1 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**.

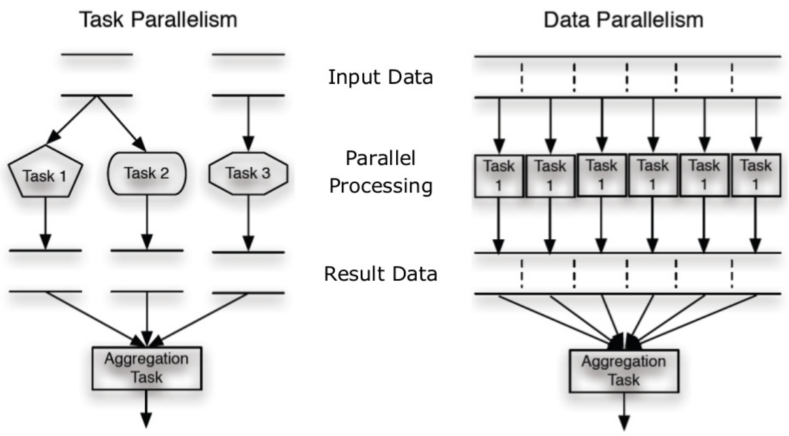

Figure 1: Difference between task and data parallelism.

1.1 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

1.2 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

1.3 Data parallel patterns

1.3.1 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.

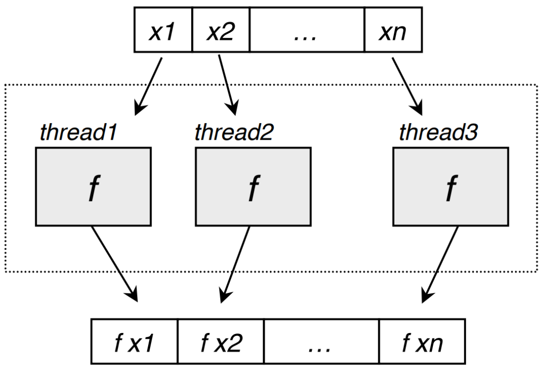

Figure 2: Parallel maps map a function across each element in parallel.

Listing 1: Parallel map implementation

```haskell
parmap :: (a -> b) -> [a] -> [b]
parmap f [] = []
parmap f (x:xs) = let fx = f x in
                  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

1.3.2 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.

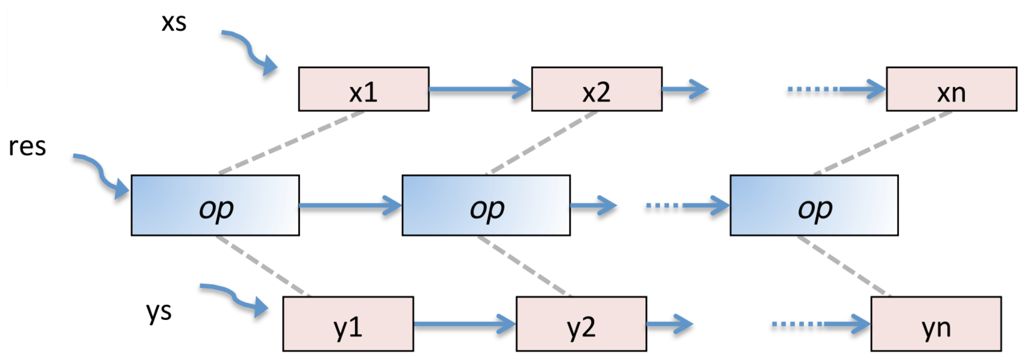

Figure 3: Parallel `zipWith` over two input lists `xs` and `ys`

1.3.3 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 2: Type signature of a parallel fold.

```haskell
parFold :: (a -> a -> a) -> a -> [a] -> a
parFold f z l = ...

-- Examples of fold uses
sum = parFold (+) 0
product = parFold (*) 1
```

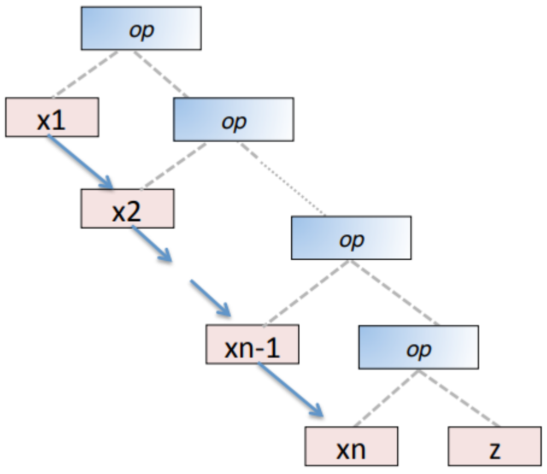

Figure 4: Sequential right fold.

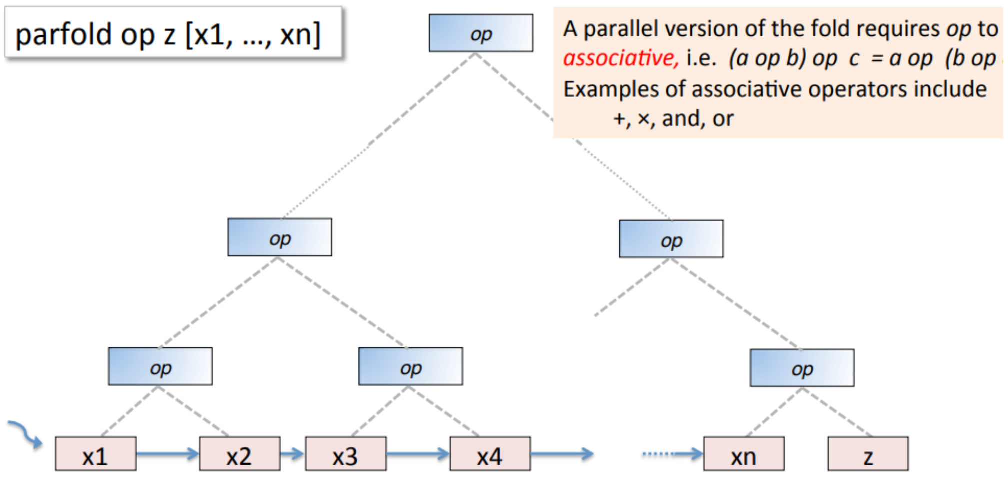
parfold op z [x1, ..., xn]
op
A parallel version of the fold requires op to associative, i.e. (a op b) op c = a op (b op
Examples of associative operators include
+, ×, and, or
op
op
op
op
op
x1
x2
x3
x4
xn
z

Figure 5: 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.

1.3.4 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.

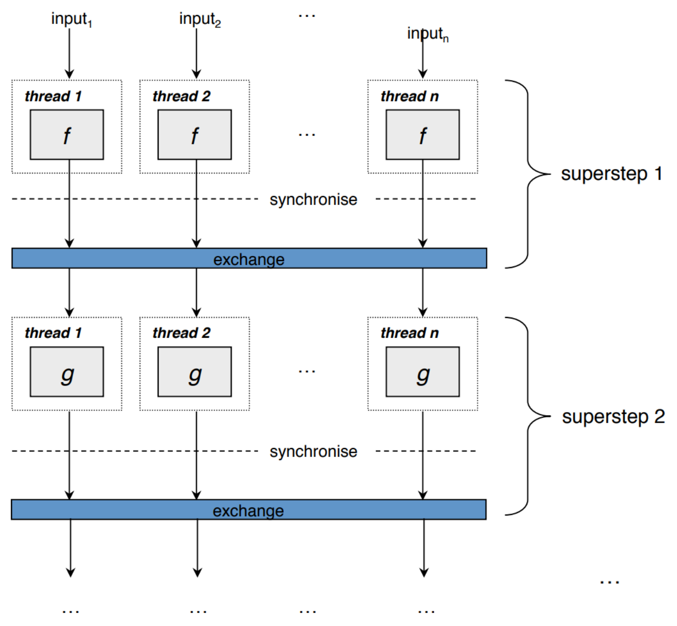

Figure 6: The bulk synchronous parallelism process.

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.

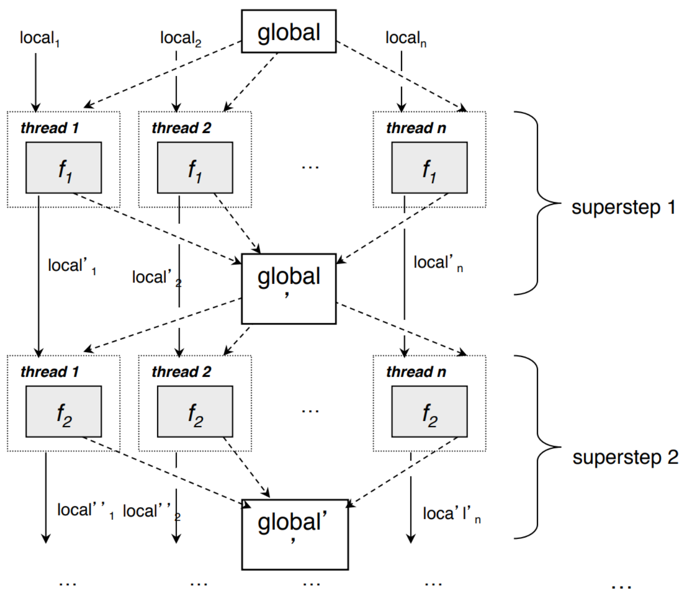

Figure 7: BSP local and global data exchange process.

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

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

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.

1.3.5 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.

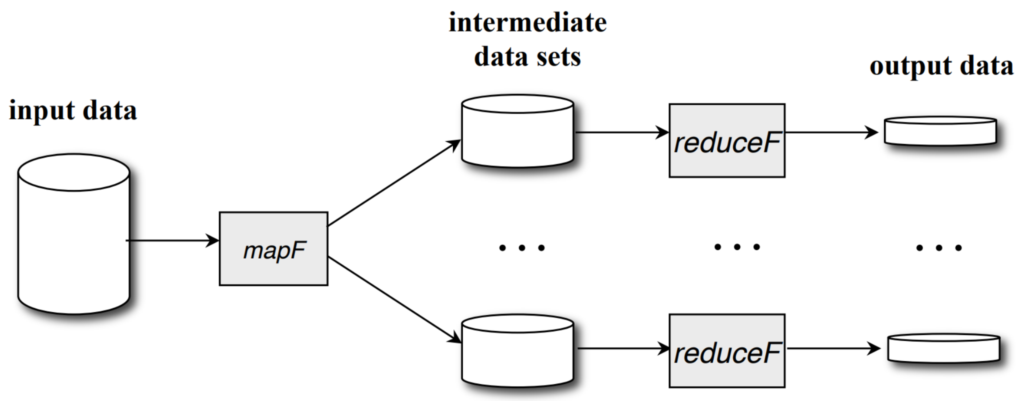

Figure 8: Map-reduce with parallel reduction step.

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.

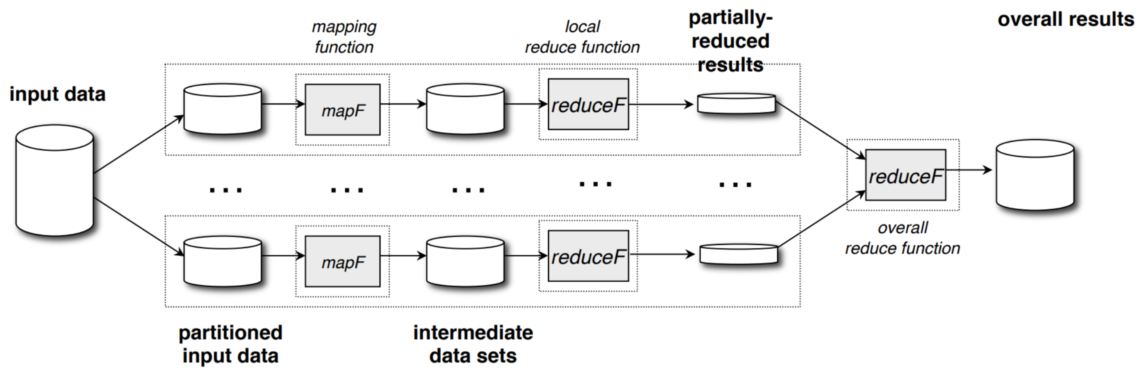

Figure 9: Map-reduce with parallel mapping step.