159.341 Programming Languages, Algorithms & Concurrency

Concurrent Programming
Types of Parallelism

Daniel Playne
d.p.playne@massey.ac.nz

Concurrent Programming

- Hardware Classification
- ***Types of Parallelism***
- Algorithm Design
- Paradigm
- Program

Types of Parallelism

Another useful way to think about problems is to consider the kinds of parallelism that can be exploited in a problem.

The types of parallelism:

- ***instruction-level*** parallelism
- ***thread-level*** parallelism
- ***data*** parallelism
- ***task*** parallelism
- ***hybrid*** parallelism
- ***ad-hoc*** parallelism

Instruction-Level Parallelism

The lowest level of parallelism is ***instruction-level parallelism*** that exploits parallelism by using multiple function units to execute instructions simultaneously.

The two main approaches for instruction-level parallelism are:

- pipelining
- multiple issue

Instruction-Level Parallelism

Pipelining is similar to the idea of a factory line where different teams work on different parts of the problem at the same time.

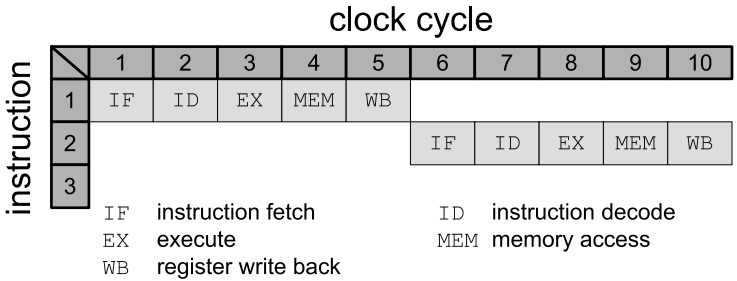
When a factory produces a run of cars, each factory worker doesn't wait for the first entire car to be completed before starting the next.

At any one point in time, many factory workers are working on different tasks for different cars at the same time.

Instruction-Level Parallelism

Likewise executing instructions on a computer can be split into different functions - *fetch instruction*, *decode instruction* etc.

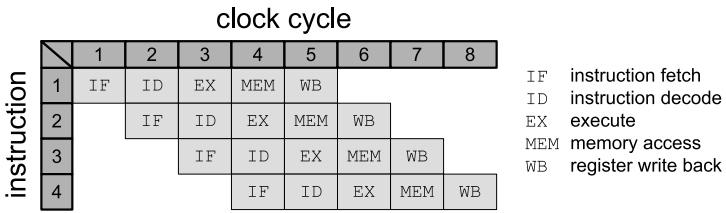
Executing each instruction after the previous one completes would look like this:



Instruction-Level Parallelism

However, when multiple instructions are executed one after another the functions can be pipelined.

The time to compute a single operation remains unchanged, but different functions of many operations can be computed at the same time.



Instruction-Level Parallelism

Multiple Issue allows different functional units to simultaneously execute different instructions in a program.

Multiple issue requires instructions that can be executed at the same time (usually relies on speculation).

Speculation is a process where the compiler (static multiple issue) or processor (dynamic multiple issue) makes a guess about an instruction and executes code based on this guess.

Instruction-Level Parallelism

For example, the processor may guess that the operation `z = x+y` will result in a positive value and simultaneously compute `z = x+y` and `w=x`.

```
z = x + y;
if(z > 0) {
   w = x;
} else {
   w = y;
}
```

This must be reversible so that the processor can go back and execute `w = y` if the guess was wrong.

Instruction-Level Parallelism

Instruction-level parallelism is generally implemented either in the compiler or in the processor hardware. This level of parallelism can be exploited in a single processing core/unit and does not require multiple cores.

The next level of parallelism is one that the programmer has control over - ***thread-level parallelism***

Thread-Level Parallelism

Thread-level parallelism is a form of parallelism where a program is split into multiple threads that each perform some part of the computation.

Unlike instruction-level parallelism, this is usually explicitly defined by the programmer.

This requires the programmer to analyse their program and determine how the computation can be split up into multiple threads and manage the communication and synchronisation between them.

Thread-Level Parallelism

Some problems are trivial to split into multiple threads and require no communication at all between different threads - known as *embarrassingly parallel problems*.

Most require threads to communicate data and/or synchronise with each other to ensure they compute the correct result.

Multi-threading libraries for programming languages must support the creation of threads and mechanisms for them to communicate/synchronise with each other.

Data Parallelism

Data parallelism considers how the data of an application can be processed in parallel.

With data parallelism we consider how can the same task be performed on different pieces of data.

This type of parallelism is well-suited to SIMD machines where the data is distributed to the different processing cores and the same set of instructions is executed for each data item.

Data Parallelism - Conway's Game of Life

A good example of a data parallel solution is Conway's Game of Life (GoL).

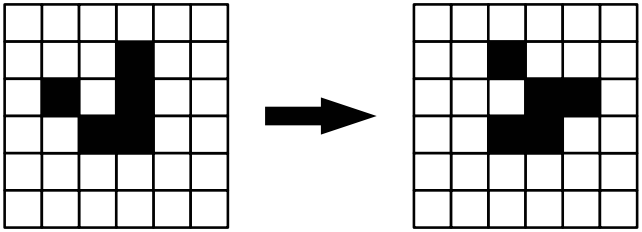
In GoL, there is a 2D grid of cells that are either *alive* or *dead*. At each generation, every cell will count the number of *alive* neighbours and update the state of the cell according to the following rules.

1. Any *alive* cell with < 2 *alive* neighbours dies.
2. Any *alive* cell with 2 or 3 *alive* neighbours survives.
3. Any *alive* cell with > 3 *alive* neighbours dies.
4. Any *dead* cell with 3 *alive* neighbours becomes an *alive* cell.

Data Parallelism - Conway's Game of Life

Each generation of Conway's Game of Life will apply these rules to every cell to compute the next generation.

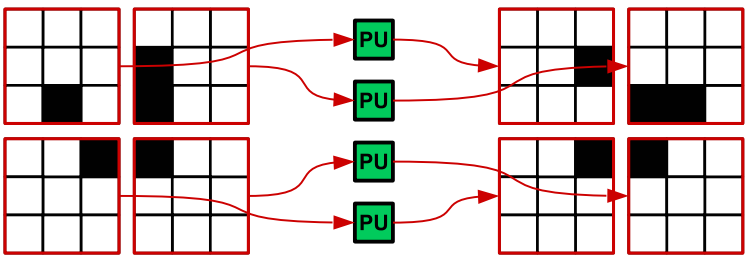
Two grids are required to ensure that the values from the previous generation are always used the cells.



Data Parallelism - Conway's Game of Life

The computation of a cell's new state is independent of every other cell, so the cells could be assigned to different threads to compute.

Depending on the architecture, each thread may be assigned a single cell or a group of cells.



Data Parallelism

Graphical Processing Units (GPUs) are a good example of an architecture designed for data-parallel processing.

GPUs are highly parallel processing units designed for rendering 3D graphics. The data in a 3D scene (vertexes/pixels) are divided up between the many processing cores in a GPU.

Each data item can be processed without any communication between cores. The data-parallel model of computing is how GPUs are able to effectively use thousands of cores while most CPUs still have in the region of 2-32.

Task Parallelism

Task parallelism exploits parallelism in a problem by dividing the program up into separate tasks to be computed (mostly) independently.

Each task can be scheduled onto a different processing unit on the machine which can execute that task independently.

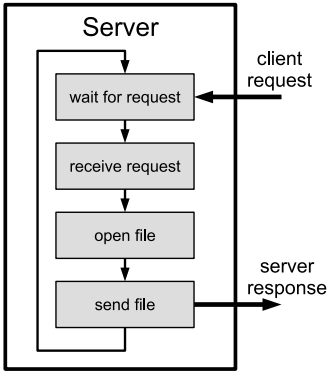
This type of parallelism is well-suited to MIMD machines as each task generally accesses separate data and performs a different sequence of instructions.

Task Parallelism

A good example of task parallelism is a web server handling requests from clients.

When the server receives a request, it will read the requested file and send it to the client.

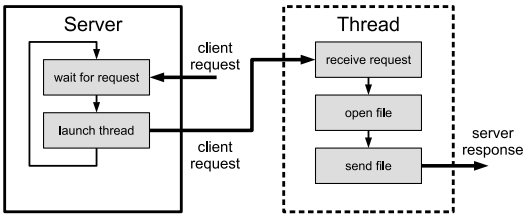
However, it can only respond to one request at a time. If a file takes a long time to send, all other requests will stall.



Task Parallelism

Responding to each request could be viewed as a separate task, the server could launch a new thread to respond to each new request.

If one request takes a long time to process, it will not block any other (smaller) requests from being processed.



Hybrid Parallelism

Data and ***task parallelism*** are not mutually exclusive approaches.

Many applications may make exploit both types of parallelism in different parts of the program or even at the same time.

Example a program may be split into separate tasks that each make use of data parallelism.

Ad-hoc Parallelism

Finally, some programs may not have an obvious structure to the parallelism that can be exploited (does not fit into either data or task parallelism).

Such applications may still be computed by multiple threads in an *ad-hoc* or unstructured approach.

This approach to parallelism is much harder to describe as, by its very nature, there is no consistent approach to designing programs this way.

Exploiting Parallelism

There is not just *one* way to parallelise a particular problem or just *one* type of parallelism that can be exploited.

The choice of how to develop a parallel program and what type of parallelism to target depends on:

- The nature of the problem
- The machine you are going to target
- The language/API/libraries you are going to use

Types of Parallelism - Hardware

The type of parallelism a program utilises is often closely tied to the type of machine it is intended to run on.

In general, ***task-parallelism*** is more suited to MIMD architectures than SIMD architectures.

Computing different tasks when each core must execute the same instruction is difficult.

Types of Parallelism - Hardware

Data-parallelism is generally well-suited to SIMD architectures where each core executes the same instruction on different data.

Game of Life - the process of updating a cell is the same so each core will execute the same instructions on its own set of cells.

MIMD machines can still compute data-parallel programs, the sequence of instructions the cores execute just happen to be the same.

Types of Parallelism - Memory

The *memory* architecture of a machine may also influence the design of the program.

Distributed memory often has less of an impact on a task parallel solution as separate tasks usually have their own data.

Data-parallel methods can still be implemented on distributed memory machines but communication between machines is usually required.

Types of Parallelism - Memory

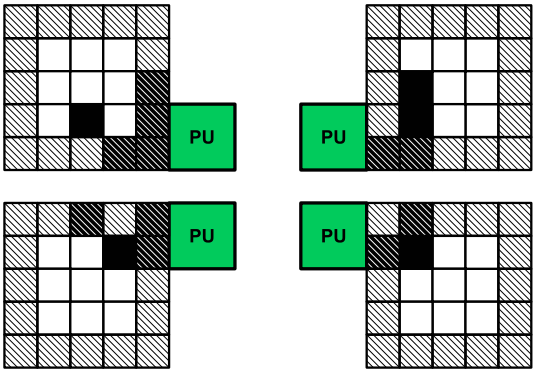
Game of Life - each cell must access the neighbouring 8 cells during an update.

If the sections of the grid are divided between different areas of memory, the processing units must communicate to exchange cells on the board of the sections.

For this type of problem, additional cells are often stored around the border of each section which are then updated after each generation.

Types of Parallelism

In this arrangement there is duplication of data as each cell on the border of a region will be stored in more than on location.



Summary

Types of parallelism

- ***instruction-level*** parallelism
- ***thread-level*** parallelism
- ***data*** parallelism
- ***task*** parallelism
- ***hybrid*** parallelism
- ***ad-hoc*** parallelism