INF5380/9380 NORBIS course
High-Performance Computing in Bioinformatics

Introduction to parallel programming

Torbjørn Rognes
torognes@ifi.uio.no

Department of Informatics, UiO
15 March 2022

UiO : Universitetet i Oslo

The need for parallelisation

- We want to process ever larger amounts of data
- We want to do it in reasonable time
- The processing power of a single CPU core is not increasing very much any more; CPU frequencies have almost stopped at about 3GHz
- Moore's law about the doubling of the number of transistors every 18 months still apply, but mainly results in more cores instead of faster cores.
- Parallel programming is required to tap the potential of modern computers.

Hardware enabling parallel execution

- Grid computing, multiple individual computer systems (e.g. Folding @ Home)
  - Communicates over Internet
- Multiple computers in one cluster (e.g. Saga or Fox)
  - With a fast network between them
- Multiple cpus in one computer (e.g. a server)
  - shared memory
  - several ordinary cpus
  - graphics processing units (gpus) or other co-processors
- Multiple cores in one cpu (e.g. most laptops)
  - 2, 4, 8, 16, 32 or even more cores is now common in many cpus
- Multiple execution units in one core
  - Two or more logical/arithmetic units
  - Hyperthreading (HT)
- Bit-level parallelism = SIMD within a register (SWAR)
  - Wide registers (e.g. 64, 128, 256... bits) that can be divided into smaller units (e.g. 8 bits) that can be operated on independently

Definition of serial and parallel systems

- We are considering basic operations like:
  - add two numbers (or other mathematical/logical operations)
  - compare two numbers
  - retrieve a number from memory
  - store a number in memory
  - jump to somewhere else in the program (if some condition)

- A serial system can only perform one of these operations per unit of time

- A parallel system can perform two or more of these simultaneously

A serial vs a parallel program

- A problem may involve N separate independent steps
- If we have one processing core available, this problem can be solved in N units of time
- If we have k processing cores available, the problem may ideally be solved in N/k units of time
- If k>=N, the problem may be solved in 1 unit of time

Parallelisation may be easy or hard

Easy problems:

- Some problems are easy or trivial to parallelise
- "Embarrassingly parallel"
- Problem may be divided into parts that can be solved independently.
- No communication between processes needed.
- Example: Searching multiple queries against a database. Each search is totally independent of the others.

Hard problems:

- Other problems are hard or impossible to parallelise
- Difficult or impossible to divide the problem into independent parts. E.g. each step depends on the previous step.
- Extensive communication between processed required

Example 1: Easy

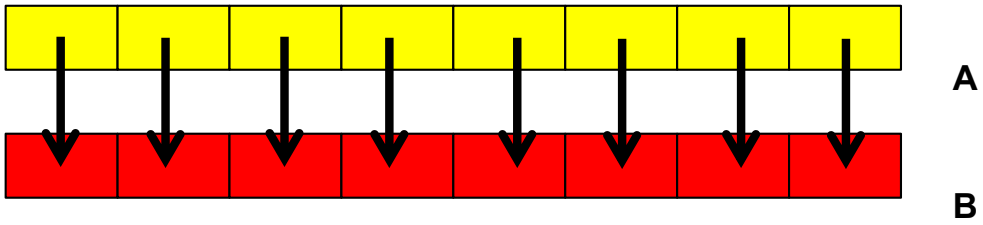
```
/* compute b[i]=2*a[i] for i=0 to N-1 */

#define N 8
int a[N] = { 94, 43, 48, 58, 1, 27, 28, 53 };
int b[N];

for(int i=0; i<N; i++)
{
        /* independent */
        b[i] = 2 * a[i];
}
```

Example 1: Difficult

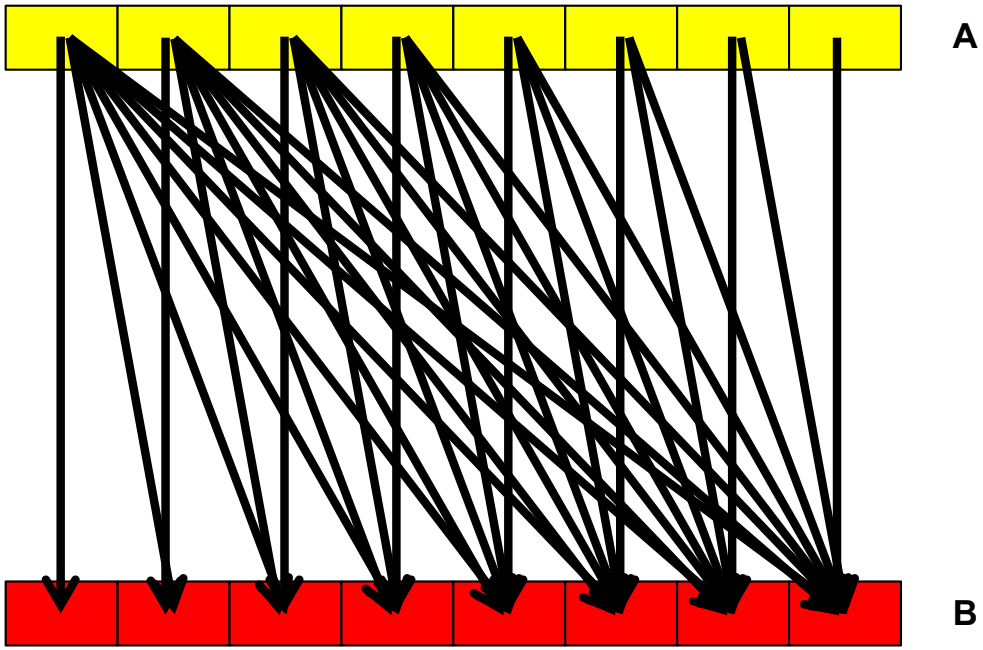
```
/* compute b[i]=sum(a[0], a[1], ..., a[i]) for i=0 to N-1*/

#define N 8
int a[N] = { 94, 43, 48, 58, 1, 27, 28, 53 };
int b[N];

b[0] = a[0];

for(int i=1; i<N; i++)
{
        /* dependent on previous value */
        b[i] = b[i-1] + a[i];
}
```

Data dependencies

Easy case



Difficult case



Two main parallel computer models

- Distributed-memory systems: processes communicate over a network
  - grid
  - cluster

- Shared-memory systems: processes communicate through shared memory
  - single machine
  - multiple cpus, cores, execution units

Distributed-memory models

- Computers in a cluster with separate memory communicating by sending messages to each other over a network
- Communication may be direct node to node or using broadcasting (one to many)
- MPI = Message Passing Interface
- openmpi
- **MapReduce** is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster.

Shared-memory models

A Parallel Random-Access Machine (PRAM) is a model of a shared-memory parallel computer. Four categories are defined:

- **EREW**: Exclusive Read and Exclusive Write
- **ERCW**: Exclusive Read and Concurrent Write
- **CREW**: Concurrent Read and Exclusive Write (most common)
- **CRCW**: Concurrent Read and Concurrent Write

Notes:

- **Exclusive** read or write means that only one process may read or write to the same memory simultaneously.
- **Concurrent** read or write means that only one process may read or write to the same memory simultaneously.
- With **concurrent write**, it is necessary to define the result of the operation (arbitrary, based on rank, sum/max, ...)

Coding on shared-memory architectures

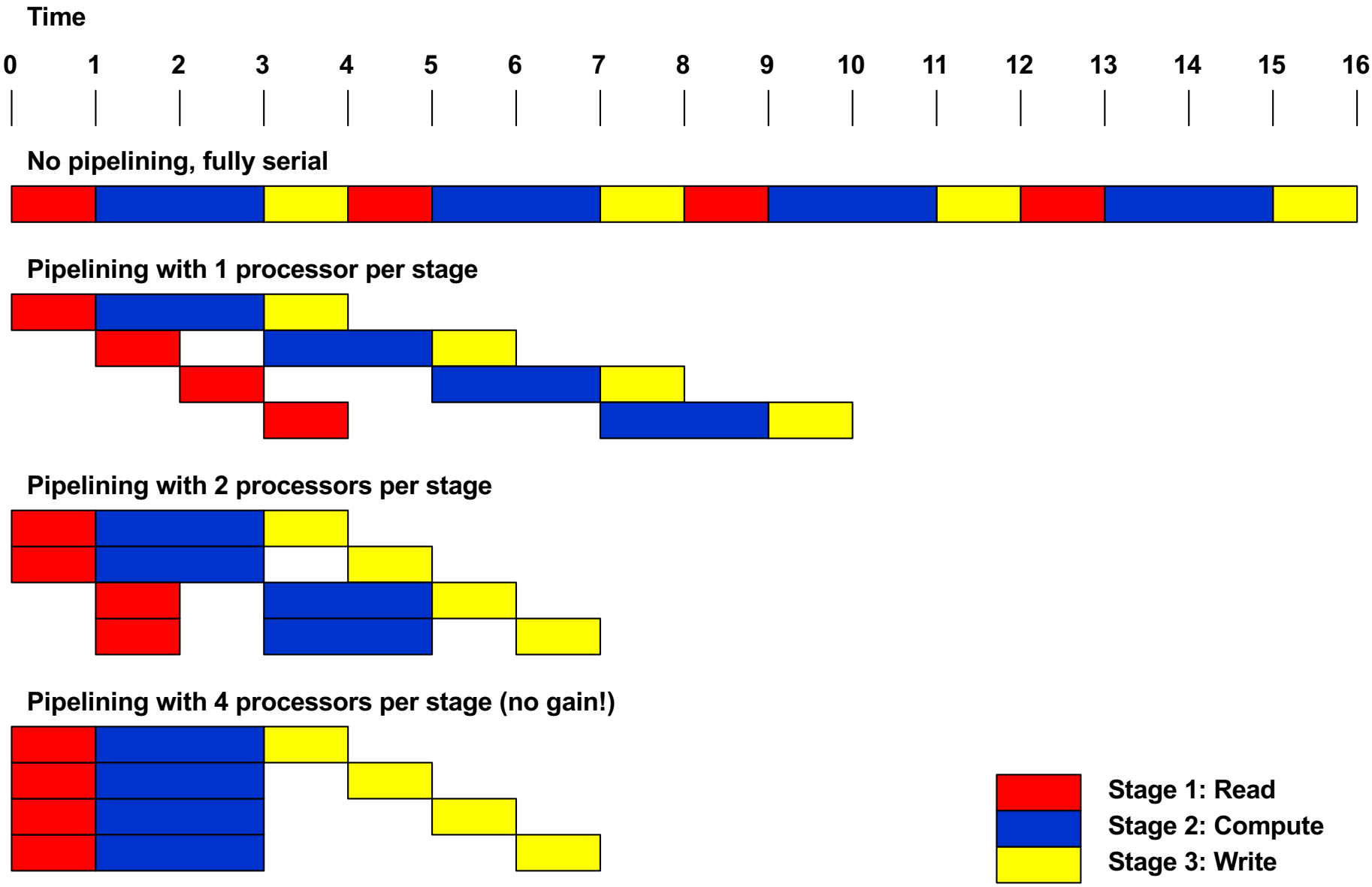
- OpenMP: We give hints to the compiler about parts of the code that may be parallelised, but leave most of the job to the compiler.

- Threads (pthreads): We manage creation and termination of separate threads ourselves and take care of synchronisation.

Pipelining

- We have a problem that requires, for instance, 3 sequential stages of operations on each data value:
  1. read (red)
  2. compute (blue)
  3. write (yellow)
- We cannot write data simultaneously
- We can have multiple (N) execution units that perform the read and compute stages.
- How can we do this quickly?
- Use pipelining!

Pipelining example



Flynn's taxonomy

SISD: Single Instruction stream, Single Data stream

- Perform one operation on one value (traditional)

SIMD: Single Instruction stream, Multiple Data stream

- Perform the same operation on many different values simultaneously (GPU, SIMD-instructions, vectorisation)

MISD: Multiple Instruction stream, Single Data stream

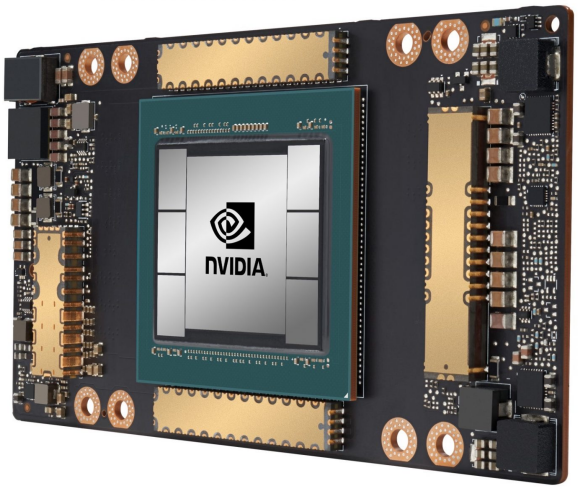
- Perform many different operations on the same value simultaneously (Uncommon, for fault detection)

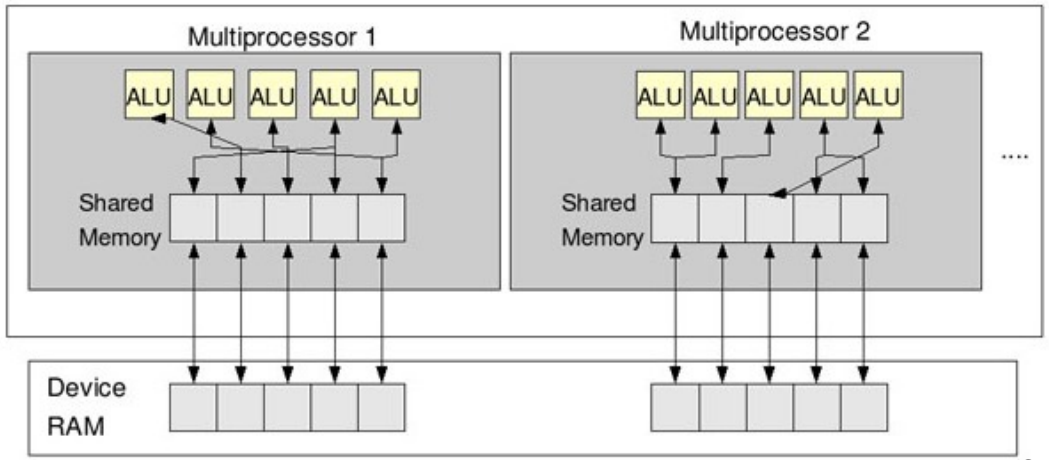
MIMD: Multiple Instruction stream, Multiple Data stream

- Perform many different operations on many different values simultaneously (multiple core, distributed systems)

Graphics Processing Units (GPUs)

- Originally made for games, now also for high-performance computing
- Manufacturers: Nvidia, ATI and others
- Thousands of cores total
- Divided into X multiprocessors, each with Y stream processors
- Relatively simple processors
- All stream processors within one multiprocessor execute the same code in sync.

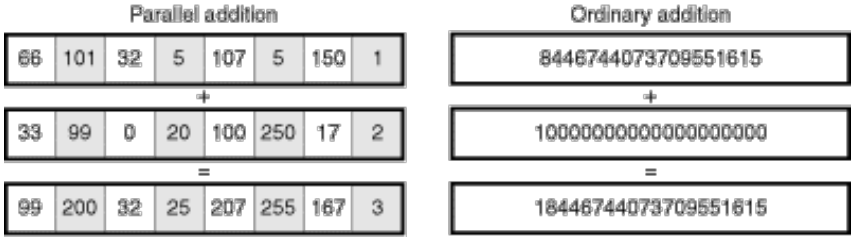


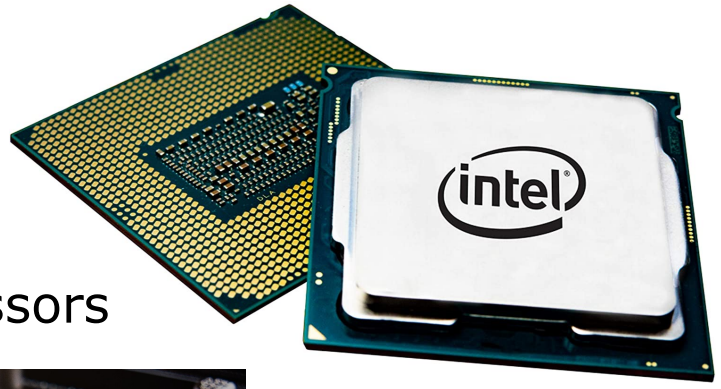


Sources: Nvidia, ATI

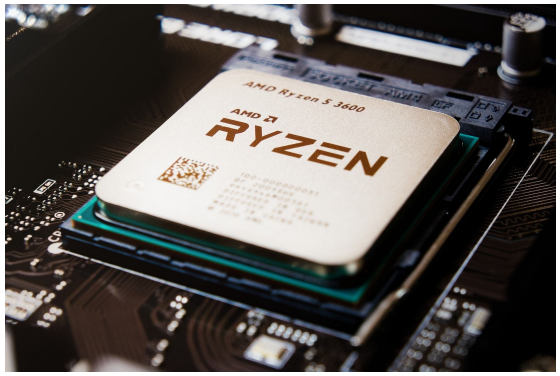
SIMD technology in ordinary CPUs

- SIMD = Single-Instruction Multiple Data
- Perform the same operation in parallel on multiple data streams
- Kind of vector processing
- Used for high-speed processing of various media like images, sound, video, as well as cryptography
- Allows up to 64 simple operations to be executed in parallel





- Embedded in most modern microprocessors



Sources: Intel, AMD, Apple

Writing SIMD code

- Assembler

```
__asm__ (”psubsb xmm15, xmm12”);
```

- SIMD Intrinsics

```
#include <tmmintrin.h>
E = _mm_subs_epi8(E, R);
```

- C (with optimizing compiler)

```
for(int i=0; i<16; i++)
    E[i] = E[i] – R[i];
```

Popular parallel programming model

- The server/client or parent/child model
- One main process (server or parent) spawns one or more child processes (clients or children)
- The parent process usually:
  - handles user input
  - creates the child processes
  - allocates work to each of the child processes
  - collects the results from the child processes
  - ensures all children are finished
  - handles user output
- The child process usually:
  - waits for work to become available
  - carries out the work
  - submits results
  - loops until asked to terminate or no more work is available
  - terminates

Creating a new process or thread

- Create a new process:

```
int fork(void)
```

Returns the process ID of the new child process to the parent, while the child process receives the value 0. Both the parent and the child now executes in parallel from the same starting point, identical data etc.

- Create a new thread:

```
int pthread_create(pthread_t *restrict thread,
                   const pthread_attr_t *restrict attr,
                   void *(*start_routine)(void *),
                   void *restrict arg)
```

Starts a new thread that executes the start_routine function.

Waiting for a child to finish

- Wait until any child process has finished:

```
pid_t wait4(pid_t pid,
            int *stat_loc,
            int options,
            struct rusage *rusage)
```

  Waits (pauses the calling process) until a given child process with given process id has terminated. Returns information about resource usage.

- Wait until a given child thread has finished:

```
int pthread_join(pthread_t thread,
                 void **value_ptr)
```

  Waits (pauses the calling thread) until the given child thread has terminated. Returns result value from child thread.

Thread synchronisation

- Synchronisation is often necessary for correct operation in parallel systems.
- Critical sections need to be protected.
- If two or more threads need to update a common variable they need to do it in an orderly manner.
- We need to guarantee mutual exclusion of certain events.
- Mechanisms:
  - lock
  - monitor
  - semaphore
  - condition variable
  - mutex
  - ...
- Requires hardware support for certain atomic operations

Example without synchronisation

- Example of withdrawal of money from bank account without proper synchronisation.
- Two different invoices of $100 and $200 are to be paid from a bank account.

Thread 1	Thread 2	Account
Read amount: $1000		$1000
	Read amount: $1000	$1000
Deduct $200: $800		$1000
	Deduct $100: $900	$1000
Write amount: $800		$800
	Write amount: $900	$900

Example with synchronisation

- With proper synchronisation using a lock
- Critical sections are indicated

Thread 1	Thread 2	Account
Acquire lock	Acquire lock	$1000
Lock it	Wait	$1000
Read amount: $1000	Wait	$1000
Deduct $200: $800	Wait	$1000
Write amount: $800	Wait	$800
Unlock it	Wait	$800
...	Lock it	$800
...	Read amount: $800	$800
...	Deduct $100: $700	$800
...	Write amount: $700	$700
...	Unlock it	$700

Thank you!