Parallel design patterns ARCHER course

General Overview

Reusing this material

About the course

- This is a more abstract course than many others, but we have plenty of practicals to get hands-on with the concepts
- Many courses take a bottom-up approach
  - This course will now look at things from the top, down
- Two important ideas
  - Reusable patterns
  - All the options we have for applying these

- Typically look at 1 or 2 patterns per lecture
  - Abstractly describe and relate to languages, hardware and applications
  - Practicals look at implementing patterns

Basis of this course

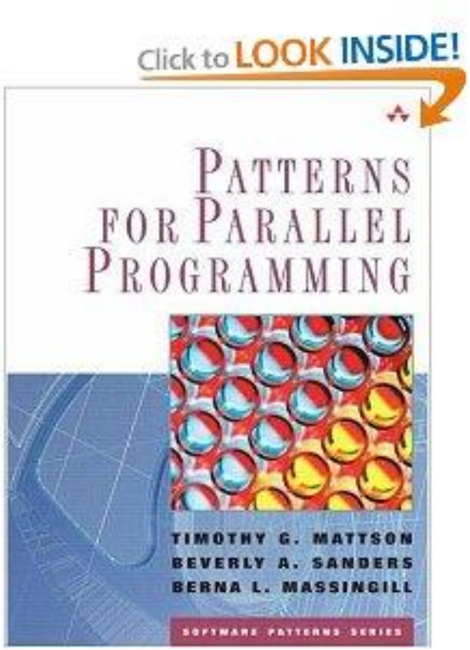
Patterns for Parallel Programming

Mattson, Sanders, Massingill

Addison Wesley (2005)

ISBN-10: 0321228111

ISBN-13: 978-0321228116



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

epcc

THE UNIVERSITY OF EDINBURGH

Timetable

Day 1

09:30 Intro and Overview
10.00 Comparing parallel algorithms
10:40 Practical
11:00 Break
11:30 Geometric decomposition
12:10 Practical
13:00 Lunch
14:00 Recursive data, task parallelism, divide and conquer
14:45 Practical
15:30 Break
16:00 Pipelines, event based coordination
16:45 Practical
17:30 Finish

Day 2

09:30 Actors
10.10 Practical
11:00 Break
11:30 Implementation strategies, SPMD, master/worker
12:15 Practical
13:00 Lunch
14:00 Loop parallelism, Fork/join
14:40 Practical
15:30 Break
16:00 Active messaging and vectorisation
16:40 Practical
17:30 Finish

Day 3

09:30 Distributed arrays, shared data, shared queue
10:20 Practical
11:00 Break
11:30 Practical
12:30 Summary
13:00 Lunch
14:00 Practical
15:30 Finish

Plus optional individual consultancy session to talk about these concepts in relation to your area/codes

Day 1

09:30 Intro and Overview

10.00 Comparing parallel algorithms

10:40 Practical (parallelizing pollution code via geometric decomposition)

11:00 Break

11:30 Geometric decomposition

12:10 Practical (parallelizing pollution code via geometric decomposition)

13:00 Lunch

14:00 Recursive data, task parallelism, divide and conquer

14:45 Practical (parallelizing pollution code via geometric decomposition)

15:30 Break

16:00 Pipelines, event based coordination

16:45 Practical (pipelining pollution code)

17:30 Finish

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

- Sometimes start with unfamiliar 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

“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

This course is not about the gang of four design patterns!

- *Design patterns for parallel codes rather than serial codes*

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

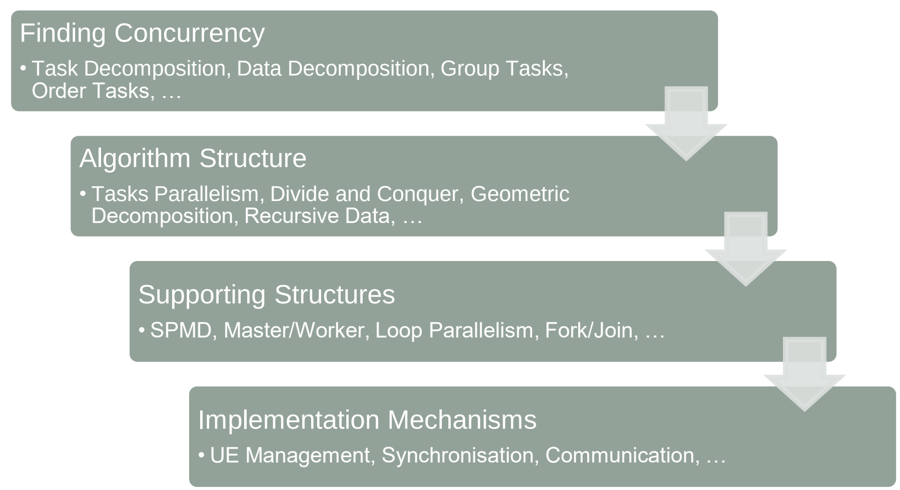
- UE Management, Synchronisation, Communication, …

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

Parallel Algorithm Strategy & Implementation Strategy

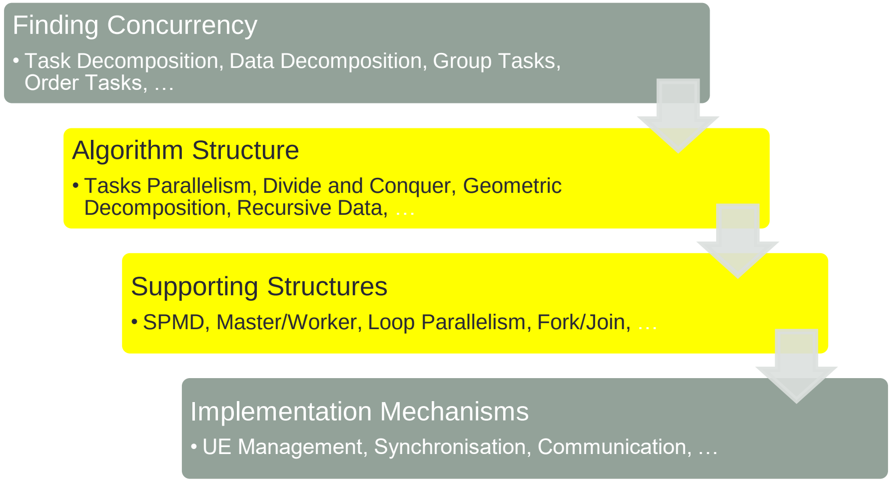
- Patterns can be grouped into "Strategies" or "Design Spaces"
- The grouping is sometimes referred to as a Pattern Language
  - "Pattern Language - a collection of design patterns, guiding users through the decision process in building a system"

- Parallel Algorithm Strategy
  - *aka* "Algorithm Structure Design Space"
- Implementation Strategy
  - *aka* "Supporting Structure Design Space"
  - distinct from "Implementation Mechanisms Design Space"



The focus of this course

On algorithm structure and supporting structures



- 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 ARCHER training courses

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

- Input information:
  - Knowledge of the problem we are parallelising/optimising
    - E.g. dependencies amongst tasks and any implied temporal constraints

- These patterns can be thought of as parallel algorithm templates

The *Algorithm Structure* Design Space

- 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 parallel 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

Criticism of Design Patterns

- We think Parallel Design Patterns 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

The importance of evaluation

- Often there are multiple approaches possible
  - Evaluate the emerging design and ensure that it is appropriate
  - This strategy is an iterative process
- Design quality
  - Simplicity
  - Flexibility, efficiency
- Suitability for target platform
  - How many PEs are available, how is data shared, will the time spent doing useful work be significantly greater than managing the parallelism
  - Sequential equivalence

The earlier you realise an approach isn't going to work, the less wasted effort this implies!