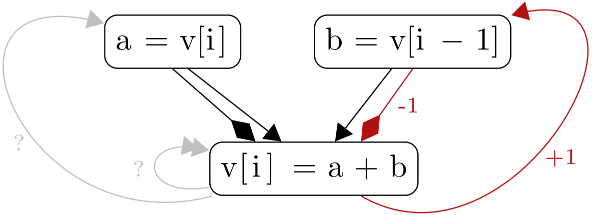
Dependencies within a single iteration of the loop

Dependence Graph with Distance Vectors



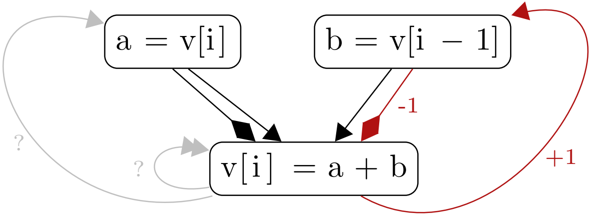
Dependencies across loop iterations (focusing on `v`)

Dependence Graph with Distance Vectors



Dependencies you might imagine, but that are in fact not there

Dependence Graph with Distance Vectors

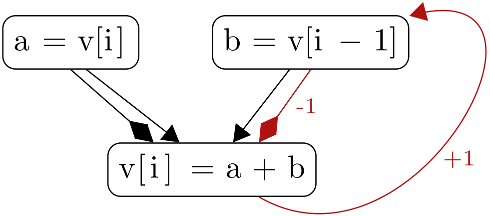


Dependencies you might imagine, but that are in fact not there

Group Exercise: Parallelism in Loops

Discuss in groups of 2-3:

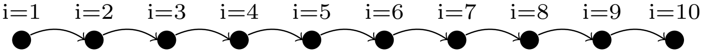
- Can the loop from before be parallelized?
- Which forms of parallelism are available?
  (SIMD, MIMD, Thread-Level?)
- Which role do the dependencies play?



Iteration Space Graphs

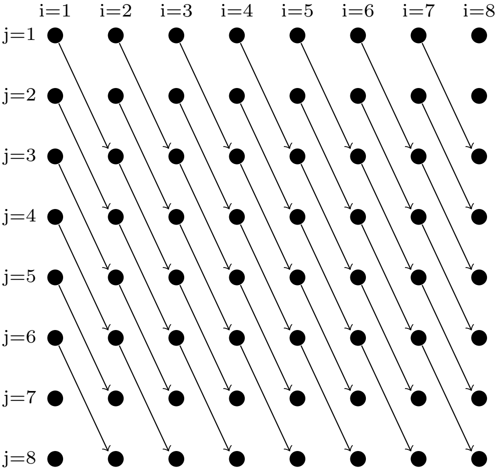
Represent dependencies between loop iterations:

- Points represent iterations
- Arrows dependencies between them
- Example from before:



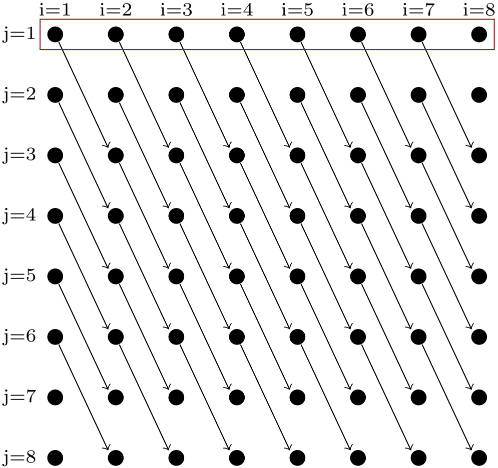
Iteration Space Graphs of Nested Loops

Iteration space graphs also work in higher dimensions:



Iteration Space Graphs of Nested Loops

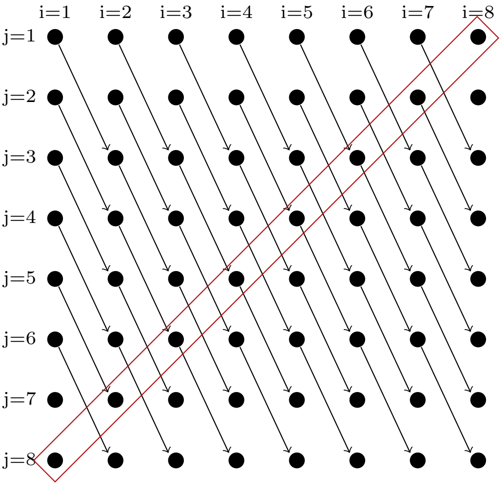
Iteration space graphs also work in higher dimensions:



Lots of (data-level) parallelism immediately visible.

Iteration Space Graphs of Nested Loops

Iteration space graphs also work in higher dimensions:



. . . also when we rotate the matrix.

Iteration Space Graphs of Nested Loops (2)

A matching program for the iteration space graph from before:

```
void foo(int v[10][10])
{
  for(int i = 1; i < 9; i++) {
    for(int j = 1; j < 8; j++) {
      v[i+1][j+2] = v[i][j] + 1;
    }
  }
}
```

Summary

- Forms of parallelism:
  - Pipelining / instruction-level parallelism (MIMD)
  - Data-level parallelism (SIMD)
  - Thead-level parallelism
  - Most modern computers allow to exploit all three forms
- Amdahl's Law:
  Amount of *sequential* code limits speedup.
- Dependencies:
  - Impose an ordering on the execution of operations
  - Data-dependencies: RAW, WAR, WAW
  - Control-dependencies
  - Loop-carried dependencies and distance vectors
  - Iteration space graphs

Todays TD

Explore data-level parallelism and its relation to dependencies:

- Reason about dependencies in programs (rather interactive)
- Play with vectorization, i.e., the compiler extracts SIMD-like data-parallelism in loops
- Some practical considerations with caches
- …