XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 20: Summary

Format for this year's exam

The final assessment for this module will be an open book exam that will be taken under exam conditions using Gradescope...

- The Gradescope assessment will available from **1600 Thursday 25th May2** (0900 UK time).
- You will have **2 hours** to complete the exam unless you have been informed otherwise.
- You will submit your answers directly via Gradescope.
- You will need all of the material covered in lectures 1–19 for this final assessment.
- **No marks** will be given for copying material verbatim from your lecture notes.

Code

You may be asked to **read** snippets of C-code, with may include OpenMP, MPI, and/or OpenCL.

You will **not** be expected to write **working** code.

You **may** be asked to write **pseudo-code**.

- No specific pseudo-code format is expected.
- The important thing is to be **clear**.
- If relevant, you should use the **correct** function names from OpenCL or MPI, and the **correct** `pragmas` for OpenMP.
- The working of standard C (*e.g.* loops *etc.*) should be obvious.

Exam structure

The exam will consist of 2 main questions, each of which has numerous subquestions.

Each main question will be marked out of 25, so the total mark will be out of 50.

- This will be added to the 50% for the courseworks.

Each main question may address multiple architectures, *e.g.* the same problem for shared memory CPU, distributed memory CPU, and/or GPGPU.

- In this sense, follows more the organisation of material in this lecture (Lecture 20).

Past papers

Past papers are available on Minerva.

XJCO3221 $\rightarrow$ Assessment Information $\rightarrow$ Past Exams

As per School policy, answers are not provdided.

Prior to 2019-20, this was a **closed book exam** that had some "bookwork" questions asking you to *e.g.* define parallel speed-up, derive Amdahl's law *etc.*

Since the exams have been online, there are **no such questions**.

- Will assess **understanding** and **application** of the material.

This lecture

Want to understand **general parallel programming concepts** that transcend particular architectures.

- Architectures, parallel frameworks *etc.* will change in future.
- General parallel programming concepts will **not**.

In this final lecture we will summarise all of the material by **parallel concept** rather than by architecture, API *etc.*

- Easier to see the commonalities.

Previous lectures

This module has been structured to focus on one parallel architecture after another.

1. Shared memory CPU (with OpenMP); Lectures 2-7.
2. Distributed memory CPU (with MPI); Lectures 8-13.
3. General purpose GPU (with OpenCL); Lectures 14-19.

The practical elements of the module **(worked examples, worksheets and courseworks)** also followed this structure.

- Some **mentions** were out-of-order, *e.g.* OpenMP barriers mentioned in Lectures 11 and 17.

Why parallel?

Lectures 1 and 4

Parallel hardware allows **simultaneous** computations.

- Necessary to improve performance as clock speeds limited by physical constraints.
- Subset of **concurrency**, which is 'in the same time frame' but could be *e.g.* time-sharing on a single core.

Want to attain good **scaling** - decrease in parallel computation time $t_{\rm p}$ for increasing number of **processing units** *(threads, processes* etc.*)* p.

Measuring parallel performance

Lecture 4

Two useful metrics for parallel performance are the **speed-up** S and **efficiency** E

$$S = \frac{t_{\rm s}}{t_{\rm p}} = \frac{\text{(serial execution time)}}{\text{(parallel execution time)}} \quad ; \quad E = \frac{t_{\rm s}}{p t_{\rm p}}$$

Achieving $S = p$ (*i.e.* $E = 1$) usually regarded as ideal, but difficult to achieve due to various **overheads**.

- Synchronisation, load balancing, communication, extra calculations, . . .
- **Super-linear scaling** $S > p$ possible (but rare) due to memory cache.

Laws for maximum parallel performance

Lectures 4, 19

Strong scaling is when the problem size n is fixed.

- Covered by **Amdahl's law**: $S \leq \frac{1}{f+\frac{1-f}{p}}$.
- $f =$ fraction of code in serial.

Weak scaling allows n to increase with p.

- Related to the **Gustafson-Barsis law**: $S \leq p + f(1-p)$.

The **work-span model** provides another estimate for the maximum S from **task-graphs** *[Lecture 19]*.

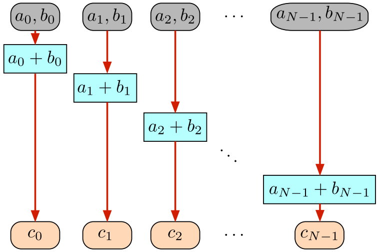
- $S \leq (\text{work})/(\text{span})$, with **work** and **span** determined from the task graph.

Loop parallelism and data dependencies

Lectures 3, 5, 9, 15

Initially looked at parallelising **loops**.

- If there are no **data dependencies**, is **data parallel** or a **map**.
- Often referred to as **embarrassingly parallel**.
- Standard example is **vector addition**.



Synchronisation
Lectures 7, 9, 11, 17

All but the simplest parallel problems require **synchronisation** between the processing units at one or more points.

- Often implemented as **barriers** or **blocking communication**.
- For instance, between levels of the binary tree in reduction.

Can lead to reduced **performance**.

- *e.g.* the extra operations required to achieve synchronisation.

Can also lead to **deadlock** *[Lectures 7, 9]*.

- When one or more processing units wait for a synchronisation even that never occurs.

Load balancing and task parallelism

Lectures 13 and 19

Idle time is an example of poor **load balancing** *[Lecture 13]*.

- Want processing units to never be idle — good load balancing.
- **Serialisation** is the extreme case when only one processing unit is active at a time.

Can improve load balancing by using a **work pool**, where independent **tasks** are sent to processing units as soon as they become idle.

- This is an example of **dynamic** scheduling; can also be **static**.

Parallelising by **tasks** rather than loops is known as **task parallelism** *[Lecture 19]*.

- Increasingly supported by modern parallel frameworks.

In general, there will be **dependencies** between the tasks.

- Can represent as a **task graph**, with **nodes** representing tasks and **directed edges** denoted the dependencies.
- For tasks that take the same time, can define the **work** as the total number of tasks, and the **span** as the length of the critical path.

Data reorganisation

Lecture 10

Parallel data reorganisation can be indexed by **read** locations ('gather'), or by **write** locations ('scatter').

In shared memory systems need to worry about **collisions**, an example of a **data race** *(see later)*.

In distrubuted memory systems, can exploit **collective communication methods** that are usually provided.

- One-to-many, many-to-one (also many-to-many).
- *e.g.* broadcasting, scattering and gathering.

Parallel hardware

Lectures 2, 8, 14, 16

Modern HPC clusters are increasingly using all three architectures:

- **Nodes** with one or more **multi-core CPUs** plus one or more **GPGPU**s.
- Interconnected modes makes a **distributed system**.

Most multi-core CPUs usually have **memory cache**, with issues of **cache coherency** and **false sharing** *[Lecture 2]*.

Network connectivity affects communication times, with **hypercube** often used *[Lecture 8]*.

GPU's most suited for **data parallel problems** and have multiple types of **memory** *[Lectures 14, 16]*.

Data races / race conditions

Lectures 5, 6, 18

A **data race** potentially arises when two or more processing units read the same memory location, and **at least one writes to it**.

- Shared memory CPU, or global/local memory in a GPU.
- Can lead to **non-deterministic** behaviour.

Can control using **critical regions**.

- Exclusive access by a single processing unit.
- Simple critical regions can be implemented more efficiently (*i.e.* by compiler and hardware) as **atomics**.

Lower level control

Lectures 7, 18

At a lower level, critical regions are controlled by **locks** or **mutexes**.

- **Multiple** locks can improve access to data structures.
- Improper use of multiple locks can result in **deadlock**.

At an even lower level, locks can be implemented using **atomic exchange** and **atomic compare-and-exchange** *[Lecture 18]*.

- **Lock-free data structures** are desirable whenever possible.

Explicit communication

Lectures 9, 10, 12, 15, 19

If memory is distributed (in some sense), may need to use **explicit communication**.

- Can be **point-to-point** between processes.
- Also one-to-many *etc.*, *i.e.* **collective communication**.
- Between CPU and GPU, *i.e.* **host** and **device**.

Communication can be:

- **Blocking**: Returns once all resources safe to re-use.
- **Synchronous**: Does not complete until sender and receiver start their communication operations.

Latency hiding

Lectures 12, 19

Can improve performance by **overlapping** communication with computation:

- Reduces the **communication overhead**.
- Known as **latency hiding**.

Often used with **domain partitioning** in HPC applications *[Lecture 12]*.

Can also overlap host-device communication with computation on a GPU *[Lecture 19]*.

- Can also perform calculations on host and device simultaneously.

The end

This is the end of the material for **XJCO3221 Parallel Computation**.