MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Concurrent Programming
Algorithm Design

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

Concurrent Programming

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

Parallel Algorithms

Like general algorithm design, there is no one method for taking a problem and designing a good parallel algorithm to solve it.

Developing parallel algorithms also have additional challenges in that they are more closely tied to hardware.

This also makes evaluating and comparing parallel algorithms more difficult.

Parallel Algorithm Design

One approach sometimes used is to take an existing sequential program and try to find pieces of it that can be parallelised.

While this is a natural approach, it is not always effective as good parallel solutions to a problem may often require very different *algorithms* / *methods* / *approaches* than a sequential program.

There are normally many different ways to parallelise a problem, the best approach often depends on the specifics of the machine you are using.

Parallel Algorithm Design

Designing new parallel solutions requires a certain creativity that cannot be boiled down to a simple recipe.

However, a methodical approach to designing parallel algorithms can help us to consider a range of options early on.

We will consider the ***PCAM*** method described in:

I. Foster. *Designing and Building Parallel Programs* Addison Wesley, 1996.

Parallel Algorithm Design - PCAM

The idea behind this method is to consider the machine-independent aspects of a problem first and leave machine-specific details until the end of the process.

Concurrency within a problem can be identified and considered regardless of what type of machine eventually computes the program.

Once the opportunities for concurrency have been identified, then the decision of whether to exploit them and map them onto a particular machine can be made.

Parallel Algorithm Design - PCAM

The design of an algorithm is broken down into four stages:

- ***Partitioning*** - divide the problem into parallel elements of computation.
- ***Communication*** - communicate/coordinate between elements.
- ***Agglomeration*** - combine elements for performance reasons.
- ***Mapping*** - assign tasks onto processing units.

Partitioning

Partitioning considers how the computation and the data of a program can be broken down into separate computational units that can be computed in parallel.

Practical considerations such as the memory architecture of a machine or how many processors the machine have are ignored at this stage.

The goal is simply to identify opportunities for fine-grained parallelism.

Partitioning

In the partitioning stage every opportunity for parallelism should be considered in as *fine-grained* a way as possible.

Fine-grained parallelism tries to separate a problem into as many small, individual units.

A large number small units provide the most flexibility and may be combined into fewer, coarser-grained units during later stages that consider machine architecture.

Communication

The tasks defined in the *partition* stage are intended to execute concurrently but are not necessarily independent.

If data computed by one task is required by another, then the tasks will need to communicate with each other.

The exact manner of this communication will depend on the partitioning, the language used to implement the program and the machine it is run on.

Communication

The communication between tasks can be conceptualised as individual communication links between tasks that send/receive data.

At this point we do not consider the hardware or actual method of communication between physical processors.

It simply determines what the necessary communication will be once it is eventually mapped onto processing units.

Communication

The communication between tasks can generally be classified according to the following categories:

- ***local/global***
- ***structured/unstructured***
- ***static/dynamic***
- ***synchronous/asynchronous***

Communication - Local/Global

Local/global communication considers the locality of tasks that communicate with each other.

Local - each task communicates with a small number of neighbouring tasks.

Global - each task communicates with many other tasks (similar order of magnitude to total number of tasks).

Communication - Structured/Unstructured

Structured/unstructured communication considers whether there is some regular structure to the tasks that communicate with each other or not.

Structured - tasks and their neighbours form a regular structure such as a grid.

Unstructured - tasks and their neighbours do not have a regular structure and may be an arbitrary graph.

Communication - Static/Dynamic

Static/dynamic communication considers whether the tasks that communicate with each other remain the same during the run-time of the program or evolve/change.

Static - tasks that communicate do not change over time.

Dynamic - tasks that communicate with each other may change during program run-time.

Communication - Synchronous/Asynchronous

Synchronous/asynchronous communication considers whether both tasks involved in the communication (sending/receiving) know of the existence and nature of communication in advance.

Synchronous - tasks communicate in a coordinated fashion, both know the intended communication beforehand.

Asynchronous - necessary communication is not known beforehand and tasks must explicitly request data from another.

Agglomeration

After the ***partitioning*** and ***communication*** stages, the parallel algorithm is still defined at a very abstract level.

At this point there is a collection of computational units/tasks and some communication between them.

In the ***agglomeration*** stage the *partitioning* and *communication* will be restructured to make an algorithm suitable for a particular class of parallel computer.

Agglomeration

In particular, separate computational tasks will be combined together or *agglomerated* to increase the granularity of the parallelism to better suit the intended machine.

Fewer, larger tasks can be beneficial by more closely matching the number of available cores and by reducing the necessary communication between tasks.

When two communicating tasks are combined together into one, the communication between them is eliminated.

Mapping

The final ***mapping*** stage considers how to allocate tasks onto the available processing units.

This problem is straightforward on shared-memory machines that provide automatic scheduling but is an important consideration for distributed machines.

The overall goal of *mapping* is to minimise the execution time.

Mapping

There are two strategies for mapping tasks onto processing units.

- Assign two tasks that can execute concurrently to different processing units to enhance concurrency.
- Assign two tasks that communicate frequently to the same processing unit to enhance locality.

Unfortunately these two strategies often conflict with each other.

Mapping

Many problems (especially data-parallel problems) can be partitioned into a number of equally sized tasks with regular communication between them.

In this case a simple, ***static*** mapping method may be used to assign each task to a separate processing unit.

It is usually reasonable to expect that (almost) identical tasks will take the same time to execute on identical processing units.

Mapping

If the problem has been partitioned into a number of unequally sized tasks with unstructured communication, the best mapping method may not be obvious.

In such cases a ***load-balancing*** method may be used to try to find an efficient way to map tasks onto different processors.

Load-balancing methods distribute tasks to processing units to try to evenly distribute the load between them.

Mapping

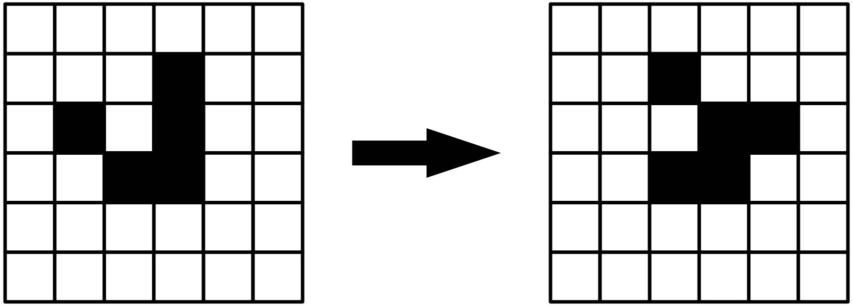
For problems that have dynamically changing computational and communication loads, a ***dynamic load-balancing*** method may be necessary.

Dynamic load-balancing methods periodically review the mapping of tasks and may reassign a task from a heavily loaded processor to another under-utilised processor.

The goal is usually to keep all available processing units supplied with computational work.

Example - Game of Life

Game of Life - compute G generations of an NxN grid.



Example - Game of Life - Partitioning

The first step in designing a parallel algorithm for this application is to identify how to partition the computation.

There are few opportunities for ***task parallelism***, could try to separate the tasks of counting the number of *alive* neighbours and the task of updating a cell.

Another approach could be to treat the computation of each generation as a separate task and use a pipeline.

Example - Game of Life - Partitioning

There are more obvious opportunities for ***data parallelism***, the application can be partitioned by separated by splitting up the cells in the grid.

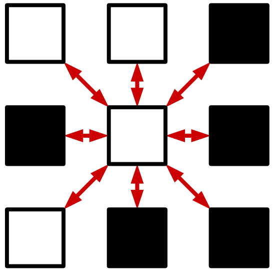
At the ***partition*** stage we are concerned only with maximising the opportunities for parallelism.

The most fine-grained way to decompose the problem is at the individual cell level.

Example - Game of Life - Communication

The next step is to consider the ***communication***.

Computing the next generation of a cell depends on the current generation of the cell and its eight neighbouring cells.



Example - Game of Life - Communication

Game of Life communication is:

- ***local*** - each cell only needs to communicate with eight neighbours.
- ***structured*** - there is a clear geometric structure in the communication.
- ***static*** - the communication does not change during the run-time.
- ***synchronous*** - each cell always communicates with all neighbours after a new generation has been computed.

Example - Game of Life - Agglomeration

The third step is ***agglomeration*** - for this step we need to know what type of machine we are going to be using.

For this step we are going to assume we will be using a standard multi-core PC which is a MIMD, shared-memory machine.

Based on this assumption we need to decide how to combine individual units of work together.

Example - Game of Life - Agglomeration

Creating a separate thread for each individual cell is unlikely to provide good performance as switching between threads is likely to take longer than each thread can execute for.

Since neighbouring cells communicate with each other, it makes sense to combine a neighbourhood of cells together into a computational task.

Although our shared-memory machine doesn't require explicit communication, accessing neighbouring values from memory will allow our program to better utilise the cache.

Example - Game of Life - Agglomeration

The question now is how many sections to divide our spatial grid into.

One option would be to divide the grid so there are the same number of sections as there are cores in the CPU - each core gets one section.

Another would be to create more grid sections than there are cores, each core will have to update more than one section but this approach may help to average out core usage from other programs.

Example - Game of Life - Agglomeration

Communication will take place through the shared memory of the machine so generations can be stored in a double buffer.

Each generation can be read from one buffer and written to the other.

The threads will need to synchronise after each generation has been computed before starting work on the next one.

Example - Game of Life - Mapping

The final step is to determine how to map the threads onto the processing units.

This stage is easy for our development as it will be performed automatically for us by the Operating system.

The mapping will (almost certainly) use ***dynamic load-balancing*** where threads are scheduled onto available cores.

Example - Game of Life - Mapping

Each thread will be scheduled to execute on whichever core the operating system determines is available.

This does mean that each thread may execute on several different cores throughout the application run-time.

It also means that the load will be balanced between different cores but the threads do have to compete with any other programs running at the same time.

Summary

The ***PCAM*** method for developing parallel algorithms consists of the following steps:

- ***Partitioning*** - divide the problem into parallel tasks.
- ***Communication*** - communicate/coordinate between tasks.
- ***Agglomeration*** - combine tasks for performance reasons.
- ***Mapping*** - assign tasks onto processing units.