Parallel Design Patterns-L01

An introduction to the course and parallel design patterns

Course Organiser: Dr Nick Brown
nick.brown@ed.ac.uk
Bayes room 2.13

PDP: Parallel Design Patterns

- Lectures:
  - Tuesdays 15:10 – 16:00 in LG.06: 40 George Square
  - Wednesdays 12:10 – 13:00 in LG.06: 40 George Square
- Practicals
  - Mondays 11:10 – 12:00 *(starting next week, week two)*
    - *In Appleton tower 6.06*
- Assessment: 100% by coursework, split in two
  - **Part One** (Report; 30%)
    - Handed out towards end of week 3
    - Deadline: 14th February, 2025 at 4pm
  - **Part Two** (Code & Report; 70%)
    - Handed out around week 7
    - Deadline: 28th March, 2025 at 4pm

Caoimhin Laoide-Kemp will also be teaching lectures and practicals

About the course

- This is a more abstract course than some others, but will lead to a practical piece of coursework and will revisit ideas you have already met in practice in Semester 1.
- So far most courses have taken a bottom-up approach
  - This course will now look at things from the top, down
- Two important ideas
  - Reusable patterns
  - Parallel frameworks
- Typically look at 1 or 2 patterns per lecture
  - Abstractly describe and relate to languages, hardware and applications
  - Practicals look at implementing patterns to solve a problem
  - Coursework: Identifying relevant patterns, and writing code to implement and apply a parallel framework

Course Text

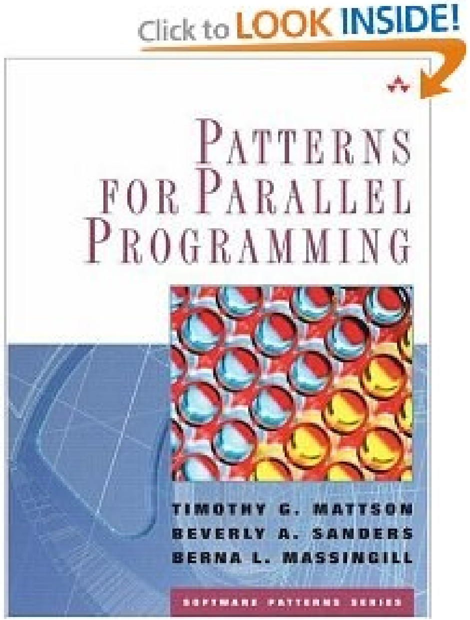
Patterns for Parallel Programming

Mattson, Sanders, Massingill

Addison Wesley (2005)

ISBN-10: 0321228111

ISBN-13: 978-0321228116



- The closest text to this course
- Covers the same patterns, generally uses the same terms

- Examinable material is covered in lectures, but if you're a book person, this is the one I'd recommend.

Planned Course Timetable 2025

Semester Week No	Week Beginning	Lectures	Topics
W1	13th January, 2025	L1, L2	Introduction, Parallel algorithm analysis, Finding Concurrency
W2	20th January, 2025	L3, P1, L4	Geometric Decomposition, Recursive Data, Task Parallelism, Divide & Conquer
W3	27th January, 2025	L5, P2, L6	Pipelines, Event-Based Coordination, Actors **Coursework 1 Handed Out** (29th January)
W4	3rd February, 2025	L7, P3, L8	Parallel framework design
W5	10th February, 2025	L9, P4, L10	Implementation Strategies, SPMD, Master/Worker, Loop Parallelism, Shared Data **Coursework 1 Due: Friday 14th February 4pm**
ILW	17th February, 2025	No lectures	**Flexible Learning Week**
W6	24th February, 2025	L11, P5, L12	**Coursework 2 Handed Out** (26th February) Distributed Array, Fork/Join, Shared Queue
W7	3rd March, 2025	L13, P6, L14	Active messaging, In-situ data analytics, Vectorisation, Memory bound codes, spatial compute
W8	10th March, 2025	L15, P7, L16	High performance data analytics, Domain Specific Languages (DSLs), fosters Methodology
W9	17th March, 2025	No lectures	*Practicals as an open surgery for coursework*
W10	24th March, 2025	No lectures	*Practicals as an open surgery for coursework*
W11	31st March, 2025	No lectures	**Coursework 2 Due: Friday 28th March, 4pm**

Some terminology

Term	Description
Task	Sequence of instructions that operate together as a group which corresponds to some logical part of the code.
Unit of Execution (UE)	To be executed a task needs to be mapped to a unit of execution – such as a process or a thread. This is a generic term for a collection of possibly concurrent executing entities
Processing Element (PE)	Some hardware element to execute the UEs. A single SMP machine might be one PE, whereas in a distributed machine (such as ARCHER) a PE would be a node.

Why Patterns?

- Motivation: The same concepts and problem types appear in many different places
- We don't want to waste time re-inventing the wheel
- We'd like a common language to talk about "ways of doing parallelism" between different, non HPC expert, stake holders
- Languages, machines and applications change frequently but ideas and concepts recur

- We often start with some problem/code in an area we know little about. Can help us know where to start.

What is a Design Pattern?

- The idea of a design pattern was first formally described by the architect Christopher Alexander in the field of architecture in his 1977 book*
- "Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" – Christopher Alexander

* Alexander, Christopher (1977). A Pattern Language: Towns, Buildings, Construction. Oxford University Press. ISBN 0195019199

"Patterns" in common use

- Sharing *n* things of type *t* amongst *m* people
  - Doesn't matter what *n*, *t*, and *m* are

- Sorting algorithms
  - As long as you have an ordering amongst any two items, you can use the same algorithm to sort strings, numbers, whatever.

What is a Design Pattern?

- A description of a problem and a strategy for its solution expressed in an abstract way independent of language, hardware, and application

- “A design pattern describes a good solution to a recurring problem in a particular context” – *Mattson et al*

- “a design pattern is a general reusable solution to a commonly occurring problem within a given context” – *Wikipedia*

Gang of Four Design Patterns

- First example of Design Patterns used in software engineering: Beck & Cunningham (1987)
- Design Patterns in the field of software engineering popularised by the “gang of four”:
  - Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
- *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1995. ISBN 0-201-63361-2)

Some “GoF patterns”

- The GoF didn’t come up with the ideas described in these patterns, but they were the first to categorize them and describe them as patterns
- Creational Patterns, e.g.,
  - Factory method, Singleton
- Structural Patterns, e.g.,
  - Decorator, Façade, Proxy
- Behavioural Patters, e.g.,
  - Iterator, Visitor, Observer (often used as part of Model-View-Controller)

We are not going to study the Gang of Four’s Design Patterns!

Parallel **Design** Patterns

- These are **design** patterns because they are used during the design of a piece of software or a system
- They should help you to think about a solution to a problem before any implementation in code
- They are **not a process**
- There is rarely *one right answer* and a good design often boils down to a number of *tradeoffs*

Patterns in a Design Process

An example from *Patterns for Parallel Programming*[1]

Finding Concurrency

- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

Algorithm Structure

- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, …

Supporting Structures

- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

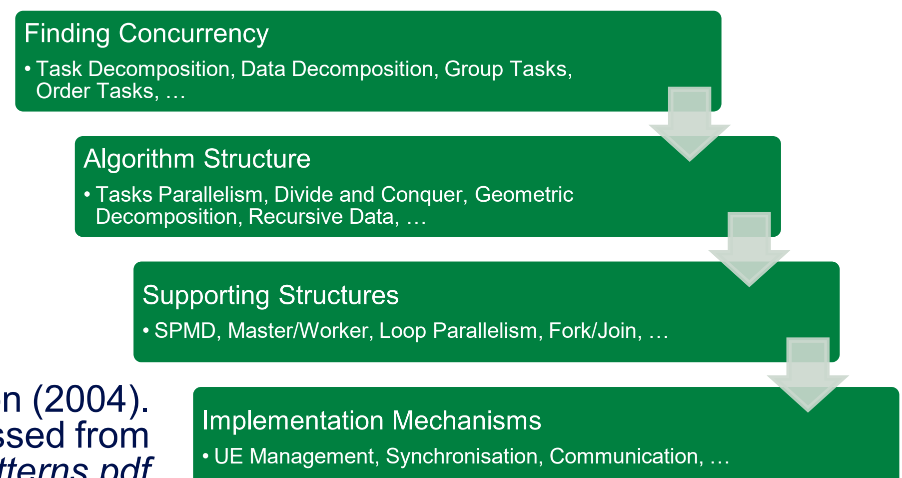
Implementation Mechanisms

- UE Management, Synchronisation, Communication, …

[1] Patterns for Parallel Programming; Mattson, Sanders, Massingill; Addison Wesley (2005)

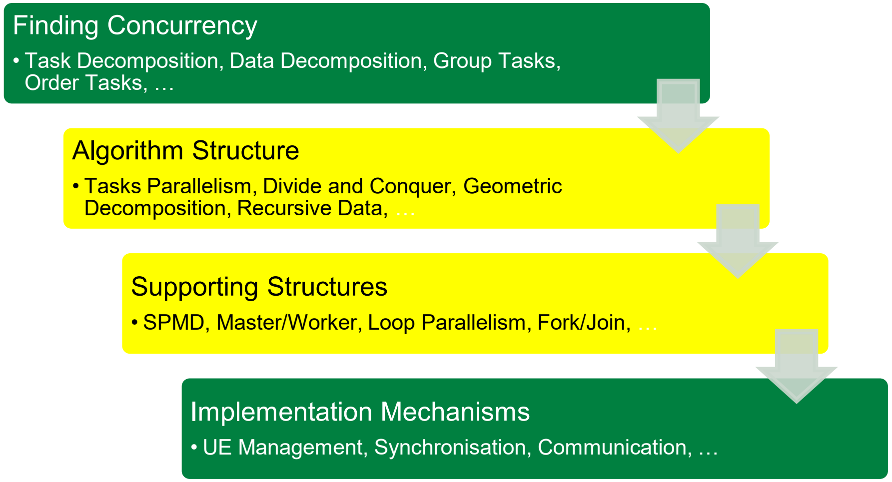
Parallel Algorithm Strategy & Implementation Strategy

- Patterns can be grouped into “Strategies” or “Design Spaces”
- The grouping is sometimes referred to as a Pattern Language
  - “Pattern Language - a collection of design patterns, guiding users through the decision process in building a system” – Kim (2004) *
- Parallel Algorithm Strategy
  - *aka* “Algorithm Structure Design Space”
- Implementation Strategy
  - *aka* “Supporting Structure Design Space”
  - distinct from “Implementation Mechanisms Design Space”



* Eun-Gyu Kim, *Parallel Patterns,* online presentation (2004). Accessed from *http://web.engr.illinois.edu/~snir/patterns/patterns.pdf*

The focus of this course



- On algorithm structure and supporting structures
- Discussion about *finding concurrency* next lecture
- Implementation mechanisms dealt with elsewhere
  - Will use implementation technologies (MPI and OpenMP) in the practicals
  - Details of how hardware, operating system and middleware can implement the parallel algorithm at run-time
  - Covered in other modules
    - Thread and Process management
    - Low-level synchronisation mechanisms
    - Communications constructs and protocols

Patterns in a Design Process

An example from *Patterns for Parallel Programming*[1]

Finding Concurrency

- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

Algorithm Structure

- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, …

Supporting Structures

- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

Implementation Mechanisms

- UE Management, Synchronisation, Communication, …

[1] Patterns for Parallel Programming; Mattson, Sanders, Massingill; Addison Wesley (2005)

Parallel Algorithm Strategy

The *Algorithm Structure* Design Space

- Input information:
  - Grouped set of tasks
  - A knowledge of dependencies amongst tasks and any implied temporal constraints
- These patterns can be thought of as algorithm templates

- Task Parallelism
- Divide and conquer
- Geometric Decomposition (Domain decomposition)
- Recursive Data
- Pipelines
- Event-Based Coordination
- Actor pattern

Patterns in a Design Process

An example from *Patterns for Parallel Programming*[1]

Finding Concurrency

- Task Decomposition, Data Decomposition, Group Tasks, Order Tasks, …

Algorithm Structure

- Tasks Parallelism, Divide and Conquer, Geometric Decomposition, Recursive Data, …

Supporting Structures

- SPMD, Master/Worker, Loop Parallelism, Fork/Join, …

Implementation Mechanisms

- UE Management, Synchronisation, Communication, …

[1] Patterns for Parallel Programming; Mattson, Sanders, Massingill; Addison Wesley (2005)

Implementation Strategy

The *Supporting Structures* Design Space

- Usually considered once the Algorithm Structure has been decided
- Can be divided into *Program Structures* and *Data Structures*

- Master / Worker
- Loop Parallelism
- Fork / Join
- Shared Queue
- SPMD
- Shared Data
- Distributed Array
- Active messaging
- Vectorisation

Components of a Design Pattern

- Pattern Name
  - Should be standard to ease communication
- Problem
- Description of the context
  - A more detailed discussion of where the pattern might be applicable
- The forces
  - Forces that act on the design using a specific pattern
  - Goals and constraints
  - Things that influence whether this is the right pattern to use
- The solution

Criticism of Design Patterns

- We teach you about Parallel Design Patterns because we think they are a useful abstraction, however there are some who criticise design patterns:
- There's nothing new or special about design patterns; they just boil down to reusing an idea and making life easier.

- Writing code to force it to look like a standard pattern can unnecessarily increase complexity
- The "parallel pattern language" is not standardised enough to be useful
  - There are different names for the patterns and strategies

Alternative Approaches

- A design pattern is only necessary because of some missing features of a given programming language
- Parallel Programming "Systems"
  - e.g. DPnDP (Siu, et al)
- Parallel Skeletons

- There are also quite a few alternative approaches
  - e.g. Foster's Methodology
    - *Partitioning: Discover as much parallelism as possible*
    - *Communication: Determine communication (local & global)*
    - *Agglomeration: Group individual tasks into larger tasks*
    - *Mapping: Map tasks to UEs*

Parallel Frameworks

- The other important strand to this course
- A good software engineer will use libraries and frameworks wherever appropriate
- Someone still has to write these libraries and frameworks
- This course aims to introduce some basic ideas about how these might be put together

- Even in a specific code, developing with flexibility and reuse in mind can help significantly when adding additional features.

A look to the future

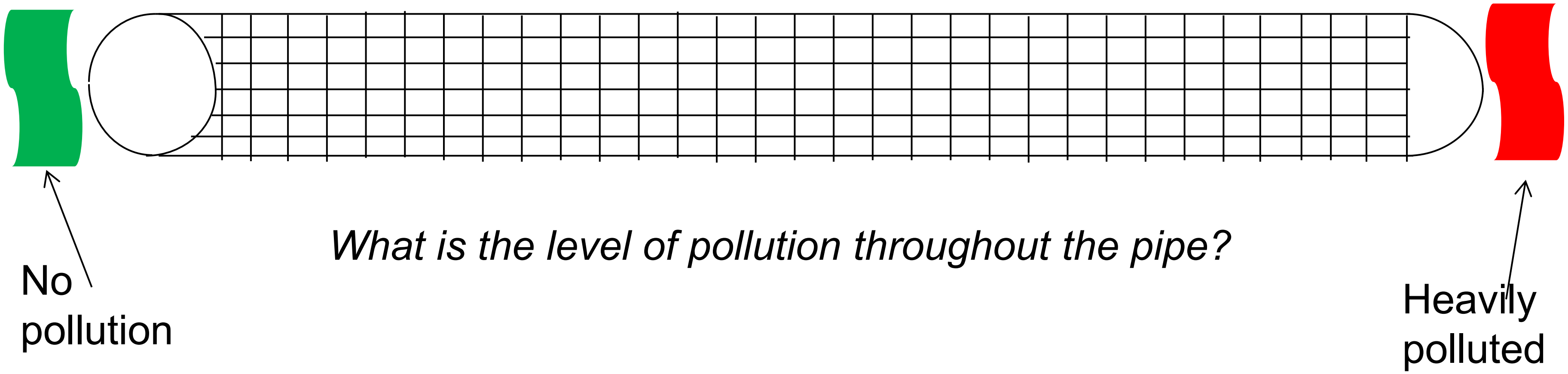
- We will also look at how HPC might change in the future and consider the impact this will have on how we write parallel codes
  - In terms of changes to hardware forcing us to write applications differently
  - New techniques which are not yet in the mainstream, often build on or complement the well known patterns we will study
  - Some patterns are more mature and/or readily used than others

Summary

- Parallel Design Pattern
  - A description of a problem (with context) and strategies for its solution, independent of language, architecture and application
  - Not a process
  - Useful, but not a silver bullet

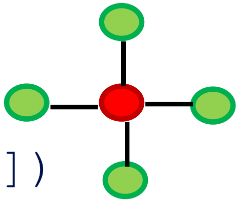
- Parallel Framework
  - Ideas from these recurring parallel patterns can be expressed in code to produce parallel frameworks that other programmers can use

Next weeks practical



What is the level of pollution throughout the pipe?

- Jacobi iteration solving Laplace's equation for diffusion in 2 dimensions $\nabla^2 u = 0$



```
for all grid points
    u_new(i) = 1/4*(u[j,i-1]+u[j,i+1]+u[j-1,i]+u[j+1,i])
```

- Works in iterations, solving to a specific residual (accuracy)
  - As we parallelise this, the overall number of iterations and residual should be the same as the serial code which is a nice check

Overview of serial code

```
double * u_k = malloc(sizeof(double) * (ny+2) * (ny+2)), * u_kp1 = malloc(sizeof(double) * (nx+2) * (ny+2)), *temp;
initialise(u_k, u_kp1);
double rnorm=0.0, bnorm=0.0, norm;
int i, j, k;
for (i=1;i<=nx;i++) {
   for (j=1;j<=ny;j++) {
      bnorm=bnorm+……….
   }
}
bnorm=sqrt(bnorm);
for (k=0;k<MAX_ITERATIONS;k++) {
   for (i=1;i<=nx;i++) {
      for (j=1;j<=ny;j++) {
         rnorm=rnorm+……….
      }
   }
   norm=sqrt(rnorm)/bnorm;
   if (norm < CONVERGENCE_ACCURACY) break;
   for (i=1;i<=nx;i++) {
      for (j=1;j<=ny;j++) {
         u_kp1[i]=0.25 * ……..
      }
   }
   temp=u_kp1; u_kp1=u_k; u_k=temp;
   rnorm=0.0;
}
```

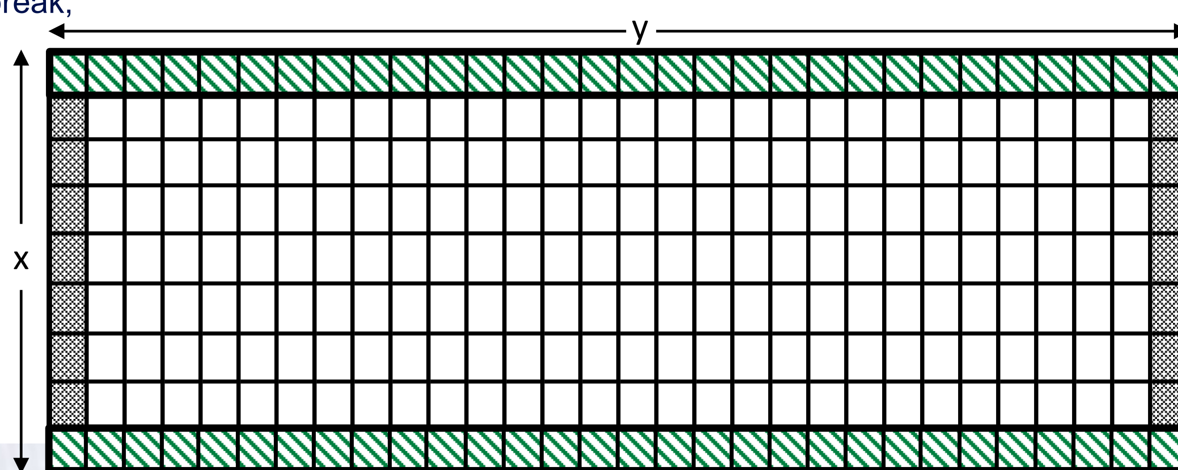
Sets the pollution values at each end of the pipe and the rest to be zero (initial guess)

Compute the initial absolute residual

Compute the absolute residual of the current solution, then divide this by bnorm to get the relative residual (how far we have progressed)

Termination criteria (level of accuracy met)

Jacobi iteration to progress the solution



Your task

- Parallelise it!
  - In 1D using geometric decomposition
  - Start with the simplest approach to halo swapping and then add in extra complexity to optimise this