A FUNCTIONAL APPROACH TO PARALLEL GREEDY ALGORITHMS

Joseph Molloy

Bachelor of Engineering (Software Engineering)

Supervisor: Gabriele Keller

Assessor: Ben Lippmeier

June 2013

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF NEW SOUTH WALES

Contents

Abstract

When writing parallel algorithms, the sharing of state between multiple cores can lead to non-determinism. Determinism gives us many benefits, such as ease of reasoning regarding debuggability and correctness. Haskell, and its extension Data Parallel Haskell (DPH), possess many virtues that make them appealing for programming deterministic parallel algorithms. In particular, functions are referentially transparent, meaning that they can safely be performed in parallel. In theory, this would suggest that we can easily parallelise our algorithms written in such a language. In some cases, this is so, however, graph algorithms are one area where this is rarely the case. Graphs are generally complex and highly connected data structures, and hence, threads need to share information somewhere. The question is, in functional language paradigm, where in our program do we break the rules of referential transparency to minimise the ill-effects of non-determinism.

In this thesis I propose an extended model of deterministic reservations as a solution to to this problem for a subset of graph algorithms. I take advantage of Haskell's type system to provide a more rigorous interface to deterministic reservations, by isolating mutable and immutable data structures, and explicitly defining relationships and requirements between them. I compare my model to the original implementation, and look at the work needed to integrate my work into DPH. My deterministic reservations achieves excellent speedup of over 4x on 8 cores, and good overall performance.

Acknowledgements

Firstly, I would like to thank my supervisor, Gabrielle Keller, for her constant support and encouragement over the last year. She has constantly challenged me to make my thesis a worthwhile contribution, and given me that needed push when I was stuck. Secondly I want to appreciate some of brilliant work on which my thesis builds on, particular that of Guy Blelloch and the Carnegie Mellon PBBS team, the UNSW Programming Languages and Systems group and the Haskell and GHC development teams.

Also I have to thank Nick, Dane, Leah, Sarah and particularly Rachel, who have each been constant supports in their own way throughout my honours year. Thank you to you all for acting as sounding boards for my frustrations and ideas, and just being there. Finally, I would like to thank my family, particularly mum for the constant supply of nutritious brain food, and dad for his constant support and encouragement.

1 Introduction

Researchers have agreed that controlling determinism is an essential consideration when designing parallel algorithms [3]. Controlling the level of determinism can address many concerns when reasoning about an algorithm. These include programmability, correctness, performance and debugging. It is well established that if two operations commute, then they can be performed in parallel, as the order of their execution does not impact the final result [44]. In [6], Blelloch et al present deterministic reservations as a programming pattern for encapsulating commutative operations, thereby enforcing the deterministic properties of an algorithm.

In the functional programming language Haskell, all functions have referential transparency. This means that a function and its input can be placed with the result, and the result of the program will not change [43]. This approach eschews side-effects and mutable data. If the result of a function depends only on its input, and no side-effects are produced, then any two such functions can be run in parallel, with the guarantee of commutativity.

These qualities make Haskell a powerful platform for parallel programming. We can automatically run our programs in parallel, and be sure, mathematically, that we will get the same result. We do not have to worry about our parallel operations commuting, as they are referentially transparent. However, there are limitations with this approach. When dealing in parallel with complicated, interconnected data structures such as graphs, we have to break referential transparency somewhere, as multiple threads need to share information between them. The question is, in the functional language paradigm, where in our program do we break the rules to minimise the ill effects of non-determinism.

In this thesis, I propose deterministic reservations as a solution to this question, presenting an adaptation of deterministic reservations to the functional programming paradigm to support my case. In doing so, I also extend deterministic reservations to provide a clearer and more descriptive interface for the programmer. Part of this extension includes a referentially transparent interface to deterministic reservations as a compelling approach to tackling graph problems in pure functional languages.

A Haskell implementation of flat data parallelism, REPA, already uses this property to great effect [24]. With fixed-size, regular n-dimensional arrays, flat data parallelism is a well-researched market leader. Designed around a SIMD model (Single Instruction, Multiple Data), a sequential operation can be performed on multiple data in parallel. However, many interesting algorithmic problems involve complicated algorithmic structures, such as trees and graphs. Flat data parallelism doesn't deal well with such data structures, and nested data parallelism implementations, such as Data Parallel Haskell (DPH) have been proposed as an alternative for working with such data structures in parallel [8].

DPH provides parallel array comprehensions over data structures represented as nested parallel arrays. It optimises these operations using fusion to remove the need for temporary arrays, leading

to fast, referentially transparent code. However, sometimes working with mutable data structures is easier, and in some scenarios, necessary for fast performance. In this thesis, I also consider deterministic reservations in this context, arguing that my work on deterministic reservations can be easily incorporated into DPH. This would provide the language extension with a compelling story for tackling graph algorithms.

The original deterministic reservations implementation, as part of the PBBS suite was developed using the Cilk++ framework [6, 30], utilising task parallelism and work-stealing in their approach. I specifically investigate whether speedup multiples similar to Blelloch's can be replicated in a higher-level language such as Haskell using gang parallelism. I approach the problem in the following way:

- In 2.3, I look in detail at deterministic reservations, and define a class of problems which are suited to such a control structure.
- Section 3 presents my Haskell extension of deterministic reservations, based on Leshchinskiy's Vector library, and I look at how to integrate the work into DPH.
- In Section 4, I evaluate the performance of our approach, focusing on multicore performance.
- Section 5 summarises the work needed to integrate my work into DPH, and evaluates the merit of such an effort, in light of these results.

2 Review of Prior Work

2.1 Data parallelism

Data parallelism has been recognised as an market leader for many years, supported by use in High Performance Fortran [13], and more recently on GPU programming platforms such as CUDA [36], and in the Haskell extension REPA [24]. Most implementations, such as those mentioned, work in the more restrictive model of flat data parallelism. In this model, sequential operations can be performed on an array of elements in parallel. This fits nicely with current memory architectures, and require fixed size n-dimensional arrays. However, many problems required complex, nested data structures, and the more general nested data parallelism offers promise in this area. It allows us to perform parallel computations on array elements in parallel [5].

2.2 Data Parallel Haskell

In [39], Peyton Jones et al describe their implementation of nested data parallelism as an extension of Haskell, called Data Parallel Haskell (DPH). DPH is pure Haskell, compiled through the GHC with a few extensions.

2.2.1 Vectorisation

The idea of implementing nested data parallelism through a vectorisation to flat data parallelism was first proposed in [9, 4], and implemented as part of the programming language NESL [8]. However, NESL did not provide all the useful features of a modern functional programming language. One of the main contributions of DPH to nested data parallelism was its expansion of vectorisation to support higher-order functions and algebraic data types. A large part of the DPH paper [39] (Section 4 onwards) covers DPH's implementation of vectorisation transformations.

Nested data parallelism allows us to represent structures such as trees and graphs, and even divide and conquer algorithms. However, these programs need to be converted to flat data parallel programs before they can be run. This is done by 'flattening' our nested arrays. The first proof that this was possible was provided with NESL in [8]. A nested array is flattened by concatenating each array into a contiguous section of memory. A second array describes each sub-array as a segment of our new concatenated array. Essentially, our internal representation is very different from what the programmer sees. Not only do we need to vectorize our data, but our code as well. As Lippmeier et al found, this can drastically degrade the runtime complexity of the resulting vectorised program [31]. In a simple example, taken from [39]:

```
f :: Float → Float f x = x*x + 1
```

For every such function we build its lifted version f_L thus:

```
f_L :: [:Float:] →[:Float:]
f_L x =(x *_L x) +_L (replicateP n 1)
        where n = lengthP x
```

If we have a function over a nested array, such as

```
g :: [:Int:] → [:Int:]
g xs = mapP f xs
```

We replace map f with f_L, the lifted version of f over arrays, equivalent to a mapping of f over each element.

```
g xs = f_L xs
```

However, what if our code requires a lifted version of g?

```
h :: [:[:Int:]:] → [:[:Int:]:]
h xxs = map P g xxs
```

Then we will need a doubly lifted version of f.

```
g_L :: [:[:Int:]:] →[:[:Int:]:]
g_L xs =f_LL xs
```

This can cause complications if the nesting depth of our arrays is unbounded, which in many tree structures, it is. Blelloch work with NESL solved this problem, making vectorised nested data

parallelism feasible [8]. He notes that a doubly lifted version can be represented in terms of a singly lifted version, coupled with concatenation and unconcatenation operations.

```
f_LL :: [:[:Float:]:] →[:[:Float:]:]
f_LL xss =unconcatP xss (f_L (concatP xss))
```

With the right logical representation of our arrays, *concatP* and *unconcatP* can be performed in constant time.

2.2.2 Array representation

In [31], Lippmeier et al present a solution to this problem, using virtual segment descriptors. This breakthrough meant that a flattened array in DPH did not have to be contiguous in memory, but can be internally represented by a series of segments scattered through separated memory blocks.

Standard arrays in Haskell, and the vector library are parametric. While 'boxing' each element is very flexible, it introduces performance overheads. Firstly, needing to following a pointer to each element, or 'unboxing' increases memory traffic, particularly when traversing an array. Secondly, it reduces opportunities for caching by reducing memory locality [39]. Parallel arrays in DPH, represented by [: a :] are also parametric, however, at the inner workings of DPH, we want to avoid the performance overheads that come with boxed arrays mentioned above. Internally, DPH provides a new array type PA which have a non-parametric representation [10]. For example, $PA(Int, Int)$ is held as two identical-sized arrays of unboxed integers, instead of an array of pointers to pairs of Ints. Part of the vectorisation process involves transforming functions which operate over [: a :] into new versions which accept non-parametric arrays of type $PA\,a$.

As arrays of type $PA\,a$ are non-parametric, our standard polymorphic functions over collections won't work. A standard function such as $lengthPA :: PA\,a \to Int$ can't be written, as $lengthPA$ includes no information about the representation of $PA\,a$, and hence, nothing is known about the length of $PA\,a$ either. The $PAElem$ type class in Figure 1 is used to work around this.

```
class PAElem a where
        data PA a
        indexPA :: PA a → Int → a
        lengthPA :: PA a → Int
        replicatePA :: Int → a → PA a
        ...more operations...
```

Figure 1: $PAElem$ type class, from [39]

Hence for each type a that is contained in a parallel array, the non-parametric representation of the array is specified, as well as operations over that array. Included in DPH are instances of the $PAElem$ typeclass for structured data types such as Int, $Float$ and $Bool$, as well as product

types, sum types and functions. However, Peyton Jones et al note that this approach does not yet allow for polymorphic functions over parallel arrays [39]. For example in the following function,

```
firstRow :: [:[:a:]:] → [:a:]
```

$[: a :]$ can't be replaced with $PA\,a$ as $PA\,a$ is non-parametric, and the type of a is not known. Also, since DPH supports separate compilation and polymorphic recursion, we can't simply generate a set of monomorphic functions for each type-specific call to $firstRow$. The authors state that they are looking towards using something like typeclasses to solve this problem.

2.2.3 Fusion

Once the compiler has vectorised the nested parallel arrays, DPH performs a fusion step to remove the need for any temporary arrays. Operations over parallel arrays such as map and filter are 'fused' together into a single loop. This process is described in detail in [11]. Often this requires radically changing the order of computations on the array, but we can be sure this optimisation won't change the final result, as all our operations are pure.

This purity is also a key enabler for Haskell's lazy evaluation [21], where values are only evaluated when needed. In vanilla Haskell, infinite lists can be defined, and each element of the list is only evaluated when needed. This is a powerful tool in Haskell, however, in DPH, vectorisation requires that parallel arrays follow strict semantics. In the process of flattening nested parallel arrays, the vectorisation step needs to know the size of each nested array. Hence, when an element is accessed from a parallel array in DPH, the whole array must be evaluated.

As part of the vectorisation process, DPH needs to to know the size of each nested array. In DPH, "demand for any element of a parallel array results in the evaluation of all elements." [39]

2.2.4 Gang parallelism

Internally, DPH uses the fork/join strategy to provide parallelism over parallel arrays [39]. A gang of threads is created, and the input array is divided up between them. The threads are reused through the program, with MVars providing a communication channel between the main thread and the gang workers. A simplified version is used to support the deterministic reservations implementation in this thesis. With nested data parallelism, it is tricky to find the right level of granularity, where the right balance between parallelism and thread overhead is present. By dividing work up into chunks based on the number of threads, gang parallelism adapts the granularity to suit the number of cores provided by the machine.

2.3 Deterministic reservations and associated work

2.3.1 Parallelising greedy algorithms

Certain graph operations can be modelled as a greedy sequential algorithms which process vertices or edges in a linear order. Some examples include finding the minimum spanning tree [20], maximal independent set [32], connected components using disjoint sets [15] and shortest path [26].

Greedy algorithms often provide very clear and concise approaches to a problem, and where they are applicable, do so with good complexity. Ideally, we want to parallelise any iterations of the loop where the operations have no conflicts. Operations can be performed in parallel when they commute, i.e. the order of their execution doesn't matter. As far back as 1989, Steele recognised this in his seminal work *Making asynchronous parallelism safe for the world* [44]. Both runtime and compiler approaches have been proposed to implicitly parallelise programs with commutative operations [40]. In [6], Blelloch et al argue that for a class of highly-data dependent problems, neither compiler or runtime approaches are particularly effective. Compiler implementations can't take advantage of data-related commutativity, while runtime techniques use a technique similar to software transactional memory, introducing a significant overhead to the algorithm.

2.3.2 Deterministic reservations - A novel approach

Blelloch et al propose deterministic reservations as an alternative approach. With this approach, it is up to the algorithm designer to specify which operations commute, with non-commutative operations separated into different passes over the iterates. In a greedy algorithm, the order iterates are processed is essential to both the correctness and determinism of the solution. With deterministic reservations, potentially all the iterates could be processed in parallel. However, this is obviously unlikely. If it were possible, an approach such as deterministic reservations wouldn't be necessary. Instead, a prefix of the iterates is selected, and two passes are made. The *reserve* phase checked for any data dependent conflicts, and then those that have none are *committed* in the second phase. Processed elements are then removed from the array, and a new prefix is selected from the unprocessed iterates. This is repeated until all iterates have been processed. A simplified example of this process is presented in 2.

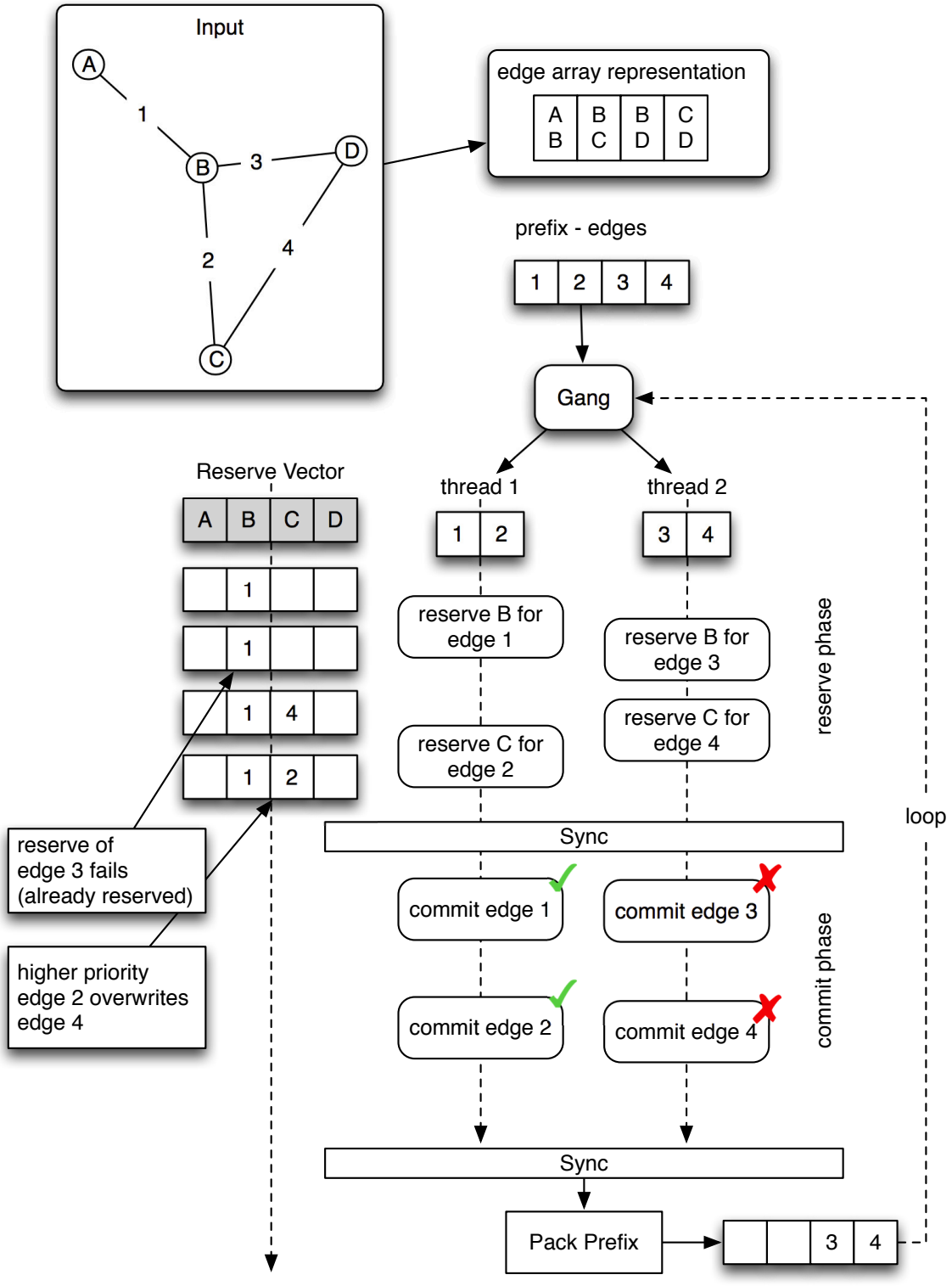


Figure 2: Deterministic reservations in action

2.3.3 Techniques for resolving data dependent conflicts

The reserve phase tries to reserve other elements in the sequence that are needed for non-commutative operations. This is done for each element in the prefix. In the commit pass, each element is only committed if the necessary components have been reserved. For example, many iterates processed in parallel may try to reserve the same vertex, but only the successful iterate operates on vertex. This reservation process is based on priority, usually represented by the original order of the iterates. Along with deterministic reservations, Blelloch et al introduce the concept of a priority reserve structure. The structure implements three operations:

- $x.reserve(p)$. A call to reserve will only reserve that location if p is smaller than the previous stored priority (lower is better). Any number of reserves commute, as the minimum value always wins out. In [6] this is implemented using an atomic compare-and-swap operation.
- $x.check(p)$. Returns *true* if p successfully reserved x. *check* doesn't commute with reserve, however, any number of checks commute with each other.
- $x.checkR(p)$. The same as *check*, however, the reservation is released if and only if p successfully reserved x. $x.checkR(p1)$ and $x.checkR(p2)$ commute if $p1 \neq p2$.

A program designer is required to adhere to two conditions when using the reserve structure with deterministic reservations:

1. Priority reserves are not called logically in parallel with either form of check.
2. All logically parallel operations must use an unique priority key.

It is currently up to the designer to enforce these rules, however, Blelloch et al suggest that it would be feasible to check these constraints at compile-time or runtime. This is currently the only area where we really give up functional transparency, and this should be addressed in future work.

2.3.4 Representing state

The reserve and commit phases are performed in two separate passes of the prefix. Hence, we need some way of maintaining state between them. Each iterate in the prefix has its own individual state, which is not shared with others. Changes to shared state are only performed once any data conflicts have been resolved. A mutable struct is maintained for each of k prefixes. These structs are reused with each new prefix selection, removing unnecessary memory operations. The contents of the struct are specified by the algorithm designer, and operated on by the reserve and commit phases.

2.3.5 Efficiently selecting a new prefix

After a prefix of iterates is processed, committed iterates have to be filtered out, and a new ordered prefix of the next n unprocessed iterates has to be selected. In a naive implementation, this would be a sequential step such as

```
newPrefix = (filter isCommitted prefix) ++ (take k remainingIterates)
```

where $k = n - numberCommitted$. In consideration of Amdahl's law [1], a repeated serial operation such as this reduces the program's upper limit of achievable parallelism. Blelloch et al compliment their deterministic reservation implementation with a highly parallel pack operation over an array and associated boolean flags. A similar implementation would be needed to be provided in a competitive DPH deterministic reservations implementation.

2.3.6 Why is it effective?

Deterministic reservations are effective as they are implemented using neither compiler or runtime techniques. It is merely a programmatic abstraction of an algorithmic technique. Implementing deterministic reservations in DPH would provide a powerful tool for solving problems which can be solved sequentially using an application of a greedy algorithm. The authors also note that the technique "gives more control to the algorithm designer and fits strictly within the nested-parallel framework" [6]. This technique of only processing a prefix of iterates is also discussed in [7].

2.4 Functional graph algorithms

There has been much recognition in the literature that writing graph algorithms in functional languages is hard, primarily due to the lack of side-effects [12, 25]. Furthermore, much of this research assumes a sequential execution model. King notes that while many graph algorithms can be implemented functionally without any state, they require at least some notion of mutable state to be efficient [25].

State becomes even more important when designing a parallel algorithm. Consider a comparison between depth-first search on a tree and graph respectively. When processing a tree, the branches can easily be divided up between processors to recursively traverse the tree. Nested data parallelism helps by improving the granularity and balancing the workload between processors. However, the properties of graphs pose much more intricate problems for parallel algorithms.

In a graph, series of edges can form cycles, and vertices may have multiple parents. Our processing threads need to share information between themselves about which vertices they have visited, to make sure they avoid redundantly visiting any vertices. A standard approach is to keep a collection (e.g. list or set) of the vertices visited at any given point. With each vertex visited, this collection is updated. Clearly, for efficiency, this collection has to be mutable, as it would be prohibitively expensive to continually create new collections with each added vertex.

2.4.1 Encapsulating state

King notes in [29, 28] that the key is to encapsulate any references to state, and this is the approach taken with the State Transformer (ST) monad in Haskell. Mutable operations can still be performed within a function, however they are encapsulated inside the $runST$ function. This means that while our function may internally modify state, our function is still externally side-effect free. This approach doesn't scale to multiple processors. The ST monad traces a computation thread to ensure that externally, it is still referentially transparent.

With multiple processors running in parallel, we have multiple computational threads. Hence, we need to look towards other methods to control stateful operations, such as deterministic reservations, to efficiently implement graph algorithms in a parallel functional language such as DPH.

2.4.2 Trading referential transparency for state

We can't always encapsulate state within a referentially transparent function. At some level, our concurrent threads need to share information, and waiting for a synchronisation step to do so is often not feasible if we want to extract sufficient parallelism. In [2], the concept of M-structures are presented as a mutable data structure for functional languages which provide implicit parallelism. In Haskell, this concept is implemented as an MVar [23, 19], and is used to provide gang parallelism (Section 2.2.4), for both DPH and deterministic reservations. The authors note that when combined with lazy, parallel evaluation, programs using MVars can have significantly more parallelism that their pure functional counterparts. This parallelism comes at the cost of referential transparency, and the benefits that come with it. They continue by saying that imperative constructs such as M-structures have their place in purely functional languages. This paper presents a similar argument. While deterministic reservations may not be a purely functional construct, they provide a useful abstraction for parallel programs that allow the programmer to write in a mostly functional style.

2.5 The relationship between determinism and parallelism

In a single threaded world, randomness generally doesn't rear its head unless asked and randomised algorithms can have excellent amortised runtimes [34]. Even if there is non-determinism present in the problem statement, a single processor will merrily proceed one step after another. It's another story when multiple cores are working together. It can make our programs hard to reason about and debug, and often they will fail only in infrequent edges cases.

While the challenges of non-determinism have been covered comprehensively in other literature [16, 17, 27, 38, 44], Blelloch et al note that determinism comes in degrees, and there is disagreement as to what degree of determinism is worth paying for [6]. Removing non-determinism can often also

restrict opportunities for parallelism, Blelloch et al suggest there is a trade-off between parallelism and the benefits of determinism, namely simplicity, programmability and debug-ability.

This thesis requires an understanding of two types of determinism, namely:

- *Internal determinism*, in which key steps of the program are also deterministic.
- *Functional determinism*, where all components of a program are independent and safe to run in parallel, due to the absence of side-effects.

Blelloch et al advocate for internal determinism as a 'sweet-spot', giving a good balance of the trade-offs suggested above. A large part of their paper is dedicated to the discussion of how to effectively implement internal determinism for parallel operations. Informally, a program/algorithm can be said to be internally deterministic if there is a unique trace through the program for any unique input. This idea can be modelled with a directed a-cyclical graph representing the possible flow through operations of the program. Internal determinism is a stronger requirement than external determinism, which only requires a particular output for every distinct input. The paper proposes that internal determinism is the "sweet-spot" for certain classes of nested data parallel programs where there is no non-determinism in the problem statement. They develop certain data-structures and algorithmic techniques built on this principle, such as deterministic reservation, that are heavily used in their implementations [6].

As discussed in 2.4.2, Haskell provides the guarantee of functional determinism within pure functions (excluding certain operations, such as unsafePerformIO, of course). The absence of side-effects is great for determinism. Essentially, if our program produces the correct result sequentially, then correctness won't be a concern if we scale to multiple-cores. As always, there is a trade-off. By eschewing mutable data structures, our parallel program may actually run slower than the sequential one! Each thread needs to keep it's own copy of any structure that it mutates, and these changes need to be merged together in a synchronisation step. This can be very expensive, even with the advanced functional data structures used in Haskell such as finger-trees [18].

Any program that conforms to functional determinism can also be classed as internally deterministic. However, the class of internal determinism programs has a more general definition. in [6], Blelloch et al build on the work of [35] to define internal determinism in terms of the flow of execution through the program, represented as a directed acyclic graph (DAG). A program is internally deterministic if for a particular input, the same path is always traced through the DAG. Consequently, The result of any two parallel operations must not depend on the sequence that the operations were performed in. That is, (and this is the key to deterministic reservations), any two parallel operations must commute.

In [2], Barth et al note that commutativity (and atomicity) are useful considerations when considering operations on mutable data structures in parallel. Their concept of an M-structure, also discussed in 2.4.2, takes advantage of commutativity and atomicity to produce determinate results under regardless of operational execution order.

2.6 Benchmark algorithms

To support their implementation of deterministic reservations, the Problem Based Benchmark Suite (PBBS) paper implements five algorithms which take advantage of deterministic reservations [6], namely:

1. Spanning Tree
2. Minimum Spanning Tree
3. Maximal Independent Set
4. Delaunay Triangulation
5. Delaunay Refinement

Only 1 and 3 have been provided using the Haskell implementation of deterministic reservations at this stage, and hence the literature surrounding these problems is discussed below. The two Delaunay algorithms presented in [6] both use the underlying concept of reserve and commit phases over iterative prefixes which underly deterministic reservation. However, neither use the control structure as it is presented in the paper. For this reason, along with the inherent complexity of the algorithms, they are not discussed in this thesis, or developed as benchmarks in Section 3.

Spanning Forest Algorithms A general spanning forest algorithm and a minimum spanning forest algorithm are specified as problems in the PBBS. A spanning forest is a generalised form of a spanning tree, in that it also covers unconnected graphs. Sequentially, a spanning forest can be generated greedily by processing the edges in an arbitrary order, i.e. depth-first search, then returning the collection of spanning trees for each isolated component.

The Shiloach-Vishkin algorithm for finding spanning trees in parallel is presented in [41]. It is based on a concurrent-read, concurrent-write architecture, and runs in $O(\log n)$ steps, where n is the number of vertices. However, the minimum spanning tree problem is more specific, and the PBBS uses a variant of Kruskal's algorithm described in [26]. Kruskal's algorithm sorts the edges by weight, then greedily adds them one by one as in the standard spanning forest algorithms. A optimisation also is implemented where only the smallest k edges are sorted, as not all edges will be included in the final spanning tree.

Both the general and minimum spanning tree implementations in the PBBS use deterministic reservations to make sure that vertices haven't already been added to the tree. This technique is discussed in Section 2.3.

The PBBS uses an optimal disjoint-set data structure in its spanning forest algorithms. An array is used as the underlying data structure, with vertices represented by integers. In an array a, the parent of a vertex v is given by $a[v]$.

Max Independent Set *For a connected undirected graph $G = (V, E)$, return $U \subset V$ such that no vertices in U are neighbours and all vertices in $V \backslash U$ have a neighbour in U.*

The max independent set (MIS) is another problem on graphs solved sequentially using a greedy algorithm. However, in a parallel implementation, there is a chance that two processors may add adjacent vertices to the set. The PBBS implementation uses deterministic reservations to make sure no two adjacent vertices are added in each pass.

In [7], a variety of approaches to parallelising greedy sequential MIS algorithms are explored. Each approach is based on giving vertices a priority, defining the order in which are processed. Vertices are added to the set as soon as they have no higher priority neighbours. Once a vertex is added, it and its neighbours are removed from the graph. None of its neighbours can be added in the future, so they are removed from the array. The authors state that a naive implementation will require $O(m)$, (where m is the number of edges) work on each step. This leads to $O(m \log^2 n)$ work. Optimisations are presented, including deterministic reservations, used in the PBBS implementation.

Another parallel approach to the MIS problem is Luby's randomised algorithm [32]. It runs in $O(\log |E|)$ time on $O(|E|)$ processors of a CRCW-PRAM. However, in [7], the authors note that with a modest number of processors, it is difficult for these parallel algorithms to outperform a simple sequential greedy algorithm.

2.7 The need for a benchmark suite in DPH

Deterministic reservations were presented as part of the initial implementation of the Problem Based Benchmark Suite (PBBS) [42]. This thesis presents work that could contribute towards a set of Haskell, and eventually DPH solutions for the problems presented in the benchmark. DPH already has a set of benchmarks, accessible at [33]. However, they are out of date and many, such as the barnes-hut presented in [39] are unusably slow. Implementing the PBBS for DPH would provide a solid set of robust benchmarks that were, from the start, designed with nested data parallelism in mind [6]. Solutions to the suite problems would also present a set of examples for prospective DPH users, giving an indication of the suitability for different problems in the suite.

The PBBS is

> "...designed for comparing parallel algorithmic approaches, parallel programming language styles, and machine architectures across a broad set of problems."[42]

Looking forward to this eventual possibility, the solutions to the spanning tree and maximal independent set problems presented in this thesis have been designed within the PBBS problem specifications. This includes the input and output formats specified for each particular problem.

2.8 Evaluation framework

The goal of this thesis is to investigate and evaluate the suitability of deterministic reservations for inclusion into DPH. Part of this evaluation will be quantitative, and the other qualitative.

Quantitative measurement

The quantitative component will focus on the speedup factor produced over multiple cores. This will be compared to the highly optimised version presented in the original implementation [6].

In the suite, each problem specification is accompanied by

- Input and output formats
- Input generators
- Code for checking the correctness of the output
- Timing code
- Both sequential and parallel baseline implementations

In this thesis, the first three are used in my testing and evaluation framework. I integrate the timing code from previous DPH benchmarks into our solutions. Taking the same approach as Blelloch et al, my benchmarks are only timed once the input files are loaded into memory in their respective representations. The input files of 10 million elements are over 600mb, and including the reading from disk would obscure results on IO-bound machines. Only the deterministic reservations function, $speculative\_for$, is included in the timings for a problem. As mentioned, this allows us to focus on the performance of deterministic reservations.

Benchmarks are run on a machine with 2 $\times$ 2.0GHZ Intel 4-core E5405 Xeon Processors, a 1333MHz bus, and 8GB of main memory. This allows testing of benchmarks up to 8 cores. The PBBS baseline will also be run on the same machine for each algorithm and input to give a reference result.

The use of tools such as Threadscope [22], and Haskell's heap profiler can provide useful numeric insights into the efficiency of developed programs [37]. I used Threadscope to look at the load balancing characteristics of my implementation and detect unexploited parallelism.

Qualitative Analysis

Deterministic reservations is designed as an inherently imperative control structure. Much of the work in this thesis involves adopting deterministic reservations for the functional language Haskell. Assuming that similar multicore performance to the original implementation can be achieved, the success of this adoption depends on the analysis of inherently qualitative aspects of the work. The following will be analysed:

- Is the control structure easy to use?
- How does deterministic reservations fit with the idiomatic code style of Haskell?
- How much work does deterministic reservations save the user?
- How much of my prototype implementation is adaptable for DPH?

3 Contributions

In this section, I present my contributions towards the integration of deterministic reservations into DPH. In the process I present the challenges faced by such an integration, and provide the solutions as the main contributions of this paper. In my approach, I utilise core components from the DPH package to present a prototype of deterministic reservations in Haskell. I underpin my approach using Leshchinskiy's Vector library to represent data structures and handle destructive updates. The vector library also provides the array data structures underpinning DPH, and hence, much of my prototype work is directly applicable to future work integrating deterministic reservations into DPH. I also take advantage of DPH's gang parallelism for the same reason. These dependencies are illustrated in Figure 3 on page 19. It is worth noting that I have not considered some of the restrictions and features of DPH in the prototype, such as vectorisation and parallel array marshalling. This allows me to focus on the core concerns related to the adaptation of deterministic reservations in a functional language paradigm.

In this Section, and Section 4, I present code snippets from my implementation. The full source code, along with a commit history, can be found at http://bitbucket.org/jamolloy/dph_thesis

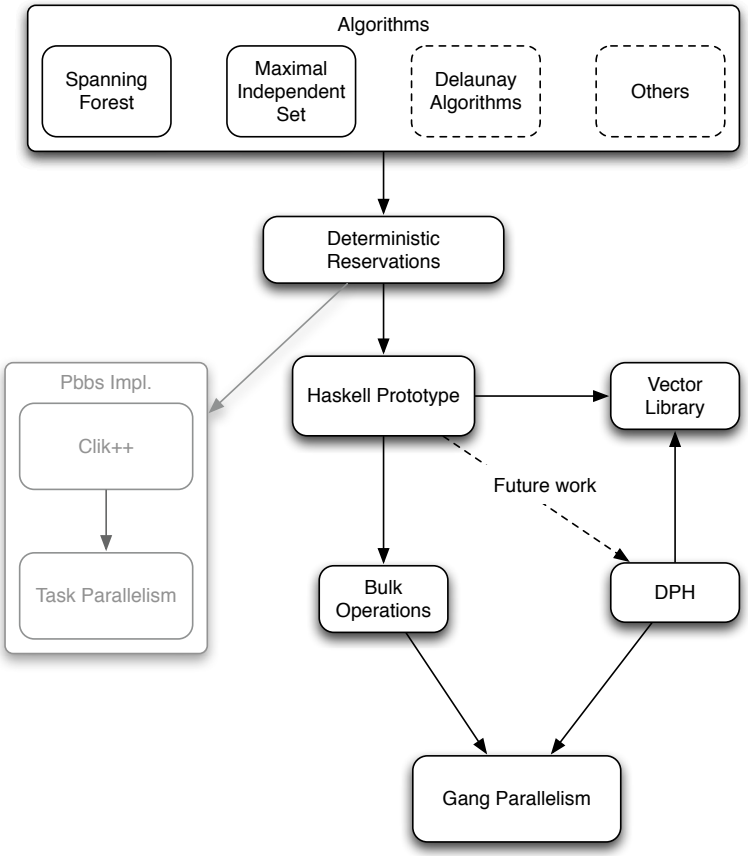


Figure 3: Deterministic reservations in context

3.1 Deterministic reservations

I have taken Blelloch's concept of deterministic reservations, and adapted it to work in a lazy, functional programming paradigm. This requires substantial changes to Blelloch's model, which I believe improve on the original. We turn his programming pattern inside-out to expose the previously implicit properities of deterministic reservations. The significant contributions are:

- Our *speculative_for* function has a pure functional interface. All mutating operations are specified by the user, but performed internally within the pure control structure.
- In my model, the distinction is made between a step's private state, and state shared with other steps. This is important for identifying operations over structures which need commutativity analysis. This requires a new deterministic reservations interface, and for deterministic reservations, it signifies a shift from an algorithmic pattern to an API.
- Implicit requirements of deterministic reservations are encoded explicitly using Haskell's type system, to better enforce the relationships between components, and reduce the quantity of boiler-plate required on the users behalf.
- The use of functional dependencies and types to express deterministic reservations output in terms of only the algorithm, the immutable input and the prefix size.

As I discussed in Section 2.4, writing graph algorithms in pure functional languages is challenging. This paper presents significant contributions in this area. With the need to preallocate mutable structures and perform destructive updates, adapting deterministic reservations to a pure functional language requires going against the grain of the language. It is almost as if there are two Haskell worlds, the pure functional land, and the lower level of mutable structures and thread control. My approach to deterministic reservations hides this complexity from the user, while still achieving excellent performance .

My approach to deterministic reservations also makes use of strict semantics where necessary. When designing parallel algorithms in a lazy language, one often has to consider evaluation semantics to ensure that thunks from sequential steps aren't left unevaluated for parallel cores. Disregarding this can introduce both non-determinism and performance issues. With my Haskell based deterministic reservations, the appropriate evaluation semantics are enforced internally within the control structure.

3.1.1 Representing steps in Haskell: The DRStep typeclass

In deterministic reservations, the reserve and commit passes exhibit different behaviour depending on the algorithm being designed. This ad-hoc polymorphism is provided by the type class *DRStep* (Figure 4 on page 21). I make a significant change from the original PBBS implementation, separating a step's private and shared state into two separate data structures. The *DRStep* typeclass

reflects this, along with the mutable dependency between the two data structures. Separating private and public state gives multiple improvements over the original deterministic reservations. It forces the algorithm designer to separately declare components that are shared between steps. This is important, as operations over these structures need to be considered for commutativity relationships.

```
class DRStep step input mut output  | mut → step, step → mut
                                    , step → input, step → output
                                        where
    newStep :: ST s (step s)
    reserve, commit :: input → mut s → step s → Int → ST s Bool
    buildSharedStructures :: input → ST s (mut s)
    returnResult :: mut s → ST s output
```

Figure 4: DRStep typeclass

Our typeclass defines five operations, the two original deterministic reservations operations, *reserve* and *commit*, as well as three new definitions, *newStep*, *buildSharedStructures*, and *returnResult*.

- *reserve* and *commit* both share the same type definition. My definition is different to that presented in the original deterministic reservations as immutable input, private step state and shared state have been separated. This allows prefix steps to be represented as unboxed arrays in the future, as discussed in Section 4.4. Unboxed vectors can't contain references to other data structures, as they can only contain elements of a primitive type such as *int*, *bool* or *pair*. An input index is also passed as a parameter, referring to the index of the original array element being processed in that step.

- *createStep* provides the concrete constructor for each step, initialising the private mutable state using STRefs. The PBBS implementation uses a copy constructor to create an array of prefix steps. I take a different approach, applying a ST monadic action for each prefix state. This operation takes no parameters, as steps are reused with each new selected prefix. Any state initialisation is done at the start of the reserve step. In the DPH implementation, we will move to a non-parametric unboxed array representation for our steps, and this constructor function will no longer be needed.

- *buildSharedStructures* encapsulates the creation of the mutable shared structures. The immutable input is passed as a parameter as the mutable data structures may depend on some property of the input, such as size.

- *returnResult* allows the algorithm designer to specify the output structure that deterministic reservations returns, whether that be a boolean result or the maximal independent set. My

approach couples the result of *speculative_for* with the operation itself, for a particular input and algorithm, the output should be deterministic. In this way, the responsibilities of deterministic reservations are naturally expressed within the type system.

3.1.2 The *speculative_for* function

In this Section, I present pure functional interface to Blelloch's *speculative_for* operation. I take advantage of type classes, functional dependencies and type equality constraints to achieve this more rigorous definition. In the previous section, I described how my DRStep typeclass presents an interface for the original reserve and commit operations, and extend it to include other operations such as *buildSharedStructures* and *returnResult*. My extended *speculative_for* wraps all these operations so that its behaviour is defined in terms of only the algorithm and its input.

```
speculative_for :: (G.Vector v a, DRStep step input mut output, v a ~ input)
        ⇒ Int → Algo step → input → output
```

The *Algo a* type is defined as

```
type Algo step = forall s. ST s (step s)
```

The reader may notice similarities to the type of *runST* in the State Tranformer monad in our *Algo* type. I take this approach to allow the user to specify the stateful algorithm they wish to run from a pure context. The stateful operations are performed within the *speculative_for* function. I also place a constraint on the input to *speculative_for*, stating that the input must be some kind of iterable vector. This constraint enforces the behaviour of deterministic reservations, which is at its core defined as a parallel speculative loop over an array.

In Figure 5 on page 24, the key differences between Blelloch's model and my own are illustrated. In Blelloch's model, the shared structures and prefix steps are built outside the *speculative_for* loop, and passed in wrapped up in a struct. After *speculative_for* is run, it is then up to the programmer to process the mutable structures for a result.

I take a much more integrated approach. As discussed in Figure 4 on page 21, all of the auxiliary operations mentioned are performed within *speculative_for*. This is a significant step foward in presenting a functional interface to deterministic reservations, and it means that all the operations over mutable data structures are handled internally inside *speculative_for*. This approach results in a determinsitic reservations where the output is properly deterministic on the three inputs; the granularity of the speculative loop prefix, the algorithm and the input. The same can't be guaranteed in Blelloch's model.

3.1.3 Extending the DRStep typeclass - Defining explicit relationships

As part of my approach to deterministic reservations, I broaden the interface to require the user to specify, the algorithm, input, shared structures, and output format. Our *speculative_for*

function then takes only the immutable input as a parameter. This is a very different approach to Blelloch's, where the algorithm, input and shared structures are combined first, passed into the *speculative_for* function and processed into a result afterwards.

There are five key components to a deterministic reservations based algorithm:

1. Input, which should be immutable in a functional approach
2. The private state of a prefix step
3. The state shared between steps
4. The algorithm, represented as two operations over the input, private step state, and shared state.
5. Converting the mutable structures processed in the algorithm into an immutable output

In Blelloch's approach, most of these elements were implicitly implied outside the deterministic reservations interface. In my approach, each of these elements are explicitly represented in the type of the *DRStep* typeclass. This is illustrated in Figure 5 on page 24. In the expanded model, the user must spell out each component for deterministic reservations. This approach has merit. All of these components are intrinsic to any application of deterministic reservations to a problem. My model just forces the user to make a distinction between the different parts, and deterministic reservations handles the relationships between them.

The DRStep typeclass asserts that each algorithm using deterministic reservations will have its own type of shared state, and for simplicity, this will be unique to the algorithm. In other words, as declared in our *DRStep* typeclass, the algorithm step type and shared state are functionally dependent on each other. Two other dependencies are also specified from algorithm type to both the input and output types. An algorithm, represented as a step must have a particular type of input and output.

These relationships, which are not defined explicitly by Blelloch, play an essential role in applying deterministic reservations to a problem. When combined with the type definitions of each function in the type class, they act as constraints, guiding the programmer in the application of deterministic reservations. In Figure 4 on page 21, the relationships are represented using functional dependencies.

In Section 2.2.1, I discussed DPH's vectorisation step, part of which converts parametric types into non-parametric representations using multiple arrays. In a DPH implementation of deterministic reservations, this technique can be used to represent prefix steps array in an unboxed fashion. A step is made up of its private state, and the mutable collections shared between all the prefix steps. The array of private states can be vectorised into a series of unboxed arrays, while retaining references to the mutable shared states. This is illustrated in Figure 6 on page 25. With this

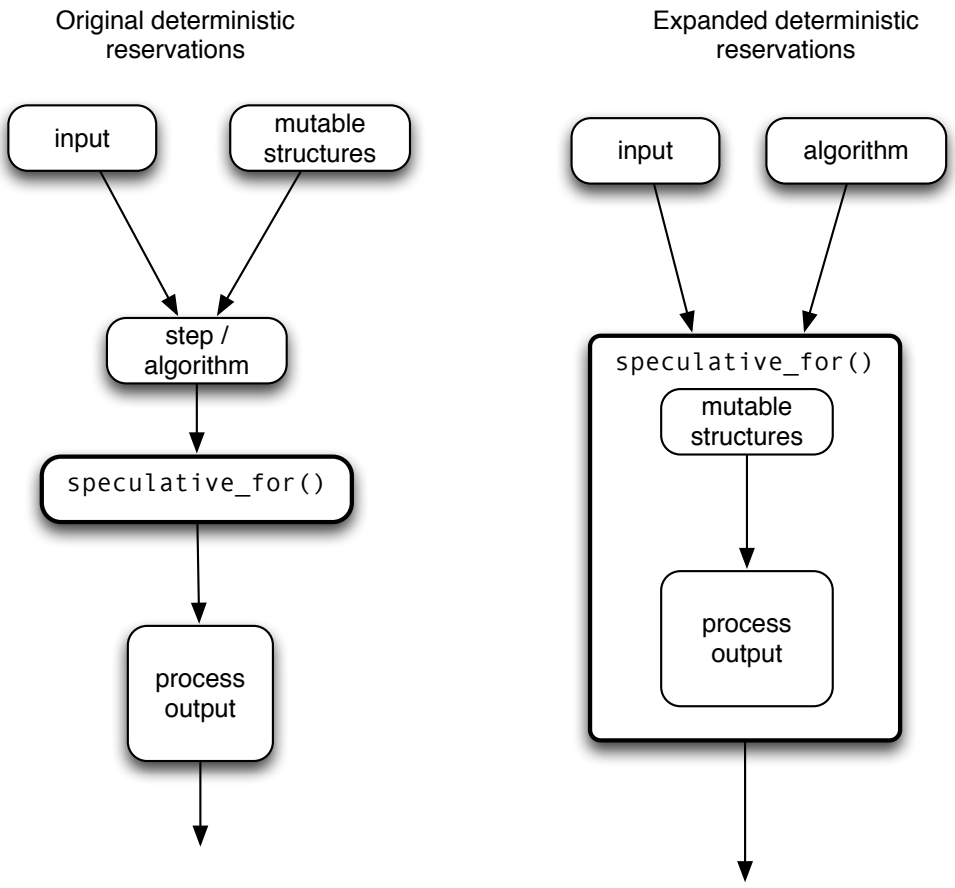


Figure 5: Different approaches to deterministic reservations. Blelloch's (left), mine (right)

approach, we can remove the need for STRefs, which require additional pointer traversals as they are a boxed construct.

3.1.4 Taking advantage of higher-order functions

The low-level work is done in the bulk module. The bulk function sends off work to the gang, passing through references to all the mutable data structures, along with the thread id and number of threads. The first step is to partially apply an operation (*reserve* or *commit*) to the shared mutable data. This hides shared data until it is needed, inside the operation itself.

I then use another partial application, this time of the $bulk'$ function, to package all the data and functions every gang member will need to operate on its slice of the prefix vector. In $bulk'$, I take advantage of the vector library's stream fusion to efficiently map our reserve or commit operation over the prefix.

Not only does the operation have to be mapped over each prefix iterate, but the results need to be tracked. In idomatic Haskell, the results of reserve or commit would be returned as an array from our *mapM* operation. Instead, I use a pre-allocated mutable array of boolean flags to reduce memory allocation overheads. $Freserve$ and $Fcommit$ handle both evaluation and updating of

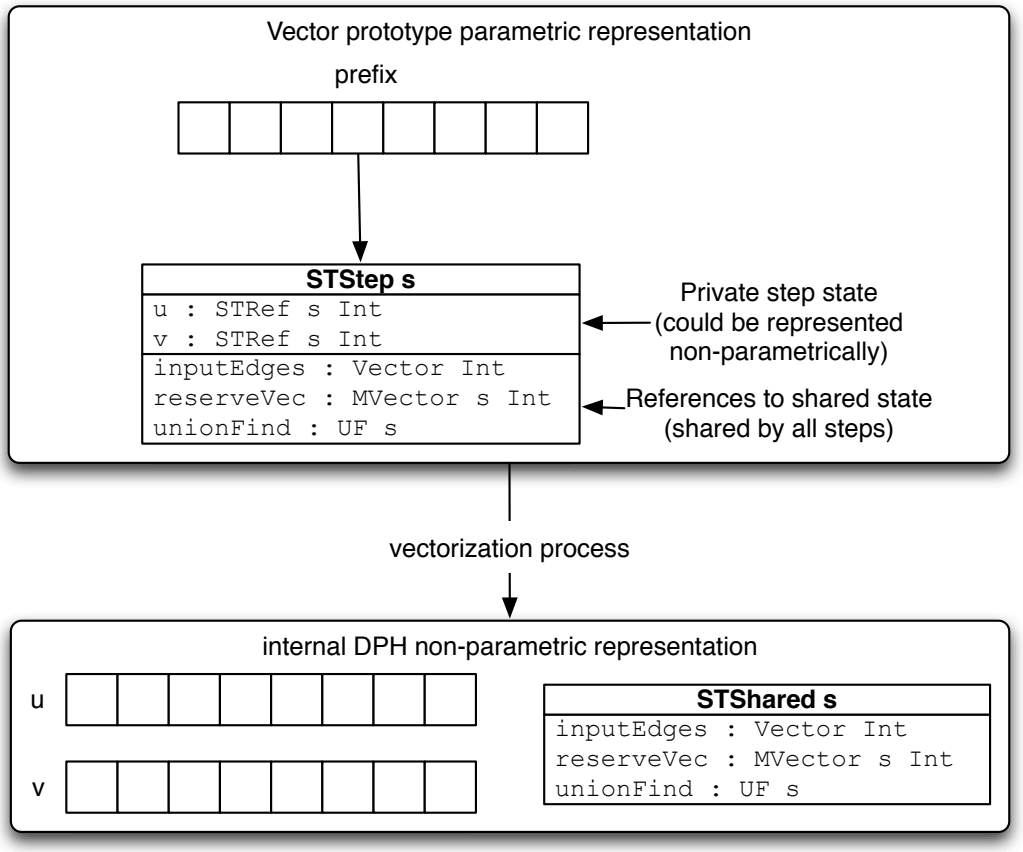


Figure 6: A non-parametric representation of DRSteps

these flags, leading to a neat encapsulation of all the conditional logic and mutation performed by *speculative_for*. I propose that this presents a neater and more understandable solution than the PBBS implementation, resulting from the use of type classes, partial application and higher-order functions.

3.1.5 Unleashing the gang

To provide parallelism, I utilise a simplified version of the gang parallelism framework implemented as part of DPH. A reference to the prefix array is passed to each gang member, who calculates the slice of the array which they will operate on. The gang member then sequentially maps either the reserve or commit operation over this slice, depending on the phase. The number of gang members is generally based on the number of available cores.

To extract as much parallelism as possible, the work needs to be equally divided among the gang members. Processing some iterates might take longer than others, so optimal load balancing won't necessarily be achieved just by dividing the input array into equal parts. The original implementation uses work-stealing parallelism to help balance out the loads between cores [6]. Our results in Section 4 show that I actually achieve good load distribution between cores, even

without work-stealing.

At some point, referential transparency has to be broken to let our operations perform side-effecting actions on shared data structures. The gang infrastructure makes this happen. It removes the restrictions of the ST monad, and runs the *bulk′* function in the IO Monad. The user will never see this, and naively assume that their algorithm is still referentially transparent in the ST monad. However, this is only the case if they have correctly defined their reserve and commit operations in terms of commutative operations. I would argue that this is much less taxing than managing race parallel operations at the thread level itself.

DPH's gang parallelism doesn't support work-stealing. In his presentation of deterministic reservations, Blelloch takes advantage of work stealing as part of the Cilk++ task parallelism. Work-stealing comes with an additional overhead, which depending on the scenario, may or may not result in a improved multicore performance through better load distribution. My results in Section 4 show that a comparable speedup can be achieved even without load balancing.

3.1.6 Sometimes it pays to be strict

Haskell provides powerful lazy evaluation semantics which normally allow the evaluation of computations only when needed [21]. A thunk is allocated representing the computation, and its evaluation is delayed until needed by the program. This is possible due to the purity of Haskell's functions. However, in deterministic reservations laziness needs to be managed carefully. Laziness introduces unnecessary overheads when we know we are definitely going to need the result of the computation later. We use forced strictness in the following cases to improve performance:

- When writing the result of *reserve* and *commit* operations to the mutable keep array. We know we are going to access these results almost immediately. This is particularly important, as we want to avoid doing any significant memory allocation in parallel phases of the program.
- Calculating the new prefix indices in packPrefix

3.1.7 Packing the prefix

My current prototype uses a linear pack to remove committed iterates from the prefix, as illustrated in Figure 7 on page 27. I manage to achieve good serial performance using $filter$ and $zipWith$ operations. I take advantage of stream fusion, discussed in Section 2.2.3, over vectors to achieve this. Both $zipWith$ and $filter$ operations are supported in DPH, and hence this model should be easily transferable. DPH provides parallel implementations of these operations and the vector fusion I utilise, hence I would expect a reasonable parallel speedup with DPH.

```
let remainingIndicies = U.filter (> 0) $ U.zipWith (*) finalKeep prefixIndicies
    let numCommitted = roundSize - (U.length remainingIndicies)
    let !newPrefix = remainingIndicies U.++ (U.enumFromN offset (min (n-offset) numCommitted))
```

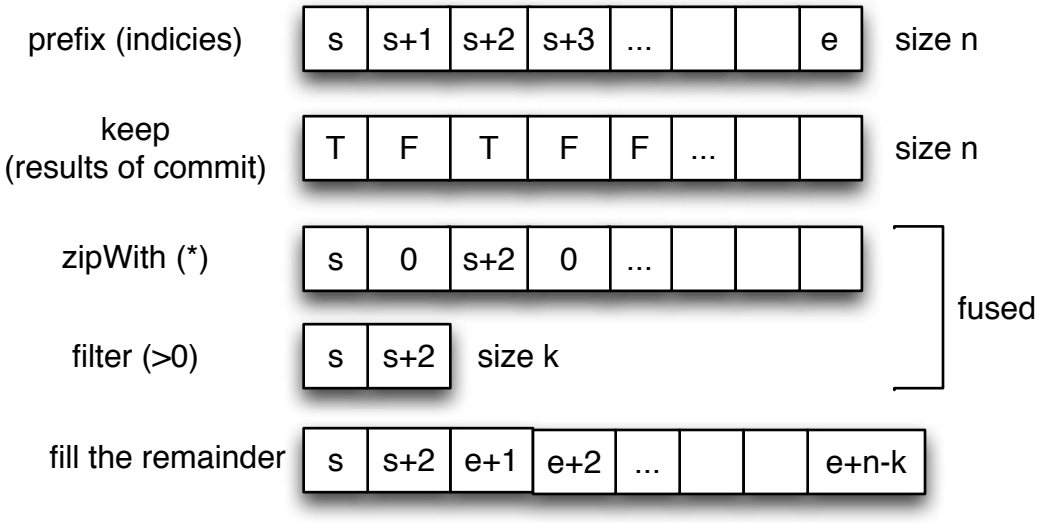


Figure 7: Sequential packPrefix over vectors

Depending on the algorithm, and the prefix granularity, on average around 10% of iterates will need to be reprocessed in the next prefix selection [6]. With each prefix loop, a new vector prefix is being created. Ideally, unnecessary allocations during the deterministic reservations process should be avoided.

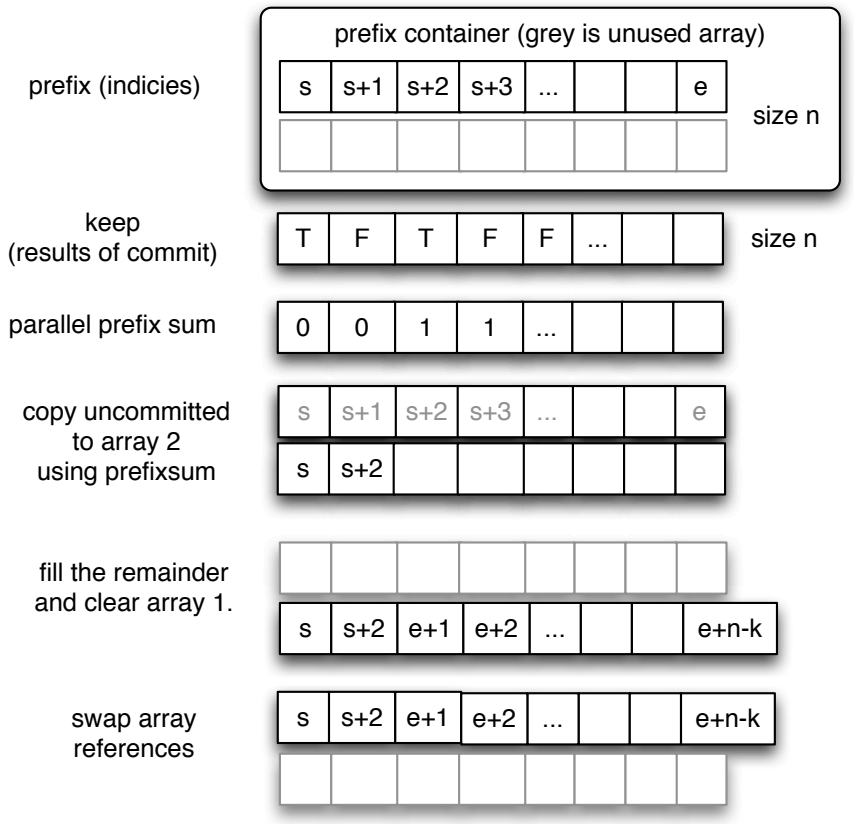


Figure 8: A parallel packPrefix using mutable arrays

As mentioned in Section 2.3.5, the original implementation uses a highly parallel low-level implementation, modelled in Figure 8 on page 28. As the prefix remains the same size through out the execution, we can pre-allocate our arrays to save memory allocation work when packing our prefix. If a prefix is of size k, then 2 arrays of size k are allocated. The process of packing the array is as followed:

1. To work out the offsets of to-pack elements, a parallel prefix sum is performed over the results of the commit.

2. Uncommitted iterates are then copied into a contiguous section at the start of second empty array.

3. The remainder of the second array is filled with new elements

4. The initial array is cleared, and the arrays swapped so that the second array becomes the primary.

The first three steps can all be done in three parallel passes. The final step is a small constant operation. The current serial implementation is very fast, due to the fusing of the zip and filter

operations over an unboxed array. However, there are garbage collection overheads due to the continual reconstruction of the prefix. To minimise the garbage-collector overhead, we want to pre-allocate all memory that used in our deterministic reservations operation. Implementing a mutable packable prefix in the future is an important step towards this.

3.2 Using deterministic reservations

In this section, I present implementations of two PBBS benchmarks utilising my adaptation of deterministic reservations. For each, I take the same algorithmic approach as the PBBS solutions.

3.2.1 Spanning Tree

My spanning tree solution is an almost literal translation of the PBBS original. I make use of the State Transformer (ST) monad to reflect the imperative nature of both the reserve and commit operations. Both reserve and commit operations are more verbose than their C++ counterparts due to the need to explicitly read and write from our mutable arrays. An edge array is taken as input, with each edge represented by a pair of vertices, which are in turn represented as integers.

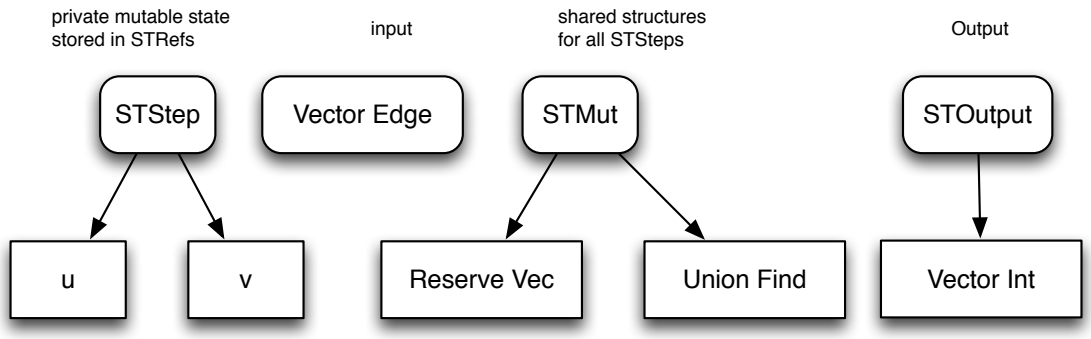
$$\begin{aligned} &type\, Vertex = Int \\ &type\, Edge = (Vertex, Vertex) \end{aligned}$$

I make a distinction between the step's private and shared data and reflect this in the data constructors, illustrated in Figure 9 on page 30. As the end goal is to provide an implementation of deterministic reservations for DPH, we want to rework deterministic reservations to achieve a more concrete separation between private and shared data. This will allow the vectorisation of the STSteps.

To accompany our spanning tree algorithm I implement a high performance, mutable, union-find data structure in the ST monad which closely follows the PBBS implementation. However, as with $speculative\_for$, a recursive find function is used to navigate up the tree. I represent our disjoint set using a mutable unboxed vector of integers. The root vertex of any set has the parent -1. Path compression and ranking are also implemented to give an optimal complexity of $\Omega\left(\alpha\left(n\right)\right)$ [14]. Care is also taken to ensure that no thunks are left unevaluated, when linking or finding vertices.

3.2.2 Maximal Independent Set

In the standard greedy maximal independent set (MIS) algorithm, vertices are taken in order and added to the set when none of their neighbours are already in the set. A parallelisation of this




```
data STmut s = STmut (Vector Edge) (ReserveVec s) (UF s)
data STStep s = STStep (STRef s Vertex) (STRef s Vertex) (STmut s)
type STinput = U.Vector Edge
type SToutput = U.Vector Int
```

Figure 9: Spanning tree representation

greedy graph algorithm is perfectly suited to deterministic reservations. I implement Blelloch's MIS algorithm [7] below.

The input graph is modelled as an adjacency array, internally represented as a vector of unboxed vectors. Each index of the initial vector corresponds to a vertex, and its element points to a vector containing the indices of all connected vertices. The array is built and frozen before we run our MIS algorithm. It is worth noting that deterministic reservations should always iterate over an immutable input structure, and only use mutable data structures internally.

The are numerous ways to represent the current state of our algorithm, however, fundamentally, only three disjoint sets need to be represented:

- Vertices in the MIS
- Vertices not in the MIS
- Vertices yet to be processed

I use an unboxed vector of integer flags, where an integer value corresponds to one of the above states. This is the only shared mutable structure between the prefix steps in the algorithm. These components and their relationships are illustrated in Figure 10 on page 31.

As a two dimensional array with variable length components, an adjacency array, illustrated in Figure 11 on page 31, is more typical of the type of data structure we would represent using DPH than the edge array used in the previous example. DPH can represent and perform parallel operations over such a structure very efficiently. However it doesn't provide the tools to efficiently calculate a result such as the maximal independent set, where the array iteration order impacts the result.

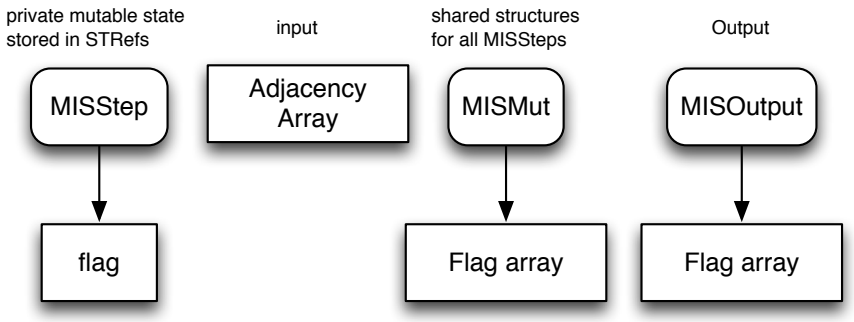


Figure 10: Maximal independent set data structures

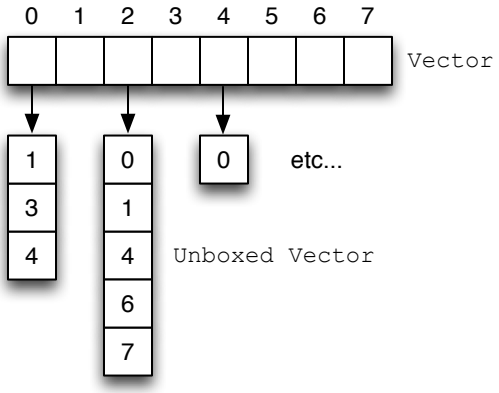


Figure 11: Adjacency array representation

The significant work is done in the reserve phase. Neighbours are ranked with a priority based on the order they are processed. For a particular vertex, all its neighbours of a higher priority are checked. There are three cases:

1. A neighbour was inserted into the graph in a previous iteration. It is inferred that our current vertex cannot be in the MIS, and it is marked as such.

2. A higher priority neighbour has neither been added or excluded from the MIS. It is inferred that it is being processed in parallel with the current vertex, and we leave our vertex to be reprocessed in the next round.

3. None of the neighbours have been inserted and the vertex is inserted into the graph.

The reserve function always returns true, as we always want to commit the result of this neighbour checking process, so that other vertices in the next prefix loop can see the result.

The commit function is simple in this instance. The result of the reserve operation is written to the shared structure, and true is returned if a decision was made on the vertex's MIS membership. After all elements have been committed, through repeatedly selecting prefixes, the flag array will contain the result. Vertices will be marked either as in or out of the MIS.

4 Evaluation

For each graph algorithm, three different types of graphs were used

1. random graph
2. grid graph
3. rMat graph

To test an algorithm, each graph was run for 1 through to 8 cores, and each test repeated 3 times, with the results aggregated. The same size input as the PBBS specifications (10 million vertices) was used.

4.1 Spanning Tree Results

The spanning tree benchmark achieved a good speed up of 4.2 over 8 cores for this algorithm (Figure 13a on page 33). The achieved speedups were also consistent over different inputs (Figure 12 on page 33).

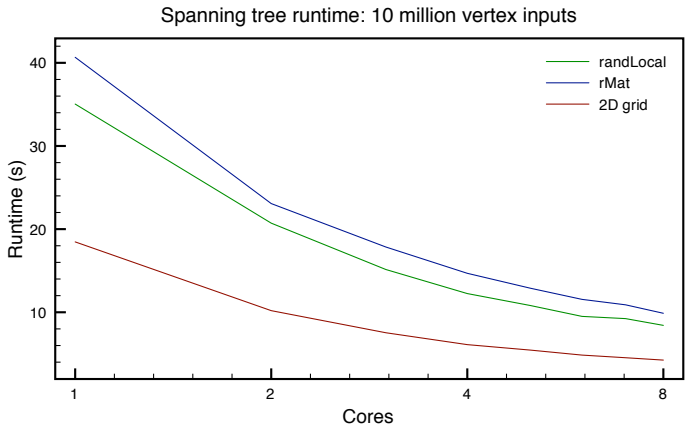
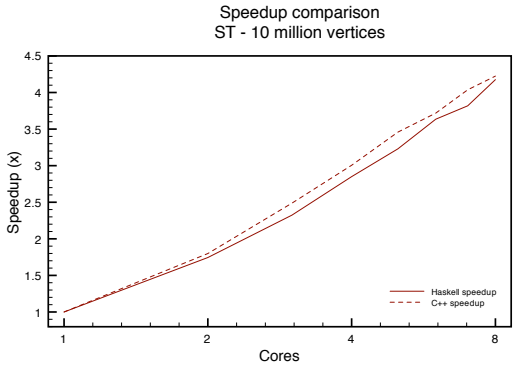
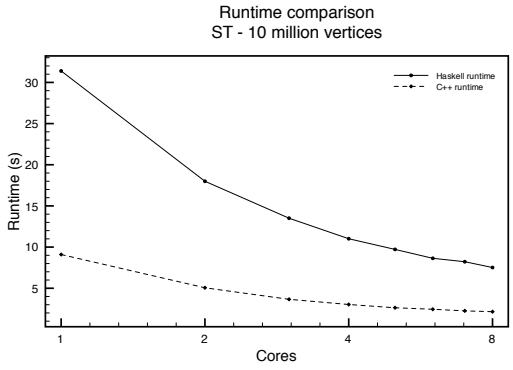


Figure 12: Performance over different graph types



(a) ST speedup comparison



(b) ST runtime comparison

Figure 13: Spanning tree results, compared to original C++ implementation.

4.2 Maximal Independent Set Results

The MIS benchmark was tested with an input sized 10 million, I achieve a speed up of nearly 4x over 8 cores for this algorithm (Figure 15a on page 34). Of particular note is the excellent speedup with 2 cores, of 1.85x. The achieved speedups were also consistent over different inputs (Figure 14 on page 34). When utilising 8 cores, runtime increases. This is not a defect of my approach, but results from system and GC overheads on the last core of the test machine.

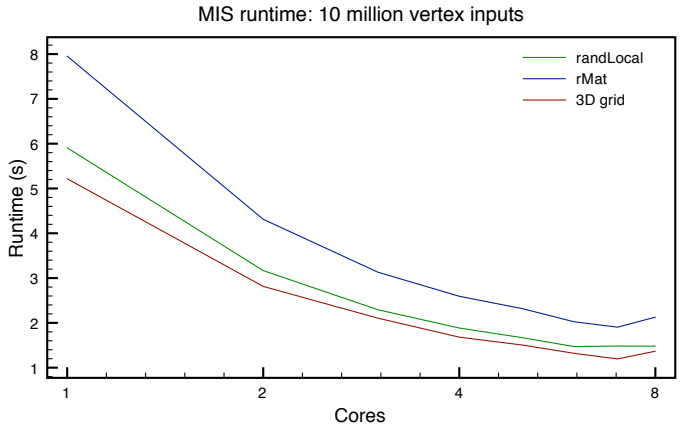
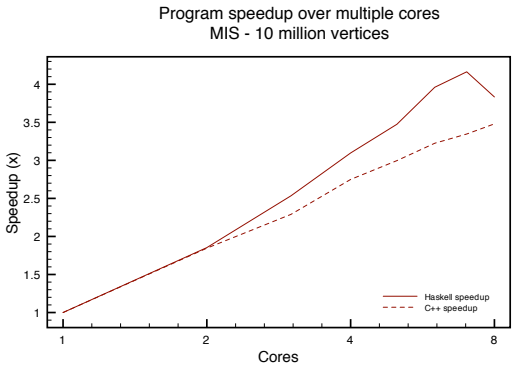
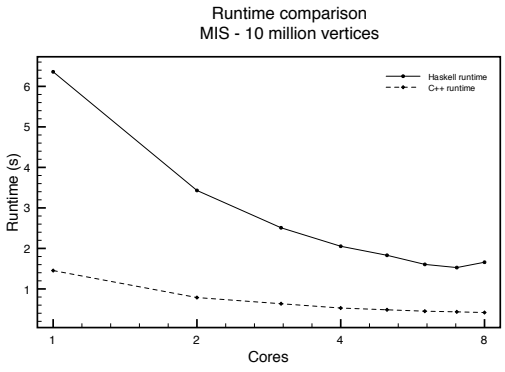


Figure 14: Performance over different graph types for MIS algorithm



(a) MIS speedup comparison



(b) MIS runtime comparison

Figure 15: Maximal independent set results, compared to original C++ implementation.

4.3 Correctness

As part of the PBBS testing framework, the output of an algorithm is checked for correctness after each test execution. The results of this process showed that both algorithms built on our deterministic reservations implementation are correct, whether on one core or eight. The output of an algorithm built on deterministic reservations gives the same result, regardless of the number of cores it is executed on, assuming the prefix size is the same. Our implementation adheres to this rule.

4.4 Absolute performance

Figures 13 and 15 show that while my deterministic reservations approach scales well, it is consistently 4-5 times slower that Blelloch's original C++ implementation. It should be noted that this is still a good result. As mentioned in Section 3, our prefix steps are represented within a boxed vector, and use STRefs for mutable state. There is a significant overhead to this representation, in both GC and pointer traversal overheads compared to an unboxed representation. Furthermore, operations on these steps occur within the inner-most loop. Moving to an unboxed representation should easily account for much of the performance difference between my implementation and the original. We leave this as future work.

4.5 Load balancing

Load balancing is key to achieving good parallel scalability. When we apply more processing units to a problem, we want to divide the work up evenly between them. In embarrassingly parallel problems, this is easy. We just divide the array up between processors. Structures such as graphs don't divide as nicely.

As visible from my results (Figures 13a and 15a), load balancing hasn't had a large impact on the speedup, however, if we scaled our algorithms to 32 or 64 cores, the imbalance present in the MIS example could limit the scalability of deterministic reservations. In the two Threadscope profiles (Figures 17 and 18) below, the reader can see a comparison between a well and poorly load-balanced algorithm. For clarity, each run was done without garbage collection.

In both examples, the respective algorithm is run on 1 million elements using 4 cores. In Figure 17 on page 37, the spanning tree algorithm displays very good load balancing, with most cores being fully utilised during each parallel phase. The valleys of the profile represent the sequential operations between parallel reserve and commit phases.

However, Figure 18 on page 37 tells a different story. The valleys are not as sharp, and we can see that sometimes two or three cores are used. Some cores even finish well before others. In our parallel steps, our prefix is divided up between threads 4,5,6 and 7. Each thread receives an array slice depending on its number. Hence a higher numbered thread will receive a later part of the

array. This is illustrated in 16.

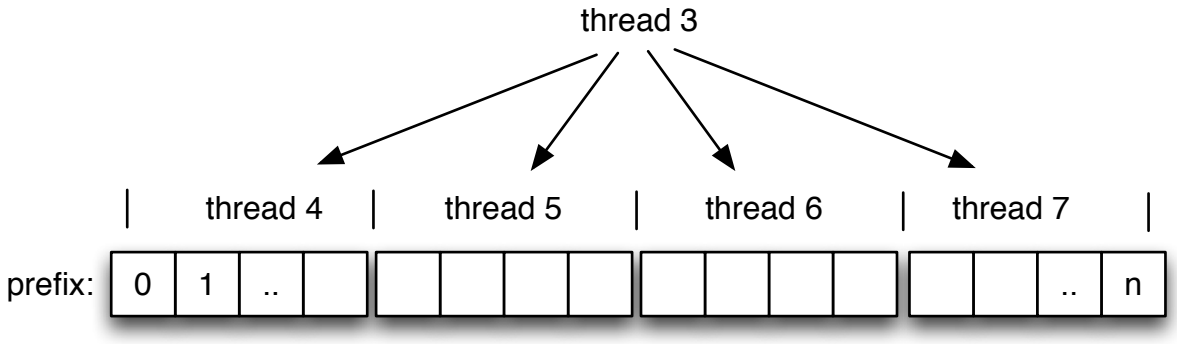


Figure 16: Dividing a prefix between threads

Upon further inspection of the stats for each HEC (Haskell Execution Context) in Figures 13a and 15a, we can see how much work each of our parallel threads are doing. In the spanning tree profile, all our threads are busy, most of the time. However, in our example of poor load balancing, the threads processing the later parts of the array nearly always do less work.

How can we tell whether our algorithm will distribute well? It depends on the algorithmic complexity of the reserve phase. Blelloch et al noted that in deterministic reservation based algorithms, most of the work is done in the reserve phase [6]. Figure 18 on page 37 supports this. Small spikes indicate commit phases, and the large blocks reflect reserve phases.

In the MIS algorithm, a node can be excluded from the set as soon as we detect that a neighbour has been added. Considering probability, our reserve step for a vertex will be a lot quicker if more of its neighbours have already been added to the set. It is likely that when processed, lower priority vertices will have higher priority neighbours already included in the MIS. It follows that threads processing later slices of the prefix are often able to quickly discard many elements, finishing their allotted slices while other threads are still working on earlier vertices.

In the spanning tree algorithm, this problem is non-existent. Underlying the reserve operation is the almost-constant time operations on the union find data structure 3.2. Hence the work is nearly always well distributed between cores. It is worth noting that the very nature of deterministic reservations as a speculative looping operation means that many greedy algorithms will exhibit sub-optimal work distribution between cores without some kind of work-stealing, as utilised in the PBBS implementation. However, even in the MIS example, the observed speedup is still good on the small number of cores we are using.

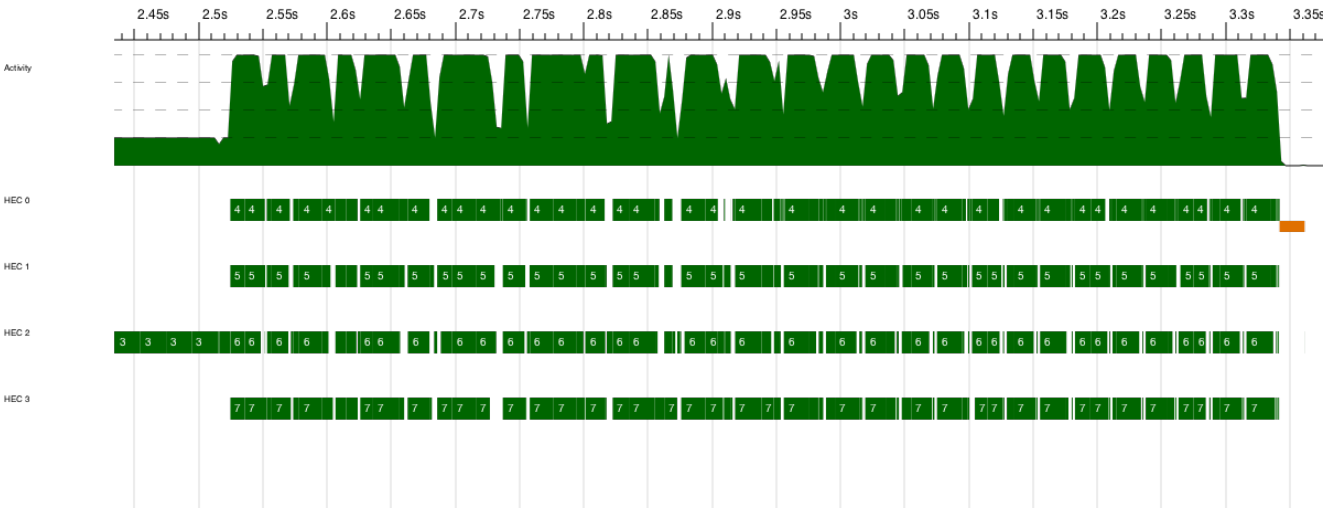


Figure 17: Core utilisation on spanning tree algorithm

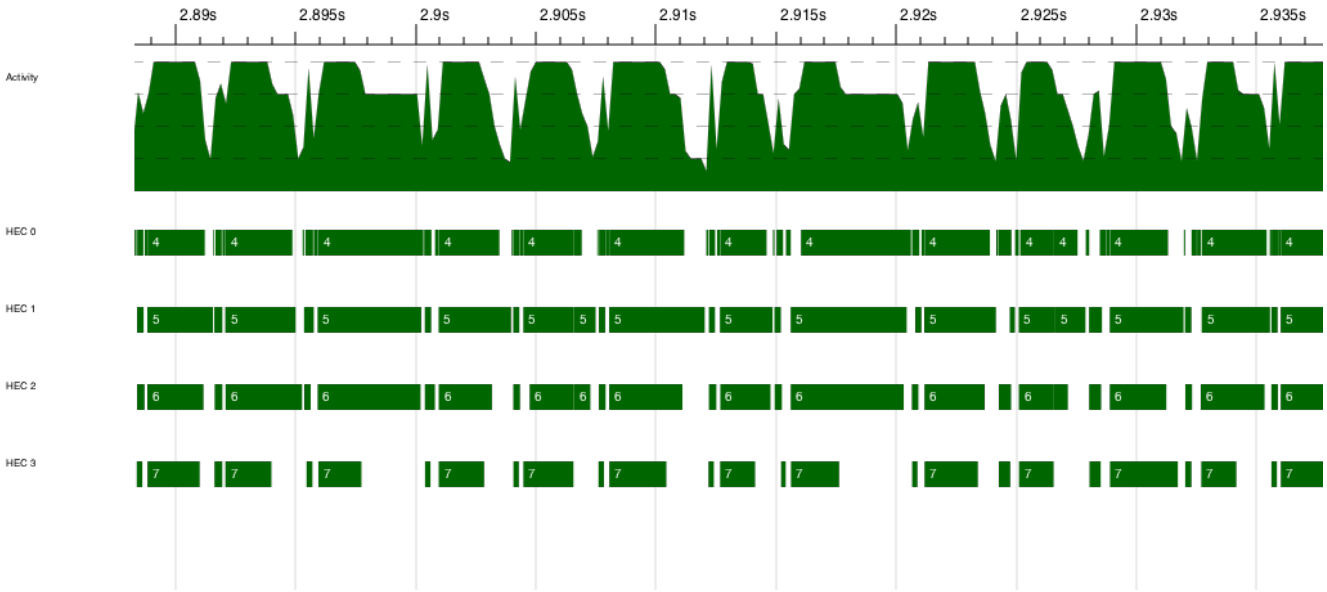


Figure 18: Core utilisation on MIS algorithm

4.6 Qualitative analysis

Blelloch et al's original deterministic reservations provides a very simple interface with just reserve and commit operations for the programmer to consider. However, the surrounding code needed to actually set up for, and process output from deterministic reservations can be confusing. Qualitatively, encapsulating these operations within an extended deterministic reservations, provides a much clearer pattern for the programmer to follow. The separation between input, mutable state and output also help for the readability of the algorithms. The tracking of state through the reserve and commit operations actually restricts the programmer to only mutating elements they declare as mutable. Through controlling stateful objects and their sharing in a much more rigorous manner, the clarity and usability of deterministic reservations is improved, while also reducing room for error. These two goals of usability and restricting stateful operations are often conflictory, and their synergy in this thesis is an impressive achievement.

My algorithm implementations are very concise and easy to read. They can be summarised concisely with the reserve/commit functions. Only in the case of the MIS algorithm is a helper function needed. Since our input is immutable, and we preallocate all our data structures, the designer doesn't need to consider strictness in too much detail. Our implementations are also fairly close functional translations of their C++ counterparts. The input needs to be read in and converted to the required format before deterministic reservations is applied. Care must be taken to strictly evaluate the datastructure before deterministic reservations is applied. Leaving the input as unevaluated thunks results in poor performance when an implementation is scaled to multiple cores.

4.6.1 Algorithmic interface

There is almost no boiler-plate code needed to run deterministic reservations. The programmer simply needs to implement their algorithm as an instance of the type class, and then call the $speculative\_for$ function with their input and choice of algorithm. Furthermore, a correct implementation will scale linearly, and still produce the correct result. Once the user has their algorithm implementation, they only have to call deterministic reservations like so:

```
result =  granularity (newStep :: Algo STStep) input
```

This is much clearer than Blelloch's model, where one needs to write the following:

```
intT m = input.nonZeros;
intT n = input.numRows;
unionFind UF(n);
reservation *R = new reservation[n];
unionFindStep<intT> UFStep(input.E, UF, R);
(UFStep, 0, m, granularity);
_seq<intT> stIdx = sequence::filter((intT*) R, n, notMax());
```

4.6.2 Code reuse

A library is only as good as the amount of work it saves the programmer, particularly work that involves writing low level, error-prone code. In Table 1 on page 39, we see that deterministic reservations achieves a very high level of code reuse. What these numbers do not illustrate is the difference in complexity between library code and algorithmic implementations.

	ST	MIS	Library
LOC	57	62	506

Table 1: Lines of code (LOC) comparison

My algorithm implementations are very modular. To implement an algorithm, the designer only has to define the data structures, and the five functions of the interface, two of which are just constructors. Figure 19 on page 40 illustrates the type of code that the designer needs to write. A comparison of this to my deterministic reservations library code in Figure 20 on page 40 suggests that the user is saved a significant amount of complex work. Many mutable structures are passed around within the library code, and the order they are operated on is essential. Furthermore, the programmer doesn't need to consider threads, fork/join constructs or even par annotations to achieve significant speedups with their algorithms. All the parallelism is hidden from the user and handled within the library.

4.7 DPH integration

My implementation of deterministic reservations should be easily adaptable for DPH. As mentioned in Section 3, core components of DPH, such as the vector library and gang parallelism underpin this approach. One obstacle is the current boxed representation of prefix steps. Prior to integrating deterministic reservations into DPH, the prefix would need to be represented as an unboxed array to allow for vectorisation.

```
instance DRStep STStep STinput STmut STOutput where
        ...
        reserve edges (STmut res uf) (STStep u v) index = do
                        (e1, e2) ← indexM edges index
                        upar ← find uf e1
                        vpar ← find uf e2
                        if upar == vpar then return False
                        else if upar < vpar then stReserve upar vpar
                        else stReserve vpar upar
                where stReserve a b = do
                                writeSTRef u a
                                writeSTRef v b
                                reserveR res b index
                                return True
        ...
```

Figure 19: Implementation of reserve for the spanning tree algorithm

```
freserve, fcommit :: DRStep step input mut output
        ⇒ Operation step s → Vector Int → Steps step s → Keep s → Int → ST s ()
freserve operation prefix steps keep i = do
    index ← indexM prefix i
    s ← V.indexM steps i
    v ← operation s index
    MU.write keep i $! v

fcommit operation prefix steps keep i = do
     b ← MU.read keep i
    when b $ do
      index ← indexM prefix i
       v ← V.indexM steps i
       b1 ← operation v index
      MU.write keep i $! (not b1)

bulkReserve, bulkCommit :: DRStep step input mut output
        ⇒ input → mut s → Gang → Vector Int → Steps step s → Keep s → ST s ()
bulkReserve input mutStuff theGang prefix prefixSteps keep
        = bulk ReserveT (reserve input mutStuff) theGang prefix prefixSteps keep
bulkCommit input mutStuff theGang prefix prefixSteps keep
        = bulk CommitT (commit input mutStuff) theGang prefix prefixSteps keep
```

Figure 20: Deterministic reservations library code

5 Conclusion

In this thesis, I put forward deterministic reservations as an attractive approach for implementing parallel graph algorithms in a functional programming language, with the goal of implementing such a library into Data Parallel Haskell. The complex, interconnected nature of graph representations makes processing them in parallel difficult. Furthermore, as of yet, parallel libraries in functional languages have no compelling story for tackling graph algorithms.

I show that deterministic reservations fits neatly within the functional paradigm. I expand the programming pattern presented by Blelloch et al, and present a more rigorous interface to deterministic reservations. The first step was to separate immutable input, private step and shared state. This provides a clearer definition to the user, and explicitly encodes important properties and relationships of deterministic reservations that were implicitly implied in the original model.

I then expanded the interface, so that all mutable data structures and shared state aren't accessible outside the deterministic reservations operation $speculative\_for$. This allows us to represent the key property of deterministic reservations using the Haskell's type system. The output of a deterministic reservations operation relies only on the input, the algorithm, and the prefix size. Our $speculative\_for$ is referentially transparent, conditional on the commutative properties of the deterministic reservations algorithm.

My expanded model achieves excellent results, both quantitatively and qualitatively, on two benchmarks. It scales well, achieving a similar linear speedup to the original model, of around 4x on 8 cores. While my implementation is slower, by around 4-5x than the original C++ code, the difference is mostly attributed to boxed operations within the inner loops. I also suggest a approach for reducing this effect as part of my contributions.

There is still some future work to be undertaken. The performance can be brought closer to the original C++ implementation by unboxing the inner loop. Deterministic reservations was initially proposed for parallelising greedy sequential algorithms such as the benchmarks used in this paper. It would be interesting to look at defining a class of algorithms suited to a deterministic reservations approach, and look at their representation. Over a small number of processors our approach works well, thanks to the trade-off between locality and load balancing. However, it would be interesting to investigate the results for different load balancing techniques. Finally, this thesis presented deterministic reservations with the aim of implementing it for DPH, and there is still some work needed to achieve this final goal.

6 References

[1] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[2] Paul S Barth and Rishiyur S Nikhil. M-structures: extending a parallel, non-strict, functional language with state. In *Functional Programming Languages and Computer Architecture*, pages 538–568. Springer, 1991.

[3] Tom Bergan, Joseph Devietti, Nicholas Hunt, and Luis Ceze. The deterministic execution hammer: How well does it actually pound nails. In *The 2nd Workshop on Determinism and Correctness in Parallel Programming (WODET'11)*, 2011.

[4] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, Cambridge, MA, USA, 1990.

[5] Guy E. Blelloch. Programming parallel algorithms. *Commun. ACM*, 39(3):85–97, March 1996.

[6] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. *SIGPLAN Not.*, 47(8):181–192, February 2012.

[7] Guy E. Blelloch, Jeremy T. Fineman, and Julian Shun. Greedy sequential maximal independent set and matching are parallel on average. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 308–317, New York, NY, USA, 2012. ACM.

[8] Guy E. Blelloch, Jonathan C. Hardwick, Siddhartha Chatterjee, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *SIGPLAN Not.*, 28(7):102–111, July 1993.

[9] Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *J. Parallel Distrib. Comput.*, 8(2):119–134, February 1990.

[10] Manuel M. T. Chakravarty and Gabriele Keller. More types for nested data parallel programming. *SIGPLAN Not.*, 35(9):94–105, September 2000.

[11] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ICFP '07, pages 315–326, New York, NY, USA, 2007. ACM.

[12] Martin Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 11:467–492, 8 2001.

[13] High Performance Fortran Form. High performance fortran language specification. 1993.

[14] Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 345–354. ACM, 1989.

[15] Bernard A Galler and Michael J Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[16] Phillip B Gibbons. A more practical pram model. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168. ACM, 1989.

[17] Robert H Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[18] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–218, 2006.

[19] C Hoareetal. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. *Engineering theories of software construction*, (180):47, 2001.

[20] Ellis Horowitz. *Fundamentals of computer algorithms*. Galgotia Publications, 1999.

[21] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: being lazy with class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[22] Don Jones, Jr., Simon Marlow, and Satnam Singh. Parallel performance tuning for haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 81–92, New York, NY, USA, 2009. ACM.

[23] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Annual Symposium on Principles of Programming Languages: Proceedings of the 23 rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 21, pages 295–308. Citeseer, 1996.

[24] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. *SIGPLAN Not.*, 45(9):261–272, September 2010.

[25] David Jonathan King. *Functional programming and graph algorithms*. PhD thesis, University of Glasgow, 1996.

[26] Joseph B. Kruskal. On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, February 1956.

[27] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. Optimistic parallelism requires abstractions. In *ACM SIGPLAN Notices*, volume 42, pages 211–222. ACM, 2007.

[28] John Launchbury and Simon L Peyton Jones. State in haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[29] John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Notices*, volume 29, pages 24–35. ACM, 1994.

[30] Charles E Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[31] Ben Lippmeier, Manuel M.T. Chakravarty, Gabriele Keller, Roman Leshchinskiy, and Simon Peyton Jones. Work efficient higher-order vectorisation. *SIGPLAN Not.*, 47(9):259–270, September 2012.

[32] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[33] Simon Marlow. Status of dph benchmarks, December 2010. available from: http://hackage.Haskell.org/trac/ghc/wiki/DataParallel/BenchmarkStatus.

[34] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[35] Robert HB Netzer and Barton P Miller. What are race conditions?: Some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(1):74–88, 1992.

[36] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, 2008.

[37] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell*. O'Reilly Media, 2008.

[38] Suhas S Patil. Closure properties of interconnections of determinate systems. In *Record of the Project MAC conference on concurrent systems and parallel computation*, pages 107–116. ACM, 1970.

[39] Simon Peyton Jones. Harnessing the multicores: Nested data parallelism in haskell. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 138–138, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] William M. Pottenger. The role of associativity and commutativity in the detection and transformation of loop-level parallelism. In *Proceedings of the 12th international conference on Supercomputing*, ICS '98, pages 188–195, New York, NY, USA, 1998. ACM.

[41] Yossi Shiloach and Uzi Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57 – 67, 1982.

[42] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proceedinbgs of the 24th ACM symposium on Parallelism in algorithms and architectures*, SPAA '12, pages 68–70, New York, NY, USA, 2012. ACM.

[43] Harald Sondergaard and Peter Sestoft. Referential transparency, definiteness and unfoldability. *Acta Inf.*, 27(6):505–517, January 1990.

[44] Guy L Steele Jr. Making asynchronous parallelism safe for the world. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 218–231. ACM, 1989.