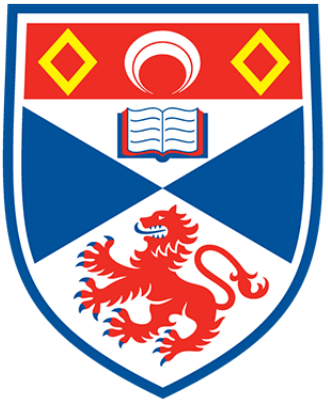


University of
St Andrews

CS4204 Concurreny and Multicore Architectures

Revision Notes

May 12, 2018

Lecturer:
Kevin Hammond
Susmit Sarkar

Submitted By:
140011146

Contents

1 Introduction

Parallel programming is about programming on multicore/manycore machines. There are generally two forms of parallel programming, low level parallelism with pThreads, OpenMP, CUDA etc and higher level parallelism with function languages such as Haskell.

In the past, the performance of consumer processors was improved from increased clock speed as transistors grew smaller. However as we are reaching the limits of improvements to silicon and hardware technology, consumer processors have started to take the direction of focusing on parallel architectures with multicore systems. Purely functional languages are beneficial for programming parallel systems thanks to the absence of side effects, making it easy to evaluate sub-expressions in parallel. The only issue is the trivial work required to evaluate sub-expressions is not out balanced by the overhead needed to spawn a parallel thread to deal with it.

1.1 What is Multicore?

Multicore architectures refer to having more than one processor on a single computer chip, where a processor is defined as a logical processing unit rather than a physical one. For example Intel's hyperthreading allows a single processor to act as two virtual processors. Multicore architectures also share many resources as a result of being on the same chip:

- Shared memory
- Shared address space
- Shared data bus
- *Independent* instruction streams
- Shared cache

The increased amount of shared resources, such as the shared bus and shared memory and cache leads to more contention as all processors compete for resources. Multicore architectures are important in today's technological advances, not just in consumer desktop machines and mobile devices. In particular, grid/cloud computing systems, embedded systems, IoT and SIMD/MIMD architectures and GPUs all take advantage of parallel computation.

1.1.1 Manycore and megacore

In the future, it may be likely that rather than the current multicore chips, the number of processors start scaling to many/megacores with hundred or thousands of processors on a single chip.

These *manycore* machines are likely to have more, individually less-powerful processors and may communicate through some kind of communication network rather than a shared bus.

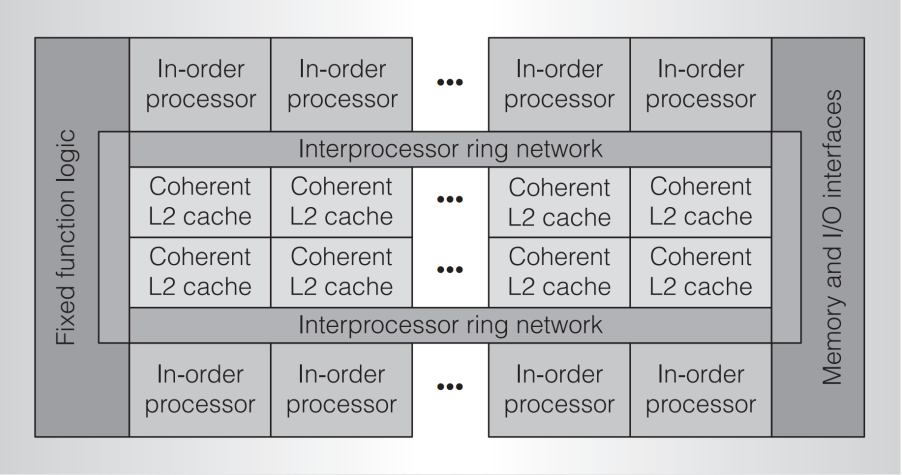


Figure 1: Schematic of the Larrabee manycore architecture.

In these many/megacore architectures, it is likely to not simply be a scaled version of today's multicore chips with more cores. As there can be many processors, there could be hundreds of dedicated but lightweight units with *few* heavyweight general purpose processors. It is also likely there would be specialised units for specific functions, such as RSA encryption or network protocols. This leads to a highly **heterogeneous processor** which is combination of many specialised units. Furthermore, this will likely *not* have shared memory, having a NUMA (Non-Uniform Memory Access) design instead.

1.2 Concurrency vs Parallelism

Concurrency is the concept that more than one thread may be executing simultaneously. Depending on the implementation, concurrent threads may execute in parallel or sequentially. Concurrency is generally used to identify logically-independent units of computation (no dependencies), for example in a user interface, which often yields relatively small-scale parallelism.

Parallelism on the other hand is the idea of executing more than one thread simultaneously on separate hardware devices. This is usually done for performance reasons, as opposed to concurrency, which is introduced for structural reasons so that independent threads can be assigned to different program elements.

1.3 Challenges of parallel programming

As the future of multicore chips moves in a more heterogeneous direction, there is a need for programs to be developed in an integrated way. It will be impossible to program each core differently and take static decisions about placement. Moreover, issues of time, energy, security etc. have to be taken into account in the design. The challenge is to think and program in a parallel way that is not the same as writing concurrent code.

> *Ultimately, developers should start thinking about* tens, hundreds and thousands *of cores* now *in their algorithmic development and deployment pipeline*
>
> **Anwar Ghuloum, Principal Engineer, Intel Microprocessor Technology Lab**

A large issue is that it is non-trivial to transform sequential code into parallel programs. In fact, many applications will actually *run slower*, especially for larger systems. Up to 2-8 or even 16 cores, it is possible to simply write modified sequential code and use multiple programs to keep processors busy, but a much larger and complicated change is required for larger systems.

The typical approach of writing concurrent code is **not** the same as parallelism. Concurrency is all about breaking down programs into independent units of computation while parallelism is about making things happen at the same time. In many cases of concurrent programming, the programs are developed at a low level of abstraction without first understanding the parallelism. The issues this brings is concurrent programs which are designed with specific architectures in mind rather than with general parallel abstractions and structure. Furthermore, concurrency only gives an *illusion* of independent threads of execution with a few **huge** threads, while parallelism is about the *reality* of threads executing at the same time with thousands or millions of **tiny** threads.

Concurrency must also deal with maintaining dependencies, as units of execution must be kept in order with dependencies to remain correct. The point of parallelism is to break dependencies.

1.3.1 Divide and conquer

The divide and conquer pattern is a simple example of parallelism where the problem only solves the program at the trivial case, otherwise it is continually divided into two or more parts, with each solved independently and the results combined.

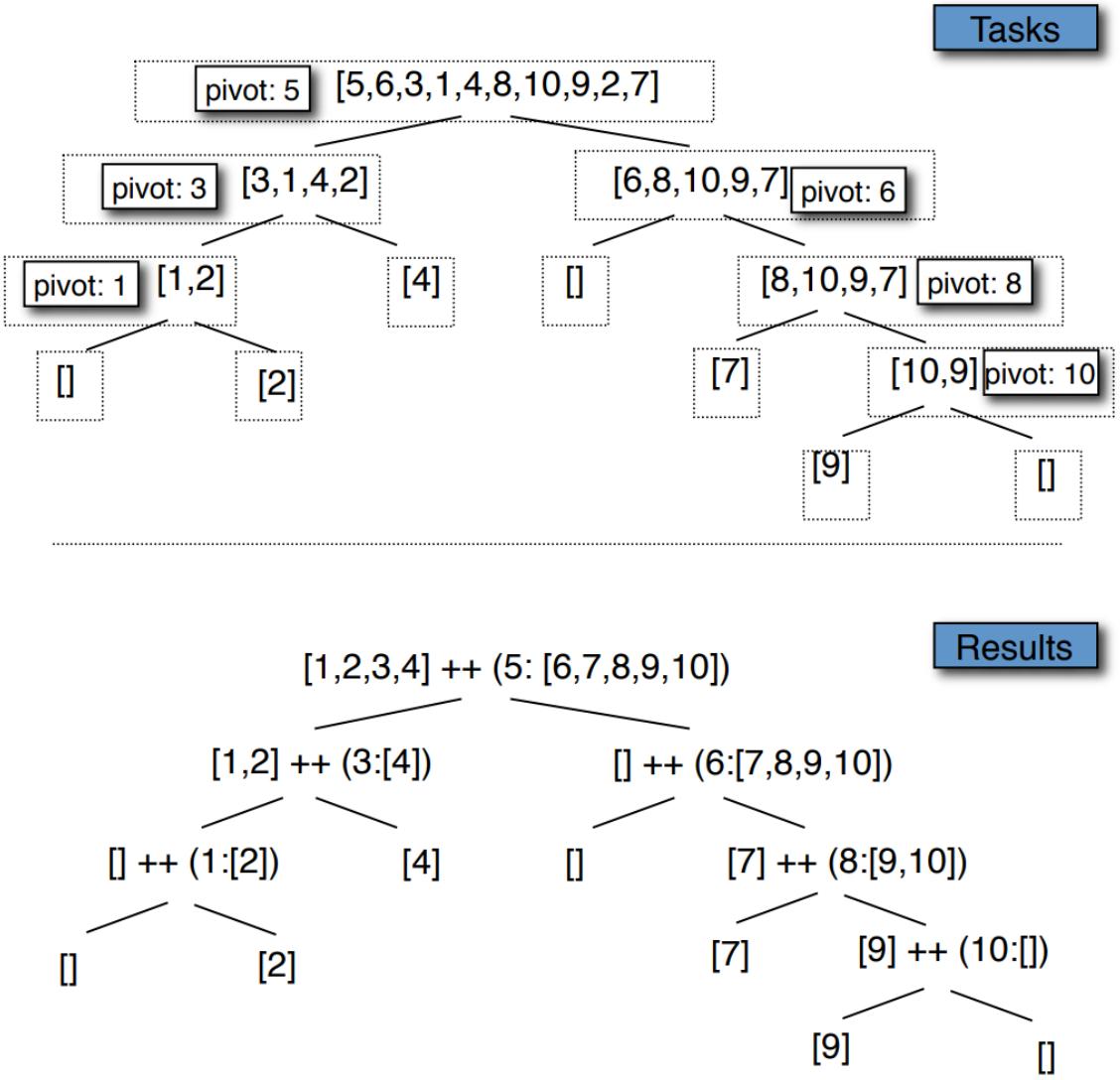


Figure 2: Quicksort using divide and conquer to the trivial case of one element.

To implement this divide and conquer algorithm with pThreads, threads have to be created and joined for each left and right of the split. This leads to too much complexity to manage and makes it difficult to scale unless the program is simple. There are issues with deadlocks, race conditions, non-determinism and communication that have to be dealt with by the systems programmer, which ideally should be abstracted away from an application programmer. Fundamentally, this means programmers must learn to think in parallel, which requires new high level constructs.

2 Parallelism basics

2.1 Definitions

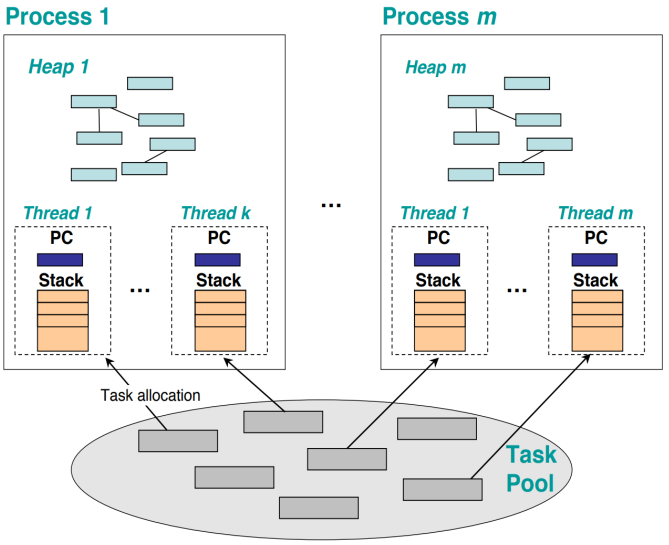


Figure 3: Diagram of how parallel processes are split up into threads and tasks.

Process - A process is an independent unit of computation with private address space and usually comprises of multiple threads. The process-thread model does not include registers so it is not hardware-specific.

Task - Indicates a unit of computation that has been identified by the programmer, for example in a workpool setting and is often used synonymously with thread.

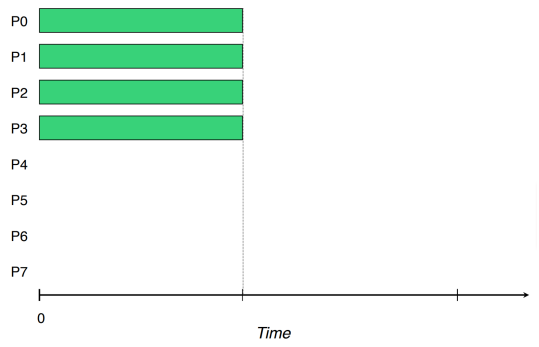
Thread - The basic unit of parallel computation. It is lightweight and shares address space with other threads in the same process, but has its own separate stack and program counter.

Filament - Filaments wind together to form threads. These are primitive units of pure computation which does no communication. Filaments don't necessarily need context switching if they are small enough because they will terminate with a result.

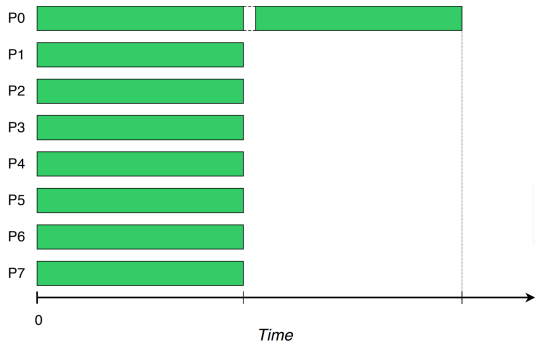
2.2 Granularity

Granularity is a **relative** measure of the ratio of the amount of computation to the amount of communication within a parallel algorithm implementation. In other words, it is a term used for the size of a parallel task in terms of its execution time.

For example, **coarse-grained** tasks are larger and relatively few in number while **fine-grained** tasks are smaller but in larger numbers. If a program is too coarse-grained, then there is not enough parallelism, resulting in poor utilisation. On the other hand if a program is too fine-grained, the overhead of thread creation and communication overtakes the benefit gained from parallel computation. A big issue in parallelism is determining the optimal granularity of a task.



(a) Too little granularity



(b) Too much granularity

The granularity is partially determined by three characteristics of the algorithm to parallelise and the hardware used to run the algorithm.

- **Structure of the problem** - Algorithms that are inherently *data-parallel*, that is few unique operations are done over many pieces of data are often fine-grained by definition, as the same operation is applied to all the data. On the other hand, if only larger subroutines can be executed in parallel which require many calculations and little communication, then they are inherently coarse-grained.
- **Size of the problem** - Given 10 data elements and 10 processing elements (PEs), then only 1 clock cycle is required to process all 10 elements in parallel. However if the problem size is increased to 100 elements, then each PE now has to work on 10 elements each. This implies that larger sized tasks are more coarse-grained by default.
- **Number of processors** - By the same token as the size of the problem, the number of processors also directly affects granularity as there are only so many limited processing units. More processors would lead to more fine-grained granularity as each processor has to do less, provided the size of the problem stayed constant.

Granularity is important in choosing the most efficient paradigm of parallel hardware for the algorithm at hand. For example SIMD architectures are best for fine-grained algorithms while MIMD architectures are less effective due to the message-passing needed between MIMD cores. This further indicates that communication speed is a factor in choosing granularity, as a fine-grained task would be heavily hampered by slow communication while coarse-grained tasks would be effected less.

2.3 Amdahl's law

Amdahl's law is a law which governs the speed-up of parallelism on a given problem. It is used as a way to determine limits on parallel optimisation. It states that if the proportion of sequential work performed by a program is s, then the maximum speedup that can be achieved is $\frac{1}{s}$. This may be expressed in the equation:

$$\text{Speedup} = \frac{1}{s + \frac{p}{N}} \tag{1}$$

where N is the number of processors available and p is the amount of time spent on parallel parts of the program that can be done in parallel.

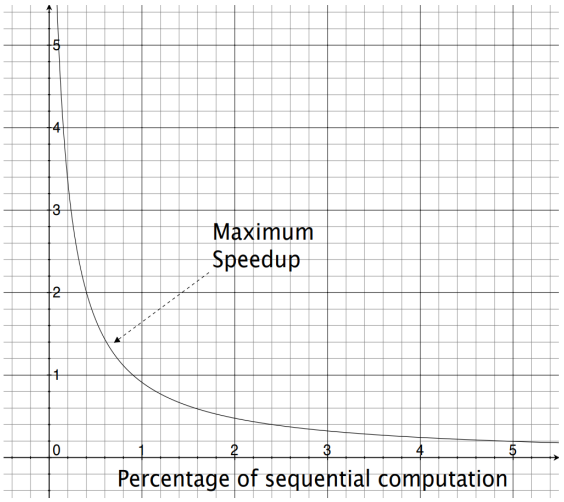


Figure 5: The effect of Amdahl's law on parallel computation. The y-axis shows the maximum speedup that can be achieved if the problem size remains fixed.

The graph in figure 5 shows that to achieve a high speedup, only a tiny percentage of the total execution time can be sequential. This means the need for sequential operations heavily dominates computation time, and a lot of parallelism is needed to get any good speedup.

2.4 Parallel Haskell

Writing explicit parallel programs is very difficult as there are many things that have to be specified by the programmer, most notably:

- Task identification
- Task Creation
- Task placement
- Data placement
- Load balancing
- Communication

Having to deal with all this at a low-level of program is prone to errors, which is why additional abstractions are required. **Glasgow Parallel Haskell (GpH)** aims to solve this by using an approach where the runtime system manages parallelism which is introduced by the parallel programmer. The goal is to create a simple parallel language extension and *automatic* control of parallelism. GpH procides a simple basic model for parallelism, programmable for both task and data parallelism. It communicates through a virtual shared heap and is architecture independent.

A purely functional language like Haskell has several advantages when it comes to parallel execution.

- No side effects - because there are no side effects, it is always safe to execute computations in parallel and the results will be the same regardless of the order. More strongly, the result will be identical to running the program sequentially.
- No race conditions - because the order of I/O operations is fully defined by the language, no race conditions or unexpected outputs can occur from interleaving I/O operations in the wrong order.

- No deadlocks - data and control dependencies in the functional language ensures that there can be no unresolved mutual dependencies between tasks.

2.4.1 The `par` and `pseq` annotation

The primitive, higher-order function `par` is used by the programmer to mark a sub-expression as being suitable for parallel evaluation. It is the basic way to introduce parallelism in GpH.

Listing 1: Example of the `par` annotation.

```
1 -- Create a spark for a and return the value of b
2 a `par` b
3
4 -- Spark x and return f x
5 x `par` f x where x = ...
```

Listing 2: Example of the `pseq` annotation.

```
1 -- Evaluate a and return the value of b
2 a `pseq` b
3
4 -- First evaluate x, then return f x
5 x `pseq` f x where x = ...
```

The `par` annotation is an example of the programmer exposing parallelism to the compiler without explicitly forking threads with concurrent methods. The expression `a ‘par‘ b` *sparks* the evaluation of `x` and returns `y`. Sparks are queued for execution in FIFO order, but are not executed immediately. The runtime system determines if the spark is converted into a real thread when it detects an idle CPU. This way the parallelism is spread among real CPUs but it is not a mandatory scheme.

While `par` creates new sparks, `pseq` and `seq` ensures sequential evaluation. The two are almost equivalent, the difference being that `seq` can evaluate its arguments in either order, but `pseq` must evaluate the first argument before the second, allowing the programmer to control evaluation order in conjunction with sparking with `par`.

Listing 3: Parallel Fibonacci

```
1 import Control.Parallel
2
3 pfib :: Int -> Int
4 pfib n
5     | n <= 1 = 1
6     | otherwise = n2 `par` (n1 `pseq` n1 + n2 + 1)
7         where
8             n1 = pfib (n-1)
9             n2 = pfib (n-2)
```

`par` is used to spark a thread to evaluate `n1` and `pseq` is used to foce the parent thread to evaluate `n2` before adding the two sub-expressions back together. When using `par`, the general rule of thumb is that the sparked computation should be required at a later time, but not too soon. The sparked computation should not be too small, otherwise the cost of forking it in parallel will be too large relative to the amount of parallelism gained. The runtime system decides on issues such as when and where threads are created and the communication between threads. Most notably, it determined whether a thread is created or not. This parallelism model is therefore called **semi-explicit** as the programmer only marks which areas can be parallelised, and the runtime system decides whether to execute the sparks in parallel based on information during execution such as

the system load and processors available. This approach eliminates the complications of low-level parallelism, leaving responsibility of these issues to the runtime system.

2.4.2 Sparks

GpH uses an **evaluate-and-die** mechanism where sparks that are not converted during runtime are simply not run at all. This is especially true in cases where the sparks contain little computation which finishes in the parent thread before the spark has a chance to be converted. This way, there is no need to spark new threads if there is not enough hardware but does have an issue of the shared spark pool being a single point of contention.

Haskell programs typically have one main thread for root computation. Programs may be forked as **Haskell Execution Contexts** (HEC) which are larger, more heavyweight evaluation engines, often implemented using operating system threads. Each HEC maintains pools of lightweight sparks and threads. **Sparks** are runtime head objects that have been marked for *possible* parallel evaluation and are held in a spark pool.

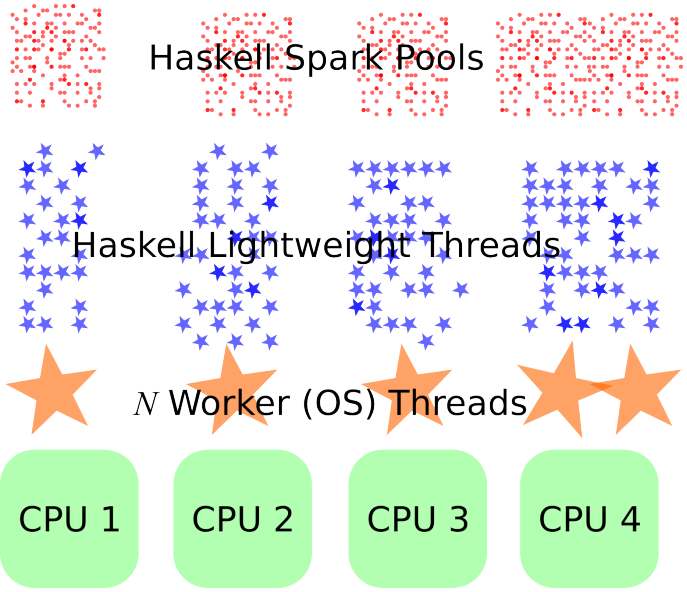


Figure 6: Visualised difference between Haskell threads and spark pools and how they are mapped to OS threads.

Haskell threads are created from sparks automatically as needed by the runtime system. A thread has a **Thread State Object** (TSO) that contains the following:

- A private stack
- Private registers
- A pointer to the spark that created the thread
- Stats (BLOCKED, RUNNING etc.)
- Other information needed to run the thread

All threads share a common heap for dynamic memory.

3 Parallel patterns

A **pattern** is a common way of introducing parallelism which helps with the program design and helps to guide the implementation. Often a pattern may have several different implementations, for example a *map* can be implemented as a *task farm*. Different implementations may then have different performance characteristics. There are primarily two forms of underlying parallelism: **data parallelism** and **task parallelism**.

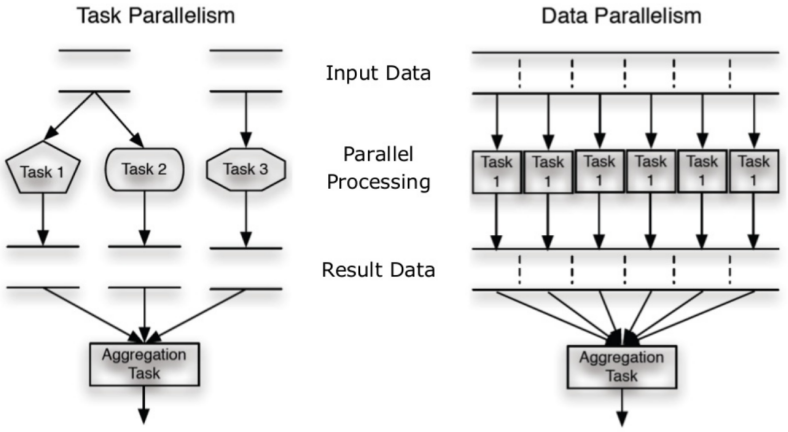


Figure 7: Difference between task and data parallelism.

3.1 Data parallelism

Data parallelism comes from parallelism that is primarily extracted from the structure of the data that operations are applied to. Operations are applied *independently* to several data items, for example the same operation to all elements of a list or array. Data parallelism is generally more regular (that is to say all parallel tasks have similar sizes and functionality) and involves less complex programming structures. Typically data-parallelism may produce large amounts of very fine-grained parallelism which is a good fit for massively parallel architectures like GPUs or SIMD vector architectures.

Examples of data parallel patterns include:

- Parallel maps
- Parallel scans
- Map-reduce

3.1.1 Parallel maps

Parallel maps are one of the simplest forms of data parallelism and is also one of the most useful. Sequential maps are very commonly used in sequential Haskell to apply a function to every element in a list. Parallelising this simply involves applying the function in parallel to every element, creating extra threads to evaluate parts of the list.

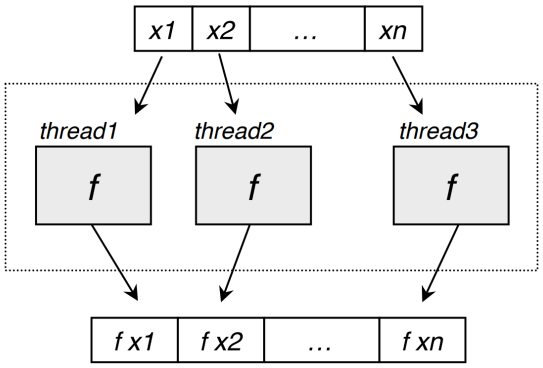


Figure 8: Parallel maps map a function across each element in parallel.

Listing 4: Parallel map implementation

```
1 parmap :: (a -> b) -> [a] -> [b]
2 parmap f [] = []
3 parmap f (x:xs) = let fx = f x in
4                   fx `par` (fx : map f xs)
```

`parmap` can be used anywhere where standard sequential `map` is used. Although this can be done by simply replace all instances of `map` with `parmap`, it may not lead to efficient parallelism. There are a few caveats that need to be taken into account to achieve good parallel performance:

- All elements of the data structure must already have been evaluated before the `parmap` is applied
- There must be no dependencies between the results of the parallel map

3.1.2 Parallel zipWith

A `zipWith` is a kind of map that works over two input lists. It maps a function across a pair of elements rather than single elements from one list. The operation *op* is allied in parallel to each pair of elements *x1*,*y1* etc. to give the resulting list. The final result is evaluated using a given strategy. As usual, the function calls are only worth evaluating in parallel is the operation *op* is expensive.

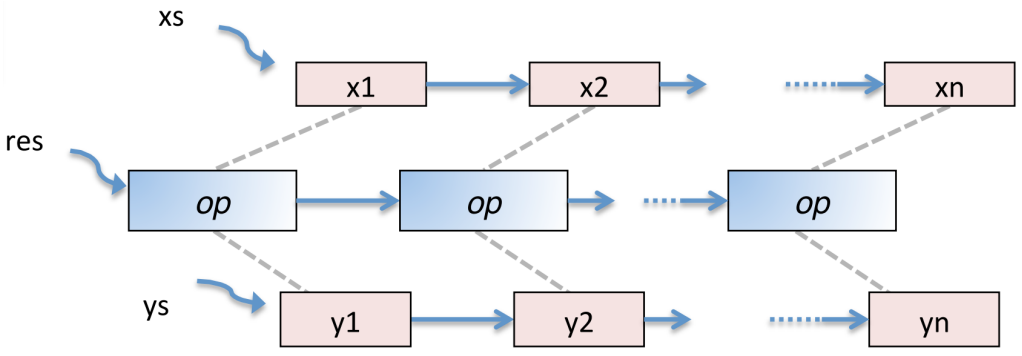


Figure 9: Parallel `zipWith` over two input lists `xs` and `ys`

3.1.3 Parallel fold (reduce)

A fold is a more complex pattern that applies an operator *between* each pair of elements in a list. They can be easily parallelised, however care must be taken over the properties of the operator being used.

Listing 5: Type signature of a parallel fold.

```
1 parFold :: (a -> a -> a) -> a -> [a] -> a
2 parFold f z l = ...
3
4 -- Examples of fold uses
5 sum = parFold (+) 0
6 product = parFold (*) 1
```

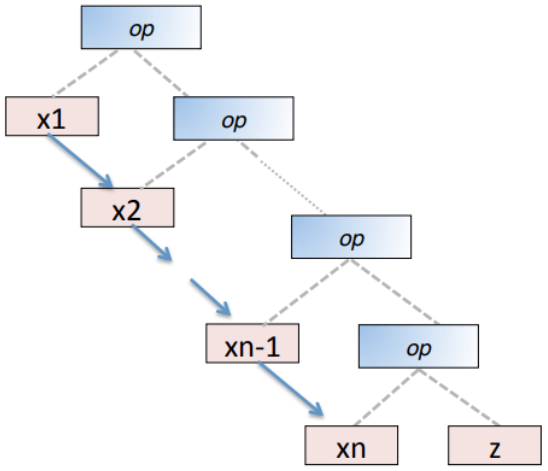


Figure 10: Sequential right fold.

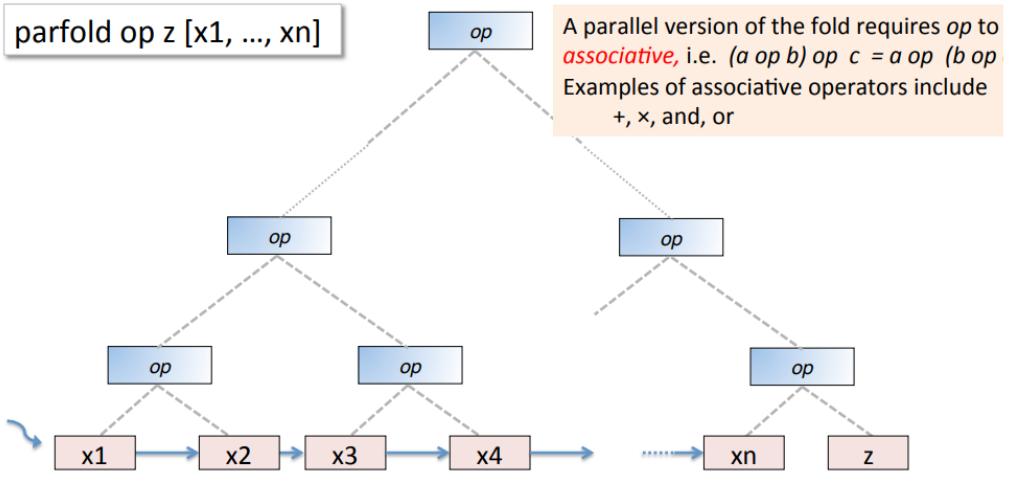


Figure 11: Parallel fold.

The parallel version of fold requires **associative** operators because the order of applying the operators cannot matter. This allows the internal computations of the fold to be reordered to give better parallel behaviour. Each operation can be executed by a separate thread and the results

combined independently in a tree-like manner. This could not have been done with an associate operator, as the elements of the list cannot be reordered and keep the same result.

3.1.4 Bulk Synchronous Parallelism (BSP)

Bulk synchronous parallelism is a more complicated and sophisticated data-parallel pattern. BSP computations proceeds in a series of **supersteps** where all threads perform the same computation on different data on each superstep, similar to a parallel map. The difference is that after each superstep, all threads synchronise and exchange some or all data with other threads as needed. The threads are synchronised by barrier, meaning all threads wait until data exchange is completed for all other threads before starting the next superstep. This is potentially very expensive when all other threads are blocked waiting for one or two threads to finish but guarantees that all threads have produced and exchanged information which means no deadlock or livelock.

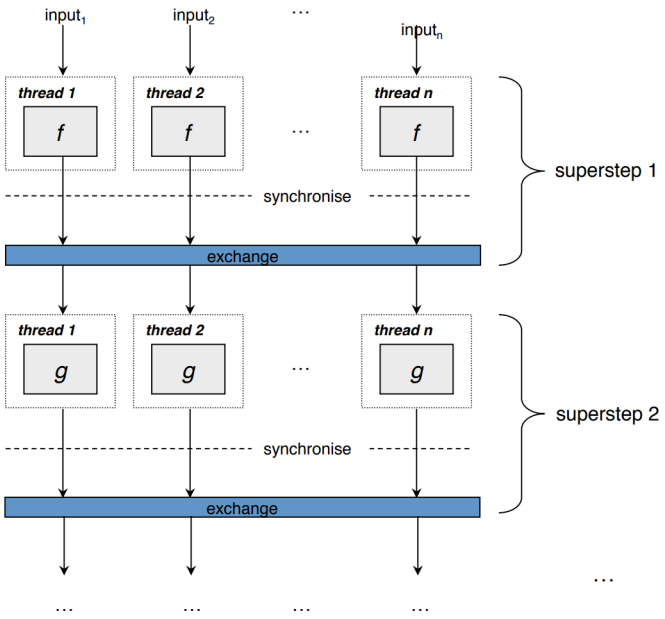


Figure 12: The bulk synchronous parallelism process.

During each superstep, each thread works on its own inputs using *private local data* and *shared global data*. This global and local data is exchanged between each superstep. The global data from all threads is combined to give a new list of global data, which is then passed as a single value to all of the threads that are evaluating the next worker task. Each thread has its own local state which may be altered as a result of the computation at each superstep. Threads also have access to the global state. During the exchange, each thread produces new local and global state. The global state can only be changed during the exchange.

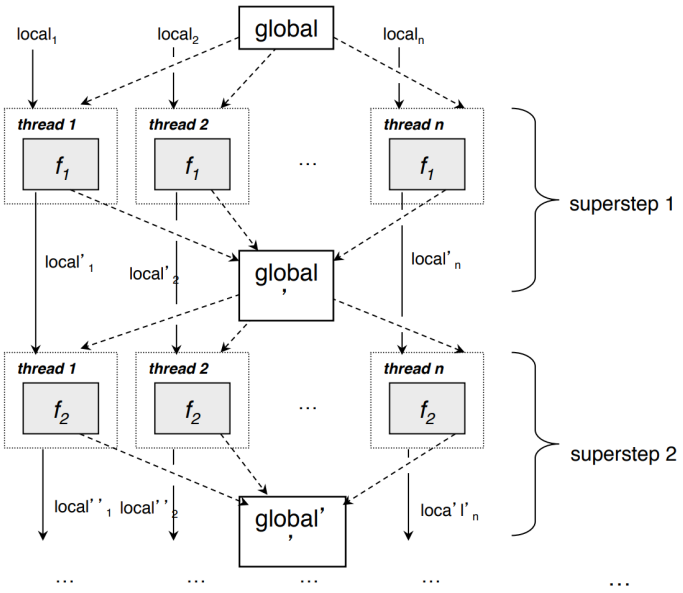


Figure 13: BSP local and global data exchange process.

BSP works well when each computation in every superstep has a similar size. The weakness is that in the synchronisation phase, if an operation takes different amounts of time for different inputs, individual threads may be blocked for significant periods of time waiting for other threads to complete.

The BSP pattern also provides a simple model of parallel execution cost. Since the global synchronisation waits on all threads, the amount of time taken to execute a superstep is always the same for all threads. Further, it follows that the exchange step has the same fixed constant size. This gives the following equation for the time required to execute a complete sequence of m BSP supersteps

$$\sum_{i=1}^{m} f_i + c_{ex} \times (m-1) \tag{2}$$

where

- f_i is the maximum cost of the operation at step i
- c_{ex} is the cost of exchange/synchronisation
- m is the number of supersteps

This calculation assumes that there are enough processors available to execute all of the threads that are created to execute the pattern.

3.1.5 Parallel map-reduce

Map-reduce is a common pattern that has been applied to commercial, large, distributed server farms for dealing with big data. It uses a combination of *map* to map a function across the data and a *reduce* to reduce the results to simpler values. It works by splitting the input into a large number of similarly sized tasks that can be processed independently using the *map* operation. Then each intermediate group of data can be processed with the *reduce* function.

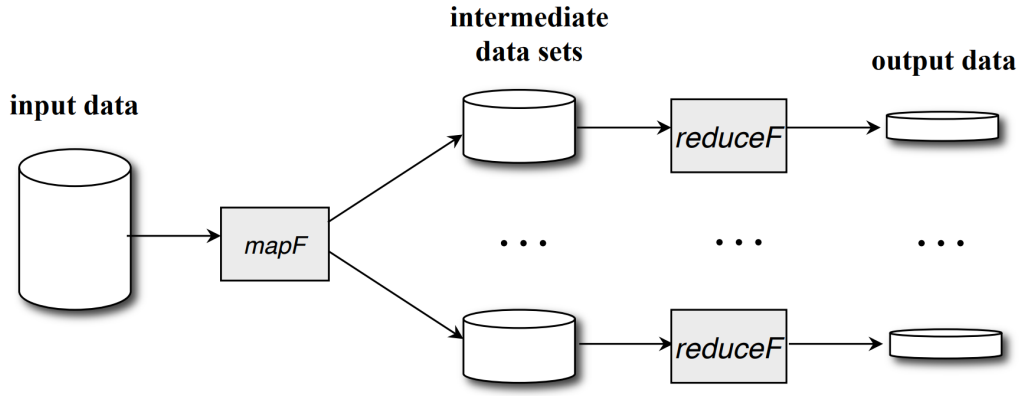


Figure 14: Map-reduce with parallel reduction step.

Quite clearly the reduce operations can be run in parallel with data parallelism, using a parallel map to map the reduce function to the data. However, it is also possible to increase the parallelism by running the map operations in parallel by splitting the data and them mapping each map in parallel. If the reduction operation is associative, each set of intermediate results may then be reduced independently: a local reduce function is applied to each intermediate set of results and an overall reduce function applied at the end of get the final result.

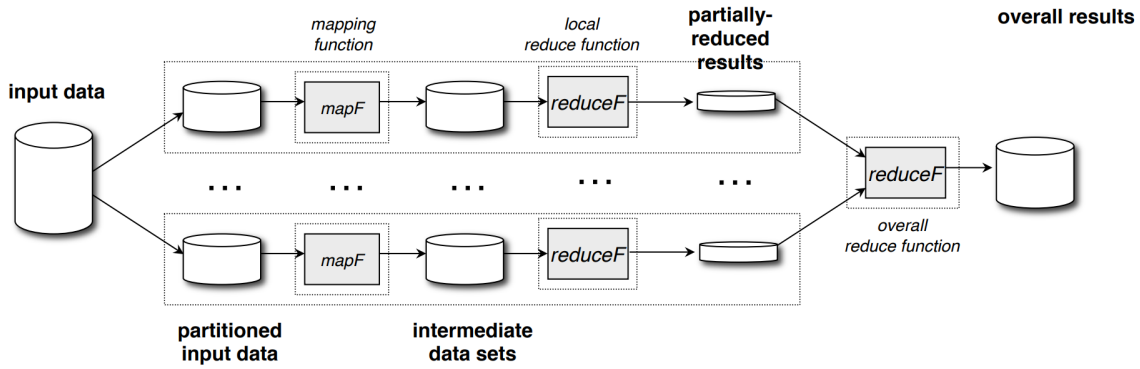


Figure 15: Map-reduce with parallel mapping step.

3.1.6 Parallel scan

3.2 Task parallelism

In contrast to data parallelism, task parallelism comes from the control flow in a program. This is more flexibly than data-parallelism but often exposes less parallelism and is harder to conceptualise. Further, task-parallelism has a less regular size and structure, as each part of the control flow can have different functionality, making it difficult to manage at runtime.

Examples of task parallel patterns include:

- Piplines
- Divide and conquer
- Task farm

Task parallelism aims to deal with issues in data parallelism, mostly to do with irregular thread granularities. For example in BSP, if the thread granularities are not balanced, some cores may remain idle, wasting possible computation time. This is where task parallelism comes in. It is parallelism that does not come purely from the structure of the data, where computations are not the same for each item and where there are irregular granularities.

In general, task parallelism is much harder to handle compared to data parallelism for the following reasons:

- Less regular
- Harder to map to the available resources
- Can give much less parallelism
- Can be harder to identify the patterns
- May need specialist support to implement

However, because many problems cannot be solved by data parallelism, task parallelism must be used in these cases.

3.2.1 Producer-consumer

Producer-consumer is one of the simplest patterns of task parallelism. One thread produces values which another thread consumes. If the producer produces only one simple value, then there is very limited scope for parallelism, however if the producer produces a lazy list or incrementally constructed structure, then each element can be consumed as soon as it is produced.

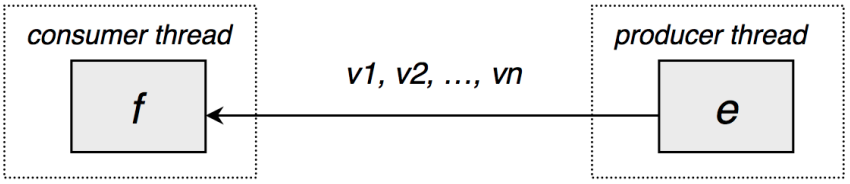


Figure 16: Producer-consumer parallelism. If the original sequential operation is `f $ e`, then the parallel version is `f $|| rdeepseq e`.

The producer-consumer pattern can be easily implemented using `` `par` ``

Listing 6: Sequential producer-consumer

```
1 f $ e = f e
```

Listing 7: Parallel producer-consumer

```
1 f $| e = e `par` f e
```

This works because by exploting Haskell's lazy evaluation to pass the result from the producer to the consumer as it is produced.

3.2.2 Parallel pipelines

Pipelines are an extension of the producer-consumer parallel pattern. In essence the pipeline is a combinations of a series of producer-consumer operations over a stream of inputs.

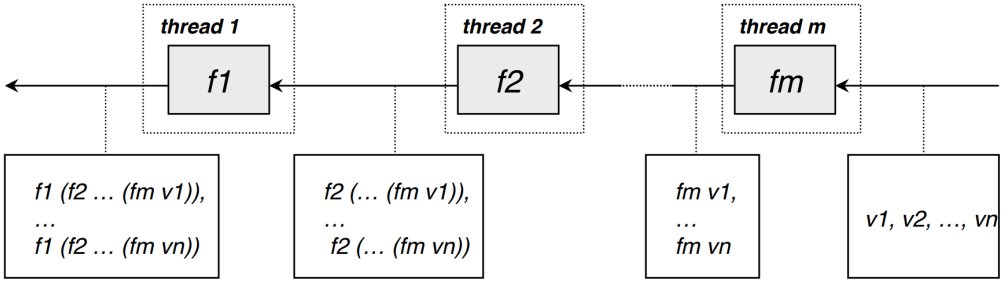


Figure 17: Parallel pipeline where each thread executes a different function f.

In the sequential version, a pipeline simply applies on function after another to the input `x`, giving `pipeline3 f1 f2 f3 x = f3 $ f2 $ f1 $ x`. The parallel version uses the `$||` operator to add a strategy to each stage that is applied to the input argument in parallel with the evaluation function.

3.2.3 Parallel streams

Streams are the natural extension of a pipeline where the operations are repeated over multiple values. The stream produces results in the order that the inputs are given, in other words, there is exactly one output produced for each input value. This is done simply by applying a `map` to each argument function

Listing 8: Parallel stream pattern on a three-stage pipeline.

```
1 parstream3 f1 f2 f3 = parpipeline (map f1) (map f2) (map f3)
```

This completely encapsulates the three-stage pipeline as a streaming definition. Of course this can be generalised to an arbitrary number of stages using a list instead of specifying each function seperately. The drawback to this generic method is that each function must have the same type

Listing 9: Arbitrarily sized `parstream` using a list of functions `fs`.

```
1  parpipeline fs z =
2    foldr (\f x -> f $|| rdeepseq $ x) z fs
3
4  parstream fs l =
5    parpipeline (map map fs) l
```

3.2.4 Divide and conquer

The divide and conquer pattern is one that commonly occurs in parallel computation and is a pattern that can be easily identified. It works by splitting the data into two or more independent parts so that each part can be worked on in parallel and the results combined at the end to give the final result. Sometimes we wish to divide until the problem is small enough to be trivially solvable, but this depends on the optimal granularity.

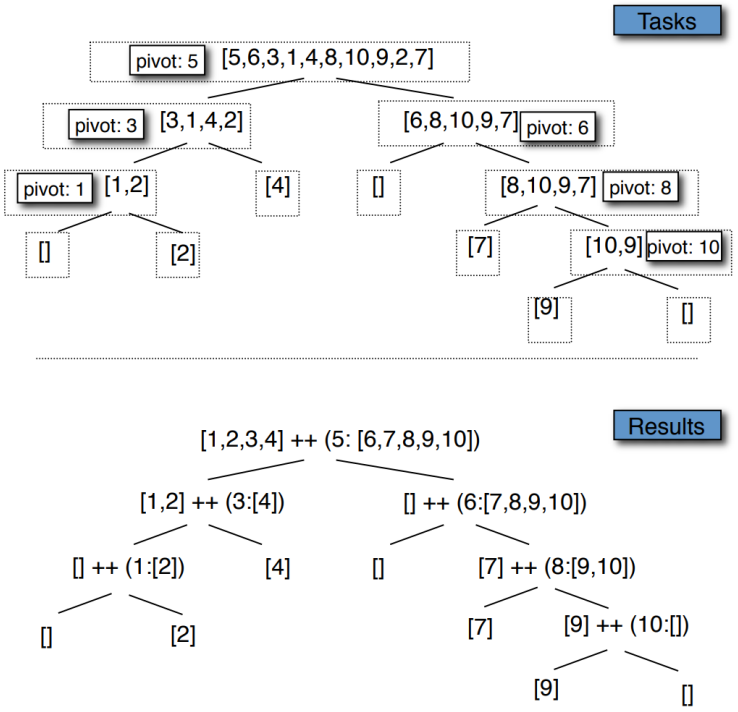


Figure 18: Example of divide and conquer with quicksort, dividing the input until the trivial case.

The strength of divide and conquer comes from the fact that it is completely recursive to the base case. In other words, each sub-divided task can be further divided with the same divide and conquer approach. The has several benefits:

1. A large number of tasks can be generated very rapidly
2. Can be quickly spread across a parallel system - this is especially important for distributed memory systems
3. Communication paths are well-defined and hierarchical - again important for distributed memory systems
4. No single processor becomes overloaded with the task of distributing work and combining results as the division can be done recursively on separate processors

The concept of **thresholding** is important to ensure the right granularity is found. We do not want tasks that are too small or trivial as the overhead of creating threads for these tasks will be greater than the amount of work gained from parallelising the work. The idea is to deliberately create larger tasks either once enough parallelism is created, or when the new tasks created fall below some threshold size. Tasks smaller than the threshold are not further divided and are processed sequentially.

This allows one to control the size of the parallel tasks that are created by varying the `threshold` value. It should be noted that if the size of the input list is large, the calculation of the length of the list can be expensive. In this case, it may be more effective to track the depth of the tree rather than the length of the input and only parallelise down to a certain depth.

There are three components to the **general divide and conquer** pattern

1. The input data is split into two or more parallel tasks
2. When the tasks are small enough, based on some threshold or depth, a sequential worker function is applied to the input

3. The results from the split tasks are combined to give an overall result that is returned to the next higher level in the tree

This gives a general pattern than can be used in a variety of situations.

Listing 10: A general divide and conquer pattern

```
dc :: Strategy b -> (a -> [a]) -> (a -> Bool) -> ([b] -> b) -> (a -> [b]) -> a -> b
dc strat splitf thresholdf combinef seqworkerf input = combinef results
  where results =
    if thresholdf input then
      seqworkerf input
    else
      parMap strat (dc strat splitf thresholdf combinef seqworkerf) (splitf input)
```

3.2.5 Task farms/workpools

In a task farm (also called master-slave) system, a number of worker processes are coordinated by a single master process. The master process decides which task should be executed by which worker, then it allocated the task to the chosen worker and receives the result from each work and merges the results into the result stream.

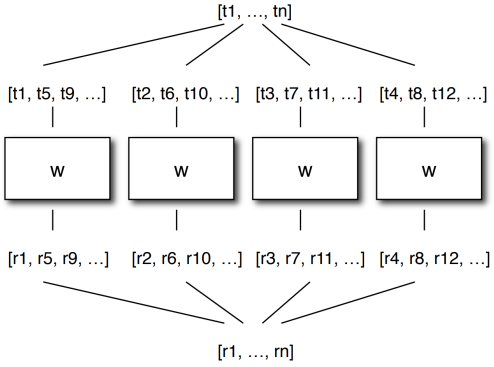


Figure 19: A parallel task farm with 4 workers.

In general, the tasks are allocated to workers equally. Each coarse grained worker can execute a fine grained task, which saves on thread creation overhead.

Listing 11: General parallel task farm implementation.

```
taskFarm :: Strategy [b] -> (a -> [b]) -> Int -> [a] -> [[b]]
taskFarm strat f nWorkers tasks = concat results `using` parList strat
  where results = unshuffle nWorkers (map f tasks)

unshuffle :: Int -> [a] -> [[a]]
unshuffle n xs = [takeEach n (drop i xs) | i <- [0..n-1]]
    where takeEach :: Int -> [a] -> [a]
      takeEach n [] = []
      takeEach n (x:xs) = x : takeEach n (drop (n-1) xs)
```

3.2.6 Workpools

A parallel workpool is similar to a task farm except that it tracks which worker has completed a task before assigning it the next task from the list of input tasks. In other works the tasks are

dynamically allocated as they are completed. This allows the workpool to deal with collections of tasks, where the tasks may have widely differing sizes.

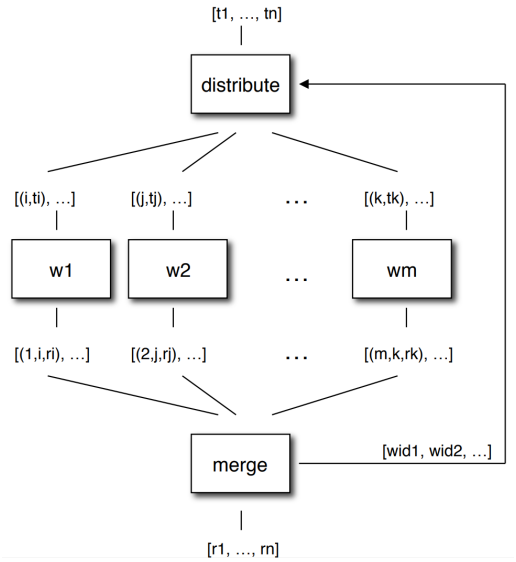


Figure 20: Parallel workpool, which dynamically allocates tasks to workers as they are completed.

3.2.7 Parallel search

Parallelism can be used to speed up search in a data structure. In a search problem, we are looking for the existence of some value, so each individual search can be run in parallel. Moreover, any solution is acceptable, so the search an stop when the value is found.

Listing 12: Parallel search by applying `parmap` on the `find` operation.

```
1  parSearch :: (a -> Bool) -> [a] -> Bool
2  parSearch find l = any (== True) (parmap find l)
3
4  any p [] = False
5  any p (x:xs) = p x || any p xs
```

3.2.8 Branch and bound parallelism

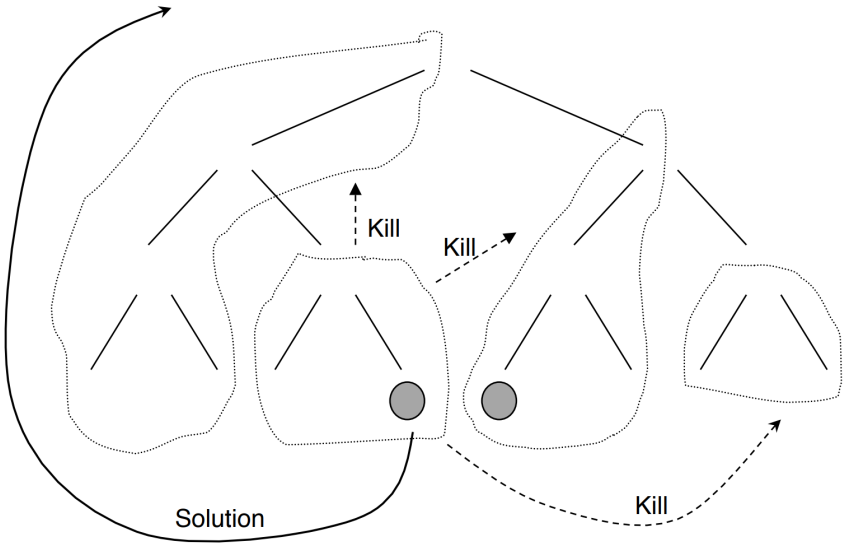


Figure 21: In branch and bound, parallel search threads are killed once the solution is found.

4 Parallel skeletons

A skeleton is an implementation of a parallel pattern that allows pluggable higher-order functions. They are like templates which can be instantiated with concrete worker functions. For example, take the general divide and conquer function

Listing 13: A general divide and conquer pattern

```
1  dc :: Strategy b -> (a -> [a]) -> (a -> Bool) -> ([b] -> b) -> (a -> [b]) -> a -> b
2  dc strat splitf thresholdf combinef seqworkerf input = combinef results
3    where results =
4      if thresholdf input then
5        seqworkerf input
6      else
7        parMap strat (dc strat splitf thresholdf combinef seqworkerf) (splitf input)
```

It requires five arguments (excluding the strategy)

- The `splitf` function - which specifies how to divide the input
- The `thresholdf` function - which specifies when is the input simple enough to stop parallelising
- The `combinef` function - which specifies how to combine the results
- The `seqworkerf` function - which specifies what should actually be done to the input. Here it is also the sequential work, or base case of the divide and conquer
- The input data

By passing in the appropriate functions into the `dc` function, we are instantiating the divide and conquer skeleton.

Skeletons may be divided into two types

1. Algorithmic skeletons - that capture some high-level parallel pattern

2. Behavioural (implementation) skeletons - that define some specific behaviour on a particular parallel architecture

Most skeletons are algorithmic, but often include some behavioural information.

Finally, skeletons can be defined for most languages using a library that provides functions as arguments. In languages like Java, they can be defined as abstract classes for the programmer to override.

4.1 Advantages and disadvantages of skeletons

4.1.1 Advantages

- Skeletons match common patterns of parallelism. If the pattern can be identified, then it is known which skeleton should be used as the skeletons are the general implementations of the pattern
- Skeletons help structure parallelism by providing hooks that just need to be instantiated
- Skeletons are general - they can be applied in many settings and used across different languages
- Skeletons often have strong cost models, making it easy to calculate the cost of a skeleton

4.1.2 Disadvantages

- Skeletons are fixed, meaning there is only a small fixed set of skeletons which cannot be altered if they do not fit the algorithm
- Skeletons cannot be nested as their implementation does not allow this. Further, combining cost models is difficult or impossible
- Skeletons may impose a rigid structure. If the problem uses a skeleton, the parallelism cannot be changed, making it difficult to refactor a program as the skeleton cannot be removed

5 Evaluation strategies

Evaluation strategies use lazy higher-order functions to separate the two concerns of specifying the algorithm and specifying the program's dynamic behaviour. A function definition can then be split into two parts: the algorithm and the strategy.

Listing 14: The strategy type specifies the "evaluation" type.

```
1 type Strategy a = a -> Eval a
```

The separation of concerns makes both the algorithm and the dynamic behaviour easier to comprehend and modify. Changing the algorithm may entail specifying new dynamic behaviour and conversely, it is easy to modify the strategy without changing the algorithm. Strategies have several desirable properties:

- Strategies are powerful - simple strategies can be composed, or passed as arguments to form more elaborate strategies
- Strategies can be generic over all types in a language
- Strategies are extensible - allowing a user to define new application-specific strategies
- Strategies are type safe - the normal type system applies to strategic code

It should be possible to understand the meaning of a function without considering its behaviour. Strategies also help fight the laziness which only evaluates when needed.

The idea of strategies is to abstract the algorithm from the behaviour, for example different strategies can be applied to the same algorithm that results in different parallel behaviour.

Listing 15: Evaluate `qsort xs` completely

```
1 qsort xs `using` rdeepseq
```

Listing 16: Evaluate `qsort xs` as a data parallel computation

```
1 qsort xs `using` parList rdeepseq
```

5.1 Basic evaluation strategies

There are three most basic strategies which specify simple evaluation:

- **r0** - no evaluation
- **rseq** - evaluate the expression
- **rpar** - spark the expression

Listing 17: Definitions of basic strategies

```
1  r0 :: Strategy a
2  r0 x = return x
3
4  rseq :: Strategy a
5  rseq x = x `pseq` return x
6
7  rpar :: Strategy a
8  rpar x = x `par` return x
```

Strategies all do some kind of evaluation and return a result. The `r0` strategy does nothing as it is there for the sake of the type system to specify a strategy that does *no* evaluation.

5.2 Using strategies

The `` `using` `` is a keyword used to apply a strategy, for example

```
1  using :: a -> Strategy a -> a
2
3  f x `using` rpar --spark f x
4  f x `using` rseq --evaluate f x
```

The line `e `using`strat` specifies that expression `e` uses strategy `strat`.

The `withStrategy` function can also be used to apply strategies. It is defined as

```
1  withStrategy :: Strategy a -> a -> a
2
3  --Example
4  withStrategy rseq (qsort xs)
```

It evaluates `e` with strategy `strat` and is most useful for building new strategic functions and skeletons

```
spark = withStrategy rpar
```

5.3 Evaluation

The **Eval** type wraps up values and is defined as a monad

```
data Eval a = Done a

instance Monad Eval where
  return = Done
  m >>= k = case m of
    Done x -> k x

-- Puts values into the monad
return :: Eval a

-- Sequences two Eval operations
(>>=) :: Eval a -> (a -> Eval b) -> Eval b

-- Extracts results from the monad
runEval :: Eval a -> a
```

In Haskell, the evaluation is **lazy**, which means it only evaluates what is needed for a result. For data structures like lists, this means only the parts of the structure that are needed are created, which allows infinite structures to be created. By default, Haskell uses the `rseq` strategy for evaluation.

5.3.1 Deep evaluation

To force evaluation as part of a strategy, we need to evaluate the entire result, for example if we wanted the entire list result. To do this, the `rdeepseq` strategy is needed. `rdeepseq` forces the evaluation when needed, which fully evaluates the expression.

```
rdeepseq :: NFData a => Strategy a
rdeepseq x = rnf x `pseq` return x
```

It is usually better than `rseq` but may evaluate twice. Further, it forces eager evaluation.

5.3.2 NFData class

The `NFData` class is a Haskell type class which incldues the `rnf` operations, whose purpose is to fully evaluate its argument. It is defined using `seq` but does not guarantee the evaluation happens in order.

```
class NFData a where
  rnf :: a -> ()
  rnf x = x `seq` ()
```

5.3.3 Strategy composition

6 Haskell implementation

6.1 Synchronisation

Most information is local to a Haskell Execution Context (HEC) and normally HECs do not need to communication except during synchronisation. Synchronisation is needed when

- Garbage collection is needed
- A thread becomes blocked on another thread (black hole)
- When a HEC updates a result that a thread on another HEC is blocked on
- When exceptions are thrown to another HEC
- When performing foreign function calls
- When load balancing happens

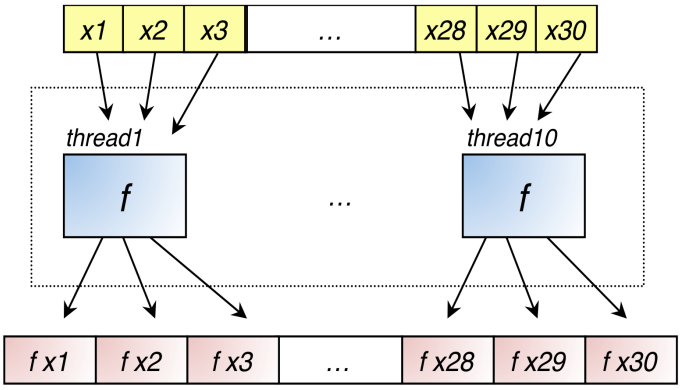
Most synchronisation is needed when values have to be shared across threads. The GHC creates black holes during garbage collection for each shared closure on the stack, though this is done lazily. Threads then block when they encounter a block hole.

Each HEC has its own allocation area which comprises of a number of memory blocks. The garbage collector is invoked when the area becomes full and then proceeds in parallel with the normal execution. This is needed to ensure that one HEC does not write over another HEC's heap. During garbage collection, HECs share information about shared heap objects such as results in data replication. Mutable objects are locked at this stage.

6.2 Improving granularity

6.2.1 Chunking

Chunking involves grouping multiple data items together before passing it as work to a thread. This increases granularity by allowing a single task to deal with more work and help reduce thread creation overhead.



6.2.2 Buffering

Buffering can be done to ensure threads are not flooded with work. In this approach, only the first few threads are passed data to work on and the rest are created as the threads complete.

6.3 Tuning for multicore

There are four main issues in tuning parallel code for multicore architectures:

1. Good granularity
2. Cache/memory locality
3. Minimising locking costs
4. Limiting thread creation overheads

6.3.1 Core affinity

Core affinity is the concept of keeping the same threads on the same core. This is done for reasons of cache and power usage. By keeping the threads on the same core, data may be still kept in the cache, which allows for better usage. Furthermore, for power usage reasons it may be better to maintain similar usage levels on all cores. The following flags in GHC allow a programmer to define the affinity

- `ghc -qa`, use the OS to set affinity for worker tasks
- `ghc -qm`, don't automatically migrate worker tasks between cores
- `ghc -qw`, migrate to the current core when a worker task is woken

6.3.2 Virtual cores

In some modern CPUs, a single real core is split into two or more virtual cores. This gives the illusion of having more cores, allowing idle units to do more work, but with simulaneous multithreading. Using virtual cores has many disadvantages compared to real cores:

- Real cores don't share a memory pipeline - This means there will be fewer cache misses as real cores will have separate caches. This leads to a smaller chance of threads stalling due to writing back changed data
- Real cores have more real hardware units - more intensive code can use all available units, so having two virtual cores on one real core would create conflict over resource usage
- Simultaneous multithreading can be much slower than multi-core due to cache thrashing

7 Locks and lock-free

In programming with locks, we want to be explicit about goals and trade-offs. A benefit in one dimension often has costs in another. For example an increase in performance that prevents a data structure from being used in some particular setting. The ultimate goal is to increase performance, either as time or resources used. The trade-off is generally if an implementation can scale well enough to out-perform a good sequential implementation.

There are three kinds of parallel hardware

1. Multi-threaded cores - increase utilisation of a core or memory bandwidth. Long latency oeprations are tolerated and the peak operations per core (ops/core) is fixed.
2. Multiple cores - increase operations per core (ops/core). However caches and off-chip resources don't always cale proportionately

3. Mult-processor machines - increase operations per core and often *do* scale cache and memory capacities and bandwidth proportionately. Has NUMA (non-uniform memory access) memory effects.

7.1 Locks

Locks are the basic building block for concurrency control. It provides a guarantee that other threads do not interfere with the current thread if the lock is being held. In a shared-memory system, threads can *always* interfere, so locks must be strict. Different languages have their own implementations of locks

- C `pthread_mutex`
- Java `synchronized` blocks
- Haskell `MVar`

The locks themselves are typically implemented with lower-level operations such as Test-and-Set (TAS) and Compare-and-Swap(CAS). Furthermore, assembly languages have specific instructions that can be used for atomic lock operations, such as `lock xchg` in x86 and `ldrex/strex` in ARM.

7.1.1 Comapre and swap

The compare and swap operation is used to implement both concurrency primitives like semaphores/-mutexes, as well as more sophisticated lock-free and wait-free algorithms.

The idea behind compare and swap is to read some memory location and remember the old value. Based on that old value, a new value is computed. A thread then tries to swap in the new value using CAS, where the comparison checks the location still being equal to the old value. If the CAS fails, it has to be repeated.

Listing 18: Compare and swap

```
1  int compareAndSwap(int *b, int exp, int place)
2  {
3    // b is a pointer to the locatio we think contains exp
4
5    int result;
6
7    atomic
8    {
9      // read current value of location b
10     result = *b;
11
12     // update b only if we guessed right
13     if (result == exp) *b = place;
14   }
15   return result;
16 }
```

7.1.2 Test and set

Test and set is an instruction used to write (set) a memory location and return its old value as a single atomic operation. If multiple processes access the same memory location, and if a process is currently performing a test-and-set, no other process may begin another test-and-set until the first process's test-and-set is finished.

Listing 19: Test and set

```
bool testAndSet(bool *b)
{
  // b is a location holding true or false

  bool result;

  atomic
  {
    // read current value of location b
    result = *b;

    // atomically set it to TRUE
    *b = TRUE;
  }
  return result;
}
```

If two or more threads try to `testAndSet` at once, only one return `FALSE`, the rest return `TRUE`. The lock is initialisation to `FALSE`, which means the lock is available.

Listing 20: Implementing a lock with test and set

```
void acquireLock(bool *lock)
{
  while (testAndSet(lock))
  {
    // do nothing
  }
}

void releaseLock(bool *lock)
{
  *lock = FALSE;
}
```

Each call tries to acqurie the lock, returning `TRUE` if it is already held.

7.1.3 Issue with locks

Using a test-and-set implementation of locks causes contention on the lock variable, as each thread spins in a while loop. Furthermore, each thread has to atomically set the lock variable on each call, which must be done sequentially to ensure atomicity. The spinning on the while loop is also a waste of system resources as they are wasted clock cycles.

The general problem is that there is no logical conflict between two failed lock acquires while a physical conflict in the cache does occur. For a good algorithm, we wish to only introduce physical conflicts if a logical conflict occurs. For example

- Successful lock-acquire and failed lock-acquire at the same time
- Successful value insertion and failed value insertion at the same time

We do not want to introduce physical conflicts in cases of two failed lock acquires, or two successful insertions. A fix to this issue would be to spin when the lock is held, rather than spin trying to acquire the lock all the time

Listing 21: Spin while the lock is held to reduce contention

```
1  void acquireLock(bool *lock)
2  {
3    do
4    {
5      while (*lock);
6    } while (testAndSet(lock));
7  }
```

By spinning while the lock is held and only trying to test and set when it is free, contention is reduced, making it significantly more scalable to more threads. However, this leads to another problem of stampedes. Once the lock is freed, all the threads waiting for it try to access it at the same time, again causing contention at that particular moment. To alleivate this issue, **backoff** algorithms can be used.

By using back-offs, each threads starts spinning and watches the lock for c iterations. If the lock does not become free, the threads spin locally for s iterations without watching the lock. Setting different values for c and s will have a different effect on the efficiency and concurrency of the program.

Lower value of c

- Less time to build up a set of threads that will stampede
- Less contention in the memory system
- Risk of a delay in noticing when the lock becomes free when no thread is watching

Higher values of c

- Less likelihood of a delay between when a lock is released and a wating thread notices

Lower value of s

- More responive to the lock becoming available
- If the lock doesn't become available then the thread makes fewer accesses to the shared variable (less contention)

The rule of thumb with back-off heuristics is to generally spin for a duration that's comparable with the shortest back-off interval and exponentially increase the per-thread back-off interval, resetting it when the lock is acquired. Finally a maximum back-off interval should be used that is large enough that waiting threads don't interfere with the other threads' performance.

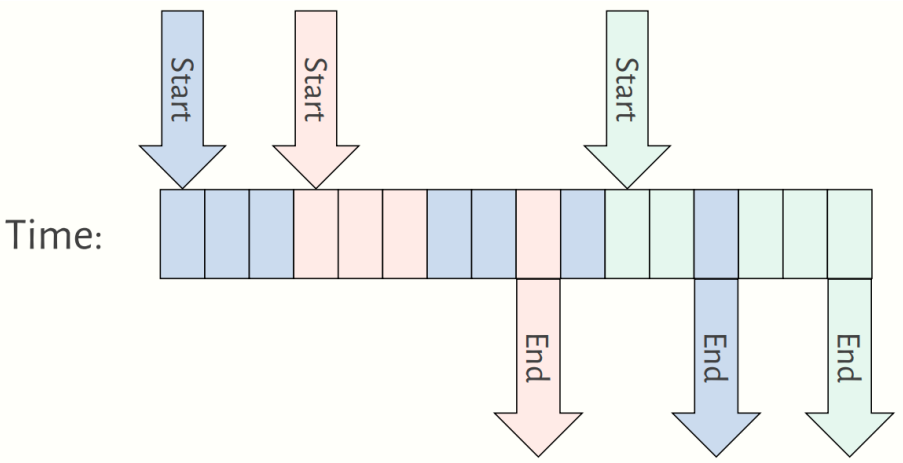
7.2 Non-blocking algorithms

To be able to scale to a large number of cores, we need critical sections to be rare and short. A lock implementation may involve updating a few memory locations, increasing the amount of critical sections needed. Using data structures may help reduce this to only specific memory locations. If we try to shrink critical sections then the time in the lock implementation becomes proportionately greater. So the idea moving forward is to try and make the cost of the operations in the critical section lower or try to write the critical sections correctly *without* locks.

These are known as non-blocking algorithms, in which failure or suspension of any thread *cannot* cause failure or suspension of another thread. Traditional non-blocking algorithms are lock-free is there is guaranteed system-wide progress, and wait-free is there is guaranteed per-thread progress.

7.2.1 Wait-freedom

Wait-freedom is the strongest non-blocking guarantee of progress, combining system-wide throughput with starvation-freedom. An algorithm is defined as wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes. In other words, a thread will finish its own operations if it continues executing steps.



General construction techniques for wait-free algorithms exist (called universal constructions) which uses CAS as a building block. However, their performance does not in general match even simple blocking designs. There are a few approaches that can be taken for wait-free algorithms:

- Queueing and helping strategies - everyone ensures the oldest operation makes progress. However, often a high sequential overhead and therefore has limited scalability
- Fast-path/slow-path constructions - Start with a faster lock-free algorithm and switch over to a wait-free algorithm if there is no progress. If done carefully, we can obtain wait-free progress overall

In practice, progress guarantees can vary between operations on a shared object, for example using a wait-free find, then a lock-free delete.

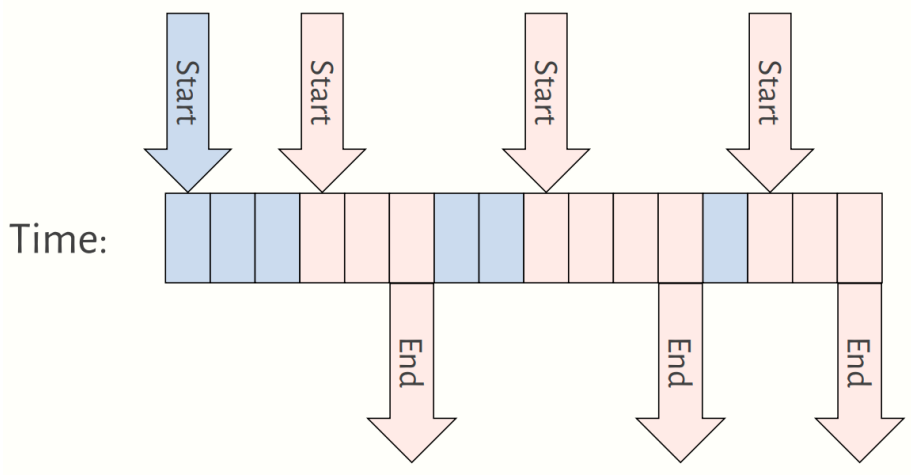
Listing 22: Wait-free using atomic increment

```
1  void inc (Object* o)
2  {
3    atomic_increment(o -> r);
4  }
5
6  void dec (Object* o)
7  {
8    if (0 == atomic_decrement(o -> r))
9    {
10     delete o;
11   }
12 }
```

This works because we defer the atomic operation to the hardware, which will likely use locks, but on this higher-level software, only a single atomic operation is needed, which guarantees wait-free progress.

7.2.2 Lock-freedom

Lock-freedom allows individual threads to starve, but guarantees system-wide throughout. An algorithm is lock-free if, when the program threads are run for a sufficiently long time, at least one of the threads makes progress. In other words, this means that some thread will finish its operation if all threads continue taking steps. All wait-free algorithms are lock-free.



Lock-free algorithms ensure that one thread only has to repeat work if some other thread has made *real progress*, for example an insertion operation has to start again if it finds that a conflicting update has occurred. In particular, if one thread is suspended, then a lock-free algorithm guarantees that the remaining threads can still make progress, so there will be no deadlock where two threads contend for the same mutex lock. The difference between wait-free and lock-free is that wait-free operation by each process is guaranteed to succeed in a finite number of steps, regardless of the other processors.

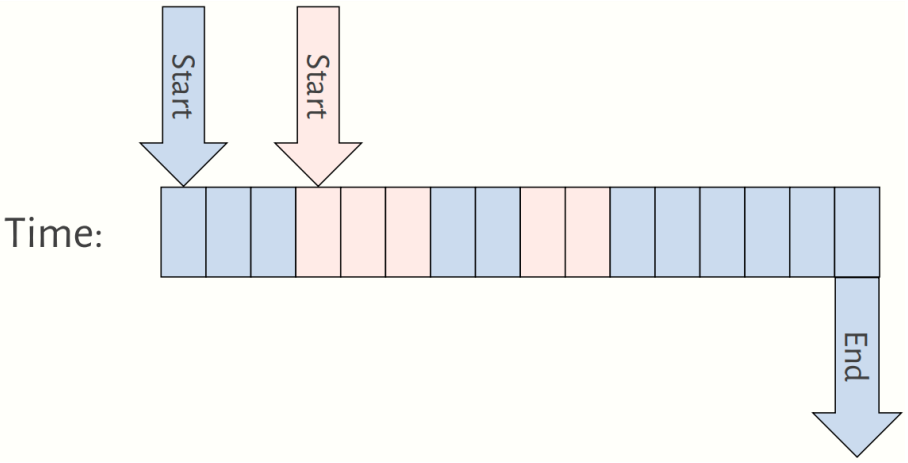
Listing 23: Lock-free algorithm to increment a counter

```
1  void inc(int counter)
2  {
3    int val = counter;
4    while(!CAS(val, &counter, val+1))
5    {
6      val = counter;
7    }
8  }
```

This is not wait-free, as there is no guarantee that any particular thread will succeed. However, it is also not locking, because there is no critical section where one thread could block and prevent any other thread from accessing a shared lock.

7.2.3 Obstruction-freedom

Obstruction-freedom is the weakest natural non-blocking progress guarantee. An algorithm is obstruction-free if at any point, a single thread executed in isolation for a bounded number of steps will complete its operation. In other words, a thread will finish its own operation if it runs in isolation. All lock-free algorithms are obstruction-free.



Obstruction-free algorithms only ensure that none of the low-level steps leave a data structure broken. It does this by demanding that any partially completed operation can be aborted and the changes made rolled back. A **contention manager** is used to reduce the likelihood of live-lock.

Listing 24: Obstruction-free algorithm to increment a counter

```
1  void inc(int counter)
2  {
3    while (true)
4    {
5      int val = LL(counter); //LL - weak load-linked
6      if (SC(counter, val+1)) //SC - store-conditional
7      {
8        return;
9      }
10   }
11 }
```

Because this algorithm requires two atomic steps (LL then SC), the LL operation on one thread will prevent an SC on another thread from succeeding.

7.3 The ABA problem

The ABA problem is a problem that occurs when memory deallocation and reclamation happens. THe main issue is when a location is read twice, has the same value for both reads where the fact that the *value is the same* is used to indicate that nothing has change. However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking that nothing has changed, even though the work done by the second thread violates that assumption. The sequence of events is generally as follows:

- Process P_1 reads value A from shared memory
- P_1 is pre-empted, allowing process P_2 to run
- P_2 modifies the shared memory value A to value B and back to A before preemption
- P_1 begins execution again, sees that the shared memory value has not changed and continues

Although P_1 can continue executing, it is possible that the behaviour will not be correct due to any hidden modifications in shared memory.

A common case of the ABA problem is encountered when implementing a lock-free data structure.

If an item is removed from the list, deleted and then a new item is allocated and added to the list, it is common for the allocated object to be at the same location as the deleted object. A pointer to the new item is therefore sometimes equal to a pointer to the old item even though the value has changes, which is an ABA problem.

7.3.1 Solutions against the ABA problem

- Add a tag/stamp to the pointer. For example, an algorithm using CAS on a pointer might use the low bits of the address to indicate how many times the pointer has been successfully modified. Because of this, the next CAS will fail even if the pointer addresses are the same because the tag bits will not match.
- Extra guards during deallocation/reallocation
- Hardware support

8 Memory consistency models

8.1 Sequential consistency