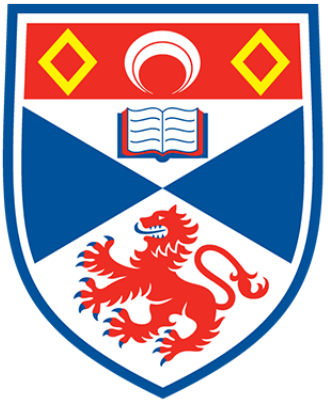


University of
St Andrews

CS4204 Concurreny and Multicore Architectures

Revision Notes

May 4, 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.

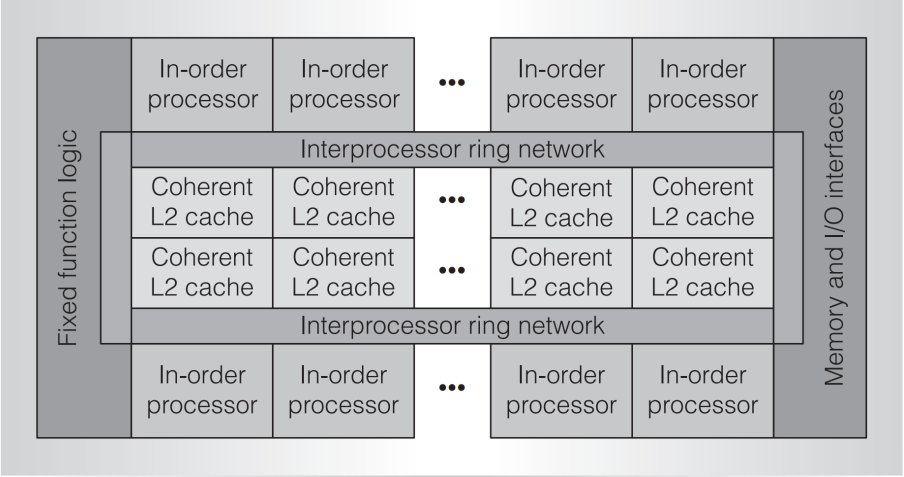


Figure 1: Schematic of the Larrabee manycore architecture.

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.

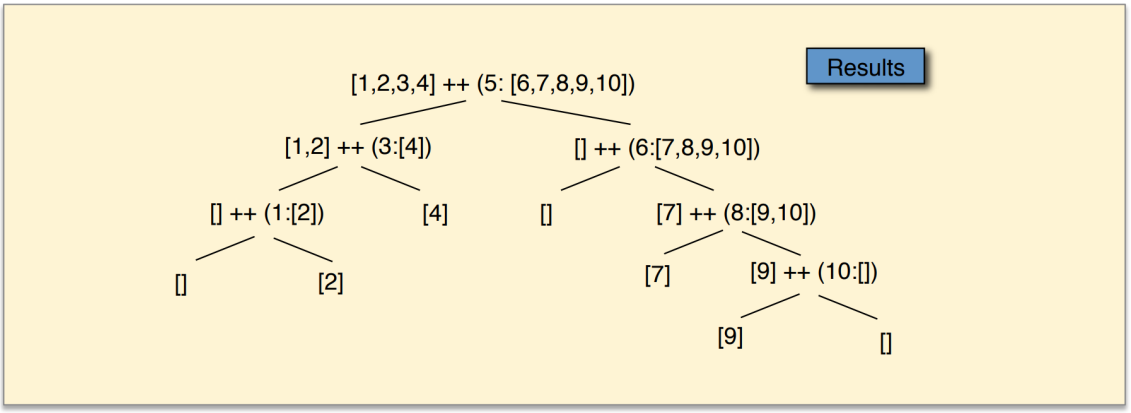


Figure 2: Quicksort using divide and conquer to the trivial case of one element.

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

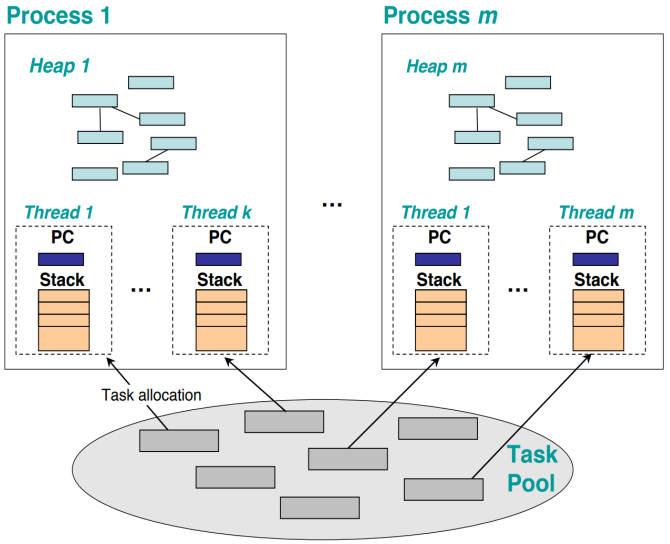


Figure 3: Diagram of how parallel processes are split up into threads and tasks.

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.

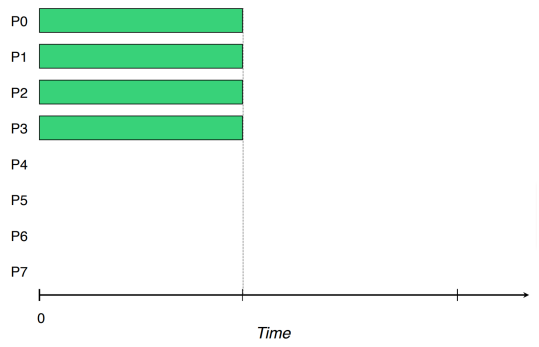
Thread - The basic unit of parallel computation. It is lightweight and shares address space with other threads in the same process, but has its own separate stack and program counter.

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.

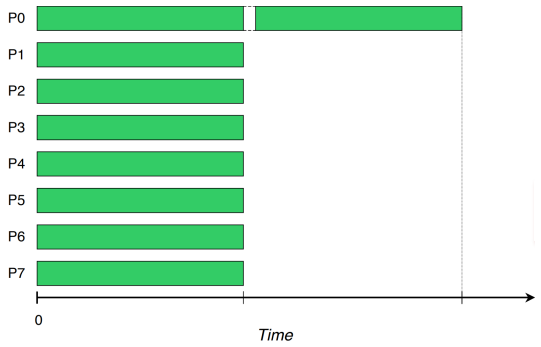
2.2 Granularity

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.

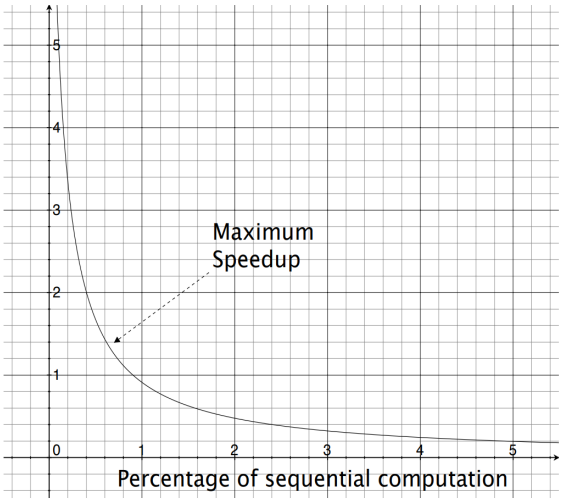


Figure 5: The effect of Amdahl's law on parallel computation. The y-axis shows the maximum speedup that can be achieved if the problem size remains fixed.

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.

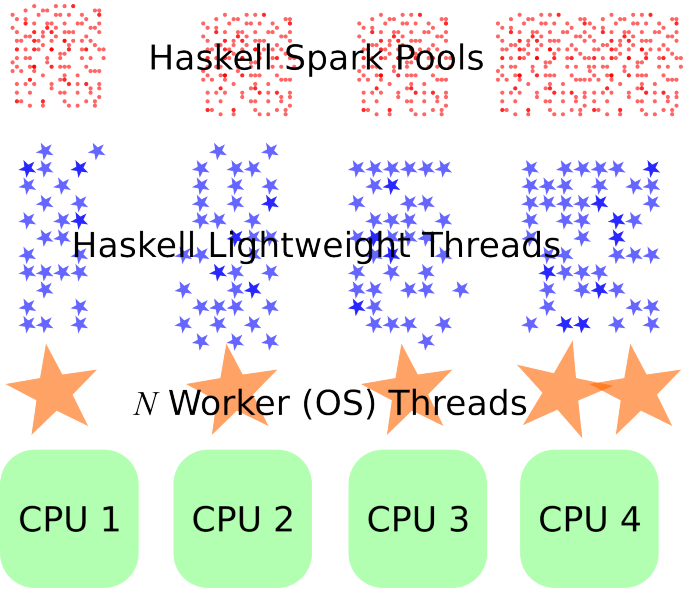


Figure 6: Visualised difference between Haskell threads and spark pools and how they are mapped to OS threads.

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**.

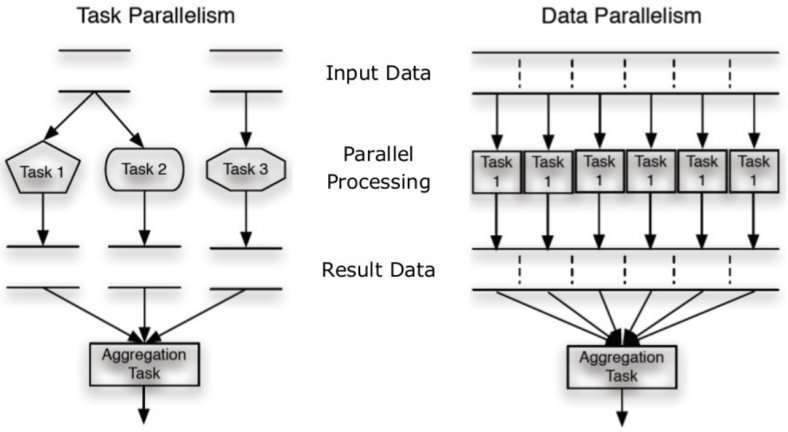


Figure 7: Difference between task and data parallelism.

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

3.2 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

3.3 Data parallel patterns

3.3.1 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.

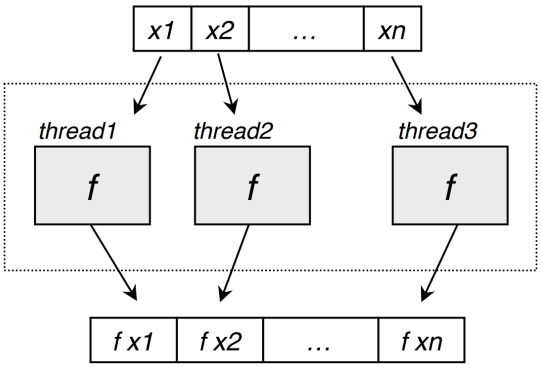


Figure 8: Parallel maps map a function across each element in parallel.

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

3.3.2 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.

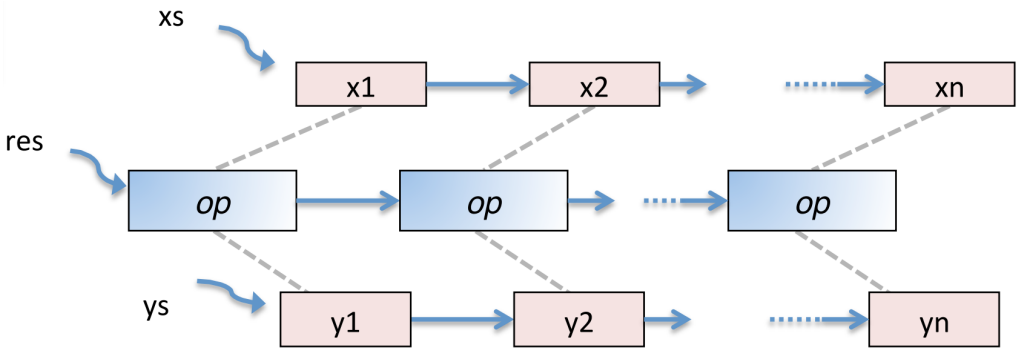


Figure 9: Parallel `zipWith` over two input lists `xs` and `ys`

3.3.3 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
```

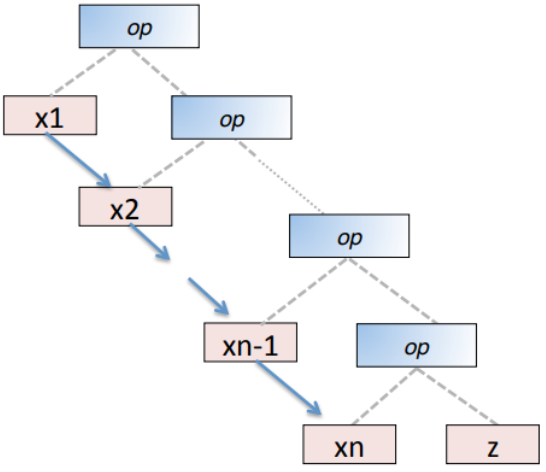


Figure 10: Sequential right fold.

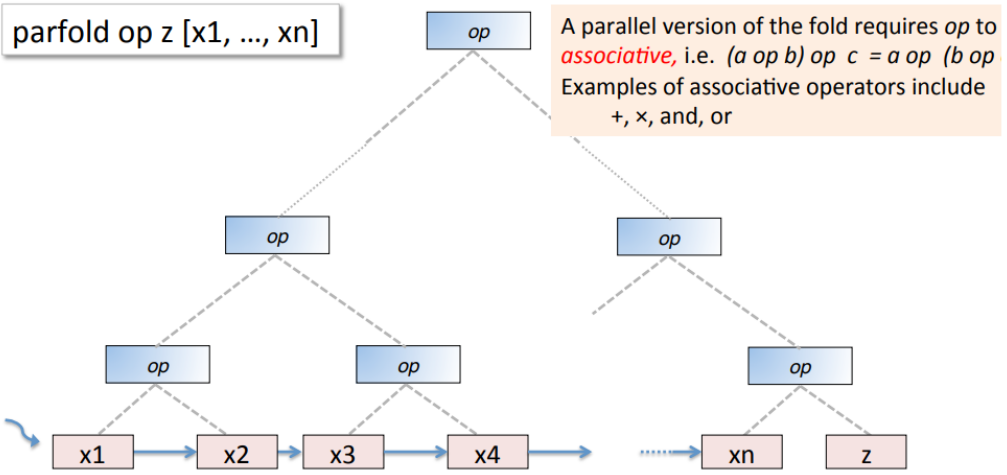


Figure 11: Parallel fold.

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.

3.3.4 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.

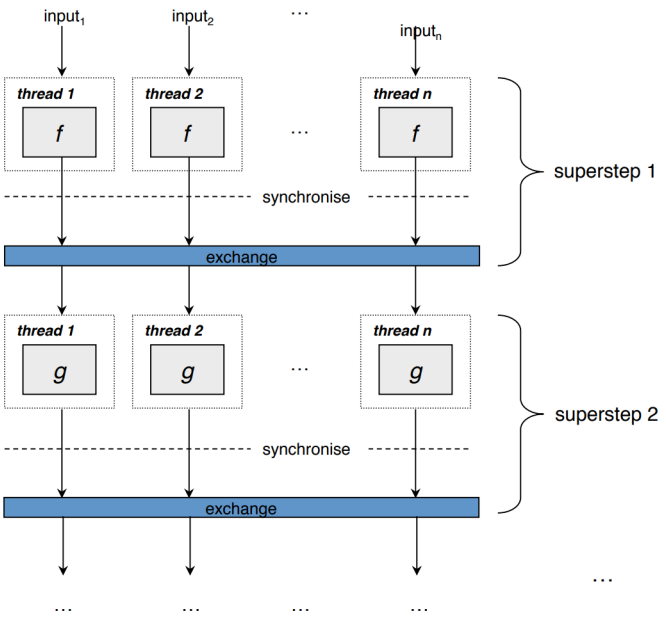


Figure 12: The bulk synchronous parallelism process.

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.

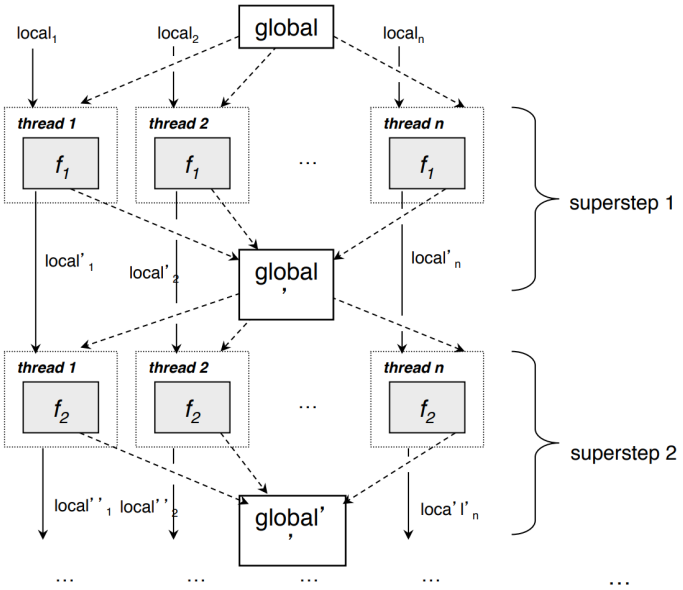


Figure 13: BSP local and global data exchange process.

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.

3.3.5 Parallel map-reduce