Concurrent programming languages

Week 1: Introduction

Stefan Kuhn
Introduction

This session intends to:

• Clarify the content and teaching mode of the module
• Give a basic overview of the area
• Clarify terminology
This module focuses on practical concurrent programming.

We will look at the practical aspects of implementing concurrent algorithms.

We will study the features of some current parallel languages.

We are going to use different programming languages for this:

- Java
- Scala
- Erlang
- C++
Introduction

- This module is called Concurrent Programming Languages.
- The term *concurrent* is used as a general term encompassing *parallel* and *distributed* computing (as usual, the terminology is not 100% uniform).
- Concurrent: More than one action happening in progress. Must not actually be at the same time, can be achieved e.g. via time slices. May look like thing happen at the same time, but they do not actually do so. A system like X on Linux can run multiple applications on simple single core processors.
Introduction

- **Parallel:** Several processes are carried out simultaneously. Requires appropriate hardware support (e.g. multi core processors). If programmed correctly, this can speed up tasks.

- **Distributed computing:** Uses several computers, which communicate using messages. Computers and processes run in parallel.

You will find different views on this!
Introduction

• Distributed computing in our context is where one overall task is run on several machines

• A client-server architecture, e.g. a webserver where some things run on the server (say as Java servlets) and others on the client (say as JavaScript) is a type of distributed computing, but not the focus here

• In “real” distributed computing, there are different architectures (peer-to-peer, client-server etc.) as well

• For distributed computing, message passing is the most important technique
Introduction

• For a long time, parallel or distributed computing was not available for the average user

• Computers had a single processor with one instruction executed at a time

• Hardware which went beyond this was unaffordable outside big corporations and government

• Network connections were also expensive or non-existent, so distributed computing was also not easy to do
Introduction

- Programming, programming languages and programming training were very much for the sequential case
- Nowadays, parallel features are commonplace: PC processors are multicore, there are GPUs, functional units like cache memory exist in multiples etc.
- Networked computers are standard as well
- With this, parallel features have been added to programming languages or new parallel languages have been designed
Introduction

IBM BG/Q (Blue Gene) Compute Chip with 18 cores (PU) and 16 L2 Cache units (L2)

Example of typical parallel computer cluster

https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial
Introduction

- Parallel and distributed computing are closely related.
- Since distributed computing is less tightly coupled, it is possible to make more resources available in distributed computing than in parallel computing.
- On the other hand, distributed computing requires more work in programming to actually make use of the resources.
Introduction

• Parallelism is commonly used in all sort of tasks in the real world
• At a construction site, many workers work in parallel
• Factories run several assembly lines in parallel
• It can speed up and make results possible which would not be achieved by a single worker in realistic time frames
• Individual humans are not good in parallel processing, typically we can do one thing at a time, and the use of multi tasking is debated
Introduction

- Parallelism can speed up (some) tasks, if it can help to solve more complex tasks is different
- In human societies, specialisation of labour is a key element
- This is less common with parallelism, typically there is “more of the same”
Introduction

- **Amdahl’s law** gives the theoretical speedup possible with parallelizing some part of the code.
- In its simplest form, it says \( speedup = \frac{1}{1-p} \)
- With \( p \): fraction of code that can be parallelized
- This assumes an infinite number of processes, so with \( p=1 \) the speedup is infinite.
Introduction

• More relevant seems a consideration of the number of processes:

\[ speedup = \frac{1}{(1-p)+\frac{p}{N}} \]

• Where N is the number of processes
• This does not consider overheads
Introduction

• The speedup is limited by the non-parallel part:

\[ speedup(s) \leq \frac{1}{1-p} \]

• e.g. if half of the code is serial, the execution time can be shortened by 50% maximum
Introduction

- This means there are bounds beyond which more processes don’t help:

<table>
<thead>
<tr>
<th>N</th>
<th>P</th>
<th>0.5</th>
<th>0.9</th>
<th>0.95</th>
<th>0.99</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.82</td>
<td>5.26</td>
<td>6.89</td>
<td>9.17</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>1.98</td>
<td>9.17</td>
<td>16.80</td>
<td>50.25</td>
<td></td>
</tr>
<tr>
<td>1,000</td>
<td>1.99</td>
<td>9.91</td>
<td>19.62</td>
<td>90.99</td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td>1.99</td>
<td>9.91</td>
<td>19.96</td>
<td>99.02</td>
<td></td>
</tr>
<tr>
<td>100,000</td>
<td>1.99</td>
<td>9.99</td>
<td>19.99</td>
<td>99.90</td>
<td></td>
</tr>
</tbody>
</table>

Speedup with increase of processes (N) and parallel part (P)
https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial
Introduction

- The good news is that often the serial part is small and gets smaller with larger problems.
- The serial parts may be initialization or I/O, which is either constant or linear with problems size, whereas the parallizable parts may be the actual algorithm, which could be polynomial or worse.
Introduction

- **Gustafson’s law** (or Gustafson–Barsis's law) considers sequential part constant:

  \[ \text{speedup} = s + p \times N \]

- That gives a lot more possibilities, since it is essentially linear

- It means not that we can solve a problem with a given dataset in even shorter time, but we can take more data and still solve it in (roughly) the same time

- That means parallelization can make problems *scaleable*
Flynn’s taxonomy is a classic taxonomy of (concurrent) computer systems:
Introduction

Single Instruction, Single Data (SISD):

- Only one instruction stream is being acted on by the CPU during any one clock cycle
- Single Data: Only one data stream is being used as input during any one clock cycle
- Used on old-fashioned PCs (or other computers)
Introduction

Single Instruction, Multiple Data (SIMD):

- **Single Instruction**: All processing units execute the same instruction at any given clock cycle
- **Multiple Data**: Each processing unit can operate on a different data element
- **Suitable for specific problems with high degree of regularity**
- **Used in GPUs or so-called vector machine (supercomputers of the 1980s)**
- **Signal processing, often via FPGAs, is another area**
Multiple Instruction, Single Data (MISD):

- Multiple Instruction: Each processing unit operates on the data independently via separate instruction streams.
- Single Data: A single data stream is fed into multiple processing units.
- Not common in practice
- Pipelining could be considered a case of this: Instructions are split in phases, and with every clock cycle, different phases from several command are executed
Multiple Instruction, Single Data (MISD):

- Pipelining:

<table>
<thead>
<tr>
<th>Instr. No.</th>
<th>Clock cycle</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>IF</td>
<td>ID</td>
<td>ID</td>
<td>EX</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.
Multiple Instruction, Multiple Data (MIMD):

- Multiple Instruction: Every processor may be executing a different instruction stream
- Multiple Data: Every processor may be working with a different data stream
- Most modern computers fall into that category
- It includes clusters, grid computers, also multi-core PCs (that does not mean all these computers work the same way)
Introduction

In an MIMD environment, the following scenarios exist:

- **Shared memory**: All processors have direct access to common physical memory
- Typically the case in multi-core processors
- Those are normally of the Symmetric Multi-Processor (SMP) type: Single address space, equal access, equal speed
- This is normally not the case for the cache memory, but that is typically automatically managed
There is a distinction between uniform memory access (UMA) and non-uniform memory access (NUMA). In UMA, access is at uniform costs for all memory. In NUMA, access can be at different costs (but is still direct). Parallel Random Access Machine (PRAM) is a name for an “ideal” shared memory computer (a UMA machine) in theoretical work.
In an MIMD environment, the following scenarios exist:

- **Distributed memory**: Processes can only “see” local memory
- Other memory is accessed by communicating with other processes
- Clusters are built like that
- Real architectures vary widely
In an MIMD environment, the following scenarios exist:

- **Distributed shared memory**: There is shared and distributed memory

As usual, there are pros and cons:

- Shared memory is easier to handle, but can cause race conditions
- Distributed memory forces more careful algorithm design
- Distributed memory can make capacities available not possible otherwise
Another term used is **symmetric multiprocessing** (SMP), which is essentially the same as a shared memory system.

Sometimes this is restricted to systems without caches (but then, caches are transparent mostly).

Generally, it means a system where processors have access to resources at uniform cost.

(Asymmetric multiprocessing is largely obsolete, early systems had for example one processor for the OS and one for applications.)
Introduction

- **Massively parallel** or massively parallel computers are not strictly defined (a similarly vague term is supercomputer, which tend to be parallel)
- What was massively parallel once is no longer today
- A GPU can be considered massively parallel as well as a large cluster or grid computer
- Sometimes only clusters with specialist interconnect hardware are considered massively parallel, not those with commodity interconnect (ethernet)
Introduction

- When discussing shared and distributed memory so far, we were talking about hardware – processors and RAM.
- Software (the operating system or application software) can have an influence as well.
- In standard operating systems, **processes** have their own memory, so several processes form a shared memory system.
- **Threads** share memory, so multiple threads form a shared memory system.
Introduction

• Hybrid systems have shared and distributed memory
• An example would be a cluster with multi-core CPUs in each node
• Quite common today
We discussed speedup earlier.
This was speedup in latency, which is simple $\frac{\text{time}_{\text{old}}}{\text{time}_{\text{new}}}$.
So if the task takes half the time, we have a speedup of 2.
There is also speedup in throughput, which refers to CPU instructions.
Speedup is not always restricted to number of processors, there are (rare) cases of super-linear speedup.
Introduction

- Concurrent applications require programs written in a particular fashion
- Programming languages may support that more or less well
- In that sense, there are no concurrent programming languages, but languages suitable for certain types of concurrent programming
- We are looking are how some languages support concurrent programming
- In some cases, libraries can make languages parallel (OpenMP, MPI)
Summary

- Concurrency is the simultaneous execution of programs
- Parallel and distributed are subtypes of it
- Multiple Instruction, Multiple Data (MIMD) is the typical model used today
- Shared memory and distributed memory is an important distinction