Parallel Implementations on the GPU of Algorithms for the Minimum Dominating Set Problem Using Dynamic Programming on Tree Decompositions.

T.A. Jimkes

Abstract

Two different algorithms that solve the Minimum Dominating Set problem are implemented in both a sequential, and parallel fashion. The first on the CPU, and the latter using GPU programming. The aim of this work is to find out to what extent these algorithms can be implemented on the GPU, and how these four different implementations compare on different instances. The experimental results show that when the tree width k grows, parallelization of these algorithms provides reductions in run time that can reach multiple orders of magnitude. The sequential implementations do outperform their parallel counterparts for very small bag sizes, due to the overhead associated with the GPU implementations.

A thesis presented for the degree of
Master of Science

Department of information and computing sciences
Utrecht University
Netherlands
July 11, 2019

Contents

1 Introduction **3**
1.1 Previous work . . . 3
1.2 Our contribution . . . 5

2 Preliminaries **5**
2.1 Minimum dominating set problem . . . 5
2.2 Parallel programming on the GPU . . . 6
2.2.1 CUDA programming . . . 6
2.3 Tree decompositions and nice tree decompositions . . . 7
2.3.1 Nice tree decompositions . . . 7

3 Minimum Dominating Set and tree decompositions **8**
3.1 A basic approach using dynamic programming and tree decompositions . . . 9
3.1.1 The leaf procedure . . . 9
3.1.2 The introduce procedure . . . 9
3.1.3 The forget procedure . . . 10
3.1.4 The join procedure . . . 10
3.1.5 Resulting complexity . . . 11
3.2 Reducing complexity by altering the state set . . . 11
3.2.1 The leaf procedure . . . 12
3.2.2 The introduce procedure . . . 12
3.2.3 The forget procedure . . . 12
3.2.4 The join procedure . . . 13
3.2.5 Resulting complexity . . . 13
3.3 An optimal approach . . . 13
3.3.1 Consistently storing κ-values . . . 14
3.3.2 The $vanRooij_2$ algorithm . . . 14
3.3.3 Interpretation of state . . . 16
3.3.4 Resulting complexity . . . 16
3.4 Additional changes . . . 17

4 A parallel implementation on the GPU **17**
4.1 Base-3 configurations . . . 18
4.2 The introduce and forget procedures . . . 19
4.3 A parallel implementation of the *Alber* join procedure . . . 19
4.3.1 Splitting the table into q-based segments . . . 19
4.3.2 Deriving the correct configuration . . . 20
4.3.3 Considerations . . . 20
4.3.4 Expected effect on complexity . . . 21
4.4 A parallel implementation of van Rooij's join procedure . . . 21
4.4.1 Memory management . . . 21
4.4.2 Expected effect on complexity . . . 22

5 Experimental setup **22**
5.1 Method . . . 23
5.2 PACE data . . . 23
5.3 Obtaining nice tree decompositions . . . 24

6 Experimental results **25**
6.1 Parallel vs sequential implementations . . . 25
6.2 Parallel implementations: **P3k** vs **P4k** . . . 27
6.3 Sequential implementations: **S3k** vs **S4k** . . . 28
6.4 Varying block size of the parallel implementations . . . 28
6.5 Comparing algorithms . . . 29
6.6 A hybrid approach . . . 31

7 Conclusion **32**

8 Discussion **33**

9 Further research **34**

Appendices **38**

A Tables **38**

1 Introduction

The minimum dominating set problem is part of a complexity class that is called NP-Complete. These problems have the property that they are relatively computationally hard, as there is no algorithm that can solve them in polynomial time (assuming that NP is not equal to P). Because of this, many different approaches have been formulated to tackle problems like these in an attempt to reduce computational complexity. One of these approaches, that applies to graph problems, is using tree decompositions. These decompositions are created using such a graph and have a width as a property. A general tree width k can be derived from a graph, which represents a certain lower bound on the complexity of tree decompositions that can be created from the graph. In most cases, there is more than one possible tree decomposition possible for a given graph. This tree width can be used in representing the computational complexity of algorithms that solve graph problems using tree decompositions.

Due to the structure of tree decompositions (each node in the tree holds a subset of vertices of the graph), dynamic programming is often applied when these decompositions are used. A partial solution in the context of dynamic programming is a solution to only a part of the problem. Tables that hold these partial solutions called memoization tables, are used during execution to reduce the number of required computations, as it eliminates the need for calculating the value of a partial solution more than once. When there are many different partial solutions possible, these tables can grow in size rapidly. Though these tables might grow in size, the effective work that has to be done to calculate a single value for a partial solution for these algorithms is relatively little. When many small, comparable calculations have to be performed, parallelization using the GPU often is a fitting way to optimize, as the hardware is specifically made for this type of workload.

The GPU, or Graphical Processing Unit, has the property that it can perform very many simple calculations in parallel. Where the use of this piece of hardware was limited to graphics processing for a long time, in the past years there has been a surge in other applications. This, due to the recent availability of programming interfaces which are relatively easy to use. This increase in scientific and other practical usages other than graphics processing has also been due to the rapid increase in parallel processing power. When the GPU is used in a more general context, it is often referred to as GPGPU, or General Purpose Graphical Processing Unit (though in this work we will refer to it as GPU).

In this work, two different algorithms that apply dynamic programming on tree decompositions are implemented in the usual, sequential fashion. These implementations use the CPU to execute. Both algorithms are also implemented to run on the GPU, introducing parallelization. We will examine both algorithms in detail in order to find out what components of these algorithms can be implemented on the GPU and to what extent paralellization affects performance for these algorithms.

1.1 Previous work

Tree decompositions have been the source of much research in the past few decades. The work of Bodleander [3] discusses many applications of tree decompositions, as well as methods that use path decompositions. The work of Ton Kloks [14] provides an extensive exploration of what tree width is, and what some applications are. Solving some NP-hard problems on graphs can be done efficiently by using dynamic programming on tree decompositions. In [7], applications of dynamic programming in combination with tree decompositions are surveyed looking at applicability, algorithms that determine (approximate) tree width, algorithms that exploit tree decompositions and algorithms that find optimal, or approximate tree width.

Approximating tree width can be a useful approach when determining bounds for certain tree

decomposition based algorithms. Several approaches for finding an estimation are discussed in [6, 8]. One of these approaches is an $O(log\ k)$ approximation algorithm, where k is the tree width. Algorithms for finding the exact tree width have exponential time complexity [5], though a linear time algorithm is discussed in [2] that finds whether a tree decomposition exists of tree width at most k.

Applications of these types of algorithms can be in many domains of graph problems, including connectivity problems [9, 4, 10, 19] and domination problems [18, 1].

Lower bounds for different algorithms that use tree decompositions have been found in [15]. Among the algorithms presented is also the Dominating Set problem. The writers show that a lower bound on the time complexity of $(3-\epsilon)^k V^{O(1)}$ exists for the Dominating Set problem, where V is the number of vertices in the input graph and k is the corresponding tree width. These results are based on the Strong Exponential Time Hypothesis of Impagliazzo and Paturi.

Alber et al. use an approach of altering the state set that is used for dynamic programming on tree decompositions for solving the Minimum Dominating Set problem. They show that when a type of state is used that expresses indifference of being dominated, an $O^*(5^k)$ algorithm can be improved by reducing this complexity to $O^*(4^k)$. Two alternative approaches are discussed in [18]. Both introduce a state that is similar to the state expressing indifference in the work of Alber et al., but by using a different implementation, time complexity for the Minimum Dominating Set problem can be reduced to $O(n^3 3^k)$ and $O(nk^2 3^k)$ for these algorithms, respectively.

A Cut and Count approach is discussed in [9], which is used for reducing the complexity of certain graph problems. This approach is applied to problems that have a global requirement (like connectivity), which often results in $k^{O(k)} V^{O(1)}$ time complexity for naive approaches, where k is the tree width. The approach discussed reduces this to a complexity of $c^{O(k)} V^{O(1)}$, where c is a constant. This idea is expanded upon in [4] by using approaches based on linear algebra. The result is the same reduction in complexity as in [9], but with an approach that also works for weighted and counting versions of the problems, using a rank-based approach. The writers of [12] provide an extensive comparison between a straight forward dynamic programming approach, and the method used in [9], showing significant improvements when examining the latter algorithm.

A well known connectivity problem is the Hamiltonicity problem. In [10], these cycles are obtained from creating a Matching Connectivity matrix. By analyzing this data structure, the writers of the paper have found two Monte Carlo algorithms, one for directed graphs and one for undirected graphs, which solve the Hamiltonicity problem. The presented algorithms have a run time complexity of $1.888^n n^{O(1)}$ for the directed variant, and a complexity of $(2+\sqrt{2})^{pw} n^{O(1)}$ (where pw is the path width) for the undirected variant. Using tree decompositions, the algorithm of [10] is improved upon in [19], in which an $O^*((2^\omega+2)^k$-time algorithm is presented for counting Hamiltonian cycles. In addition, a $O^*((2^\omega+1)^k$-time algorithm is described that counts Steiner trees. These results are obtained by using Clifford algebras, which are mainly used in the field of quantum mechanics, in combination with Non-commutative Subset Convolution, which is method that is useful when working with determinant-based algorithms. An extensive look on different tree decomposition based approaches for solving the Hamiltonian Cycle problem is provided in [20].

There has already been research towards the effectiveness of GPU implementations of dynamic programming algorithms. In [13], different approaches are explored for creating an implementation the SAT problem that can be run on the GPU. Several of these approaches are implemented and examined, after which it is shown that these implementations can lead to a speed up of an order of magnitude. In [11], parallel implementations of the Counting SAT and Weighted Model Counting algorithms are discussed. The approaches again use dynamic programming on tree decompositions. The writers show that rather complex reasoning problems can be solved by solving instances with a tree width of up to 30.

1.2 Our contribution

The main objective of this thesis is to explore how parallel implementations on the GPU of the Minimum Dominating Set algorithms as discussed in [1] and [18] perform, and how they compare to their sequential counterparts. This is done by answering the following research questions:

1. Which components of the provided algorithms can be implemented on the GPU, and how?
2. What effect does this parallelization have on the run times of the introduce, forget and join procedures and how does this relate to bag size?
3. What effect does this parallelization have on the global run times and how does this relate to tree width k?

Section 2 will introduce preleminaries. Here we define the Minimum Dominating Set problem, provide a brief overview on how parallel programming on the GPU works, and introduce the concept of tree decompositions and nice tree decompositions. After this, an overview of both algorithms is provided in Section 3, along with an important correction to one of the two algorithms, as described in the literature [18]. When the reader has become familiar with the problem domain and the algorithms, we will discuss what components of the algorithms can be, and are, implemented on the GPU in Section 4. Section 5 will discuss the experimental method, the tree decompositions that were used and how these tree decompositions were transformed to fit the algorithms. In Section 6, the experimental results will be discussed. Our conclusion and a small discussion are provided in Sections 7 and 8, respectively. Finally, some ideas and suggestions for further research can be found in Section 9.

The results of this work show that, for the right values tree width k, parallelization using the GPU can result in a speed up in the multiple orders of magnitude for the provided algorithms. For individual nodes, when bag size exceeds a certain threshold, be it for a join, introduce, or forget nodes, all instances prefer a parallel implementation. Only for very small instances do the sequential implementations outperform their parallel counterpart. This is the result of overhead that comes with the GPU implementations.

2 Preliminaries

This section will cover the central concepts that are used in this work: the minimum dominating set problem, parallel programming using CUDA, tree decompositions and nice tree decompositions.

2.1 Minimum dominating set problem

A dominating set D on the graph G is a set of vertices such that, for all vertices $v \in V$ either of the following two statements holds:

1. The vertex v is part of the dominating set D, or
2. The vertex v is not part of D, but is adjacent to at least one vertex in D

The Minimum Dominating Set problem is centered around finding a dominating set that is of minimum size.

This work focuses on the variant of the problem, where the solution is the size of the smallest dominating set that can be found in the graph G. We fix this problem variant because of memory

constraints. It should be mentioned however, that a minimum dominating set can be found by performing n runs of a slightly adjusted variant of the algorithm through a method called self-reduction (e.g. see [3]). Here, the input is a graph and two sets of vertices, where one set contains vertices that are fixed in the dominating set, and the other set contains the rest of the nodes:

1. Find the size of the smallest dominating set D using the original algorithm.
2. For each vertex v, fix its inclusion in the dominating set D and execute a variant of the algorithm that allows for this. If the result of the algorithm changes, the node is not part of the minimum dominating set.

2.2 Parallel programming on the GPU

The main difference between the Graphical Processing Unit, or GPU, and the Central Processing Unit, or CPU, is that a GPU consists of many individual processing units, each with a limited amount of processing power, caching memory, and instructions available. The CPU on the other hand, consists of one, or few processing units. Each CPU processing unit has plenty of processing power, and is optimized for more complex sequential tasks.

Each individual process that is executed on the GPU is called a thread [16]. Each thread is part of a bundle of 32 threads called a warp. This bundling and executing of threads is done by a streaming multiprocessor, or SM, of which a GPU has many. Warps are executed simultaneously and perform the same operation. This way of executing is called SIMT, or Single Instruction Multiple Threads. Since warps are executed as a unit, all threads start at the same time and have to wait for each other to finish. If threads within a warp have different paths of execution, for example due to clauses in an if-statement, branching occurs. The streaming multiprocessor solves this by executing all different paths in sequence. This is why it is recommended to avoid such branching structures.

These fundamental differences in the CPU and GPU, have as a result that the CPU is optimized for complex sequential processes, and the GPU is more viable for many, small, similar calculations that can be computed in parallel.

2.2.1 CUDA programming

For this research, the CUDA toolkit v10 was used [17]. This toolkit provides a framework and debugging methods for writing GPU code in C++, or C. When this approach is used, the GPU is accessed through an abstraction, which means that the developer no longer has to take into account individual warps and SM's, but thinks in threads, blocks, grids, and logical CUDA cores. For a developer, the GPU consists of many CUDA cores, cache associated with such a core and shared memory. Where local cache (consisting of several kilobtyes) is accessible for only a single thread, shared memory is accessible for all threads during execution. The latter is slower to write to and to access, but it is vastly larger than the local cache. In a CUDA project, there is the distinction between the *host* and the *device*. The former is the system, including the CPU and internal memory, the latter is the individual NVidia GPU, which has its own memory. Executing code on the device is done through *kernels* [1]. These are methods that are compiled to run on the device.

When a kernel is launched from the host, it is launched very many times at once. Kernel executions are distributed over processing units and they are queued if none are available. These executions are performed using a set structure of *grids*, *blocks*, and *threads*. Each kernel execution has an associated grid, block and thread id, which is used for determining what data is associated

[1] Not to be mistaken by the kernels in the FPT theory about preprocessing.

with this specific execution. By defining dimensions for these blocks and grids relative to the data that is used, one can optimize processing unit usage and memory usage.

When many kernels are launched on the device, all of these do not communicate directly with the host. Executions occur locally on the device, which means that results also have to be stored on the device until they can be copied back to the host. Because of this, kernel executions often are performed using the following sequence of operations:

1. First, allocate memory for possible preliminary data and an array that will hold the results of all kernel executions.
2. Then, copy the preliminary data to the device, this is called a *memcopy* and it is a relatively slow operation.
3. After that, all kernels are executed and the result of each individual kernel is stored in the array. The index in the array that is associated with a certain kernel execution is derived from the combination of the grid, block and thread id.
4. Lastly, as all kernels have finished executing, the array containing the results is copied back to host, and the memory that was allocated in step one is deallocated.

CUDA provides one more dimension of parallelization that does not strictly follow the concept of SIMT processing. Multiple *streams* allow for execution of different commands in sequence, or in parallel. As a block is executed by a certain CUDA core, multiple streams allow for more optimal use of these cores when the device is not fully ocupied. As will be shown in section 4.3, this extra method of parallelization can be very useful.

2.3 Tree decompositions and nice tree decompositions

A tree-decomposition of a graph $G = \{V, E\}$, consists of a pair $(\{X_i | i \in I\}, T = \{I, F\})$ having $\{X_i | i \in I\}$ a family of subsets, or bags, of the vertices in V, one for each node in T, such that:

1. The union over all bags is the set V.
2. For all edges $(v, w) \in E$, There exists an $i \in I$ with $v \in X_i$ and $w \in X_i$.
3. For all $i, j, k \in I$: if j is on the path from i to k in T, then $X_i \cap X_k \subset X_j$.

The treewidth of a graph is a measure of how 'tree-like' the graph is. The width of a tree decomposition is the size of the largest bag, minus one. The tree width of the corresponding graph is the smallest width over all possible tree decompositions for that graph. One is subtracted from this specific bag size, because the result of this is that the tree width of a tree is exactly one.

2.3.1 Nice tree decompositions

A tree decomposition can be transformed in linear time to a nice tree decomposition [14]. This form of tree decompositions is called 'nice', because it allows simple formulations of dynamic programming algorithms over these decompositions. For a tree decomposition to be nice, the following conditions must hold [14]:

1. The number of children for a node is at most two.
2. If a node i has two children j, and k, then $X_i = X_j = X_k$, i.e. their bags are equal.

3. If a node i has a single child j, then either
   - $|X_i| = |X_j| + 1$ and $X_i \subset X_j$, or
   - $|X_i| = |X_j| - 1$ and $X_j \subset X_i$.

In addition to the above conditions, we introduce two additional conditions for practical purposes:

1. The root bag must be of size exactly one.
2. All leaf bags must be of size exactly one.

When these extra constraints are applied, the resulting nice tree decomposition has five different types of nodes:

1. A root node. This node has a bag size of exactly one.
2. Leaf nodes also have bag size one.
3. Introduce nodes. The bag size of these nodes is exactly one more than that of their children.
4. Forget nodes are the opposite of introduce nodes, as they contain one fewer vertex than their child.
5. Join nodes have exeactly two children. Both of those children have the exact same bag as their parent.

Since all nodes in the nice tree decomposition are one of these five types, we can implement algorithms that have a set procedure per type. This makes it a lot easier to use dynamic programming in combination with such tree decompositions. The root node stores the solution to the problem and often does not require a separate procedure. The different procedures in the dynamic programming algorithm are as follows:

1. Leaf nodes are instantiated using static values.
2. Introduce nodes introduce a single vertex to the bag.
3. Forget nodes remove one vertex from the bag.
4. Join nodes have the same bags as their children and as a result, each partial solution in the join node is associated with a partial solution in each child. The join procedure will have to reduce these multiple partial solutions to a single partial solution.

3 Minimum Dominating Set and tree decompositions

As discussed in Section 2, the algorithms that are used in this project use four different main operations which are associated with their respective nodes: leaf, introduce, forget, and join.

A configuration c is a state assignment for each vertex in the bag of a certain node. The memoization table in each node consists of all possible configurations for the vertices present in the bag. The value that is associated with a certain configuration in a node x, $A_x(c)$, is a partial solution. What this value represents differs from algorithm to algorithm. Nodes that are in-between nodes that have been evaluated, and nodes that have yet to be evaluated are on the so called boundary. A partial solution that is on the boundary represents a solution for the set of vertices that are present

in the current node and in the sub-tree of the node. Vertices that have not yet been introduced are not part of that partial solution. As a result, two partial solutions for the same sets of vertices can often be interchanged.

In this work, we make the distinction between nodes and vertices. Vertices are part of the input graph. Nodes are part of the tree decomposition and contain bags of vertices.

3.1 A basic approach using dynamic programming and tree decompositions

In this section, we will discuss an approach that is based on the discussion in [18]. It has a computational complexity of $O^*(5^k)$ and it uses the following set of states: $\{1, 0_1, 0_0\}$. A vertex has state 1 if it is included in the partial solution. A vertex has state 0_1, or 0_0 if it is not included in the partial solution, and it is dominated, or not dominated, respectively.

3.1.1 The leaf procedure

We know that the bag of a leaf node contains only a single vertex. This is why we can set the table values for all of the possible states by just setting three values.

$$A_x(\{1\}) = 1$$
$$A_x(\{0_1\}) = \infty$$
$$A_x(\{0_0\}) = 0$$

A state assignment of 1 results in a dominating set that includes the only vertex in the partial solution. The size of this set is, of course, one. The configurations that exclude the single vertex from the dominating set result in a size of zero, for state 0_0, and infinity for state 0_1, which represents infeasibility (the vertex cannot be dominated, since there are no other vertices in this partial solution).

3.1.2 The introduce procedure

The introduce node is more complex. The algorithm uses the values from the child nodes's table. Here, v is the newly introduced vertex and the node y is the single child of node x in the tree decomposition.

$$A_x(c \times \{0_1\}) = \begin{cases} A_y(c) & \text{if } v \text{ has a neighbour with state 1 in } c \\ \infty & \text{otherwise} \end{cases}$$
$$A_x(c \times \{0_0\}) = \begin{cases} A_y(c) & \text{if } v \text{ does not have a neighbour with state 1 in } c \\ \infty & \text{otherwise} \end{cases}$$

Adding a vertex with state 0_1 requires that the vertex has a neighbour in the bag with state 1. If this is not true, the state assignment is impossible and we assign value ∞. If the vertex does have a neighbour with state 1, the new configuration is assigned value $A_y(c)$. We don't have to change this value, since we have added no new vertex to the dominating set. The same is true for adding a

vertex with state 0_0, though in this case it is required that the newly added vertex has no neighbour in the current bag (as the state assignment indicates).

Introducing a vertex with state 1 is more complicated. When a vertex is introduced with state 1, we can make a selection of the configuration with the lowest value for $A_y(c')$ that *matches* the configuration c, where a matching configuration c' is defined as follows:

$$\begin{aligned}&\text{For all } u \in X_y \backslash N(v) : c_x(u) = c_y(u)\\&\text{For all } u \in X_y \cap N(v) : \text{ either } c_x(u) = c_y(u) = 1, \text{ or } c_x(u) = 0_1 \text{ and } c_y(u) \in \{0_1, 0_0\}\end{aligned}$$

In words, if a neighbour of v in node x is dominated, we can choose to do the same for another neighbour that has state assignment 0_0 in the child node y. This results in the following equation:

$$A_x(c \times \{1\}) = \begin{cases} \infty & \text{if } v \text{ has a neighbour that has state } 0_0 \\ 1 + \min\{A_y(c') | c' \textit{ matches } c\} & \text{otherwise} \end{cases}$$

When the vertex has a neighbour with state 0_0, we cannot set the state for the current vertex to 1. If this is not the case, we explore all different configurations in the child node that adhere to the *match* condition and take the minimum value.

3.1.3 The forget procedure

In a forget node, we choose between two feasible state assignments (thus excluding 0_0, as every vertex must be dominated, or be part of the dominating set) for the forgotten vertex v:

$$A_x(c) = \min\{A_y(c \times \{1\}), A_y(c \times \{0_1\})\}$$

We take the minimum because we are calculating the minimum size of the dominating set.

3.1.4 The join procedure

The join node, which is the main focus of this work, has the most complex procedure and because of that, dominates the computational complexity of the entire algorithm. We say that configurations c_x, c_l, and c_r *match*, if for each vertex v in the nodes' bags the following holds:

$$\begin{aligned}&\text{Either } c_x(v) = c_l(v) = c_r(v) = 1,\\&\text{or } c_x(v) = c_l(v) = c_r(v) = 0_0,\\&\text{or } c_x(v) = 0_1 \text{ and } c_l(v), c_r(v) \in \{0_0, 0_1\}, \text{ but not both } 0_0.\end{aligned}$$

When a vertex is dominated and has state 0_1, there are several options for choosing what states are assigned in the child nodes by definition of the *match* condition. If at least one of the two vertices is dominated in the child, we make sure that it is also the case in the parent. Since this is true, we can then choose either of the 0_1 or 0_0 states, as long as we select the minimum value.

Using this new definition of a *match*, we can calculate the values in the table A_x. The function $\#_1(c)$ is used to indicate the number of one assignments in the configuration c.

$$A_x(c_x) = \min_{c_x, c_r, c_l \text{ match}} A_l(c_l) + A_r(c_r) - \#_1(c_x)$$

The number of ones included in the configuration has to be subtracted from the sum, since otherwise they would be counted twice. Due to the *match* condition, we find the minimum combination of partial solutions in the children.

3.1.5 Resulting complexity

The asymptotic complexity of the entire algorithm is the dominated by the complexity of the join procedure. The complexity of this procedure is the result of *match* condition. There is a total of five different combinations of states that can occur at a single position in the configuration of a parent and both its children $\{c_x(v), c_l(v), c_r(v)\}$: $\{1, 1, 1\}$, $\{0_0, 0_0, 0_0\}$, $\{0_1, 0_1, 0_0\}$, $\{0_1, 0_0, 0_1\}$, and $\{0_1, 0_1, 0_1\}$. Since these five different combinations have to be explored for each vertex in the bag in the join procedure, we get a complexity of $O^*(5^k)$. All other types of nodes are straightforwardly computed in $O^*(3^k)$.

3.2 Reducing complexity by altering the state set

The algorithm presented in [1], which we will call *Alber* for simplicity, improves on the $O^*(5^k)$ run-time of the previous algorithm, by introducing a new type of state, namely: $0_?$. This state is assigned to a vertex in the configuration and it indicates that a vertex is not part of the dominating set D, but it is not known whether the vertex is dominated or not. This algorithm replaces state 0_0 with the new $0_?$ state. Effectively, we can express the new type of memoization table $A_{Alber}(c)$ as follows:

$$A_{Alber}(c) = \min\{A(c')\} \tag{1}$$

c' *subject to:*

$$\begin{aligned} &c(v) = c'(v) = 1, \text{ or} \\ &c(v) = c'(v) = 0_1, \text{ or} \\ &c(v) = 0_? \text{ and } c'(v) \in \{0_1, 0_0\} \end{aligned}$$

This definition is true coordinate-wise: for each position in the configuration that we have a $0_?$, we take the minimum of the two partial solutions that have a 0_1, or a 0_0 at that same position.

Since the new state does not indicate whether the vertex is already dominated or not, and we are trying to find the minimum dominating set, we can select the minimum value of either state assignment. For simplicity, we will denote the memoization tables with the new states as $A(c)$, instead of $A_{Alber}(c)$.

From this definition of $0_?$ follows the concept of monotonicity for the $0_?$ and 0_1 states:

$$A(c \times \{0_?\}) \leq A(c \times \{0_1\}) \tag{2}$$

Since we select the minimum value over both states 0_1 and 0_0, we know by definition that the value for the $0_?$ state is smaller than, or equal to the value for the 0_1 state.

3.2.1 The leaf procedure

The leaf procedure is almost the same as that of the previous algorithm:

$$\begin{aligned} &A_x(\{1\}) = 1 \\ &A_x(\{0_1\}) = \infty \\ &A_x(\{0_?\}) = 0 \end{aligned}$$

The only change here is that the 0_0 state of the previous algorithm is replaced with a $0_?$ state. The value $0_?$ follows from Equation 1.

3.2.2 The introduce procedure

The table in the introduce node is calculated as follows:

$$A_x(c \times \{0_1\}) = \begin{cases} A_y(c) & \text{if } v \text{ has a neighbour that has state 1} \\ \infty & \text{otherwise} \end{cases}$$

$$A_x(c \times \{0_?\}) = A_y(c)$$

$$A_x(c \times \{1\}) = 1 + A_y(\phi_{N(v)}(c))$$

Here, the ϕ function replaces the states for the vertices that neighbour v to state $0_?$ if they have state 0_1. This change is made, since we know by definition of monotonicity that a configuration containing $0_?$ will always have a solution smaller than, or equal to a configuration having a 0_1 at the same position.

3.2.3 The forget procedure

The forget procedure is equal to the corresponding procedure in the original algorithm:

$$A_x(c) = \min\{A_y(c \times \{1\}), A_y(c \times \{0_1\})\}$$

We again exclude the last state, since forgetting a vertex means that it has to have a certain, feasible state and as was mentioned, a configuration containing state assignment 0_0 is not feasible, since all vertices have to be dominated or have to be part of the dominating set.

3.2.4 The join procedure

The most significant change is the new join procedure. By using the $0_?$ state, the total number of combinations of states that have to be reviewed in the join procedure is reduced from five to four.

We introduce the notion of two configurations c_l and c_r from child nodes l and r *dividing* the configuration c if either one of the following conditions hold for each position vertex v in the configuration:

1. $c(v) = \{1, 0_?\} \implies c_l(v) = c_r(v) = c(v)$
2. $c(v) = 0_1 \implies c_l(v), c_r(v) \in \{0_?, 0_1\}$ and $c_l(v) \neq c_r(v)$

In words, if a vertex is assigned configuration $0_?$, or 1, this should also be true in the configurations of the children. If a vertex is assigned state 0_1, then the configurations of the children should have state 0_1, or state $0_?$ assigned to that vertex, but they cannot have the same state assigned. This limitation is the key to the reduction in complexity. The table values A_x are calculated as follows from the children l and r:

$$A_x(c) = \min\{A_l(c_l) + A_r(c_r) - \#_1(c) \mid c_l \text{ and } c_r \text{ divide } c\} \quad (3)$$

To avoid counting 1-assignments twice, the term $\#_1(c)$ is subtracted from the sum of the two values. If a vertex is assigned state 0_1 in c, then it is sufficient that either one of the two states in the children have the same state. If we can ensure that the vertex with state 0_1 is dominated in one of the two children, we can choose to use the $0_?$ state assignment in the other child, since we have ensured that the vertex is dominated. By choosing the $0_?$ state assignment in the other child, we get the minimum size of D for the partial solution.

3.2.5 Resulting complexity

In Section 3.1.5 it was shown that the complexity of the respective algorithm has its origin in the different combinations of states that have to be explored in the join procedure. Since in this algorithm we reduce this number from five to four, the complexity changes with it.

The time complexity of $O^*(4^k)$ can be derived by looking at the *divide* condition in the join. As will be discussed more extensively in Section 4.3, There are $\binom{b}{q}2^{b-q}$ different configurations for a given bag size b and number of 0_1 states in the configuration q. This means that we have a total of $\sum_{q=0}^{b}\binom{b}{q}2^{b-q}$ different configurations for a given bag size. For a given q, the divide condition generates another 2^q configurations, which means that the total complexity comes to $\sum_{q=0}^{b}\binom{b}{q}2^{b-q}2^q$, which is equal to 4^k by Newtons Binomium.

3.3 An optimal approach

Further improvements are made by van Rooij et al. [18], who give two algorithms. The first of the two has a computational complexity of $O(n^3 3^k)$ which already is a significant improvement over the previous algorithm. The second algorithm in the work of [18] reduces this complexity even further to $O(nk^2 3^k)$. This is an improvement, since by definition $k \leq n$.

The forget, introduce, and leaf procedures of the latter approach are the same as those of the *Alber* algorithm discussed in Section 3.2. These procedures are combined with the join procedure of the introduced $O(n^3 3^k)$ algorithm, which results in the need for table transformations to and from a compatible format.

3.3.1 Consistently storing κ-values

The first of the two mentioned algorithms, which we will call *vanRooij*$_1$ for simplicity, differs from the second algorithm, which we will call *vanRooij*$_2$, in that the tables store more extensive information. A value in a table of the *vanRooij*$_1$ algorithm $A_{vanRooij}(c, \kappa)$ describes the number of solutions that are associated with that configuration, and a dominating set size of κ, where κ can range from 0 to n. This algorithm uses the following state set: $\{1, 0_0, 0_+\}$. With the new state 0_+, we again have to define the value for this state assignment in the memoization table $A_{vanRooij}(c, \kappa)$:

$$A_{vanRooij}(c, \kappa) = \sum A(c', \kappa) \tag{4}$$

c' *subject to:*

$c(v) = c'(v) = 1$, or
$c(v) = c'(v) = 0_1$, or
$c(v) = 0_?$ and $c'(v) \in \{0_1, 0_0\}$

As was the case for the $0_?$ state in the *Alber* algorithm, this definition is true coordinate-wise: for each position in the configuration that we have a 0_+, we take the sum of the two partial solutions that have a 0_1, or a 0_0 at that same position.

For simplicity, we will refer the the $A_{vanRooij}(c, \kappa)$ algorithm as just $A(c, \kappa)$. Here, we are counting solutions of a certain size κ for a given configuration c, by using the 0_+ state as the sum of the 0_1 and 0_0 states.

From this definition, we can derive monotonicity in the following form:

$$A(c \times \{0_0\}, \kappa) \leq A(c \times \{0_+\}, \kappa) \tag{5}$$

The join procedure for *vanRooij*$_1$ is fairly simple. The reason why this is possible, is that we do not have to consider multiple matching state assignments when calculating new partial solutions:

$$A_x(c, \kappa) = \sum_{\kappa_l + \kappa_r = \kappa + \#_1(c)} A_l(c, \kappa_l) \times A_r(c, \kappa_r) \tag{6}$$

The positions in the configuration that have a 1, or a 0_0 require to be the same for the configurations for the children. Since we are indifferent about whether vertices with a 0_+ state are dominated, we only have to consider 0_+ states for those positions in the respective configurations of the children. This indifference means that we do not have to consider multiple configurations, since all three of the states in the configurations in the parent table need to match all three of the states in the configurations in the tables of the children.

3.3.2 The *vanRooij*$_2$ algorithm

The *vanRooij*$_2$ algorithm combines the join operation from the *vanRooij*$_1$ algorithm that is discussed in Section 3.3.1, with the leaf, introduce and forget procedures from the *Alber* algorithm discussed

in Section 3.2. A key difference between these algorithms is that the *Alber* algorithm only stores the size of the smallest dominating set for a configuration in a partial solution, whereas the $vanRooij_1$ algorithm stores the number of solutions for a given size of the dominating set. This means that there is a major difference between the $vanRooij_1$ algorithm and the $vanRooij_2$ algorithm: the $vanRooij_2$ requires that the tables are transformed to and from different state domains.

The algorithm in [18] describes the following method for performing the join procedure. First, the memoization tables are expanded to contain κ-values. For children $y \in \{l, r\}$, we expand the table with $b + 1$ values, where b is the size of the bag, for each configuration and fill it with the following formula:

$$A'_y(c, \kappa) = \begin{cases} 1 & \text{if } A_y(c) = \kappa \\ 0 & \text{otherwise} \end{cases}$$

Note that here we are not counting partial solutions, but partial solutions stored in the original table. Where the tables in $vanRooij_1$ have a κ ranging from 0 to n, here we only expand the table to hold $b + 1$ additional values per configuration. This is done using the parameter ξ_y, which holds the lowest value in the table A_y. Since we know that we don't have to store lower values than this in the expanded tables (since they simply do not exist by definition of ξ_y) and because we do not have to store values higher than $\xi_y + b + 1$ (since these would never occur in an optimal solution), we can limit the range for κ, which becomes $\xi_y \leq \kappa \leq \xi_y + bagSize + 1$.

After expansion, the tables need to be transformed to use the right set of states. We do this by using the following formula:

$$A_x(c_1 \times \{0_+\} \times c_2, \kappa) = A_x(c_1 \times \{0_0\} \times c_2, \kappa) + A_x(c_1 \times \{0_1\} \times c_2, \kappa) \tag{7}$$

We plug in the $0_?$ state from the *Alber* algorithm to create the following equation to perform the translation and to introduce the 0_+ state:

$$A_x(c_1 \times \{0_+\} \times c_2, \kappa) = A_x(c_1 \times \{0_?\} \times c_2, \kappa) + A_x(c_1 \times \{0_1\} \times c_2, \kappa) \tag{8}$$

The formula is applied coordinate-wise. This means that for each position in the configuration, all $O(k^2 3^k)$ table entries are passed. As we saw in Equation 4, the 0_+ state is defined as the addition of the 0_0 and 0_1 states. In this instance, we need to transform to the 0_+ state by using the $0_?$ state and the 0_1 state. This means that we effectively interpret the $0_?$ state as being the 0_0 state. To see why this is correct, we look at Equations 2 and 5. By combining these definitions we can derive that:

$$A(c \times \{0_?\}) \leq A(c \times \{0_+\})$$

After expansion and transformation of the child tables, the join procedure from Equation 6 is performed for the following range for κ: $\xi_l + \xi_r - (b + 1) \leq \kappa \leq \xi_l + \xi_r + b + 1$. After this procedure is executed, the tables have to be transformed and reduced in order to be compatible with the introduce and forget procedures. The transformation occurs using Equation 9:

$$A_x(c_1 \times \{0_1\} \times c_2, \kappa) = A_x(c_1 \times \{0_+\} \times c_2, \kappa) - A_x(c_1 \times \{0_0\} \times c_2, \kappa) \quad (9)$$

The reason that we transform back using Equation 9, instead of translating back to the 0_0 state directly (which would then again be interpreted as state $0_?$), is that in the latter process, we would lose monotonicity (see Equation 2). This monotonicity was lost during the join as described in Equation 6, as this was not taken into consideration when combining the values of the two child nodes. If we were to translate back directly to state 0_0, we could no longer interpret the 0_0 state as the $0_?$ state, as monotonicity no longer holds. We can, however, translate back to the 0_1 state.

After the transformation, the table A'_x is reduced by using the following method:

$$A_x(c) = \min\{\kappa \mid A'_x(c, \kappa) \geq 1 \ \xi_l + \xi_r - (b+1) \leq \kappa \leq \xi_l + \xi_r + b + 1\} \quad (10)$$

From the κ-values that are larger than zero, we choose the smallest, since this is the smallest size of the dominating set D for which solutions are possible (for the given configuration). Since we have chosen for a more compact representation of the expanded tables by storing a min value ξ_y, we have to add both of these values to the optimal value of κ.

3.3.3 Interpretation of state

The join procedure for the *vanRooij*$_2$ algorithm as described in Section 3.3.2 is not the same as the one that is given in [18]. In Section 3.3.2 we perform an asymmetrical transformation, first performing a transformation that introduces the 0_+ state, but then transforming back to the 0_1 state, instead of transforming back to the $0_?$ state. The original implementation in *vanRooij*$_2$ uses a different implementation. Here, the transformation of the child tables is performed using the following Equation:

$$A_x(c_1 \times \{0_0\} \times c_2, \kappa) = A_x(c_1 \times \{0_?\} \times c_2, \kappa) - A_x(c_1 \times \{0_1\} \times c_2, \kappa) \quad (11)$$

And transforming the parent table back is done using Equation 9.

The $0_?$ state and the 0_+ state are incorrectly assumed to be equal, since the 0_+ state is fundamentally different from the $0_?$ state. The $0_?$ state is the effect of minimizing over the possibilities of being dominated, and remaining undominated (as defined in Equation 1). The 0_+ state is the result of taking the sum over these possible values (see Equation 4).

3.3.4 Resulting complexity

The discussed indifference in the join procedure is the reason that the computational complexity of this algorithm is less than that of the *Alber* algorithm. The tables in the join procedure of the *vanRooij*$_2$ algorithm have size $O(k3^k)$. Since, for each configuration, at most $O(k)$ multiplications have to be performed during evaluation, we get a complexity of $O(k^2 3^k)$. The state transformations also have complexity $O(k^2 3^k)$, since $O(k3^k)$ calculations have to be performed in $O(k)$ coordinate-wise steps. In total, n nodes in the tree decomposition have to be evaluated, which means that the total time complexity adds up to $O(nk^2 3^k)$.

3.4 Additional changes

Aside from the correction discussed in Section 3.3.3, several small adjustments to the original algorithms were made in order to make them slightly easier to implement. These changes had no significant impact on run time what so ever and they did not change the outcome of any of the algorithms. In practice, we count the number of vertices that are part of the solution (the dominating set) at a different moment in the algorithm. By counting a vertex as part of the dominating set as it is forgotten, rather than when it is introduced, some of the other methods used can be slightly simplified.

The introduce procedure is slightly altered, by replacing the function for the addition of a vertex with state 1 to the following:

$$A_x(c \times \{1\}) = A_y(\phi_{N(v)}(c)) \tag{12}$$

Since we now have to count a vertex as it is forgotten, the forget procedure is also changed:

$$A_x(c) = \min\{A_y(c \times \{1\}) + 1, A_y(c \times \{0_1\})\} \tag{13}$$

The effect that this has on the join procedure of the *vanRooij* algorithm, is that we no longer have to take into account the number of ones that are in a configuration:

$$A_x(c, \kappa) = \sum_{\kappa_l + \kappa_r = \kappa} A_l(c, \kappa_l) \times A_r(c, \kappa_r) \tag{14}$$

This also has an effect on the transformations that are part of the $vanRooij_2$ algorithm. If we create the the $A'(c, \kappa)$ tables starting at index zero, we can simplify the reduction in the last step of the join. In practice, we alter the expansion method as follows:

$$A'_y(c, \kappa) = \begin{cases} 1 & \text{if } A_y(c) = \kappa + \xi_y \\ 0 & \text{otherwise} \end{cases} \qquad \text{Having } \kappa \text{ ranging from 0 to } \mathit{bagSize} + 1 \tag{15}$$

By changing the range of the second parameter of the $A'(c, \kappa)$ function, we can simplify the reduction step as follows:

$$A_x(c) = \min\{\kappa \mid A'_x(c, \kappa) \geq 1\} + \xi_l + \xi_r \tag{16}$$

4 A parallel implementation on the GPU

Copying memory to device and then copying memory back for the leaf procedure would cause far too much overhead for such a simple procedure (merely instantiating 3 values in an array). This is

why the leaf procedure is shared among all implementations discussed in this work (sequential, as well as parallel). Our implementations perform each procedure sequentially, by copying memory to the device and copying the solutions of the kernel executions back to the host.

It should be noted here, that we could also have chosen the approach of performing the full algorithms on the GPU, thus eliminating overhead. We have chosen not to do this, opting for the approach of executing each procedure on the GPU individually using memcopies, for several reasons:

1. We were not necessarily very interested in GPU versus CPU implementations for small problem instances, as we do not expect parallelization to have a significant effect on run time (as these instances have relatively few partial solutions that can be found in parallel).

2. Memory constraints. Host memory is often (and also in this case) more abundant than device memory. As tables can become quite large, and many tables might have to be stored for later use, we could be severely limiting the instances that can be used by only using the device memory.

3. Ease of implementation. The modularity of the implemented algorithms allowed for selecting a procedure during execution. This allowed us to isolate newly implemented procedures and test them.

4.1 Base-3 configurations

The problems discussed in this work all use a state set containing three states. A configuration of vertices that is associated with a certain partial solution should be able to represent all three of these states for each vertex, resulting in the need for a base-3 configuration. Recall that each configuration c is associated with a single partial solution in the memoization table: $A(c)$.

In order to reduce storage requirements and complexity, standard 32-bit integers are used as configurations. Using these integers makes it so that a positional index in a memoization table can be used as a configuration. One problem arises from this approach, which is the fact that integers are encoded in bits, and not trits, which is what we need for the base-3 configurations. We use the concept of Binary Coded Ternary, or BCT. This encoding uses two bits in the bit string to represent one trit. Conversion from and to BCT makes it so that this encoding can be used. The functions used for configuration permutations, translations, etc., have been implemented using bit shifts wherever possible, in order to reduce overhead.

Since these conversions are very frequent, one might consider other solutions to this problem. The most straightforward optimization would be to use a conversion table, which would result in a constant-time array lookup, which is of course very fast. The downside of this approach is the memory requirement. It would introduce another table of size 3^k, that would have to reside in the host memory, as well as the device memory. The former would not be such a big problem, since it is relatively easy to expand host memory. The latter would cause problems however, since memory is embedded in the device and it should be used optimally and sparingly. In addition, it is possible that the few calculations for a small value of k will outperform the approach of creating a lookup table in memory.

Another approach would be to use an array that could be directly accessed with the BCT formatted configurations. This would eliminate the overhead of conversions, it would however, introduce sparsity in tables. Since only three quarters of the BCT string is used, the resulting tables would also have values for only three quarters of the indices.

As the mentioned alternatives come with some unwanted caveats, which are the result of memory constraints, we have chosen to use the conversions as mentioned in the beginning of this section.

4.2 The introduce and forget procedures

The parallel introduce procedure distributes the workload over an array of threads. Since each configuration in the child node is associated with three new configurations in the parent, we launch a total of 3^{b_c} threads, where b_c is the bag size of the child node. The implemented kernel performs the same operation as a single iteration in the sequential implementation does.

The forget procedure, simple as it is in itself, is parallelized similarly to the introduce. A thread is launched for each configuration in the parent node, which means that a total of 3^{b_p} kernel executions will occur, where b_p is the bag size of the parent node. The method used in the kernel itself is the same as is used in iterations of the sequential approach.

4.3 A parallel implementation of the *Alber* join procedure

As was discussed in Section 3.2, the join procedure contains a divide condition. This condition creates the need for exploring pairs of configurations from the children that contain either a 0_1, or $0_?$ state at the positions where the parent's configuration has a 0_1 state.

Let q, be the number of 0_1 states in a configuration c: $q_c = \#_{0_1}(c)$. We use this property q_c to divide the configurations in comparable segments, which are consequentially more manageable by the GPU. This, because the workload that has to be performed per configuration (exploring different combinations of configurations from the children) is dependent of the number of 0_1 states in the configuration. The set of configurations that we need from both children to perform the join operation for a single configuration for the *Alber* algorithm is 2^{q_c}, and will grow exponentially with q. This introduces the problem that configurations with more 0_1 states require significantly more computations to determine the minimal value for $A(c)$.

4.3.1 Splitting the table into q-based segments

For each q_c, we split the $\binom{b}{q}2^{b-q}$ corresponding configurations, where b is the bag size of the join node, into segments that are manageable for the GPU, as different values of q require different amounts of work. Note that the total of $O(3^k)$ configurations is the result of summing over all of these segments: $\sum_0^b \binom{b}{q}2^{b-q} = 3^b$.

As was mentioned in Section 2.2, we can distribute work over multiple streams in order to optimize GPU usage, due to the possibility of executing multiple streams in parallel. For a given q, the respective number of kernels are launched and assigned to a stream. This happens for all values of q, in an decrementing fashion, thus starting out with the stream that has to perform the most work, and gradually working towards smaller workloads. As multiple streams are launched, their workloads are queued if there is no capacity on the GPU available at that point. Since we start out with the largest workload, we can queue the other, smaller workloads, thus reducing idle time and optimizing GPU usage.

Recall that each thread has a unique identifier, which is derived from the respective combination of a grid id, block id, and thread id. This unique combination, which we will call m, ranges from 0 to $\binom{b}{q}2^{b-q}$, which is the size of the set of configurations associated with a given q. Each kernel is also given the corresponding value for q as a parameter at execution. We can split the value of m into two factors α, and β, where α is the index of the positioning of the q 0_1 states in the ordered set, and β is the index of the rest of the configuration in the ordered set (consisting of only $0_?$ and 1 states). One can see that α ranges from 1 to $\binom{b}{q}$, and β ranges from 1 to 2^{b-q}. As the 2^{b-q} different possible sub-configurations of the remainder of the configuration can only consist of two states, we can infer this part of the configuration directly from the BCT string β.

Since we are working with the BCT format, each m can be viewed as a BCT string that is the concatenation of the two BCT strings represented by α and β. Since β ranges from 1 to 2^{b-q}, one can see that we can easily determine the value of β by looking at the first $b-q$ positions of the BCT string m, which also means that we can obtain the value for α by looking at the rest of the BCT string.

4.3.2 Deriving the correct configuration

We define p_α as a set of integers $p_\alpha = \{p_0, ..., p_{q-1}\}$, where each integer is strictly larger than the preceding index and each integer describes the position of one of the q 0_1 states. We can view the set of possible combinations for p_α as a combinatorial number system. Such a system consists of combinations of numbers from a given ground set, where the combinations have a strict ordering and no combination occurs twice. This ordering of combinations is derived from the ordering of the set of numbers in the ground set. This means that for each value of α we have a single corresponding combination of positions p_α. With this knowledge, we can find the correct value of p_α by using the following algorithm, which uses α, q, and bag size b:

Algorithm 1 Find combination

1: **procedure** FINDCOMBINATION(α, b, q)
2: $\quad A \leftarrow [\ ]$
3: $\quad r \leftarrow \alpha$
4: $\quad cs \leftarrow b$
5: $\quad$ **for** s in descending values in the range q **do**
6: $\quad\quad$ **while** $r - \binom{cs}{s} < 0$ **do**
7: $\quad\quad\quad cs = cs - 1$
8: $\quad\quad r = r - \binom{cs}{s}$
9: $\quad\quad A.append(cs)$
$\quad$ **return** A

The algorithm determines for each of the q 0_1 states what their respective position is in the configuration, given a value α. Going from right to left in the configuration, we check if the current position has a 0_1 state. If this is not the case, we move over one position. This is done until a set of q positions is found.

The for-loop on line 5 executes exactly q times. The inner while-loop executes $O(b)$ times, since the value for cs reduces in each iteration. The binomial coefficient has time complexity $O(b)$, which means that the execution of this algorithm has a time complexity of $O(b^2)$. Now that we have found the correct combination of positions of the 0_1 states using α, and the correct composition of the rest of the configuration using β, we can easily translate this information to a configuration in constant time.

4.3.3 Considerations

Recall that a total of 2^q computations have to be performed per configuration, followed by a reduction (see Equation 3). Since we segment the configurations by q, we know that the workload per configuration for a given segment is equal. The first, more straight-forward approach to a GPU implementation is to assign all of the 2^q workload to a single thread. The reduction over all these values will then also be performed in the same kernel, which means that all of these executions can be done fully in parallel using streams.

A more complex approach would be to assign each of the 2^q operations in the min-term of the join of the *Alber* algorithm it's own thread. Each thread will perform the corresponding operation and will then wait until all threads corresponding to a single configuration are synchronized. After synchronisation, a block reduction step of complexity $O(\log 2^q) = O(q)$ is performed over all threads that correspond to a single configuration and the resulting value is stored. The implementation used for this research however, uses the first of the two approaches. This, due to a time constraint. Other suggestions that might improve run time will also be discussed in Section 9

4.3.4 Expected effect on complexity

As was explained in Section 3.2, the complexity of this algorithm is dominated by the $O(4^k)$ complexity term. The total work that has to be done is divided into q loads, where each load requires $\binom{k}{q}2^{k-q}2^q$ work. Theoretically, if there was no restriction on hardware, we could perform these q work loads fully in parallel using streams. In practice however, this is of course not the case and we are limited by the capacity of the GPU.

4.4 A parallel implementation of van Rooij's join procedure

The implementation of the $vanRooij_2$ algorithm is more complex than that of the *Alber* algorithm. This is due to the fact that the algorithm requires that tables are transformed to and from certain state set domains. Where we could perform the operations in the q sequential steps in the *Alber* algorithm fully in parallel, it is not quite as easy for the $vanRooij_2$ algorithm.

As was discussed in Section 3.3, the state transformations of the tables are not symmetrical: the tables of the two child nodes are transformed using Equation 8, and the transformation of the table in the parent node is performed using Equation 9. In a BCT string, we use three different values to indicate a certain state assignment: 0, 1, and 2. In the code, each state is associated with one of these values in the BCT string: a 0_+ state, or $0_?$ is a 2, a 1 state is a 1, and a 0 can be either a 0_0, or a 0_1 state. After transformation of the child tables, these associations are still true, though state 0_0 is used instead of state 0_1 and a 0_+ state has replaced the $0_?$ state. When we transform back, we replace the 0_+ state with a 0_1 state. Now, a 0 in the BCT string indicates state assignment $0_?$, and a 2 indicates a 0_1 state assignment. This is corrected by swapping 0 and 2 in the configuration, which means that we need an additional $O(3^k)$ swap operation in order to correct all the configurations.

4.4.1 Memory management

The join procedure of the $vanRooij_2$ algorithm involves transformations of tables to different state domains in order to be able to use a lower complexity join procedure. Since copying memory from host to device causes overhead, these memcopies are to be limited as much as possible. In order to do this, device memory has to be used sparingly and creatively, by overwriting tables that are no longer used with newly created tables. The process used is described in the following steps. For simplicity, we use the notation of memory blocks A, B, and C. The size of these memory blocks is determined by the size of the tables expanded with κ-values. We make a distinction between blocks, which are used in CUDA programming, and memory blocks, which are used here for efficient memory use.

The tables for the children and the parent are represented by A_l, A_r and A_x, respectively. The tables that have been expanded with κ values are denoted by A'_l, A'_r and A'_x. The process is as follows:

1. Allocate memory blocks A and B for tables A_l and A'_l, respectively. Also allocate memory block C for table A'_r.
2. Copy A_l from host to device.
3. Fill table A'_l by expanding A_l using Equation 15.
4. Perform a state transformation on table A'_l and overwrite table A_l in memory block B using Equation 8.
5. Overwrite A'_l in memory block A with A_r, by copying the table from host to device.
6. Fill table A'_r by expanding A_r, again using Equation 15.
7. Perform a state transformation on table A'_r, by overwriting table A'_r, which is in memory block C, again by using Equation 8.
8. Use tables A'_l, and A'_r, located in memory blocks B and C, respectively to fill table A'_x located in memory block A using Equation 14.
9. Perform another state transformation on table A'_x, by overwriting data in memory block A and by using Equation 9.
10. We now have a table where states 0_1 and $0_?$ are swapped. This is corrected by swapping the states and overwriting memory block B
11. Reduce the newly transformed table A'_x in memory block B by overwriting the data in memory block C and by using Equation 16.
12. Memory block C now holds the final table, this table is copied back to host and all memory blocks are freed.

This method ensures that we limit the number of memcopies between host and device, and it makes sure that we use as little memory as possible on device. As one can see, state transformations do not require the data to be written to a different memory block, since these operations can be performed in place.

4.4.2 Expected effect on complexity

As was explained in Section 3.3, the transformations from one state set to another has to be done in b sequential steps, where b is the bag size. This is a strong limitation when it comes to parallelization. As a result, we not only have to perform the transformations and join procedure one sequentially after the other, but we also have to perform the transformations in sequential steps. It is expected that this difference in viability for parallelization will become apparent during experimentation.

5 Experimental setup

Experiments were run on a consumer-grade system. The parallel algorithms ran on an Nvidia GTX 1050 Ti, which has a total of 768 CUDA cores. The card is clocked at 1290 MHz and has 4 GB of DDR5 memory. The system also includes a Ryzen 2700X CPU, which has 8 cores and 16 virtual threads (though the sequential algorithms ran on a single core), which has a base clock of 3.7 GHz. A total of 8 GB of DDR4 memory clocked at 3200 MHz was used on the host.

5.1 Method

Each of the four algorithms were run for all of the tree decompositions from our dataset (see Section 5.2). For each individual run, the following data was collected:

1. Global run time of the algorithm
2. Run time for each individual node
3. Bag sizes and node types

The collected data was processed to provide an overview of minimum, maximum and average run times for each type of node and for each occurring bag size. The graphics card that was used for the parallel implementations requires some time to start up. This is due to the driver having to connect to the device before any interaction can take place. This is why for each tree decomposition, a preliminary dummy run was executed using one of the two parallel algorithms. This ensures that the run times are accurate. Of course, these dummy runs were not taken into account when data was collected.

5.2 PACE data

We used graphs and tree decompositions from the PACE challenge data set. The PACE challenge is an annual competition that invites computer scientists to compete against each other to find the fastest possible algorithms to solve a given problem in the field of FPT-algorithms. The competition held in 2017 provided competitors with a repository with graphs and corresponding tree decompositions. These tree decompositions however, are not nice (in the sense that they are not structured as is defined by the definition of a nice tree decomposition). We had to implement the make-nice algorithm from [14]. This implementation is discussed in Section 5.3.

The tree decompositions included in the PACE set have varying tree widths. The tables in the nodes have size $O^*(3^k)$. If the memoization table for a node is calculated, it has to be stored in memory when its parents memoziation table is be calculated. This means that, depending on the structure of the tree and the bag size, memory might be heavily used. Depending on the type of node, several tables have to be available for the respective procedure. For a leaf node, this is negligible, as the table will always (at least in the context of this research) exist of only three integer values. The introduce and forget nodes require $3^b + 3^{b+1}$ space, where b is the bag size of the smallest node. This is the case because we need the table of the child to calculate the table of the parent. The join requires space for one more table; one for the parent, and one for each child. That means that, in the case of the *Alber* algorithm, the join node requires $3 * 3^b$ space, where b is the bag size, which is equal for all three nodes. This space requirement is even more (a factor $O(k)$) for the join node of the $vanRooij_2$ algorithm. The following example provides insight for on limiting this restriction might be. If we have a tree width of 20, we have at least one bag of size 21. The space required in bytes will then be:

$$\frac{3^{21} \times 32}{8} = 41841412812$$

We multiply by 32, since 32-bit integers are used. This is a total of more than 40 gigabytes of memory that is required for a single introduce, or forget type node memoization table. One can

image that, when multiple memoization tables are needed for a procedure, and when multiple nodes have to be stored for later use, this space requirement will be too vast for our hardware.

Due to this very strict limitation, all trees that had a tree width higher than 18 were filtered out. This, because our system is able to handle tables of bag size 19 (which are around 4.5 gigabytes in size). After filtering the tree decompositions, preliminary runs revealed which of the resulting tree decompositions were suited for experimentation. After these runs, several more trees were discarded. This, because during their runs, the required memory would exceed system memory, which would result in a very significant and unfair drop in performance. What remained was a total of 116 tree decompositions.

Table 1: For each tree width k the number of tree decompositions is given that has at least one join node of size $k+1$. The last column shows the percentage of tree decompositions of tree width k which have at least one join node of size $k+1$

k	none	at least one	%
6	0	2	100
7	0	11	100
8	0	9	100
9	0	34	100
10	0	31	100
11	0	14	100
12	0	12	100
13	0	2	100
14	0	1	100

Table 1 provides an overview of the presence of join nodes of size $k+1$ for trees that have tree width k. As was explained in Section 3, the join procedure determines the computational complexity of both the *Alber* algorithm and the $vanRooij_2$ algorithm. As we can now see, all tree decompositions that have a join node also have a join node of size $k+1$. This means that for all of the tree decompositions that have a join node (only one single tree decomposition in our set does not have single join node) the global run time (the execution time for the algorithm on the entire tree decomposition) should be at least heavily impacted by the join procedure.

5.3 Obtaining nice tree decompositions

In order to obtain nice tree decompositions (our version, see section 2.3), several modifications had to be performed to the tree decompositions in the PACE data set.

The implemented method passes all the nodes in a given tree decomposition twice. The fact that the algorithm takes slightly longer is irrelevant, since its running time is not included in the benchmarks. During the first pass, all nodes that have more than two children are transformed. For such a node, we remove children until the node has only two children left. Then, we append the removed children to one of the two remaining children. This operation is performed recursively, until we no longer have to remove any children. The result is a skewing in the tree, which is preferred over the other option: distributing nodes evenly over children to create a symmetric branching structure. This is true, since in a more breadth-oriented structure, more evaluated memoization tables have to be stored in memory for later use. After this first pass, the second pass will reduce the leaves and the root to a bag size of one. This pass will also insert paths of successive forget nodes and

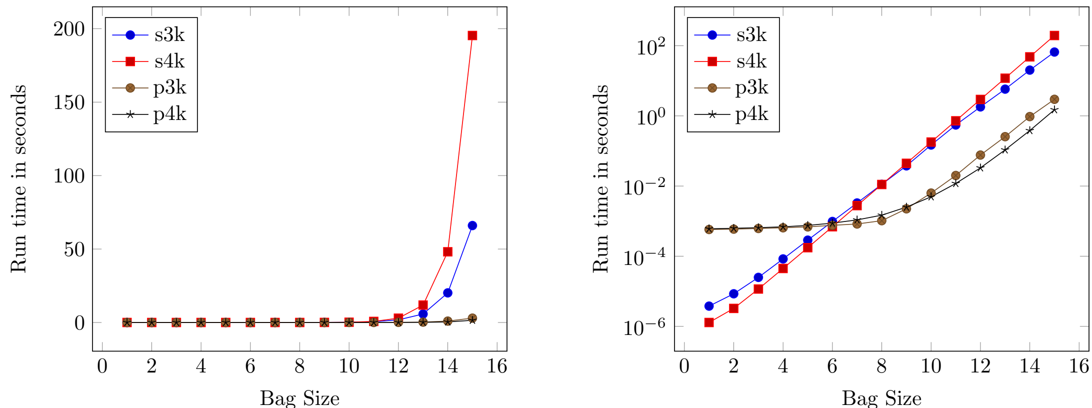
introduce nodes between nodes in the graph that have bags that differ more than one vertex. The result is a nice tree decomposition.

6 Experimental results

The sequential and parallel implementations of the *Alber*, and *vanRooij*$_2$ algorithms are denoted as follows: **S4k** is the sequential implementation of the *Alber* algorithm. **P4k** is the parallel implementation of the *Alber* algorithm. We use an **S**, or **P** to denote whether the implementation is sequential or parallel, respectively. The **4k** is derived from the $O^*(4^k)$ time complexity of the *Alber* algorithm. Using the same logic, we have **P3k** and **S3k** for the *vanRooij*$_2$ algorithm.

6.1 Parallel vs sequential implementations

When looking at Figure 1, one can see that the sequential algorithms have run times much lower than their parallel counterparts when bag size of the join nodes is very small, and as bag size increases, the run times increase exponentially. This consistent exponential increase contrasts with the parallel algorithms. One can see that there is a difference of multiple orders of magnitude between the run times for lower bag sizes for the sequential algorithms, while the run times are fairly consistent for these bag sizes for the parallel algorithms. The is the result of the copying of memory from and to device causing a bottleneck. As run time increases, one can see that also the parallel implementations increase in run time exponentially. This is due to the limited capacity of the used hardware: as the workload increases, not all blocks can be processed concurrently. When there is too much work to handle in parallel, blocks are queued until CUDA cores become available.



(a) The run times in seconds for the join nodes for all four algorithms.

(b) The results plotted against a logarithmically scaled y-axis.

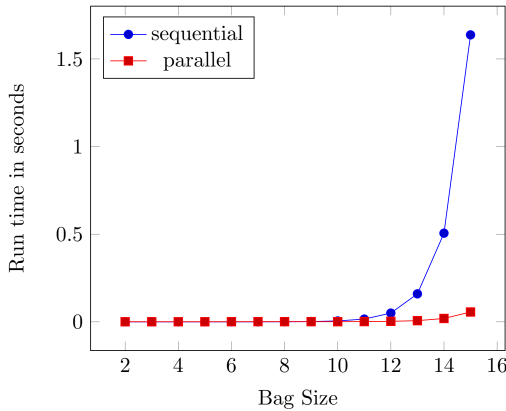
Figure 1: For each bag size, the run times for the join procedures are plotted for all four algorithms. Figure 1a shows the results with a linearly scaled y-axis, and Figure 1b uses a logarithmically scaled y-axis. The values are presented in Table 4.

At bag size ten, the difference in run time is already a factor of 36 between the sequential, and parallel implementation of the *Alber* algorithm. When the bag size is increased to 15, this becomes a factor of 130.

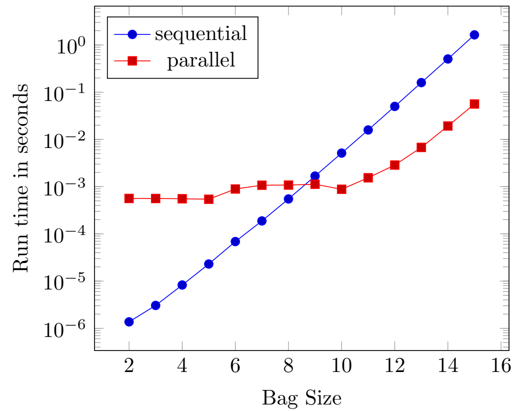
The trends that can be found in the join node between the sequential and parallel algorithms, can also be found in the introduce and forget nodes. Figures 2 and 3 show the average run times per bag size for the introduce, and forget nodes, respectively.

The run times for the introduce, and forget nodes stay fairly low when compared to those of the join node. The result of this, is that the parallel implementations overtake the sequential implementations at a later point, i.e. at a larger bag size. For both the introduce and forget procedure, this happens at bag size nine. One can also see that for both the introduce procedure and the forget procedure, run times associated with smaller bags are dominated by overhead, which was also the case for the join nodes. Only when the bag size increases to such an extent that the work exceeds the capacity of the GPU do we see an exponential increase in run time.

When bag size 15 is reached for the introduce node, the difference between the parallel and sequential algorithms is a factor of 29. The factor for the forget node becomes 13 at bag size 14. These numbers, along with the average run times, are a lot lower for the introduce, and forget node than they are for the join node. This is due to the relative simplicity of the introduce, and forget node. These nodes have a computational complexity of $O(3^k)$, whereas the **P3k** join has complexity $O(k^2 3^k)$ and the **P4k** algorithm has complexity $O(4^k)$.



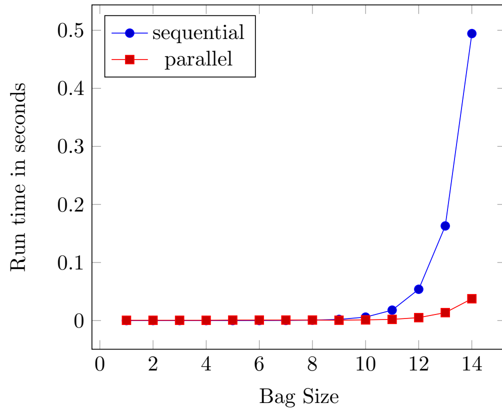
(a) The run times in seconds for the introduce nodes for the sequential and parallel algorithms.



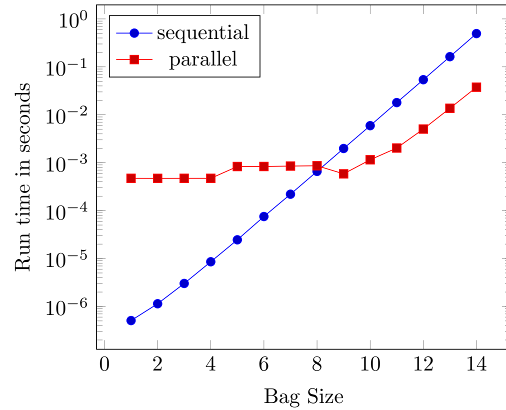
(b) The results plotted against a logarithmic y-axis.

Figure 2: The average run times in seconds for the introduce nodes in the sequential, and parallel algorithms. Figure 2b shows the results using a logarithmically scaled y-axis. The values are presented in Table 2.

Tables 2, 3, 5, and 6 in Appendix A also show the minimum, and maximum run times for each bag size and each node type. One can see that these values differ only marginally. This is due to the fact that the dynamic programming algorithm performs exactly the same operations for each run for a given tree decomposition. Because of this, the differences between the minimum and maximum run times can be credited to outside factors, like other processes running on the system. Table 2 shows are more significant difference between the minimum and maximum run times for the introduce procedure. This is due to the ϕ function discussed in Section 3.2.2. The amount of work that has to be done per configuration when this ϕ function is executed, is dependent on the number of neighbours that the introduced vertex v has in the current bag. This dependence of the run time of the algorithm on the structure of the graph is a property that is unique to the introduce procedure and is the reason why there is more variation in run time.



(a) The run times in seconds for the forget nodes for the sequential and parallel algorithms.

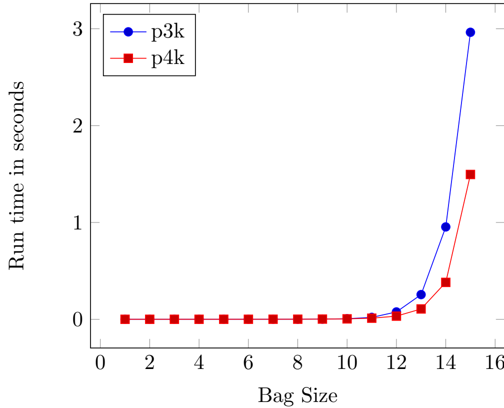


(b) The results plotted against a logarithmic y-axis.

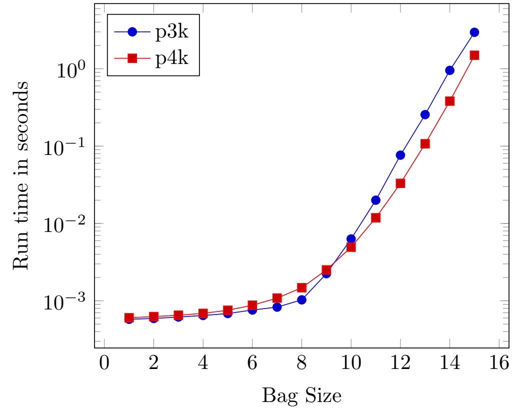
Figure 3: The average run times in seconds for the forget nodes in the sequential, and parallel algorithms. Figure 3b shows the results using a logarithmically scaled y-axis. The values are presented in Table 3.

6.2 Parallel implementations: P3k vs P4k

Since the parallel implementations of the *Alber* and *vanRooij*$_2$ algorithms use the same implementations for the introduce and forget procedure, we only have to look at the results of the join procedure when comparing the two. Figure 4a shows the difference in run times for the two parallel implementations. As one can see, the **P4k** algorithm outperforms the **P3k** algorithm for larger bag sizes. One can see that the algorithms have fairly equal run times up to a bag size of 10, after which **P4k** is the clear winner. From this we conclude that the *Alber* algorithm is more viable for parallel implementation for these bag sizes. This is mainly due to the state transformations that have to be performed on the memoization tables during the join of the **P3k** algorithm. These transformations cannot occur fully in parallel, but have to be performed in $O(k)$ sequential steps. What is more, is that the operation of combining the two transformed child tables has to be performed in sequence of the transformations for the **P3k** algorithm. This is in contrast with the **P4k** algorithm, for which all threads could theoretically be executed fully in parallel using streams. A more in depth discussion on the relation between hardware and these parallel algorithms can be found in Section 8.



(a) The results presented using a linear time scale.



(b) The results presented using a logarithmic time scale.

Figure 4: Comparisons for the run times of join nodes of the **P4k** and **P3k** algorithm. Figure 4a shows the results for the parallel algorithms. Figure 4b shows these results using a logarithmic scale on the y-axis. The values are presented in Table 4.

6.3 Sequential implementations: S3k vs S4k

For this comparison we also only have to look at the join procedure. Figure 1 shows the run times for the two sequential implementations. In contrast to what was discussed in Section 6.2, here the sequential implementation of the $vanRooij_2$ algorithm outperforms the sequential *Alber* algorithm. As can be seen in figure 1, the **S3k** algorithm overtakes the **S4k** algorithm at a bag size of nine. This is a surprising result, as we would expect something different when we look at the complexity of the two different operations. The complexity of the **S4k** join is $O(4^k)$. This is $O(k^2 3^k)$ for the **S3k** join. Theoretically, the **S3k** algorithm should outperform the **S4k** algorithm in run time at bag size 21 and up. When analyzing details of the implementations of both sequential algorithms, it becomes clear why this occurs.

As was explained in Section 3.3.4, the $vanRooij_2$ algorithm performs $O(k)$ multiplications for the $O(k3^k)$ configurations in the table of a join node. The $O(4^k)$ operations that have to be performed in the join of the *Alber* algorithm are in practice not of constant time. We need to perform several operations (see the divide condition in Section 3.2) for every configuration during the execution of the *Alber* algorithm; we need to find combinations that adhere to the *divide* condition. As the configurations have length $O(k)$, we add this term to the complexity of the *Alber* algorithm. If we now take the equation: $k^2 3^k = k4^k$ and solve for k, we get a value of around 6.5, which is a lot closer to the turn-over point of nine that we find empirically.

6.4 Varying block size of the parallel implementations

The results from the previous section were all from runs that used block size 1024. This is the maximum size of a block for the specific hardware that was used [17]. If a block is launched, it might be the case that redundant threads are used because not all threads are assigned an index that is within the range of the memoization table. If a single block is launched for all of the configurations (in the case of a small memoization table), then we lose the efficiency of the parallel execution. This, among other reasons, is why it might be beneficial to introduce a dynamic block size, which depends

on the type of node and the size of the memoization table. The parallel algorithms were run with varying static block sizes. As execution of threads is based on warps of size 32, it is recommended to use block sizes that are a multiple of 32. The chosen block sizes are 64, 128, 256, and 512. These runs are compared with the data that is already present, which was the result of runs that use a block size of 1024. The resulting data is presented in Tables 7, 8, 9, and 10, for the introduce and forget procedures, the **P3k** and **P4k** join procedures, respectively. A surprising result, is that these values do not differ significantly.

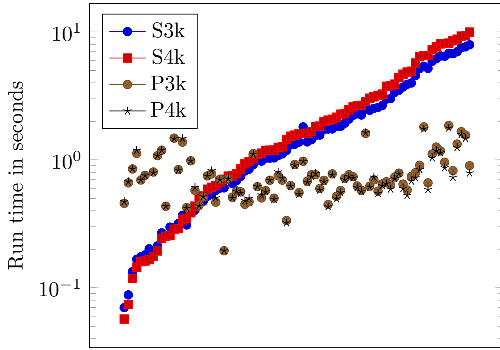
Since in theory the worst case run time of the procedures is equal to the average run time, one can assume that the differences between the minimum and maximum values in Tables 6, 2, and 3 are due to circumstantial factors (i.e. other processes that are running on the system), as was discussed in Section 6.1. If we take this difference as an error and compare this to the differences in the results for the varying block sizes, we can see that these differences never exceed the error. With this knowledge we conclude that using a variable block size does not significantly improve run times for the given bag sizes and the used hardware.

6.5 Comparing algorithms

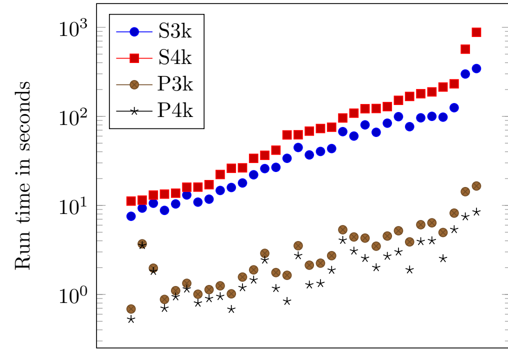
The global run times (we use the word 'global' to indicate that the time measurement was performed over the entire algorithm, not just individual nodes) for all tree decompositions are depicted in Figure 5. Figure 5a shows all runs that have a maximum run time of 10 seconds for the **S4k** algorithm, and Figure 5b shows the result for a maximum run time of 10 seconds and up (again for the **S4k** algorithm). Both figures show that the parallel algorithms outperform the sequential ones in most cases. In Figure 5a one can see that for a run time higher than around three seconds, no sequential runs are faster than the parallel ones. For lower run times, the results vary.

This variation is due to the differences in tree decompositions. As was shown in Section 6.1, sequential algorithms outperform their parallel counterparts when nodes have small bags. When a tree decomposition has very many nodes with small bags, these differences in run time start to add up. As was explained in Section 6.1 the run times for the parallel algorithms for nodes that have relatively small bags are dominated by the overhead caused by memcopies. When the global run time is not dominated by a (join) node with a large bag size, the global run time of the parallel algorithms will then also be dominated by the overhead caused by memcopies and the effective accumulation of differences in run times between them and their sequential counterparts.

From these figures, it becomes clear that the **S3k** algorithm will outperform the **S4k** algorithm when the global run times get higher. This is especially clear in Figure 5b. The parallel algorithms appear to perform comparably, as their markers are very close at every point. The difference between the two parallel implementations is depicted in Figure 6. Again, the graph is split into two figures to more clearly show the differences. Figure 6a shows maximum run times of one second for the **P4k** algorithm, and Figure 6b shows the maximum run times of one second and up (again for the **P4k** algorithm). With a maximum run time that is smaller than one second, the algorithms appear to perform comparably. When looking at Figure 6b however, a trend appears. The **P4k** algorithm outperforms the **P3k** algorithm for longer maximum run times. As was mentioned before, this is the result of nodes with large bag sizes dominating the global run time.

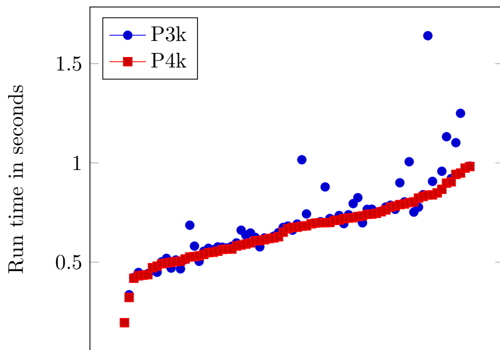


(a) Results for the tree decompositions that resulted in a run time smaller than ten seconds for the **S4k** algorithm.

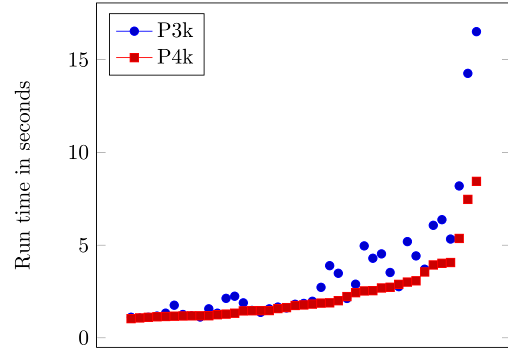


(b) Results for the tree decompositions that resulted in a run time larger than ten seconds for the **S4k** algorithm.

Figure 5: A comparison of the global run times for each algorithm. The run times are sorted based on the run time of the **S4k** algorithm. Figure 5a shows a comparison of the run times up to a max run time of 10 seconds. Figure 5b shows the run times for a max run time of 10 seconds and higher. Both graphs use a logarithmic scale on the y-axis.



(a) Results for the tree decompositions that resulted in a run time smaller than one second for the **P4k** algorithm.



(b) Results for the tree decompositions that resulted in a run time larger than one second for the **P4k** algorithm.

Figure 6: A comparison of the global run times for both parallel algorithms. The run times are sorted based on the run times of the **P4k** algorithm. Figure 5a shows a comparison of the run times up to a max run time of 1 seconds. Figure 5b shows the run times for a max run time of 1 seconds and higher.

The distribution that describes what algorithms performed the best for the tree decompositions can be found in Figure 7. Again, a clear trend appears from the data. Instances with smaller tree width are solved more quickly by the sequential algorithms, whereas instances with a larger tree width work better with the parallel algorithms. Tree width six and seven are almost exclusively dominated by the **S4k** algorithm. As we increase tree width to eight or nine, the results are more divided, but from that point and up, **P4k** is the clear winner. These results align with Figures 5 and 6. These results confirm very clearly how run times of algorithms on tree decompositions are largely determined by their largest bags, or in other words, their respective widths.

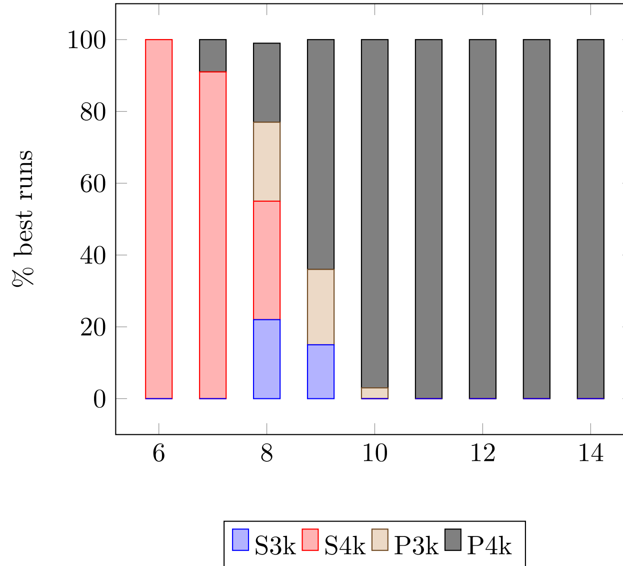


Figure 7: A bar plot of the distributions of optimal algorithms. The comparison was performed over the global run times.

6.6 A hybrid approach

The results in Sections 6.1, 6.2, and 6.3 show that for certain types of nodes, and certain bag sizes, given algorithms perform better than the others. As was mentioned, for smaller bag sizes the sequential algorithms perform better in general due to the absence of the overhead that is present for the parallel algorithms. With this information, we can introduce a hybrid algorithm, which decides what implementation of the given procedure to use during execution. This should, in theory, result in the most optimal global run times. Table 11 in Appendix A shows the optimal selection for each node type and bag size. These selections are based on the run times of the different implementations per procedure and bag size as discussed in Section 6.1. For both the introduce and forget nodes, the parallel algorithms start to outperform their sequential counterparts from a bag size of nine and up. For the join, we saw that this point was reached at bag size seven. The **P3k** algorithm outperforms the **P4k** algorithm for the join procedure for bag up to nine. From bag size ten and up, the **P4k** algorithm performs best. In Section 6.4, we determined that using a specified block size for a given node type and bag size does not significantly improve performance. This is why we exclude this factor from a hybrid approach, and why we stick to the original block size of 1024.

Figure 8 depicts the performance of the original four algorithms and the performance of the discussed hybrid implementation. For each run, the minimum run time of all four original algorithms are compared to the hybrid run time. For all of the tree decompositions, the hybrid implementation outperforms all of the other four. One can see that for very low run times, and especially very high run times, performance is comparable. As was mentioned, tree decompositions that have run times that are in these extremes are decisively solved more quickly by either the parallel, or the sequential implementations. For the tree decompositions that have run times that lie somewhere in the middle of Figure 8, it can go either way. Because of this, the run times of these tree decompositions might consist of relatively much overhead when parallel implementations are used. It might also be the

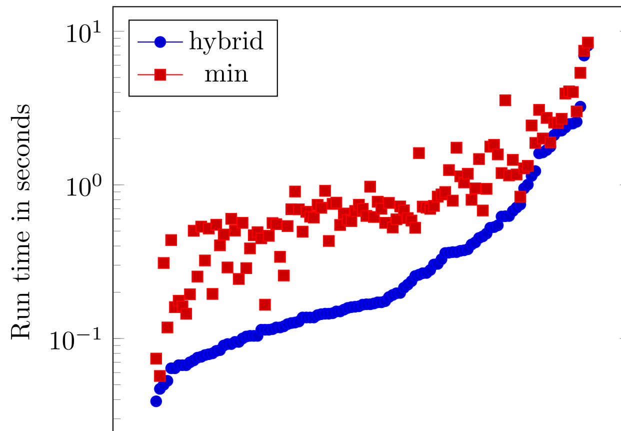


Figure 8: Minimum run time per tree decomposition for the original algorithms compared to the run times of the hybrid approach. The run times are ordered from lowest to highest, based on the run time of the hybrid algorithm. The time in seconds on the y-axis uses a logarithmic scale.

case that there are several (though relatively few) nodes that have a large bag size, which results in longer run times when sequential algorithms are used. When we select a procedure during execution, we can eliminate these factors.

7 Conclusion

This work is centered around four different implementations of algorithms for the Minimum Dominating Set problem using tree decompositions and dynamic programming. Two different algorithms are implemented, namely the *Alber* algorithm [1] and the $vanRooij_2$ algorithm [18]. Both are implemented as a sequential, and a parallel algorithm, the latter using the CUDA toolkit on a GPU. We have shown that the *Alber* and $vanRooij_2$ algorithms are very viable for parallel implementations. Depending on the bag size, speed ups of multiple orders of magnitude can be seen when comparing parallel and sequential implementations of the join procedures. These patterns do not only also strongly correlate with global run times, but also with tree width k.

The sequential implementations **S3k** and **S4k** prove to be more suited for nodes that have small bag size. This, because our parallel implementations **P3k** and **P4k** have the caveat of overhead (moving around memory, launching threads, etc.), which only becomes negligible with larger bag sizes. This overhead is very apparent in the results, where the run times per node for lower bag sizes all are very comparable. We conclude that, for the introduce and forget nodes, our parallel implementations outperform our sequential implementations from bag size nine. For the join procedure this point is at bag size seven, as this procedure is computationally much more complex.

While the complexity of the $vanRooij_2$ and the *Alber* algorithms predict otherwise, the **S3k** algorithm outperforms the **S4k** algorithm from a bag size of around nine for the join nodes. This was the result of an extra factor k that we had to add to the complexity of the *Alber* algorithm. The extra factor k comes from computations that have to be performed on configurations.

For the parallel algorithms, we see the exact opposite pattern occur. The **P4k** algorithm outperforms the **P3k** algorithm from a bag size of 10 and up. While the more complex work per configuration causes slower run times for the sequential join of the *Alber* algorithm, it proves to

be easier to create a parallel implementation which eliminates this factor (for the given bag sizes). This, in contrast with the join of the $vanRooij_2$ algorithm, which requires more sequential steps. The differences were less significant, though apparent and in line with the join procedure, for the introduce and forget procedures. This is the result of these processes being less complex, which means that differences in run times become more apparent with larger bag sizes.

By combining the individual procedures from the four implementations, we have created a hybrid algorithm. This algorithm selects the optimal implementation of the respective procedure, by looking at the bag size. This approach proves very effective, as it outperforms all four original implementations every run. This is the result of either eliminating overhead for small bag sizes, by selecting a sequential procedure, or by taking advantage of parallelization for large bag sizes by selecting a parallel procedure. We have found that varying the block size does not yield a significant effect on run times, at least not for these instances.

The trends as they are found for the different types of nodes, also appear for the global run times. This is an indication of how the global run times are dominated by the join procedures. We see that again, tree decompositions that resulted in shorter run times were consistently dominated by the sequential algorithms, while the tree decompositions that result in longer run times are all dominated by the parallel implementations. Not only for individual nodes and global run times do we see a clear pattern, a strong correlation between tree width k and best suited algorithm also continues in this trend. Again we see that small tree width works well with sequential algorithms, and large tree width prefers parallel implementations. The patterns between the parallel implementations, as well as the sequential implementations are again found when correlating with tree width, where the **S4k** algorithm is more suited for the very smallest tree widths compared to the **S3k** algorithm, and the **P4k** algorithm outperforms the **P3k** algorithm when tree width grows.

8 Discussion

Though this work provided some promising results, we were only able to experiment with tree decompositions that had a maximum tree width of 15. This limitation prevents us from exploring the potential of the parallel algorithms for higher tree width. As was mentioned, a tree width of 20 already causes the memory requirements to be exceedingly high, which makes the algorithm impractical if the tree width exceeds that point. It would be very interesting to see, however, how the results shown in this work would compare to results of graphs with tree width between 15 and 20. These runs would of course require more powerful hardware.

The run times of the parallel implementations are very dependent of the available hardware. If infinite cores were available, all threads in the q blocks that have to be executed during the **P4k** algorithm could be processed fully in parallel. The limitation here would be the capacity per thread. When q becomes large, the 2^q work that has to be performed per thread will become a bottleneck. A solution to this would be to use a block reduction procedure. Several approaches to this are discussed in [17]. The reduction over the 2^q individual calculations (see Equation 3) introduces logarithmic complexity term (as these approaches are of a divide-and-conquer nature): $O(\log(2^q)) = O(q)$. This is an operation that cannot be eliminated through parallelization. Since q ranges from 0 to $O(k)$, we can say that, for an implementation of **P4k** that uses block reduction, we have a non-parallelizable complexity term of $O(k)$. If we compare this to the **P3k** algorithm, we get a similar result. Since a transformation has to be performed in $O(k)$ sequential steps, this algorithm also has a nonparallelizable complexity term of $O(k)$. From this we conclude that, when k becomes sufficiently large, both algorithms will have comparable performance. An additional requirement is of course that we have enough memory. When k is within the range discussed in this work, we

conclude that the per-thread capacity does not bottleneck the algorithm to such an extend that it cannot outperform the **P3k** algorithm.

If infinite memory were available to us, we would not have to take into account table size. The result of this would be that we could run the implementations for all possible values of k. If we were to have limited computational power, i.e. a restricted amount of cores, we would observe a different result than what would be the case if we would have infinite cores. Recall that the computational complexity of the join procedures of the*Alber* and $vanRooij_2$ algorithms is $O(4^k)$ and $O(k^2 3^k)$, respectively. As was discussed in Section 6.5, theoretically, the $vanRooij_2$ algorithm should outperform that *Alber* algorithm, starting at a value for k of around 21. As k grows and the amount of cores stays the same, the practical benefit of performing calculations in parallel diminishes. The result of this, is that the run times of the parallel algorithms start to resemble those of the sequential ones. From this we conclude that, if memory is not a constraint, but the amount of cores is restricted, as k becomes very large, the $vanRooij_2$ algorithm will outperform the *Alber* algorithm.

For both the sequential and parallel algorithms optimizations are still possible. A lookup table for translating between integer and BCT formatted numbers would possibly introduce a significant speed up for the sequential algorithms. Though, even if run time for these algorithms would be cut in half, the parallel implementations would still be orders of magnitudes faster for high enough tree width.

The fact that varying block sizes did not affect the run times for the specific bag sizes (as discussed in Section 6.4) was surprising. One would expect that, especially for larger bag sizes, using the maximum block size of 1024 versus using a block size of 64 would yield significant improvements. Further research that uses more high-grade hardware could possibly show different results. This, because tests could be performed for larger bag sizes, and more individual processing units would be available, which introduces more capacity for parallelization.

9 Further research

We were unable to realize some optimizations for the algorithms. This section will discuss several of those optimizatons, as well as some suggestions for further research on the subject.

The **P4k** implementation that was used in this work assigns a thread to each configuration during the join procedure. The result is that each thread has to perform 2^q work, where q is the number of 0_1 states in the configuration. When q gets sufficiently large, this would result in a lot of sequential work performed by each thread. A solution to this would be to distribute the work that has to performed for those 2^q configurations over individual threads. Since the minimum value has to be determined, a block-reduce operation could be performed. Several techniques for this are available and presented in [17]. Experimentation would have to indicate what the point of diminishing return would be for this approach, as this would possibly introduce significant overhead for small values of q.

As the bag size of a node grows, the corresponding memoization table grows exponentially with it. This could possibly result in the memoization table not being able to fit in device memory. An adaptation of the approach in [11] could be used, where tables are split into chunks that are processed sequentially. Though this would introduce additional overhead from the additional memcopies, if the chunks are sufficiently large, this overhead becomes negligible in comparison to the calculations that have to be performed.

Tree decompositions could be optimized for the algorithms. We could focus more on keeping bag sizes of join nodes low. Since the join nodes are the most computationally expensive procedures of both the *Alber* and $vanRooij_2$ algorithms, decreasing these bag sizes would have significant impact

on the over all run times of the algorithms.

There are many problems that can be solved using dynamic programming on tree decompositions (as was mentioned in Section 1.1). One problem that could be used for further research on this subject would be the Counting Perfect Matches algorithm from [18]. This algorithm uses a comparable method for state transformations as the $vanRooij_2$ algorithm does. An interesting property of this algorithm is that the configurations are base-2. This would eliminate overhead from configuration transformations, and would allow for experimentation with much higher tree width. It would be interesting to see a comparison between a more optimized sequential algorithm (since no configuration transformations are necessary) and a parallel algorithm that allows for experimentation with more different values for k. Aside from the algorithms discussed in [18], many algorithms that use tree decompositions and dynamic programming could be implemented on the GPU and evaluated. The rank-based approach for solving the Steiner Tree problem and the Hamiltonian Cycle problem, as discussed in [9] is an example of such algorithms.

The run times of the introduce and forget procedures of the parallel implementations show a clear trend that is discussed in this work. One result stand out, however. Figures 2b and 3b show a small dip in run time for bag sizes ten and nine, respectively. What is more, is that these small dips are at the exact point where the run times go from staying fairly consistent, to growing exponentially. The results from the varying block sizes experiment show these same dips for all of the tested block sizes. Due to limited time, we were not able to explore this anomaly and provide an explanation. It would be interesting to see future work explain this property of the algorithms, as it might allow us to gain a better understanding of the practical implications of parallelization of these types of algorithms.

References

[1] Jochen Alber and Rolf Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In Sergio Rajsbaum, editor, *LATIN 2002: Theoretical Informatics*, pages 613–627, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[2] H. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, 25(6):1305–1317, 1996.

[3] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–23, 1993.

[4] Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Deterministic single exponential time algorithms for connectivity problems parameterized by treewidth. *Inf. Comput.*, 243(C):86–111, 2015.

[5] Hans L. Bodlaender, Fedor V. Fomin, Arie M. C. A. Koster, Dieter Kratsch, and Dimitrios M. Thilikos. On exact algorithms for treewidth. In *Algorithms – ESA 2006*, pages 672–683, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[6] Hans L. Bodlaender, John R. Gilbert, Hjálmtýr Hafsteinsson, and Ton Kloks. Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *J. Algorithms*, 18(2):238–255, 1995.

[7] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.

[8] V Bouchitté, D Kratsch, H Müller, and I Todinca. On treewidth approximations. *Discrete Applied Mathematics*, 136(2):183 – 196, 2004. The 1st Cologne-Twente Workshop on Graphs and Combinatorial Optimization.

[9] M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J. M. M. v. Rooij, and J. O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 150–159, 2011.

[10] Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Fast hamiltonicity checking via bases of perfect matchings. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC '13, pages 301–310, New York, NY, USA, 2013. ACM.

[11] Alessandro Dal Palù, Agostino Dovier, Andrea Formisano, and Enrico Pontelli. CUDASAT: SAT solving on GPUs. *Journal of Experimental and Theoretical Artificial Intelligence*, 27:1–24, 2014.

[12] Stefan Fafianie, Hans L. Bodlaender, and Jesper Nederlof. Speeding up dynamic programming with representative sets. In Gregory Gutin and Stefan Szeider, editors, *Parameterized and Exact Computation*, pages 321–334, Cham, 2013. Springer International Publishing.

[13] Johannes K. Fichte, Markus Hecher, Stefan Woltran, and Markus Zisser. Weighted Model Counting on the GPU by Exploiting Small Treewidth. In Yossi Azar, Hannah Bast, and Grzegorz Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer-Verlag Berlin Heidelberg, 1994.

[15] Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. In *Twenty-second annual ACM-SIAM symposium on Discrete algorithms*, pages 777–789. Society for Industrial and Applied Mathematics, 2011.

[16] NVIDIA Corporation. *Nvidia's next generation CUDA Compute Architecture: Fermi*. NVIDIA Corporation, 2009.

[17] NVIDIA Corporation. *CUDA Toolkit Documentation v10.1.168*. NVIDIA Corporation, 2019.

[18] Johan M. M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, pages 566–577, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[19] Michal Wlodarczyk. Clifford Algebras Meet Tree Decompositions. In *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)*, volume 63 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:18, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[20] Michal Ziobro and Marcin Pilipczuk. Finding Hamiltonian Cycle in Graphs of Bounded Treewidth: Experimental Evaluation. In Gianlorenzo D'Angelo, editor, *17th International Symposium on Experimental Algorithms (SEA 2018)*, volume 103 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Appendices

A Tables

Table 2: The average (Avg), max (Max), and min (Min) run times in seconds for the introduce nodes for both the sequential (Seq) and parallel (Par) implementations.

Bag Size	Seq Avg	Par Avg	Seq Max	Seq Min	Par Max	Par Min
2	$1.37 \cdot 10^{-6}$	$5.62 \cdot 10^{-4}$	$1.01 \cdot 10^{-4}$	$5.53 \cdot 10^{-7}$	$1.22 \cdot 10^{-3}$	$4.46 \cdot 10^{-4}$
3	$3.05 \cdot 10^{-6}$	$5.61 \cdot 10^{-4}$	$3.85 \cdot 10^{-5}$	$1.94 \cdot 10^{-6}$	$2.82 \cdot 10^{-3}$	$4.52 \cdot 10^{-4}$
4	$8.26 \cdot 10^{-6}$	$5.54 \cdot 10^{-4}$	$6.01 \cdot 10^{-5}$	$5.54 \cdot 10^{-6}$	$8.8 \cdot 10^{-4}$	$4.52 \cdot 10^{-4}$
5	$2.3 \cdot 10^{-5}$	$5.41 \cdot 10^{-4}$	$1.28 \cdot 10^{-4}$	$1.66 \cdot 10^{-5}$	$8.49 \cdot 10^{-4}$	$4.46 \cdot 10^{-4}$
6	$6.87 \cdot 10^{-5}$	$8.91 \cdot 10^{-4}$	$2.03 \cdot 10^{-3}$	$4.98 \cdot 10^{-5}$	$1.29 \cdot 10^{-3}$	$7.86 \cdot 10^{-4}$
7	$1.88 \cdot 10^{-4}$	$1.07 \cdot 10^{-3}$	$4.61 \cdot 10^{-4}$	$1.51 \cdot 10^{-4}$	$1.57 \cdot 10^{-3}$	$8.01 \cdot 10^{-4}$
8	$5.49 \cdot 10^{-4}$	$1.08 \cdot 10^{-3}$	$2.61 \cdot 10^{-3}$	$4.52 \cdot 10^{-4}$	$3.73 \cdot 10^{-3}$	$8.01 \cdot 10^{-4}$
9	$1.68 \cdot 10^{-3}$	$1.12 \cdot 10^{-3}$	$3.47 \cdot 10^{-3}$	$1.37 \cdot 10^{-3}$	$1.59 \cdot 10^{-3}$	$8.38 \cdot 10^{-4}$
10	$5.14 \cdot 10^{-3}$	$8.8 \cdot 10^{-4}$	$1.11 \cdot 10^{-2}$	$4.15 \cdot 10^{-3}$	$1.35 \cdot 10^{-3}$	$5.57 \cdot 10^{-4}$
11	$1.59 \cdot 10^{-2}$	$1.54 \cdot 10^{-3}$	$3.14 \cdot 10^{-2}$	$1.26 \cdot 10^{-2}$	$2.2 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$
12	$5 \cdot 10^{-2}$	$2.86 \cdot 10^{-3}$	$9.28 \cdot 10^{-2}$	$3.84 \cdot 10^{-2}$	$4.32 \cdot 10^{-3}$	$2.3 \cdot 10^{-3}$
13	0.16	$6.77 \cdot 10^{-3}$	0.31	0.12	$8.89 \cdot 10^{-3}$	$5.65 \cdot 10^{-3}$
14	0.51	$1.92 \cdot 10^{-2}$	0.8	0.36	$2.12 \cdot 10^{-2}$	$1.71 \cdot 10^{-2}$
15	1.64	$5.62 \cdot 10^{-2}$	1.94	1.47	$5.88 \cdot 10^{-2}$	$5.45 \cdot 10^{-2}$

Table 3: The average (Avg), max (Max), and min (Min) run times in seconds for the forget nodes for both the sequential (Seq) and parallel (Par) implementations.

Bag Size	Seq Avg	Par Avg	Seq Max	Seq Min	Par Max	Par Min
1	$5.08 \cdot 10^{-7}$	$4.72 \cdot 10^{-4}$	$5.26 \cdot 10^{-6}$	$2.76 \cdot 10^{-7}$	$7.33 \cdot 10^{-4}$	$4.44 \cdot 10^{-4}$
2	$1.14 \cdot 10^{-6}$	$4.71 \cdot 10^{-4}$	$6.64 \cdot 10^{-6}$	$8.3 \cdot 10^{-7}$	$3.1 \cdot 10^{-3}$	$4.39 \cdot 10^{-4}$
3	$3.02 \cdot 10^{-6}$	$4.72 \cdot 10^{-4}$	$7.47 \cdot 10^{-6}$	$2.49 \cdot 10^{-6}$	$4.47 \cdot 10^{-3}$	$4.43 \cdot 10^{-4}$
4	$8.57 \cdot 10^{-6}$	$4.72 \cdot 10^{-4}$	$1.99 \cdot 10^{-5}$	$7.75 \cdot 10^{-6}$	$6.52 \cdot 10^{-4}$	$4.4 \cdot 10^{-4}$
5	$2.46 \cdot 10^{-5}$	$8.32 \cdot 10^{-4}$	$5.67 \cdot 10^{-5}$	$2.3 \cdot 10^{-5}$	$1.12 \cdot 10^{-3}$	$7.84 \cdot 10^{-4}$
6	$7.53 \cdot 10^{-5}$	$8.33 \cdot 10^{-4}$	$1.29 \cdot 10^{-3}$	$6.89 \cdot 10^{-5}$	$1.11 \cdot 10^{-3}$	$7.88 \cdot 10^{-4}$
7	$2.2 \cdot 10^{-4}$	$8.5 \cdot 10^{-4}$	$1.24 \cdot 10^{-3}$	$2.07 \cdot 10^{-4}$	$1.2 \cdot 10^{-3}$	$8.01 \cdot 10^{-4}$
8	$6.58 \cdot 10^{-4}$	$8.63 \cdot 10^{-4}$	$1.9 \cdot 10^{-3}$	$6.15 \cdot 10^{-4}$	$1.23 \cdot 10^{-3}$	$8.11 \cdot 10^{-4}$
9	$1.98 \cdot 10^{-3}$	$5.85 \cdot 10^{-4}$	$2.34 \cdot 10^{-3}$	$1.88 \cdot 10^{-3}$	$1.17 \cdot 10^{-3}$	$5.32 \cdot 10^{-4}$
10	$5.92 \cdot 10^{-3}$	$1.15 \cdot 10^{-3}$	$6.39 \cdot 10^{-3}$	$5.68 \cdot 10^{-3}$	$1.72 \cdot 10^{-3}$	$1.04 \cdot 10^{-3}$
11	$1.79 \cdot 10^{-2}$	$2.02 \cdot 10^{-3}$	$1.94 \cdot 10^{-2}$	$1.73 \cdot 10^{-2}$	$2.66 \cdot 10^{-3}$	$1.75 \cdot 10^{-3}$
12	$5.39 \cdot 10^{-2}$	$5.03 \cdot 10^{-3}$	$5.66 \cdot 10^{-2}$	$5.25 \cdot 10^{-2}$	$6.22 \cdot 10^{-3}$	$4.58 \cdot 10^{-3}$
13	0.16	$1.37 \cdot 10^{-2}$	0.17	0.16	$1.58 \cdot 10^{-2}$	$1.28 \cdot 10^{-2}$
14	0.49	$3.76 \cdot 10^{-2}$	0.5	0.49	$3.77 \cdot 10^{-2}$	$3.75 \cdot 10^{-2}$

Table 4: The average run times in seconds for the join nodes.

Bag Size	S3k	S4k	P3k	P4k
1	$3.78 \cdot 10^{-6}$	$1.29 \cdot 10^{-6}$	$5.77 \cdot 10^{-4}$	$6.04 \cdot 10^{-4}$
2	$8.49 \cdot 10^{-6}$	$3.25 \cdot 10^{-6}$	$5.91 \cdot 10^{-4}$	$6.25 \cdot 10^{-4}$
3	$2.49 \cdot 10^{-5}$	$1.16 \cdot 10^{-5}$	$6.17 \cdot 10^{-4}$	$6.51 \cdot 10^{-4}$
4	$8.37 \cdot 10^{-5}$	$4.47 \cdot 10^{-5}$	$6.46 \cdot 10^{-4}$	$6.88 \cdot 10^{-4}$
5	$2.87 \cdot 10^{-4}$	$1.75 \cdot 10^{-4}$	$6.85 \cdot 10^{-4}$	$7.55 \cdot 10^{-4}$
6	$9.77 \cdot 10^{-4}$	$6.92 \cdot 10^{-4}$	$7.6 \cdot 10^{-4}$	$8.77 \cdot 10^{-4}$
7	$3.29 \cdot 10^{-3}$	$2.77 \cdot 10^{-3}$	$8.29 \cdot 10^{-4}$	$1.08 \cdot 10^{-3}$
8	$1.12 \cdot 10^{-2}$	$1.11 \cdot 10^{-2}$	$1.03 \cdot 10^{-3}$	$1.48 \cdot 10^{-3}$
9	$3.76 \cdot 10^{-2}$	$4.45 \cdot 10^{-2}$	$2.24 \cdot 10^{-3}$	$2.51 \cdot 10^{-3}$
10	0.15	0.18	$6.31 \cdot 10^{-3}$	$4.94 \cdot 10^{-3}$
11	0.55	0.72	$2 \cdot 10^{-2}$	$1.18 \cdot 10^{-2}$
12	1.8	2.94	$7.64 \cdot 10^{-2}$	$3.3 \cdot 10^{-2}$
13	5.78	11.86	0.26	0.11
14	20.17	48.14	0.95	0.38
15	65.99	195.36	2.96	1.5

Table 5: The average (Avg), minimum (Min), and maximum (Max) run times in seconds for the join nodes. These results are for the **S3k** and **S4k** algorithms.

Bag Size	S3k Avg	S4k Avg	S3k Max	S3k Min	S4k Max	S4k Min
1	$3.78 \cdot 10^{-6}$	$1.29 \cdot 10^{-6}$	$4.15 \cdot 10^{-6}$	$3.6 \cdot 10^{-6}$	$1.66 \cdot 10^{-6}$	$1.11 \cdot 10^{-6}$
2	$8.49 \cdot 10^{-6}$	$3.25 \cdot 10^{-6}$	$3.16 \cdot 10^{-4}$	$5.81 \cdot 10^{-6}$	$4.43 \cdot 10^{-6}$	$2.77 \cdot 10^{-6}$
3	$2.49 \cdot 10^{-5}$	$1.16 \cdot 10^{-5}$	$2.04 \cdot 10^{-3}$	$2.1 \cdot 10^{-5}$	$2.55 \cdot 10^{-5}$	$1.05 \cdot 10^{-5}$
4	$8.37 \cdot 10^{-5}$	$4.47 \cdot 10^{-5}$	$5.19 \cdot 10^{-4}$	$7.64 \cdot 10^{-5}$	$7.06 \cdot 10^{-5}$	$4.15 \cdot 10^{-5}$
5	$2.87 \cdot 10^{-4}$	$1.75 \cdot 10^{-4}$	$3.35 \cdot 10^{-3}$	$2.69 \cdot 10^{-4}$	$2.21 \cdot 10^{-4}$	$1.62 \cdot 10^{-4}$
6	$9.77 \cdot 10^{-4}$	$6.92 \cdot 10^{-4}$	$2.62 \cdot 10^{-3}$	$9.19 \cdot 10^{-4}$	$7.5 \cdot 10^{-4}$	$6.41 \cdot 10^{-4}$
7	$3.29 \cdot 10^{-3}$	$2.77 \cdot 10^{-3}$	$3.65 \cdot 10^{-3}$	$3.14 \cdot 10^{-3}$	$2.97 \cdot 10^{-3}$	$2.58 \cdot 10^{-3}$
8	$1.12 \cdot 10^{-2}$	$1.11 \cdot 10^{-2}$	$1.24 \cdot 10^{-2}$	$1.07 \cdot 10^{-2}$	$1.15 \cdot 10^{-2}$	$1.06 \cdot 10^{-2}$
9	$3.76 \cdot 10^{-2}$	$4.45 \cdot 10^{-2}$	$4.11 \cdot 10^{-2}$	$3.6 \cdot 10^{-2}$	$4.63 \cdot 10^{-2}$	$4.26 \cdot 10^{-2}$
10	0.15	0.18	0.17	0.14	0.19	0.17
11	0.55	0.72	0.58	0.53	0.75	0.71
12	1.8	2.94	2.17	1.75	3.01	2.88
13	5.78	11.86	5.95	5.63	12.11	11.68
14	20.17	48.14	20.56	19.78	48.73	47.58
15	65.99	195.36	66.28	65.79	196.36	194.2

Table 6: The average (Avg), minimum (Min), and maximum (Max) run times in seconds for the join nodes. These results are for the **P3k** and **P4k** algorithms.

Bag Size	P3k Avg	P4k Avg	P3k Max	P3k Min	P4k Max	P4k Min
1	$5.77 \cdot 10^{-4}$	$6.04 \cdot 10^{-4}$	$5.83 \cdot 10^{-4}$	$5.72 \cdot 10^{-4}$	$6.09 \cdot 10^{-4}$	$5.96 \cdot 10^{-4}$
2	$5.91 \cdot 10^{-4}$	$6.25 \cdot 10^{-4}$	$6.9 \cdot 10^{-4}$	$5.6 \cdot 10^{-4}$	$8.43 \cdot 10^{-4}$	$5.95 \cdot 10^{-4}$
3	$6.17 \cdot 10^{-4}$	$6.51 \cdot 10^{-4}$	$8.08 \cdot 10^{-4}$	$5.89 \cdot 10^{-4}$	$2.85 \cdot 10^{-3}$	$6.14 \cdot 10^{-4}$
4	$6.46 \cdot 10^{-4}$	$6.88 \cdot 10^{-4}$	$8.22 \cdot 10^{-4}$	$6.13 \cdot 10^{-4}$	$9.12 \cdot 10^{-4}$	$6.52 \cdot 10^{-4}$
5	$6.85 \cdot 10^{-4}$	$7.55 \cdot 10^{-4}$	$9.72 \cdot 10^{-4}$	$6.48 \cdot 10^{-4}$	$1.25 \cdot 10^{-3}$	$7.13 \cdot 10^{-4}$
6	$7.6 \cdot 10^{-4}$	$8.77 \cdot 10^{-4}$	$9.81 \cdot 10^{-4}$	$7.33 \cdot 10^{-4}$	$1.16 \cdot 10^{-3}$	$8.37 \cdot 10^{-4}$
7	$8.29 \cdot 10^{-4}$	$1.08 \cdot 10^{-3}$	$2.89 \cdot 10^{-3}$	$8 \cdot 10^{-4}$	$1.43 \cdot 10^{-3}$	$1.03 \cdot 10^{-3}$
8	$1.03 \cdot 10^{-3}$	$1.48 \cdot 10^{-3}$	$1.44 \cdot 10^{-3}$	$9.88 \cdot 10^{-4}$	$1.89 \cdot 10^{-3}$	$1.39 \cdot 10^{-3}$
9	$2.24 \cdot 10^{-3}$	$2.51 \cdot 10^{-3}$	$2.42 \cdot 10^{-3}$	$2.17 \cdot 10^{-3}$	$3.06 \cdot 10^{-3}$	$2.26 \cdot 10^{-3}$
10	$6.31 \cdot 10^{-3}$	$4.94 \cdot 10^{-3}$	$6.9 \cdot 10^{-3}$	$6 \cdot 10^{-3}$	$6.89 \cdot 10^{-3}$	$4.38 \cdot 10^{-3}$
11	$2 \cdot 10^{-2}$	$1.18 \cdot 10^{-2}$	$2.08 \cdot 10^{-2}$	$1.87 \cdot 10^{-2}$	$1.61 \cdot 10^{-2}$	$1.04 \cdot 10^{-2}$
12	$7.64 \cdot 10^{-2}$	$3.3 \cdot 10^{-2}$	$7.96 \cdot 10^{-2}$	$7.38 \cdot 10^{-2}$	$4.04 \cdot 10^{-2}$	$2.85 \cdot 10^{-2}$
13	0.26	0.11	0.27	0.24	0.13	$8.95 \cdot 10^{-2}$
14	0.95	0.38	0.98	0.92	0.42	0.35
15	2.96	1.5	3.12	2.88	1.52	1.47

Table 7: The average run times in seconds per bag size for a given block size. These results are for the introduce procedure.

Bag Size	64	128	256	512	1024
2	$5.64 \cdot 10^{-4}$	$5.75 \cdot 10^{-4}$	$5.65 \cdot 10^{-4}$	$5.76 \cdot 10^{-4}$	$5.62 \cdot 10^{-4}$
3	$5.65 \cdot 10^{-4}$	$5.74 \cdot 10^{-4}$	$5.65 \cdot 10^{-4}$	$5.71 \cdot 10^{-4}$	$5.61 \cdot 10^{-4}$
4	$5.59 \cdot 10^{-4}$	$5.66 \cdot 10^{-4}$	$5.59 \cdot 10^{-4}$	$5.63 \cdot 10^{-4}$	$5.54 \cdot 10^{-4}$
5	$5.46 \cdot 10^{-4}$	$5.54 \cdot 10^{-4}$	$5.46 \cdot 10^{-4}$	$5.52 \cdot 10^{-4}$	$5.41 \cdot 10^{-4}$
6	$8.79 \cdot 10^{-4}$	$9.1 \cdot 10^{-4}$	$8.8 \cdot 10^{-4}$	$9.07 \cdot 10^{-4}$	$8.91 \cdot 10^{-4}$
7	$1.04 \cdot 10^{-3}$	$1.09 \cdot 10^{-3}$	$1.04 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	$1.07 \cdot 10^{-3}$
8	$1.04 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	$1.05 \cdot 10^{-3}$	$1.1 \cdot 10^{-3}$	$1.08 \cdot 10^{-3}$
9	$1.08 \cdot 10^{-3}$	$1.14 \cdot 10^{-3}$	$1.09 \cdot 10^{-3}$	$1.14 \cdot 10^{-3}$	$1.12 \cdot 10^{-3}$
10	$8.56 \cdot 10^{-4}$	$8.89 \cdot 10^{-4}$	$8.55 \cdot 10^{-4}$	$8.87 \cdot 10^{-4}$	$8.8 \cdot 10^{-4}$
11	$1.49 \cdot 10^{-3}$	$1.57 \cdot 10^{-3}$	$1.48 \cdot 10^{-3}$	$1.57 \cdot 10^{-3}$	$1.54 \cdot 10^{-3}$
12	$2.85 \cdot 10^{-3}$	$2.86 \cdot 10^{-3}$	$2.86 \cdot 10^{-3}$	$2.86 \cdot 10^{-3}$	$2.86 \cdot 10^{-3}$
13	$7.02 \cdot 10^{-3}$	$6.82 \cdot 10^{-3}$	$7.05 \cdot 10^{-3}$	$6.84 \cdot 10^{-3}$	$6.77 \cdot 10^{-3}$
14	$1.94 \cdot 10^{-2}$	$1.93 \cdot 10^{-2}$	$1.94 \cdot 10^{-2}$	$1.91 \cdot 10^{-2}$	$1.92 \cdot 10^{-2}$
15	$5.65 \cdot 10^{-2}$	$5.65 \cdot 10^{-2}$	$5.74 \cdot 10^{-2}$	$5.6 \cdot 10^{-2}$	$5.62 \cdot 10^{-2}$

Table 8: The average run times in seconds per bag size for a given block size. These results are for the forget procedure.

Bag Size	64	128	256	512	1024
1	$4.69 \cdot 10^{-4}$	$4.8 \cdot 10^{-4}$	$4.69 \cdot 10^{-4}$	$4.85 \cdot 10^{-4}$	$4.72 \cdot 10^{-4}$
2	$4.71 \cdot 10^{-4}$	$4.8 \cdot 10^{-4}$	$4.7 \cdot 10^{-4}$	$4.73 \cdot 10^{-4}$	$4.71 \cdot 10^{-4}$
3	$4.71 \cdot 10^{-4}$	$4.81 \cdot 10^{-4}$	$4.71 \cdot 10^{-4}$	$4.74 \cdot 10^{-4}$	$4.72 \cdot 10^{-4}$
4	$4.71 \cdot 10^{-4}$	$4.81 \cdot 10^{-4}$	$4.71 \cdot 10^{-4}$	$4.75 \cdot 10^{-4}$	$4.72 \cdot 10^{-4}$
5	$8.1 \cdot 10^{-4}$	$8.48 \cdot 10^{-4}$	$8.1 \cdot 10^{-4}$	$8.53 \cdot 10^{-4}$	$8.32 \cdot 10^{-4}$
6	$8.11 \cdot 10^{-4}$	$8.46 \cdot 10^{-4}$	$8.1 \cdot 10^{-4}$	$8.48 \cdot 10^{-4}$	$8.33 \cdot 10^{-4}$
7	$8.28 \cdot 10^{-4}$	$8.62 \cdot 10^{-4}$	$8.31 \cdot 10^{-4}$	$8.63 \cdot 10^{-4}$	$8.5 \cdot 10^{-4}$
8	$8.45 \cdot 10^{-4}$	$8.76 \cdot 10^{-4}$	$8.46 \cdot 10^{-4}$	$8.71 \cdot 10^{-4}$	$8.63 \cdot 10^{-4}$
9	$5.7 \cdot 10^{-4}$	$5.96 \cdot 10^{-4}$	$5.69 \cdot 10^{-4}$	$5.94 \cdot 10^{-4}$	$5.85 \cdot 10^{-4}$
10	$1.13 \cdot 10^{-3}$	$1.16 \cdot 10^{-3}$	$1.12 \cdot 10^{-3}$	$1.16 \cdot 10^{-3}$	$1.15 \cdot 10^{-3}$
11	$2.01 \cdot 10^{-3}$	$2.01 \cdot 10^{-3}$	$1.99 \cdot 10^{-3}$	$2.01 \cdot 10^{-3}$	$2.02 \cdot 10^{-3}$
12	$5.26 \cdot 10^{-3}$	$5.08 \cdot 10^{-3}$	$5.24 \cdot 10^{-3}$	$5.07 \cdot 10^{-3}$	$5.03 \cdot 10^{-3}$
13	$1.41 \cdot 10^{-2}$	$1.35 \cdot 10^{-2}$	$1.43 \cdot 10^{-2}$	$1.34 \cdot 10^{-2}$	$1.37 \cdot 10^{-2}$
14	$4 \cdot 10^{-2}$	$3.89 \cdot 10^{-2}$	$4.15 \cdot 10^{-2}$	$3.76 \cdot 10^{-2}$	$3.76 \cdot 10^{-2}$

Table 9: The average run times in seconds per bag size for a given block size. These results are for the join procedure of the **P3k** algorithm.

Bag Size	64	128	256	512	1024
1	$5.6 \cdot 10^{-4}$	$5.71 \cdot 10^{-4}$	$5.61 \cdot 10^{-4}$	$5.78 \cdot 10^{-4}$	$5.77 \cdot 10^{-4}$
2	$5.94 \cdot 10^{-4}$	$6.01 \cdot 10^{-4}$	$5.93 \cdot 10^{-4}$	$5.99 \cdot 10^{-4}$	$5.91 \cdot 10^{-4}$
3	$6.24 \cdot 10^{-4}$	$6.29 \cdot 10^{-4}$	$6.23 \cdot 10^{-4}$	$6.22 \cdot 10^{-4}$	$6.17 \cdot 10^{-4}$
4	$6.57 \cdot 10^{-4}$	$6.59 \cdot 10^{-4}$	$6.52 \cdot 10^{-4}$	$6.52 \cdot 10^{-4}$	$6.46 \cdot 10^{-4}$
5	$6.91 \cdot 10^{-4}$	$6.95 \cdot 10^{-4}$	$6.88 \cdot 10^{-4}$	$6.92 \cdot 10^{-4}$	$6.85 \cdot 10^{-4}$
6	$7.29 \cdot 10^{-4}$	$7.41 \cdot 10^{-4}$	$7.31 \cdot 10^{-4}$	$7.65 \cdot 10^{-4}$	$7.6 \cdot 10^{-4}$
7	$7.68 \cdot 10^{-4}$	$7.85 \cdot 10^{-4}$	$7.7 \cdot 10^{-4}$	$7.95 \cdot 10^{-4}$	$8.29 \cdot 10^{-4}$
8	$9.75 \cdot 10^{-4}$	$9.96 \cdot 10^{-4}$	$9.95 \cdot 10^{-4}$	$1.03 \cdot 10^{-3}$	$1.03 \cdot 10^{-3}$
9	$2.1 \cdot 10^{-3}$	$2.12 \cdot 10^{-3}$	$2.11 \cdot 10^{-3}$	$2.18 \cdot 10^{-3}$	$2.24 \cdot 10^{-3}$
10	$6.12 \cdot 10^{-3}$	$6.17 \cdot 10^{-3}$	$6.15 \cdot 10^{-3}$	$6.24 \cdot 10^{-3}$	$6.31 \cdot 10^{-3}$
11	$1.96 \cdot 10^{-2}$	$1.98 \cdot 10^{-2}$	$1.94 \cdot 10^{-2}$	$1.95 \cdot 10^{-2}$	$2 \cdot 10^{-2}$
12	$7.71 \cdot 10^{-2}$	$7.72 \cdot 10^{-2}$	$7.6 \cdot 10^{-2}$	$7.51 \cdot 10^{-2}$	$7.64 \cdot 10^{-2}$
13	0.26	0.26	0.26	0.25	0.26
14	0.97	0.98	0.98	0.94	0.95
15	3.04	3.21	3.1	2.89	2.96

Table 10: The average run times in seconds per bag size for a given block size. These results are for the join procedure of the **P4k** algorithm.

Bag Size	64	128	256	512	1024
1	$5.94 \cdot 10^{-4}$	$6.08 \cdot 10^{-4}$	$6.01 \cdot 10^{-4}$	$6.38 \cdot 10^{-4}$	$6.04 \cdot 10^{-4}$
2	$6.24 \cdot 10^{-4}$	$6.36 \cdot 10^{-4}$	$6.26 \cdot 10^{-4}$	$6.31 \cdot 10^{-4}$	$6.25 \cdot 10^{-4}$
3	$6.53 \cdot 10^{-4}$	$6.61 \cdot 10^{-4}$	$6.54 \cdot 10^{-4}$	$6.5 \cdot 10^{-4}$	$6.51 \cdot 10^{-4}$
4	$6.9 \cdot 10^{-4}$	$6.98 \cdot 10^{-4}$	$6.91 \cdot 10^{-4}$	$6.92 \cdot 10^{-4}$	$6.88 \cdot 10^{-4}$
5	$7.57 \cdot 10^{-4}$	$7.65 \cdot 10^{-4}$	$7.56 \cdot 10^{-4}$	$7.62 \cdot 10^{-4}$	$7.55 \cdot 10^{-4}$
6	$8.73 \cdot 10^{-4}$	$8.82 \cdot 10^{-4}$	$8.69 \cdot 10^{-4}$	$8.92 \cdot 10^{-4}$	$8.77 \cdot 10^{-4}$
7	$1.08 \cdot 10^{-3}$	$1.09 \cdot 10^{-3}$	$1.07 \cdot 10^{-3}$	$1.09 \cdot 10^{-3}$	$1.08 \cdot 10^{-3}$
8	$1.48 \cdot 10^{-3}$	$1.49 \cdot 10^{-3}$	$1.48 \cdot 10^{-3}$	$1.5 \cdot 10^{-3}$	$1.48 \cdot 10^{-3}$
9	$2.52 \cdot 10^{-3}$	$2.52 \cdot 10^{-3}$	$2.48 \cdot 10^{-3}$	$2.52 \cdot 10^{-3}$	$2.51 \cdot 10^{-3}$
10	$5.05 \cdot 10^{-3}$	$5.03 \cdot 10^{-3}$	$4.97 \cdot 10^{-3}$	$5 \cdot 10^{-3}$	$4.94 \cdot 10^{-3}$
11	$1.24 \cdot 10^{-2}$	$1.22 \cdot 10^{-2}$	$1.21 \cdot 10^{-2}$	$1.2 \cdot 10^{-2}$	$1.18 \cdot 10^{-2}$
12	$3.48 \cdot 10^{-2}$	$3.49 \cdot 10^{-2}$	$3.47 \cdot 10^{-2}$	$3.39 \cdot 10^{-2}$	$3.3 \cdot 10^{-2}$
13	0.12	0.12	0.11	0.11	0.11
14	0.41	0.41	0.41	0.4	0.38
15	1.6	1.59	1.56	1.56	1.5

Table 11: The optimal selection for each bag size, and node type. The forget procedure has no instances of bag size 15.

Bag Size	Introduce	Forget	Join
1	sequential	sequential	**S4k**
2	sequential	sequential	**S4k**
3	sequential	sequential	**S4k**
4	sequential	sequential	**S4k**
5	sequential	sequential	**S4k**
6	sequential	sequential	**S4k**
7	sequential	sequential	**P3k**
8	sequential	sequential	**P3k**
9	parallel	parallel	**P3k**
10	parallel	parallel	**P4k**
11	parallel	parallel	**P4k**
12	parallel	parallel	**P4k**
13	parallel	parallel	**P4k**
14	parallel	parallel	**P4k**
15	parallel	n.a.	**P4k**