MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA
UNIVERSITY OF NEW ZEALAND

159.341 Programming Languages, Algorithms & Concurrency

Concurrent Programming
Parallel Programming Paradigms

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

Concurrent Programming

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

Parallel Paradigms

There are several different general parallel programming paradigms that represent high-level approaches to writing parallel programs.

The same problem may be solved with different paradigms or by some hybrid of multiple paradigms.

Identifying the paradigm you use can help in thinking through the design and can point you towards the appropriate languages/libraries for writing your program.

Parallel Paradigms

The paradigms we will consider are:

- ***Phase-Parallel***
- ***Task-Farming***
- ***Work-Pool***
- ***Data Pipelining***
- ***Divide and Conquer***

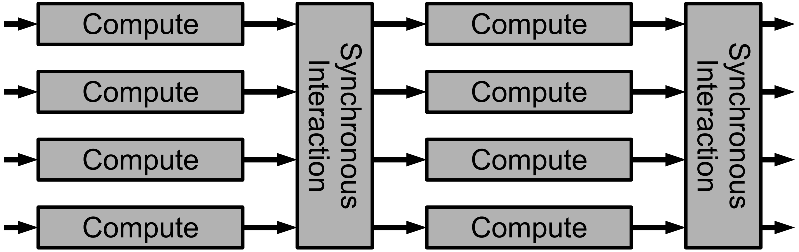
Phase Parallel

Phase Parallel is a commonly used paradigm where the program is broken down into a series of steps that each consist of two phases:

- ***Computation Phase*** - threads perform *independent* computation.
- ***Interaction Phase*** - threads interact with each other (synchronise/communicate).

Phase Parallel

A pair of these two phases can be view as a step in the paradigm.



Phase Parallel

Phase Parallel is popular for iterative programs where a number of iterations must be computed which require the previous step to be computed before the next begins.

An iteration of the problem is computed in the *computation phase* and then the threads are synchronised before the next iteration begins.

Depending on the architecture this may require communication of the results of the iteration as well.

Phase Parallel

The disadvantages of the ***phase parallel*** paradigm is that the synchronised interactions are not overlapped with the computation.

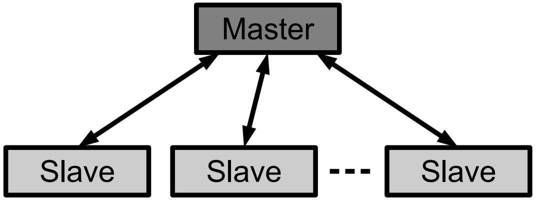
For some architectures this can have a significant effect on performance.

Load-balancing can also be a problem as each phase ends only after the slowest thread has completed its work.

Task-Farming (Master/Slave)

The Task-Farming paradigm (also known as master/slave) consists of a master with a number of slaves.

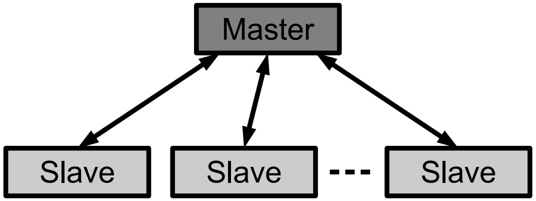
The Master executes the serial part of the program and farms out parallel portions of the program to slave threads.



Task-Farming (Master/Slave)

The slave processes/threads receive parallel tasks, compute them and then return a result to the master.

In this paradigm, communication often only takes place between the master and each slave.



Task-Farming (Master/Slave)

One advantage task-farming is the flexibility of how the paradigm can be mapped onto different machines:

- The master/slaves are processes running on different nodes of a distributed machine communicating through a network.
- The master/slaves are threads on a SMP machine communicating through shared memory.
- and more...

Task-Farming - Load-Balancing

The Task-Farming paradigm can also easily make use of ***static*** or ***dynamic*** load-balancing.

Static load-balancing - the master thread divides the tasks evenly at the beginning of the program.

The advantage of static load-balancing is that it allows the master to allocate work to itself as it may otherwise sit idle while the slaves perform the computation.

Task-Farming (Master/Slave)

Dynamic load-balancing - the master thread farms out tasks during run-time.

Dynamic load-balancing is better suited to problems that have tasks with inconsistent run-times or an unknown number of tasks.

For such problems, dynamic load-balancing allows the computing resources to be better utilised and can deal with issues such as node failures but may suffer from overheads from communication.

Task-Farming (Master/Slave)

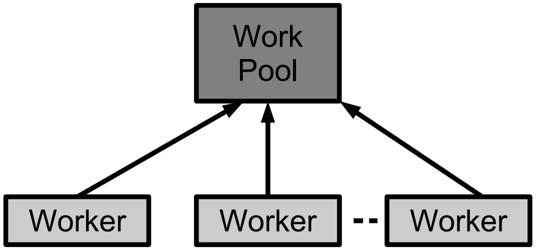
The disadvantage of the Task-Farming paradigm is that the master thread can easily become a bottleneck in the system.

If the master is unable to provide enough work to the slave processes/threads then the whole system can suffer from underutilisation.

Work-Pool

The ***Work-Pool*** paradigm is similar to the task-farming paradigm but without a central master.

The program is defined as a set of tasks that must be computed by the workers.



Work-Pool

A number of worker processes/threads are created that retrieve a task from the work-pool whenever they are free and compute the task.

When a worker has completed a task it may add zero to many new tasks into the work-pool.

When the work-pool is empty and there are no tasks currently being computed, then the program has completed.

Work-Pool

Work-pools support dynamic ***load-balancing*** as any time a worker is free, it will try to fetch a new task to work on.

It does rely on an efficient implementation of a work-pool that can be accessed by many different workers at once.

Accessing the work-pool may become a bottleneck if there are a large number of workers.

Data-Pipelining

Data-pipelining uses a functional decomposition approach. Similar to pipelining in instruction-level parallelism, the overall task being processed is broken down into different functional units that can be processed in parallel.

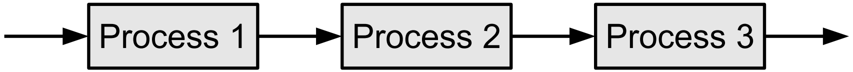
The processes are formed into a pipeline where each process will compute its particular task and then pass the result onto the next process.

The communication in this paradigm is generally very simple (one-way) and can be asynchronous.

Data-Pipelining

Dividing the functional tasks into even workloads is vital for constructing an efficient data pipeline. One task with an unbalanced (higher) workload may act as a bottleneck that can slow down the entire pipeline.

This paradigm is popular for data and image processing applications.



Divide and Conquer

The ***Divide and Conquer*** approach takes a problem and divides it into two (or more) subproblems. These subproblems can be solved independently and their results combined to give the final result.

The subproblems can (normally) be divided further into subproblems and so on. Task farming can be viewed as a form of divide and conquer where the problem is broken down at one level into many subproblems.

Divide and Conquer generally has the advantage that the combination of results can be (partially) performed in parallel.

Divide and Conquer

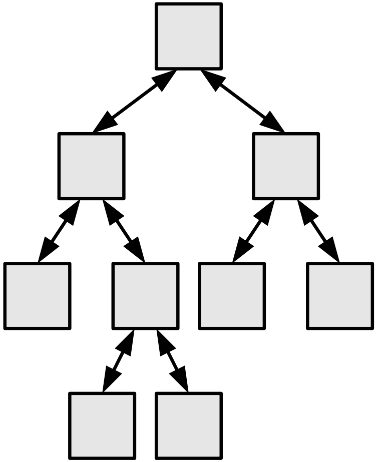
There are three main operations in a divide and conquer program - split/fork, join and compute. Split or fork is the division of a process that solves a problem into two (or more) processes that each solve a subproblem.

Join is the operation where the results from multiple processes are combined back together.

Finally compute takes place when the problem has been broken down to a sufficient point that the process simply needs to solve its subproblem.

Divide and Conquer

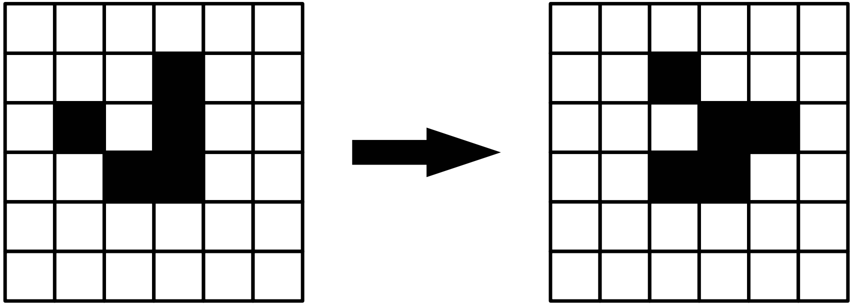
Divide and conquer is generally flexible as the problem can be split into increasingly smaller subproblems until the available resources are occupied.



Example - Game of Life

Example:

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



Example - Game of Life

In our previous lesson we decided to use a ***data-parallel*** approach to divide up the work of computing a Game of Life simulation.

The grid is broken down into a number of sections which are to be updated independently.

The process of updating the cells must be repeated for G generations ensuring that each generation is completed before the next begins.

Which paradigm should we use to implement this algorithm?

Example - Game of Life

Task-Farming - treat updating a section of the grid for each generation as a *task*.

The master thread can farm out sections to slave threads until a generation is complete at which point it starts farming out tasks for the next generation.

Similarly we could use ***Work-Pool*** and allow workers to fetch tasks from the work-pool - would need to add tasks for the next generation once a generation is complete.

Example - Game of Life

Data Pipelining - would be difficult to build a pipeline for this problem. Each generation should be finished before the next begins.

Divide and Conquer - would also be difficult to implement the algorithm with divide and conquer.

For each iteration we could allocate all sections to a single thread and allow it to create additional threads and divide the work between them. Would be repeated each iteration though.

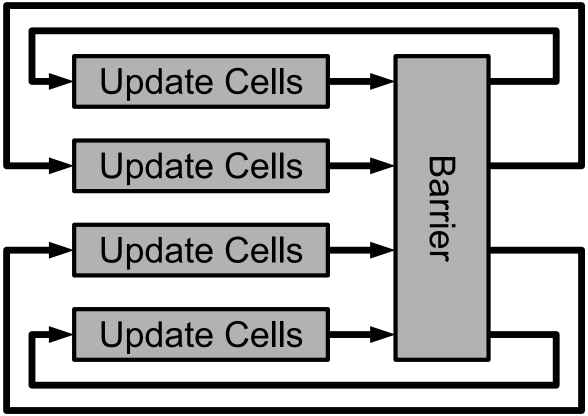
Example - Game of Life

Phase Parallel seems like the obvious choice.

Threads can be created for each section and in the compute phase they update their section.

The synchronous interaction phase synchronises the threads to ensure the iteration is finished before the next begins.

Example - Game of Life



Example - Game of Life

Load-balancing for this type of problem is generally not a large problem.

If each section of the grid is the same size then the time to update it will not vary very much.

By creating more threads than there are cores we can also make use of the operating system's thread scheduler to average out interruptions from other programs.

Summary

Summary of parallel paradigms:

- Phase-Parallel
- Task-Farming
- Work-Pool
- Data Pipelining
- Divide and Conquer