Chapter 4

PARALLEL ALGORITHMS

The purpose of this chapter is to provide a collection of parallel algorithms that form the building blocks for many parallel applications. A secondary objective is to give examples of the pod comprehension notation, and where possible, explore the use of non-strictness in data-parallel algorithms. Twelve higher-order parallel algorithms are defined: ScanPod, FoldPod, LinearRecur, SuffixScanPod, PackPod, FilterPod, HistogramPod, GradeUpPod, SortIntPod, SegmentScanPod, SegmentFoldPod, and SendWithPod. The scan algorithm forms the heart of all the parallel algorithms in this chapter.

4.1 Deriving a parallel scan function

Scan is widely accepted as a fundamental technique of a data-parallel paradigm. A step-by-step guide to the parallel implementation of scan is presented. By concentrating on the associated problem of evaluating the partial sums of the series \([x_0; x_1; \ldots; x_n]\) defined by equation (4.1), the relationship between parallelism and non-strictness is explored. Unlike existing implementations of scan, a non-strict version presented here, has the potential to scan an infinite pod if Ladner’s requirement of a fixed length input for a parallel scan no longer applies.
Chapter 4. PARALLEL ALGORITHMS

4.1.1 Sequential: the partial sums of a list

Since the partial sums function computes a series of numbers, it is natural to represent this series as a list. By reading equation (4.1) denotationally we can inductively define the partial sums function over a non-empty list as: the partial sums of the singleton list is the singleton list; the k-th partial sum of the list \([x_0; \ldots; x_k; \ldots; x_n]\) is equal to the k-th element added to the sum of the preceding \(k-1\) elements; i.e.,

\[
\text{partialSums}([x_0; x_1; \ldots; x_n]) = [x_0; \Sigma_{i=0}^{1} x_i; \Sigma_{i=0}^{2} x_i; \ldots; \Sigma_{i=0}^{n} x_i].
\]

\[ (4.1) \]

\[
\text{partialSums}(x:xs) = \text{psums}(a \cdot \text{psums}(\text{acc} + x) \cdot xs)
\]

The accumulating parameter of the auxiliary function \(\text{psums}\) is used to represent the partial sum of the preceding \(k-1\) elements of the list. The function’s complexity is \(O(N)\) for a list of length \(N\).

4.1.2 Parallel: a first attempt

Using a parallel implementation of graph reduction such as that offered by the GRIP[31] machine, there are opportunities for parallelism in an application of \(\text{partialSums}\). Yet if the entire list of results are needed from such an application, the
Chapter 4. PARALLEL ALGORITHMS

complexity of the partial Sums function can be no better than \( O(N) \). One reason for this bound on complexity is the sequential ordering of solutions caused by the \( k \) th element of the solution depending on all preceding elements. The first step towards a truly parallel implementation is attained by breaking this dependence.

---

A similar problem to that of generating the partial sums is shown in figure 4.1(a), where the sum of a one-dimensional pod of numbers (the circles represent processors) is achieved by a divide and conquer technique. The dependence between consecutive elements of the pod has been broken by utilising the associativity of addition. If we were implementing this divide and conquer algorithm on a data-parallel machine, we would impose a structure on the combination of processors such that all the combinations at each level of the evaluation tree shown in figure 4.1(a) occur synchronously, and in parallel.
Chapter 4. PARALLEL ALGORITHMS

Such a structure is enforced in the divide and conquer algorithm shown in figure 4.1(b). As would be expected, the last element of the vector contains the sum of the vector, yet the first two processors also contain their partial sums. The key to producing a parallel partial sums function is to utilise all of the processors during each iteration of the divide and conquer algorithm.

Figure 4.1(c) shows such an algorithm which uses the parallel prefix technique[1 1,66]. We can generalise from this figure that during the k th iteration of the algorithm (where k ≥ 1) a subset of the processing elements combine with a processor $2^k$ places to their left. If vec is the state of the vector during the start of each iteration, then we can express such a combination by the pod comprehension:

$$<< (x; b+a)| (x;a) << vec, |$$

$$(|x - 2''(k-1); b|) <= vec >>$$

Next, if lo is the processor identifier of the first element of the vector, then the processors in the range $-\infty$ to $lo+ 2^k$ do not combine with any processor.

This range of processors can be represented by the boolean vector:

$$<< |x; x < lo + 2^k | | (|x ; - |) <= vec >>$$

Finally, we define the parallel partial sums function by merging the above expressions to form a single pod comprehension that represents the parallel combining process, and iterate this process a logarithmic number of times. The algorithms complexity is $O(\log N)$ for an input of size $N$.

>parPartialSums :: (Pid a, Ord a, Num a, Num b) => a -> a -> <<(a;b)>> -> <<(a;b)>>
Chapter 4. PARALLEL ALGORITHMS

definition is tail recursive), into a fold type definition in which the results are available as soon as possible.

> parPartialSums' :: (Pid a, Ord a, Num a, Num b) => a -> [[a; b]] -> [[a; b]]
> parPartialSums' lo vector = eachIter' 1 vector
>
> where eachIter' k vec = let vec' = [[x; b+a]
> | ((x; a)) <= vec,
> | ((x - 2^((k-1)); b)) <= vec>>
> in [[x; if x < lo+2^((k-1)) then a else b]
> | ((x; a)) <= vec,
> | ((x; b)) <= eachIter' (k+1) vec>>

Notice how the lazy version of parPartialSums' above has no upper bound (i.e no hi as an argument). The computation potentially continues indefinitely, summing from a fixed point within a pod (represented by lo) up to 1. Because of the non-strict nature of DPHaskell’s evaluation mechanism, and the non-strictness associated with index-generators, a recurrence relation such as that shown in the lower pod comprehension can be expressed. If we want to evaluate the processors between 1 and n, then only Log 2 (n) iterations of each Ietter’ need to be evaluated before the processors in the range 1 and n satisfy the condition in the if expression. No more evaluation is required because the results are in a normal form and the wave of evaluation has passed over them.
Chapter 4. PARALLEL ALGORITHMS

4.1.4 Higher order glue and fold

Examining the definition of the sequential and parallel partial sums function, we see that only `+` is specific to the computation. The function can therefore be parameterised on the function applied at each iteration of the scan, producing scan11 as defined in the Haskell report and scanPod.

> scanPod::(Ord a, Num a, Pid a) => (b->b->b) -> a -> ((a;b)) -> ((a;b))

> scanPod fn lo vector = eachIter 1 vector

> where eachIter k vec = let vec' =<< ((x; fn b a

> | | (x; al) <<- vec,

> | (x - 2^(k-1); b)) <<= vec >>

> in <<= (x; if x < lo + 2^(k-1) then a else b

> | | (x; al) <<- vec,

> | (x; bl) <<= eachIter (k+1) vec')

Unlike other implementations of scan in data-parallel programming languages, the definition here has the potential to scan an infinite pod (this is more general than the definition of parallel scan proposed by Ladner [49]).

Having presented a scan algorithm, its definition is re-used in an implementation of fold.

To calculate the result of a parallel fold between two indices lo and hi of a pod, a parallel
Chapter 4. PARALLEL ALGORITHMS

scan is applied from lo, and the resulting pod is indexed at processor hi. This technique is similar to the program identity foldl (θ) x = last ffl scanl (θ) x, and can be used as the basis of foldPod.

>foldPod::(Ord a, Num a, Pid a) => (b -> b -> b) -> a -> a -> <(a,b)>> -> b

>foldPod fn lo hi vec = indexPod (scanPod fn lo vec) hi

Although the definition of fold has a theoretical basis in terms of the above program identity, it is not obvious that indexing processor hi of the scanned pod ensures the parallel evaluation of that processor's contents. To verify that the fold really is parallel, the fold's operational characteristics are investigated. In keeping with non-strict languages, the order of evaluation can be rather confusing. Operationally, evaluation of indexPod proceeds by: (1) setting the aim at the processor to be indexed; (2) the vector argument of the index is evaluated with that aim; (3) once the evaluation is complete, the indexed processors contents will be returned. It would be expected that the singleton aim produced by indexing a pod would produce no parallelism as only one processor has its evaluation forced. Surprisingly, the evaluation of a single processor can cause an avalanche of parallel evaluation, and follows from the interplay of the aim and the communication primitives of DPHaskell.
Chapter 4. PARALLEL ALGORITHMS

Considering the evaluation of the pod \( <(1,1), (2,2), (3,3), (4,4)> \),

by the definition of scanPod the tail recursive function will perform \( 1 + \log N \) iterations without actually evaluating the vector to be folded. During the last iteration, the result from the drawn-from generator \( (x;a)<<\text{vec} \) will be required, and vec will be evaluated. As a consequence of the index used in the fold, the aim will be set to processor four, and because of the data-dependencies inherent in the scan, the processor identified by the binding \( b \) at position \( x \) the aims required to produce the result of the fold in processor four because of the data-dependencies of scan. This fan-out is shown in figure 4.2 where the circles represent processors, and the squares represent those processors with a defined aim.

This tree differs from the one normally associated with a fold, as it only defines those processors which need to be evaluated. Evaluation with the aim is akin to a data-flow model of parallel computation where the data-dependencies required to evaluate the solution (i.e., processor four) dictate a data-flow network defined by the aim at successive iterations of the tail recursion. Once the aim is initialised, then evaluation of the function...
Chapter 4: PARALLEL ALGORITHMS

fn (+ in the example) at each iteration of the scan occurs to produce the evaluation tree shown in figure 4.3.

In summary, the definition of fold uses a scan algorithm that has a different operational nature than a fold (contrast figure 4.1(a) and 4.1(b)). However the aim is used to create just those characteristics from a general scan function in the creation of a parallel fold.

One problem with this definition of foldPod is that the overhead associated with the initialisation of the aim is always incurred, even when the extra non-strictness is not required. For example when folding a strict operator such as integer addition across a vector, the overhead of initialising the aims for non-strict evaluation is superfluous. This can be remedied by performing speculative evaluation on all those elements in the range of the pod being folded. The user can control this speculation by setting the aim to those processors between the indices to be folded. The strictPod function achieves this, a strict version of the foldPod function can now be defined as:

> strictFoldPod::(Ord a,Num a,Pid a)=> (b -> b -> b) -> a -> a -> <\((a;b)\)> -> b

> strictFoldPod fn lo hi vec

> = indexPod (strictPod lo hi (scanPod fn lo vec)) hi

This example exposes an alternative way of thinking about the aim mechanism. It can be thought of as a runtime strictness technique which gives the user the control of how much, and where parallel evaluation occurs. With the two definitions of fold, parallel analogues to Haskell’s list processing functions can now be defined.
**Chapter 4. PARALLEL ALGORITHMS**

\[ \text{sumPod, productPod :: (Ord } a, \text{Num } a, \text{Pid } a, \text{Num } b) \mapsto a \rightarrow a \rightarrow ((a; b)) \mapsto b \]

\[ \text{sumPod = strictFoldPod (+)} \]

\[ \text{productPod = strictFoldPod (*)} \]

\[ \text{maximumPod, minimumPod :: (Ord } a, \text{Num } a, \text{Pid } a, \text{Ord } b) \mapsto a \rightarrow a \rightarrow ((a; b)) \mapsto b \]

\[ \text{maximumPod = foldPod max} \]

\[ \text{minimumPod = foldPod min} \]

\[ \text{andPod, orPod :: (Ord } a, \text{Num } a, \text{Pid } a) \mapsto a \rightarrow a \rightarrow ((a; Bool)) \mapsto \text{Bool} \]

\[ \text{andPod = foldPod (&&)} \]

\[ \text{orPod = foldPod (||)} \]

\[ \text{anyPod, allPod :: (Ord } a, \text{Num } a, \text{Pid } a) \mapsto (b \rightarrow \text{Bool}) \rightarrow a \rightarrow a \rightarrow ((a; b)) \rightarrow \text{Bool} \]

\[ \text{anyPod fn low-pid high-pid} = \text{orPod low-pid high-pid} \cdot \text{mapPod fn} \]

\[ \text{allPod fn low-pid high-pid} = \text{andPod low-pid high-pid} \cdot \text{mapPod fn} \]

\[ \text{elemPod, notElemPod :: (Ord } a, \text{Num } a, \text{Pid } a, \text{Eq } b) \mapsto b \rightarrow a \rightarrow a \rightarrow ((a; b)) \rightarrow \text{Bool} \]

\[ \text{elemPod = anyPod . (==)} \]

\[ \text{notElemPod = allPod . (/=)} \]

**4.2 First order linear recurrence**

The introduction to this thesis concentrated on the lack of sequencing in some algorithms which enables data-parallel techniques to be utilised. The scan algorithm of the previous section was parallelised by using the algebraic property of associativity. This gave rise to patterns of grouping sequences of operations in such a way as to take...
advantage of data-parallelism. In this section we introduce a parallel implementation of a linear recurrence algorithm that utilises algebraic properties other than associativity. A secondary goal of this example is to show how two nested serial loops can be flattened into consecutive scan and fold operations. This technique provides a parallel implementation of nested loops without resorting to the nested parallelism offered by Blelloch's language NESL.

The first-order linear recurrence algorithm of equation 4.2 forms the heart of many scientific and numerical applications in areas as diverse as financial modelling and the solution of differential equations used in weather modelling. In these application areas, the linear recurrence often forms the computationally intense part of an algorithm, and as a consequence, many researchers have investigated parallel techniques for vector and multi-processor machines

\[ x_0 = b_0 \]
\[ x_i = (a_i \otimes x_{i-1}) \oplus b_i \]

Unlike conventional implementations that calculates all the values \( x_i \) where \( 1 \leq i \leq n \), we concentrate on calculating the single value of \( x_n \) at iteration \( n \) of the recurrence. The example is used as an introduction to suffix-scans, which are based upon the prefix-scan algorithm of the previous section. In contrast to scanning an associative operator from left-to-right across a sequence of values in the prefix-scan algorithm, a suffix-scan starts from the right and works leftwards. For example,
Chapter 4. PARALLEL ALGORITHMS

\[
\text{scan (+)} \{1; 2; 3; 4\}
\Rightarrow \{1; 3; 6; 10\}
\]
\[
\text{suffixScan (+)} \{1; 2; 3; 4\}
\Rightarrow \{10; 9; 7; 4\}
\]

The first step towards a parallel interpretation of equation 4.2 is achieved by investigating the patterns that emerge when the equation is expanded. For example, expanding four iterations of the recurrence gives:

\[
x_0 = b_0
\]
\[
x_1 = (a_1 \otimes b_0) \oplus b_1
\]
\[
x_2 = (a_2 \otimes ((a_1 \otimes b_0) \oplus b_1)) \oplus b_2
\]
\[
x_3 = (a_3 ((a_4 ((a_3 a_1 b_0) \oplus b_1)) \oplus b_2)) \oplus b_3
\]

\[
\ldots \ldots (4.3)
\]

Now if both \( \otimes \) and \( \oplus \) are associative, and \( \otimes \) distributes over \( \oplus \), then the expansions of 4.3 can be regrouped as:

\[
x_0 = b_0
\]
\[
x_1 = (a_1 \otimes b_0) \oplus b_1
\]
\[
x_2 = (a_2 \otimes a_1 \otimes b_0) \oplus (a_2 \otimes b_1) \oplus b_2
\]
\[
x_3 = (a_3 \otimes a_2 \otimes a_1 b_0) \oplus (a_0 \otimes a_2 \otimes b_1) \oplus (a_3 \otimes b_2) \oplus b_3
\]

\[
\ldots \ldots (4.4)
\]

iff \( x \oplus (y \oplus z) = (x \oplus y) \oplus z \) associativity of \( \oplus \)

\( x \otimes (y \otimes z) = (x \otimes y) \otimes z \) associativity of \( \otimes \)
Chapter 4. PARALLEL ALGORITHMS

\[ x \otimes (y \otimes z) = (x \otimes y) \otimes (x \otimes z) \] distributivity of \( \otimes \) over \( \oplus \)

We can now see that a pattern emerges from the expansions, such that monolithic summation and product functions can be used to re-express the algorithm. If \( \Sigma_{j=1}^i \) is used as a summation over values ranging from 1 up-to \( i \), using \( \oplus \) as the operator for the summation; and similarly, \( \Pi \) is used as a product from \( i \) down-to \( j \), Using \( \otimes \) as the product operator, then equation 4.5 can be re-expressed as:

\[ x_0 = b_0 \]
\[ x_i = ( \Sigma_{j=1}^i (\Pi_{k=0}^i a_{i,k}) \otimes b_{i,j}) \oplus b_i \]

The equation now has a form that is amenable to a data-parallel interpretation, as the summation and product can be implemented by the folds and scans of DPHaskell.

\[ \text{linearRecur :: (Ord a, Num a, Pid a) => (b->b->b) -> (b->b->b) -> a -> a ->} \]
\[ \text{<<(|a;b|)>> .> } \text{<<(|a;b|)>> .> b} \]
\[ \text{linearRecur otimes oplus lo hi vecA vecB} \]
\[ \text{= oplus} \]
\[ \text{= (foldPod (oplus) lo hi} \]
\[ \text{= (zipWithPod (otimes)} \]
\[ \text{= (suffixScanPod (flip (otimes)) lo vecA} \]
\[ \text{= vecB))} \]
\[ \text{= (indexPod vecB hi)} \]
Chapter 4. PARALLEL ALGORITHMS

The algorithm contains four major components: (1) a suffix-scan of $\otimes$ with a $O(\log N)$ complexity in relationship to $N$ expansions of the linear recurrence; (2) a $O(1)$ complexity pairwise application of $\otimes$ with each of the values from the scanned pod and vecB; (3) a final $O(\log N)$ complexity fold of $\otimes$. As each of these components are applied one after another, the complexity of the entire algorithm is $O(\log N)$ compared to $O(N^2)$ of a naive sequential encoding of equation 4.2. The definition of the suffixScanPod higher-order function is shown below. Notice how like scanPod, hi represents a position where scanning commences, and continues potentially indefinitely towards minus infinity.

```haskell
> suffixScanPod :: (Ord a, Num a, Pid a) => (b -> b -> b) -> a -> ((a; b)) -> (a; b)
> suffixScanPod fn hi vector = eachIter 1 vector
> where eachIter k vec = let vec' = ((x; fn a b)
>   | (x; a) < vec,
>   > (x + 2^"(k-1); b) <= vec >
>   > in ((x; if x > hi+2^"(k-1) then a else b)
> > | (x; a) <= vec,
>   > (x; b) <= eachIter (k+1) vec '>
```

The following program identities define relationships between prefix-scan, suffix-scan, fold, and the recurrence function:

```
foldPod (\) lo hi = linearRecur (\x y -> y) (\) lo hi
```

Identity 1: "scan-

recurrence"
Chapter 4. PARALLEL ALGORITHMS

\[ \lambda \text{vec} \rightarrow \text{indexPod} (\text{scanPod} (\oplus) \text{lo vec}) (\text{lo} + i) \]
\[ \equiv \lambda \text{vec} \rightarrow \text{indexPod} (\text{suffixScanPod} (\oplus) \text{lo vec}) (i - \text{lo}) \]  
Identity 2: "suffix-prefix scan" 

iff \( \oplus \) is commutative.

4.3 Packing

Packing is an important technique that is used in many algorithms. The packing algorithm defined here is based upon the packing intrinsic of Fortran 90, and is generalised to take advantage of non-strictness.

The specification of the Fortran 90 pack can be given in terms of a Haskell function over two lists:

\[
\text{packF90} :: [\text{Bool}] \rightarrow [\text{a}] \rightarrow [\text{a}]
\]
\[
\text{packF90} [J] = [I]
\]
\[
\text{packF90} [I] = [I]
\]
\[
\text{packF90} (m:ms) (x:xs) | m = x : \text{packF90} ms xs
\]
\[
\text{otherwise} = \text{packF90} ms xs
\]

The idea is that given a mask and a vector, those elements of the vector that are defined to be true at a corresponding location in the mask are packed, item-by-item, towards the left-hand side of the vector. In DPHaskell this packing towards the start of the vector poses a problem as pods have neither a start or an end.
Chapter 4. PARALLEL ALGORITHMS

If the point-of-interest is at processor one, and the mask is converted into a integer pod such that wherever there is a true we put a one, and false a zero, then scanning (+) across this integer pod produces an index-pod that identifies where each element should be communicated too. As this is only sufficient for elements to the right of the point-of-interest, a suffix-scan is used to define those elements to the left. A complete algorithm expressed in DPHaskell is shown below:

> packPod :: Pid a => a -> ((a;Bool)) -> ((a;Bool)) -> ((a;Bool))
> packPod poi mask data
> = << if x => poi
>     then ((sent—right—poi + poi-1; y)
>     else ((-sent—left—poi + poi; y)
>     | ((x;True l) <<= mask,
>     (x;  y) <<= data,
>     (x;sent—right—poi) <<= right—scanned,
>     (x;sent—left—poi) <<= left—scanned >>
>     where
>     right—scanned = scanPod (+) poi (mapPod boolToInt mask)
>     left—scanned = suffixScanPod (+) (poi-1) (mapPod boolToInt mask)
>
> boolToInt :: Bool -> Int
> boolToInt True = 1
> boolToInt False = 0
Chapter 4. PARALLEL ALGORITHMS

The complexity of this algorithm depends upon the complexity of the communication primitives of DPHaskell. A SIMD machine such as the CPP DAP can only perform regular shifts, and a general communication is performed by a $O(N)$ send in software. The effect of this is that the pack algorithm has an $O(N)$ on the DAP; however this is by no means the entire story. If the communication involved in the pack is decomposed into two phases, one for the communication of all the processors to the left of the point of interest, and another for the right, then the communication can be considered to be semi-regular as each processor to the right of the point-of-interest sends data to a processor directly to the left of itself, with this behaviour mirrored on the opposite side of the point-of-interest.

One way of implementing this regular communication is to take the index-pod used in the send, and shift those elements that have an odd index-value one place to the left, along with the index-value. Next those elements which are divisible by two have their index-value and data shifted two places to the left, and this process is iterated with growing powers of two. Shifting communication has a $O(1)$ complexity, and the number of iterations of this shifting communication is bounded on the byte-size of an integer. The communication is therefore $O(1)$, and the packing routine for the DAP therefore has the same $O(\log N)$ complexity as that for the Connection Machine without the need for specialised hardware communication networks.
Chapter 4. PARALLEL ALGORITHMS

Instead of encoding this "bit-twiddling" into the packing algorithm here (and losing the elegance of the DPHaskell function), the technique would be better utilised by incorporating it as a special case of send communication used by the data-parallel abstract machine. This special case would take effect when the index-pod of a send communication is monotonically ascending or descending, which can be tested in $O(1)$ by shifting the index-pod one place to the left, performing a comparison, and then folding logical "and" across the resulting mask. On the DAP, the fold of "and" will have a $O(1)$ because of the hardware broadcast communication network of the machine.

The packing algorithm here has the feel of a conventional data-parallel algorithm and isn't really in the spirit of functional programming. The problem concerns the mask used to identify elements to be packed, which wouldn't normally be used in a functional algorithm. Using the lower level packPod function, we can define filterPod:

```haskell
> filterPod :: Pid a -> (b -> Bool) -> a -> (a,b) -> (a,b)
> filterPod fn poi data = packPod poi (mapPod fn data) data
```

This function selects those elements from a pod that satisfy the higher-order predicate, and is analogous to the Haskell list filtering function filter.

4.4 Radix sorting

In this section we introduce two implementations of a radix-sorting[80,11] algorithm.
Chapter 4. PARALLEL ALGORITHMS

The core of the algorithm is then re-implemented in a functional style, which can be viewed as an optimisation. However, this optimisation cannot easily be incorporated into the Fortran 90 routine because of the dynamic nature of the parallel data-structures used. The example provides an interesting case-study as the radix-sorting algorithm involves a considerable amount of low level "bit twiddling" that functional programmers don't usually have to resort too. Instead of choosing a more naturally functional sorting algorithm, such as quick-sort, we explore the ease in which techniques currently used when expressing data-parallel algorithms carry through to DPHaskell.

4.4.1 The radix sorting algorithm

Given the problem of sorting a pod of integers, a sorting algorithm can be decomposed into two parts: (1) Each of the integers in the pod to be sorted is chunked into smaller parts (i.e., a 32-bit integer can be viewed as two 16-bit integers). If each integer is chunked into m parts, then m pods are created where each pod element contains corresponding chunks from the elements of the pod to be sorted; (2) These m pods are used to calculate m index-pods that if applied with the original unsorted data to a send communication, would sort the data.
Chapter 4. PARALLEL ALGORITHMS

The justification for splitting the unsorted data into pods that contain chunks with a smaller numerical range, is that an $O(N)$ sorting algorithm can be used, where $N$ is proportional to the range of values to be sorted. For example, figure 4.7 shows the steps involved in sorting five integers, where each digit of the number to be sorted represents a distinct chunk. The problem to be addressed with the $O(N)$ sort is the calculation of an index-pod which can be used in a send communication (such an index-pod is represented by the arrows in figure 4.4. One way of generating this index-pod is to calculate the number of elements that directly precede each element of the pod if sorting had taken place. As pods are potentially infinite, and sorting isn't lazy, then sorting is performed within a finite region of a pod. If the lower bound of this region is zero, then the calculated index-pod can be used by a send communication to rearrange the data. For example, using the sequence $[2; 9; 7; 9; 5]$:

$$<< (j0; 2j); (j1; 9j); (j2; 7j); (j3; 9j); (j4; 5j) >> \text{ unsorted pod}$$

$$\Rightarrow << (j0; 2j); (j1; 5j); (j2; 7j); (j3; 9j); (j4; 9j) >> \text{ sorted pod}$$

$$\Rightarrow << (j0; 0j); (j1; 3j); (j2; 2j); (j3; 4j); (j4; 1j) >> \text{ items before}$$

91
Chapter 4. PARALLEL ALGORITHMS

given the i th element in the pod to be sorted, then the i th element of the "items before" pod represents how many items would directly precede that element if the pod was sorted, i.e., three items in the sorted pod precede the first nine of the input pod; two items precede the first seven; four items the second nine. Informally, a parallel algorithm that calculates the "items before" pod would proceed as follows:

1. A variable count is used to represent the number of preceding elements. The initial value of this variable would be zero, signifying no elements precede the first element of the pod;

2. For each i in the range \( \text{minimumPod}(\text{pod}) \leq i \leq \text{maximumPod}(\text{pod}) \):

   - Those elements from the input pod that have a value equivalent to i are flagged;
   - If the flagged elements are numbered from count in increments of one, then each flagged element records how many elements directly precede it. The consequence of enumerating in ascending order from count, in conjunction with i having consecutive values in the range \( \text{minimumPod}(\text{pod}) \) to \( \text{maximumPod}(\text{pod}) \) ensures sorting is stable;
   - count is updated with the value of the last element of the flagged pod;

For a region within a pod of size N, the sort will have a \( O(\log N) \) as the enumeration of the flagged elements can be performed by a \( O(\log N) \) scan. Unfortunately, the algorithm described here has a rather large downside. If the values of the pod to be sorted are
Chapter 4. PARALLEL ALGORITHMS

proportional to $N$, then order $N$ iterations of the loop are required, which results in the algorithm having a $O(N)$.

4.4.2 The "items before": imperative

The informal algorithm described in the previous section is encoded in DPHaskell by using a technique that moulds the algorithm into a form suitable for a data-parallel machine. Given a loop with counter $i$ that ranges over the smallest to largest values in the pod to be sorted at iteration $i$, we perform the following:

- Those elements that have a value equivalent to $i$ are assigned the value one, those without zero;
- The $(+)$ function is scanned across the pod;
- count is added to the resulting pod.

The difference between this technique, and the previous informal description is that those elements of unsorted that are not equivalent to $i$ will be active during the algorithm. As only the values of the flagged elements are of interest, this discrepancy does not alter the semantics of the algorithm.

```haskell
> itemsBefore :: Pid a => Int -> Int -> Int -> a -> a
> | i > max — i = result
> | otherwise = itemsBefore
> count' (i+1) max — i low — pid unsorted result
```
A naive sorting algorithm can be defined using itemsBefore with a mixture of sending and fetching communication. Notice how the result on entry to itemsBefore is undefined at each processor; the result pod is gradually filled during successive iterations.

A complete sorting algorithm can now be encoded as:

```haskell
naiveSortPod :: Pid a => a -> a -> ((a;Int!) -> <<(a;Int!)>> -> <<(a;Int!)>>
naiveSortPod low pid high pid unsorted
= << ((low pid y dat) | (x;dat)) << unsorted,
   (x;y) <<= sortRadix
   0
   (minPod low pid high pid unsorted)
```

94
Chapter 4. PARALLEL ALGORITHMS

> (maxPod low—pid high—pid unsorted)
> unsorted
> << .. (error "bottom") .. >>
> >>

This concludes the imperative presentation of the "items before" algorithm.

4.4.3 The "items before": functional

An alternative way of calculating the "items-before" index-pod is to perform three phases that: (1) histogram the input data; (2) perform a list-scan of (+) on the histogrammed data; (3) finally assign the number of occurrences of each value to be sorted within the input pod.

Histogram Figure 4.8 shows this first stage of the algorithm, where the input data is histogrammed. Arvind identified histogramming as an important application of arrays, where a purely functional implementation should have the ability to histogram data in \(O(N)\) time. The parallel algorithm used here is \(O(\log N)\).

The combEnv function takes two environments and combines them in such a way that any elements from either of the argument environments that have the same domain value combine their respective target elements using an associative binary operator. Any mappings in either of the environments which have a disjoint domain are copied into the resulting environment.
Chapter 4. PARALLEL ALGORITHMS

> combEnv :: Ord a => (b -> b -> b) -> Env a b -> Env a b
> combEnv fn [] renv = renv
> combEnv fn lenv [] = lenv
> combEnv fn ((s,t):lenv) ((s',t') : renv)
>   | s == s' = (s,fn t t') : combEnv fn lenv renv
>   | s < s' = (s,t) : combEnv fn lenv ((s',t') : renv)
>   | s > s' = (s',t') : combEnv fn ((s,t) : lenv) renv

The definition of combEnv uses a list of pairs to represent an environment, where the lists are sorted on the values of their domain. For example, two environments represented as lists can be combined as:

\[
\text{combEnv}(+)\{[1 \rightarrow 42; 2 \rightarrow 1; 3 \rightarrow 5][0 \rightarrow 2; 1 \rightarrow 4; 5 \rightarrow 5] \Rightarrow [0 \rightarrow 2; 1 \rightarrow 46; 2 \rightarrow 1; 3 \rightarrow 5; 5 \rightarrow 5]\}
\]

The first stage of the parallel histogramming algorithm is to create an environment for each processor, where each environment contains the mapping \([\text{source} \rightarrow 1]\); where source is the data stored in a processor. If the associative function combEnv (+) is folded across this pod, then the resulting environment is a histogram of the input data.

> histogramPod :: (Pid a, Ord b) => a -> (a,b) => -> [(b,Int)]
> histogramPod = foldPod (combEnv (+))
List-scan  The second stage of the "items before" algorithm is also shown in figure 4.5, where addition is scanned down the successive elements of the co-domain of the environment (assuming the environment is sorted on the values in the domain). If the pod to be sorted has a lower bound of one, then this environment encodes the processor identifier of the last occurrence of the values in a sorted version of the input pod.

Suffix-scan  The final part of the algorithm, illustrated in figure 4.6, stores the scanned environment alongside the last element in the pod to be sorted. If this last element contains the value $x$, then one is subtracted from the target element which has $x$ as its domain. The altered environment is then passed to the left-neighbouring processor, and the process is repeated. By iterating the subtraction and copying from right-to-left in the pod, then when all the iterations are completed, and the resulting pod of environments is indexed with the original unsorted data, then the "items-before" pod is produced.
Chapter 4. PARALLEL ALGORITHMS

This iterative algorithm can be converted into a parallel algorithm by using the suffix-scan of § 4.2. If the last processor of the pod to be sorted contains the list-scanned environment, and all other processors contain [source \(\rightarrow 1\)], the suffix-scanning of \(\text{combEnv}(+)\) over the POD of environments creates an environment which when indexed by unsorted data produces the "items before" index-POD. The complexity of this algorithm is \(O(\log N)\) if the number of distinct elements in the pod to be sorted is independent of the size of the pod to be sorted; otherwise the complexity is \(O(N \log N)\) due to the \(O(N)\) complexity of \(\text{combEnv}\).

\[
\begin{aligned}
> \text{itemsBefore} :: \text{Pid} a &\rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow a \rightarrow a \\
> &\rightarrow \langle\langle a;\text{Int}\rangle\rangle \rightarrow \langle\langle a;\text{Int}\rangle\rangle \rightarrow \langle\langle a;\text{Int}\rangle\rangle \\
> \text{itemsBefore count i max} &\rightarrow \text{pid high} \rightarrow \text{pid unsorted result} \\
> = \text{zipWithPod indexEnv unsorted final-envs} \\
> \quad \text{where} \\
> \text{histogram} = \text{histogramPod low} \rightarrow \text{pid (mapPod (\"x \rightarrow[(x,1)]\) unsorted)} \\
> \quad (\text{src, tgt}) = \text{unzip histogram} \\
> \text{final-envs} = \text{suffixScanPod (combEnv (+)) high} \rightarrow \text{pid} \\
> \quad \text{(updatePod} \\
> \quad \text{(mapPod (\"x \rightarrow[(x,-1)]\) unsorted)} \\
> \quad \text{high} \rightarrow \text{pid} \\
> \quad \text{(zip src (scanl1 (+) tgt)))}
\end{aligned}
\]
Chapter 4. PARALLEL ALGORITHMS

> indexEnv :: Eq (a,b) => (a,b) -> [(a,b)] -> b

> indexEnv x xs = snd (head [ y | y<-xs, y==x])

> updatePod::(Eq a, Pid a) => <<(a;b)>> -> a -> b -> <<(a;b)>>

> updatePod vec i x = << (| a; if a == i then x else b |) | (|a;b|) <- vec >>

Comparing this algorithm with the one of §4.4.2, shows that it can be considered as an optimisation because the calculation of the index-pod only depends upon the number of distinct values in a pod, which will always be smaller than or equal to the range of the values in the pod. One problem with encoding this environment based algorithm in an imperative language, is that a vector of environments cannot easily be implemented. In contrast it is natural to encode the environment as a list, or even a tree, to speed-up lookups in DPHaskell.

4.4.4 The radix part of the sort

For fixed sized integers, the undesirable O(N) behaviour of the imperative items before, or the O(N log N) complexity of the functional encoding can be eliminated by taking advantage of the physical constraints of any computing device, and performing some "bit-twiddling". For a 32-bit integer, the range of values in the pod to be sorted will be $2^{32}$. This value is a physical upper-bound on the number of iterations of the tail-recursive function of §4.4.2. As this attributes the O(N) complexity to the items before algorithm, it has the consequence that the theoretical complexity of the function is really
Chapter 4. PARALLEL ALGORITHMS

\(O(1)\)-albeit with an enormous constant of proportionality! The trick applied in the radix sorting algorithm is to reduce this constant of proportionality to more manageable levels.

If it were possible to split a 32-bit integer into two 16-bit integers, and sort the two sub-parts independently, then the number of iterations of the tail recursion would be reduced to \(2 \times 2^{16} = 2^{17}\). The idea behind the radix sorting algorithm is to split each of the integers in the pod to be sorted into \(m\) chunks, giving an absolute range of \(m \times 2^{32/m}\). The choice of value for \(m\) which gives an optimal radix-sorting algorithm will be both machine and compiler dependent.

4.4.5 Bit twiddling in a functional language

In an imperative implementation of radix-sorting, the integer to be sorted would be split into \(m\) chunks by accessing the underlying binary representation of an integer. Unfortunately the manipulation of such a representation unnecessarily complicates matters because splitting needs to cope with the underlying representation of negative numbers. Given an integer that is split into \(m\) chunks, if the most-significant chunk contains a negative number, then that chunk will have a numerical value which is larger than any positive number (because of the 2s-complement encoding of an integer). To ensure that any ordering predicates used on the chunked numbers have their natural interpretation, we flip the sign-bit of the least significant chunk to be sorted.

In a functional language we can raise the level of abstraction of the splitting process. The reason why the underlying representation of the integer is split in the radix sorting algorithm, is that it is efficient to do so in an imperative language, such as C or
Chapter 4. PARALLEL ALGORITHMS

Fortran. Abstractly, all we need to do is to split the integer in such a way that the absolute range of each chunk is smaller than the range of the total. One simple way of doing so is to split an integer into its component digits. The function numToDigits performs this transformation, and as negative numbers are encoded by the head of the list (least significant chunk) having a real negative value, then there is no need to perform sign-changing to ensure that the standard ordering predicates are well defined.

\[
\text{numToDigits :: Int} \rightarrow \text{Int} \rightarrow \text{[Int]}
\]

\[
\text{numToDigits base x | x >= 0 = num-to-digs base x}
\]

\[
| \text{otherwise = case (num-to-digs base (-x)) of}
\]

\[
(y:ys) \rightarrow (-y):ys
\]

\[
> \text{num-to-digs :: Int} \rightarrow \text{Int} \rightarrow \text{[Int]}
\]

\[
> \text{num-to-digs base x}
\]

\[
> | x \ 'div' \ base \ = \ 0 = \ [x \ 'mod' \ base]
\]

\[
> | \text{otherwise = num-to-digs base (x \ 'div' \ base) ++ [x \ 'mod' \ base]}
\]

A complete sorting algorithm that uses the itemsBefore function can be viewed as a simple induction:

**0th iteration** If the pod is already sorted, then the index-pod will be [lower; : : : ; upper], signifying each element of the pod sends data to itself (an identity communication).
Chapter 4. PARALLEL ALGORITHMS

iteration k The itemsBefore algorithm is used to calculate an index-pod for a single chunk of the pod to be sorted. The resulting index-pod is used to permute the remaining chunks of the original pod unsorted, and the prior value of index-pod.

The gradeUpPod function calculates the index-pod required to communicate the unsorted data into its sorted form. Equation 4.6 provides a denotational relationship between gradeUpPod and itemsbefore, but operationally the advantage of gradeUpPod is that it is guaranteed to have a O(log N) complexity irrespective of the values being sorted.

\[
\text{gradeUpPod} = \lambda \ h \ d < \text{itemsBefore} 0 \rightarrow \infty \ \rightarrow \ h \ d < \ldots \ | \ldots \rangle \ \ldots \ (4.6)
\]

We can implement gradeUpPod as follows:

\[
\text{gradeUpPod} :: \text{Pid a} => a \rightarrow b \rightarrow ((a;\text{Int}) \rightarrow \rightarrow ((a;\text{Int}) \rightarrow
\]

\[
= \text{induction} (\text{mapPod} (\text{numToDigits} \ \text{base}) \ \text{unsorted})
\]

\[
\rightarrow \rightarrow ((x;x)) \ | \ (x;y)) < - \text{infPod} \rightarrow
\]

\[
\rightarrow \ | \ \text{where}
\]

\[
\rightarrow \ \text{base} = 10
\]

\[
\rightarrow \rightarrow \ \text{induction chunks ivec}
\]

\[
\rightarrow \rightarrow = \text{if} (\text{allPod} \ \text{null low} \rightarrow \text{pid chunks})
\]

\[
\rightarrow \rightarrow \rightarrow \ \text{then} \ \text{ivec}
\]

\[
\rightarrow \rightarrow \rightarrow \ \text{else} \ \text{let} \ \text{send ivec} = \text{itemsBefore} 0 \ (-\text{base}) \ \text{base}
\]

\[
\rightarrow \rightarrow \rightarrow \ \rightarrow \ \text{low} \rightarrow \text{pid high} \rightarrow \text{pid}
\]

\[
\rightarrow \rightarrow \rightarrow \ (\text{mapPod head chunks})
\]
Chapter 4. PARALLEL ALGORITHMS

> 
> (error "Bottom").. >>

> in induction (sendIPod send-ivec (mapPod tail chunks))

> (sendIPod send-iVEC iVEC)

> sortIntPod :: Pid a => a -> b -> << ([a;Int]) >> -> << ([a;Int]) >>

> sortIntPod low—pid high—pid unsorted

> = sendIPod (gradeUpPod low—pid high—pid unsorted) unsorted

The complexity of the sorting algorithm is $O(\log N)$, and can be understood by considering the complexities of each of the constituent parts of the algorithm:

- The small range of values in a pod after chunking, causes the complexity of itemsBefore to be $O(\log N)$;
- The fixed number of iterations of induction are bounded by a value proportional to base;
- The communication of sendIPod is assumed to have a $O(1)$ complexity (see the discussion in §4.3).

In conclusion, a radix-sorting algorithm has been presented where the parallelism inherent in the algorithm can be attributed to scan. The advantage of giving a functional description of the algorithm compared to a conventional imperative description is that the level of abstraction can be raised. This exposes the essence of the algorithm without bit-twiddling blurring the problem.


Chapter 4. PARALLEL ALGORITHMS

4.5 Segmented Scan

The segmented scan algorithm is yet another variation of prefix-scan of §4.1. The algorithm splits the pod being scanned into segments and then performs a scan on each individual segment. A common representation of a segment is to use a pod of Booleans where a true represents the start of each segment. Figure 4.7 shows the result of segment-scanning (+) over a pod that contains three segments. A specification of the function is given below in DPHaskell:

\[
\text{segmentScanPod fn low \_pid mask = flattenPod low \_pid . mapPod (scan1 fn)}.
\]

\[
\text{packPod low \_pid mask . groupPod low \_pid mask}
\]

Fig. 4.7 An example of segmented-scan

The algorithm is decomposed into four phases: (1) groupPod collapses each segment into the processor at the start of a segment (the definition of this function isn't given); (2) the resulting pod of lists will then be defined wherever the segment-mask is true, and these
lists are packed into consecutive locations; (3) each of the grouped lists is then list-scanned using the function \( fn \) (as zero-sized segments cannot be expressed using the segment-mask, it is safe to use \( \text{scanl1} \) on each of the grouped lists); (4) finally the lists are flattened back into a pod that has the same extent as the original (the definition of \( \text{flattenPod} \) isn't given). Although the specification of the problem is concise, it suffers from an excessive amount of redundant communication that initially groups each of the segments, and then flattens the data again. A more serious problem is that it has a \( O(N) \) complexity in the largest segment of the pod being scanned.

A solution to the adverse \( O(N) \) complexity of the specification is to merge the grouping, list-scan, and final flattening into a single monolithic unit. A way of achieving this is to apply a tailored parallel prefix technique in which the segment mask is taken into consideration during each iteration of the algorithm. In contrast to the number of iterations of the parallel-prefix technique being indirectly bounded by the size of the pod being scanned (ignoring the infinite nature of the \( \text{DPHaskell prefix-scan} \) algorithm), the idea is to let the segment mask guide the termination condition of the parallel prefix technique. Figure 4.11 shows an example of the steps required in the segment-scan algorithm.
Chapter 4. PARALLEL ALGORITHMS

Table 4.1: When to combine in the segmented-scan algorithm

Fig. 4.8 Segmented scan: A worked example

<table>
<thead>
<tr>
<th>left</th>
<th>right</th>
<th>combine?</th>
<th>justification</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>don't combine</td>
<td>combination between two elements that start different segments.</td>
</tr>
<tr>
<td>✓</td>
<td>✗</td>
<td>combine</td>
<td>combination between the element at the start, and one within the same segment.</td>
</tr>
<tr>
<td>✗</td>
<td>✓</td>
<td>don't combine</td>
<td>combination across segment boundaries.</td>
</tr>
<tr>
<td>✗</td>
<td>✗</td>
<td>combine</td>
<td>combination of elements within the same segment.</td>
</tr>
</tbody>
</table>

Table 4.1: When to combine in the segmented-scan algorithm
When considering the combination of two processors in the parallel prefix technique, table 4.1 is used to determine if the combination should occur (an informal justification of the combination is given next to each entry in the table); processors which don't combine (identified by the dashed line in figure 4.8) leave their contents unchanged. After the elements combine according to these rules, the mask value of the right-combining processor is updated by applying a Boolean "or" to the combing processors. A conventional presentation of the segment scan algorithm would iterate the combining processor for \( \log N \) iterations (for a vector of size \( N \)), or until the segment mask becomes all true. In keeping with the lazy prefix-scan algorithm of x4.1, this algorithm uses indefinite iteration with the termination conditions of each processor controlled by the value of the segment-mask. By using non-strictness, the number of iterations of the entire algorithm is determined by those processors defined by the aim of evaluation. For example, if the first processor of each segment is required, then because of the non-strictness of the lower conditional in the body of the let expression, no iterations of the parallel prefix technique are be required.

```haskell
> segmentScanPod::{(Ord a,Num a,Pid a)=>(b->b->b) -> a -> <<([a;Bool]>>)
>      -> <<([a;b])>> -> <<([a;b])>>
>
> segmentScanPod fn lo mask
>  = mapPod snd . eachIter l . zipWithPod (\m x -> (m,x)) mask
>  where eachIter k vec
```
Chapter 4. PARALLEL ALGORITHMS

> =let vec' =<< (x' (m_b || m_a,
>   if (combine m_b m_a)
>   then fn b a
>   else a)
>   | (x; (m_a,a)) | << vec,
>   (x - 2^(k-1); (m_b,b)) | << vec >>
> in << (x; if m then a else b
>   | (x; (m,a)) | << vec,
>   (x; (_b)) |<< eachIter (k+1) vec' >>

> combine :: Bool -> Bool -> Bool
> combine _ False = True
> combine _ _ = False

As this algorithm performs the same steps as a conventional prefix-scan (ignoring non-combination), the algorithm's complexity is O(log N), irrespective of the segment size.

4.6 Segmented fold

Just as the prefix-scan algorithm formed the basis of parallel fold, segmented-scan can be used as the basis of a segmented-fold. Instead of indexing the last processor in a prefix-scanned pod to denote the result of a fold, a segment-scan is applied and the last processor in each segment is identified and packed into consecutive processors. As the last element of a segment will be at a position one place to the left of the start of the next
Chapter 4. PARALLEL ALGORITHMS

segment, the segment-mask is shifted one place to the left and used in a pack. The results of the segmented fold will therefore be in consecutive processors from low—pid in the resulting pod.

\[
\text{segmentFoldPod::(Ord a,Num a,Pid a)->(b->b->b) -> a -> } <<(a;Bool)>> \\
\rightarrow \ (a;Bool)>> \rightarrow \ (a;b)>
\]

\[
\text{segmentScanPod fn low—pid mask}
\]

This completes the foundations required for the "sending with collision" function in the next section.

4.7 Sending with collision

In the final section of this chapter we define the scatter function of HPF. sendWithPod defines a general sending communication within a finite region of a pod, where any colliding data is resolved using a binary associative operator. This definition for sending communication can also be used as an alternate definition of foldPod:

\[
\text{foldPod :: (a -> a -> a) -> Int -> Int -> } <<(Int;a)> > \rightarrow \ a
\]

\[
\text{foldPod fn low_pid high_pid vec}
\]

\[
\text{indexPod (sendWithPod fn low_pid high_pid } \text{vec} 0
\]

The implementation of send WithPod provides an opportunity to use a large proportion of the functions defined in this chapter. The sending algorithm initially sorts the data to be sent, together with the index-vector of the send, using the index-vector as the sort key.
Chapter 4. PARALLEL ALGORITHMS

One way of sorting two pieces of data, when one is the sort key, is to apply the gradeUpPod function of x4.4 to the sort key, and then send the two original pieces of data using the resulting index-vector. Once the data has been sorted, those processors that send their data to the same processor-identifier will lie in consecutive processors. These runs of processors are identified, and used as the segment-mask of segmentFoldPod. The mask can be calculated by identifying an edge in the index-vector, using the pod comprehension:

\[ \langle\langle (x; a = b) | (x; a) \rangle \rangle_{\text{sort-ivec}}, \langle\langle (x-1; b) \rangle \rangle_{\text{sort-ivec}} \]

A complete algorithm is shown below in DPHaskell:

\[
\begin{align*}
\text{sendWithPod} & : (a \to a \to a) \to \text{Int} \to \text{Int} \\
& \quad \langle\langle (\text{Int}; \text{Int}) \rangle \rangle \to \langle\langle (\text{Int}; \text{a}) \rangle \rangle \\
& \quad \langle\langle (\text{Int}; \text{a}) \rangle \rangle \to \text{result}
\end{align*}
\]

\[
\begin{align*}
& \text{sendWithPod fn low-pid high-pid ivec data} \\
& \quad = \text{segmentFoldPod fn low-pid high-pid segment-mask sort-data} \\
& \quad \text{where} \\
& \quad \text{send-ivec = gradeUpPod low-pid high-pid ivec} \\
& \quad \text{sort-ivec = sendIPod send-ivec ivec} \\
& \quad \text{sort-data = sendIPod send-ivec data} \\
& \quad \text{segment_mask = } \langle\langle (x; a = b) | (x; a) \rangle \rangle_{\text{sort-ivec}}, \\
& \quad (x-1; b) \langle\langle \text{sort-ivec} \rangle \rangle
\end{align*}
\]
Chapter 4. PARALLEL ALGORITHMS

As each of the functions used in the send is $O(\log N)$, and they are applied one after another, the complexity of sendWithPod is also $O(\log N)$.

4.8 Summary

As shown in table 4.2, this chapter has described a number of useful parallel algorithms whose parallelism can be attributed to parallel scan. These algorithms illustrate the benefits of using scan and the elegance of DPHaskell, yet we haven’t covered anything really profound in this chapter. This work illustrates how conventional data-parallel techniques carry through to DPHaskell, with non-strictness making its mark on the algorithms by generalising existing approaches.

<table>
<thead>
<tr>
<th>Function</th>
<th>Section</th>
<th>Sequential complexity</th>
<th>Parallel complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>scanPod</td>
<td>4.1.4</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>foldPod</td>
<td>4.1.4</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>linearReccur</td>
<td>4.2</td>
<td>$O(N^2)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>suffixScanPod</td>
<td>4.2</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>packPod</td>
<td>4.3</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>histogramPod</td>
<td>4.3</td>
<td>$O(N)$ best case, $O(N^2)$ worst case</td>
<td>$O(\log N)$ best case, $O(N \log N)$ worst case</td>
</tr>
<tr>
<td>gradeUpPod</td>
<td>4.4</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>sortIntPod</td>
<td>4.4</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>segmentScanPod</td>
<td>4.5</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>segmentFoldPod</td>
<td>4.6</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>sendWithPod</td>
<td>4.7</td>
<td>$O(N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

Table 4.2: The complexities of DPHaskell functions