MASTER THESIS

Symbolic Model Checking with Partitioned BDDs in Distributed Systems

Author:
Janina TORBECKE
(s0191981)

Graduation Committee:
Jaco VAN DE POL
Wytse OORTWIJN
Ana VARBANESCU
Marieke HUISMAN

September 17, 2017

UNIVERSITY OF TWENTE.
Formal Methods and Tools (FMT)

Abstract

In symbolic model checking Binary Decision Diagrams (BDDs) are often used to represent states of a system in a compressed way. By using reachability analysis the system's entire state space can be explored. However, even with symbolic representation models can grow exponentially during an analysis such that they do not fit in a single machine's working memory. Both multi-core and distributed reachability algorithms exist, but the combination of both is still uncommon.

In this work we present a design for multi-core distributed reachability analysis. Our work is based on the multi-core model checking tool Sylvan and combines this with a BDD partitioning approach. As network traffic is one of the bottlenecks that are often reported in distributed reachability designs, we tried to minimize communication between machines.

Our benchmark results show that our current design does not fully utilize available hardware capacity, but nevertheless our implementation was able to achieve speedups up to 29 compared to a linear execution and up to 5.3 compared to an existing multi-core distributed analysis tool [11].

Contents

List of Figures

Chapter 1

Introduction

In recent days the application of program verification became more important and challenging as the number of software solutions and their complexity increased. For instance, in the last fifty years, the size of aircraft software grew from 1.9 million lines of code (F-22) to over 24 million lines of code (F-35) [1]. One common program verification technique is *model checking*, where an abstract model of the system is built. This model can then be used to check if the software meets the requirements imposed on it.

In model checking the a program is reflected by its state space. The state space is defined as a set of all possible combinations of variable assignments in the program. Thereby the software can be seen as a graph such that common graph analysis tools can be used to test for certain desired or undesired outcomes and properties. For instance, it can be tested if a certain error state is reachable from a given initial state of the system. Since verifying even small programs can result in exploring huge state spaces, several mechanisms to cope with this state space explosion problem have been developed. One method is to decrease the number of states, another is to use faster or more hardware resources.

There are different ways to decrease the state space, for example Partial Order Reduction or Bisimulation Minimization. Another method is to represent the state space in a compressed manner, *symbolically* instead of *explicitly*, by using *Reduced Ordered Binary Decision Diagrams* – for the purpose of convenience just called Binary Decision Diagrams (BDDs). BDDs can describe Boolean functions and are therefore capable of storing sets of states and transitions from one state to another. It is also possible to represent structures like Petri Nets or cyclic graphs as BDDs [2]. This could be useful for dense graphs – graphs with a very high number of edges compared to the number of nodes. Due to the dif-

ferent structure of BDDs compared to less restrictive graphs, in many of these cases the number of edges tends to decrease compared to the initial representation. [3] By using BDD operations the state space of a system can be generated and analyzed. Although compressing the model by a symbolic representation (symbolic model checking) yields a significant reduction of states, the size of the state space remains a limiting factor.

It is also possible to use more processing units and parallelize the computation. Recent work shows that a significant speedup factor of up to 38 can be reached with 48 cores by the parallel model checker Sylvan [4]. However, if the size of the model exceeds the size of the main memory, it is necessary to distribute the computation over multiple machines. Although many algorithms for distributed symbolic verification have been developed [5–8], most of them can process big models, but do not result in speedup. The main two reasons are that the computation is either executed sequentially (no speedup through parallelization), e.g. through horizontal partitioning [6, 9], or it is slowed down due to workload or memory imbalance or communication overhead between machines. As far as we know there is only one approach that combines both multi-core and distributed architectures. This was done by Oortwijn, van Dijk and van de Pol in [10]. Their BDD package (called DistBDD) reached speedups up to 51.1 and is even able to outperform single-machine computations with Sylvan when memory runs short. [11]

In this work we focus on symbolic model checking with BDDs and the optimization of their distribution and processing on multiple machines. In particular, we will focus on limiting communication overhead and finding a suitable BDD partitioning strategy with the goal to obtain speedups in distributed symbolic verification. As [11] has shown the advantages of combining multi-core with distributed algorithms, we will follow this approach.

1.1 Challenges for Distributed Model Checking

1.1.1 Locality

To distribute BDD exploration, the BDD needs to be split into partitions which are then assigned to multiple machines. Each machine processes a subtree of the BDD. Depending on the splitting mechanism, adjacent nodes might be located on different machines. As a result communication between machines is necessary in certain stages of the analysis. These inter-machine edges be-

tween adjacent nodes are referred to as *cutting edges*. In a distributed environment, the performance of BDD-based exploration can be improved significantly if adjacent BDD-nodes are located on the same machine such that the number of cutting edges is minimal. However, this graph partitioning problem is NP-complete [12]. There are several ways to make a good estimation of a reasonable graph distribution of explicit graphs. This operation remains challenging in symbolic reachability analysis, since it is difficult to make assumptions about the resulting graph (a BDD) obtained after applying an operation.

Obviously, locality is maximal if all nodes of the BDD belong to one machine. In this case, there are no cutting edges and no communication between machines is necessary. [13] However, this is not possible in the case that the graph exceeds a machine's memory limit. Furthermore, due to additional hardware resources in distributed systems, the communication overhead might be compensated by higher parallelization (load balancing).

1.1.2 Workload balancing

Together with the challenge of locality comes the issue of workload balancing. While certain problems need to be calculated sequentially, others are suited to be parallelized. For example, rendering 3D images is a task that is often performed independently for different parts of the resulting image. An example in terms of graph theory is the algorithm of Borůvka which allows high parallelization of calculating the minimal spanning tree for a given graph. An optimal workload balance means that tasks are distributed over all processing units (over both machines and cores within these machines) in a way such that during a computation all these units finish processing at the same or nearly at the same time without idle or waiting times during the computation. For some types of graphs there exist approaches to achieve a good workload balance while keeping the number of cutting edges low, but often a good workload balance correlates with a high number of cutting edges [14].

1.1.3 Communication Overhead

Also communication overhead is an important factor with respect to model checking in distributed environments. If only one machine is utilized or most nodes are located on one machine there is no or at least not much communication between machines. The more machines are used for the computation

and the more the nodes are spread over machines, the more likely it is that data must be shared between them. On the other hand, a good distribution over several machines might result in a better workload balance and therefore achieve higher parallelization of processing, which can lead to less processing time. There are graphs for which there are good distribution algorithms such that the workload is balanced and still communication is low [15]. However, in many cases it is difficult to achieve both a good workload balancing and minimal communication [13]. Still, there are also several algorithms that try to address communication overhead by sending data in bursts in contrast to continuous traffic [16] or optimize the network load in other ways [17].

1.1.4 Bandwidth/Network

Communication between machines and therefore the entire computation is also limited by the throughput/bandwidth and the latency of the used hardware of the network. Recent research indicates that the Infiniband [18] technology shows good results in this area. Infiniband supports remote direct memory access (RDMA), which decreases the latency. Therefore latency is not a bottleneck, but the throughput is still an issue. In our opinion Infiniband is at this point the fastest and also affordable network hardware available [10, 19]. In this study we will use this technology and will not focus on finding an even better solution.

1.2 Related Work

1.2.1 Distributed Explicit Graph Analysis

In 2015 Hong et al. developed a new system (called PGX.D) for distributed graph analysis which is able to reach speedup factors of up to 90 compared to other distributed (explicit) graph processing systems. It also shows faster results than single-machine executions. Hong et al. stated that a low overhead bandwidth-efficient communication mechanism with support for remote data pulling is important for fast graph processing. They use the Infiniband technology for a fast network communication. They also made use of selective ghost node creation to reduce traffic between machines. This is a technique where on each machine copies are created of specific nodes, for instance high-degree nodes. To achieve balanced workload, edge partitioning and edge chunking

were used, which are, according to the developers, essential for a balanced workload between cores. [19]

Research by Guo, Varbanescu, Epema and Iosup showed that the combination of GPU and CPU resources in distributed systems can have a positive impact on the performance [20].

An explicit model checker which achieves close to linear speedup is Murϕ. It uses distributed memory without synchronization between processes and a hash function to guarantee a balanced workload. [13]

Work by Inggs and Barringer on reachability analysis for CTL* model checking also shows nearly linear speedup. They use a shared-memory architecture with a work stealing mechanism to keep the workload balanced and minimize idle times of processors. [21]

1.2.2 Symbolic Model Checking

In 1997 Narayan, Jain, Fujita and Sangiovanni-Vincentelli invented partitioned-ROBDDs as a new way of constructing decision diagram representations for a given Boolean function. The Boolean space is divided into partitions and each partition is then represented by a ROBDD with its own variable ordering. This can result in a state space which is exponentially smaller than the generated state space using a single ROBDD. [22] According to [23] the core advantage of partitioned-ROBDDs is that for instance counterexamples in model checking can be located fast and that each partition can have its own variable ordering.

In 2000 Heyman, Geist, Grumberg and Schuster developed a partitioning algorithm with dynamic repartitioning for symbolic model checking with BDDs in a distributed-memory environment [8]. Their partitioning method (*'Boolean function slicing'*) is based on the work of Narayan et al. (see [22]). We will refer to it as vertical partitioning.

In 2003 Grumberg, Heyman and Schuster presented a work-efficient approach due to dynamic allocation of resources and a mechanism for recovery from local state space explosion. This means that no unnecessary hardware resources are used. [17]

Chung and Ciardo (2004) presented a distributed-memory saturation algorithm for *multi-valued decision diagrams* (MDDs) using horizontal partitioning [24]. Due to the sequential computation this method does not achieve speedups compared to a single machine version. However, this approach enables the computation of much larger graphs, because more memory is avail-

able. [6]

Two years later (2006) Chung and Ciardo developed their method further and achieved speedups of up to 17 percent compared to their previous version. This time they used vertical slicing combined with speculative firing. [7]

Not a distributed technology but another way to minimize the state space are *partial binary decision diagrams* (POBDDs) which were developed by Townsend and Thornton in 2002 [25]. As opposed to the method of Narayan et al. not a Boolean formula but an existing ROBDD representation is partitioned into multiple partial BDDs. As far as we know this approach has not yet been used in a distributed environment.

In 2015 Oortwijn, van Dijk and van de Pol presented their findings on shared-memory hash tables for symbolic reachability model checking. They stated that for the efficient use of shared hash tables it is essential to minimize the number of roundtrips which is limited by the throughput of the network (in this case Infiniband which supports Remote Direct Memory Access – RDMA). According to Oortwijn et al. linear probing is one way to achieve less roundtrips (compared to e.g. Cuckoo hashing). With DistBDD they implemented the first BDD package that combines both distributed and multi-core reachability algorithms. [10, 11]

1.3 Research Questions

In the graph analysis and especially the symbolic model checking domain there are still many challenges to solve to make optimal use of the benefits of distributed environments. Since in most of the cases of distributed reachability analysis the network is the bottleneck of the system, we focus in this thesis mostly on how to reduce communication overhead between machines and how to achieve this by combining existing multi-core and multi-machine approaches that have shown good performance in earlier research. We will base our work on the multi-core model checker Sylvan and the vertical partitioning method of [17].
This leads to the following main research question:

Research question: *How can a problem of BDD based symbolic model checking be divided between machines in a compute cluster of multi-core machines to reduce the total computation time?*

To better point out the different aspects, this question will be split into three sub-questions.

Subquestion 1: *How can principles of vertical partitioning be combined with existing multi-core model checking solutions?*

Subquestion 2: *How do different configurations regarding the partitioning policy affect the overall performance of the resulting system?*

Subquestion 3: *How does the proposed method scale with the size of the graph and the number of utilized machines?*

Chapter 2

Preliminaries on Symbolic Model Checking

2.1 Binary Decision Diagrams

Binary decision diagrams were developed by C. Y. Lee [27] and are based on the idea of Shannon expansion [28]. Any Boolean function f can be decomposed into two sub-functions in which each Boolean variable X_i is either true or false:

$$f(X_0,\ldots,X_i,\ldots,X_n) \equiv X_i \cdot f(X_0,\ldots,1,\ldots,X_n) + \overline{X_i} \cdot f(X_0,\ldots,0,\ldots,X_n) \qquad (2.1)$$

Since every variable in a Boolean formula can have two values, a binary decision tree can be built by recursively applying the Shannon expansion formula. Figure 2.1 is an example of such a tree. It represents formula $a \vee (b \wedge c)$. This tree can be transformed into a directed acyclic graph (DAG) by merging or deleting some nodes. These DAGs were invented by Bryant and are called Reduced Ordered Binary Decision Diagrams (ROBDDs). [29,30]. The following definition is taken from [4].

Definition 1 (Ordered Binary Decision Diagram)**.** *An (ordered) BDD is a directed acyclic graph with the following properties:*

1. *There is a single root node and two terminal nodes 0 and 1.*
2. *Each non-terminal node p has a variable label x_i and two outgoing edges, labeled 0 and 1; we write $lvl(p) = i$ and $p[v] = q$, where $v \in \{0,1\}$.*
3. *For each edge from node p to non-terminal node q, $lvl(p) < lvl(q)$.*

4. *There are no duplicate nodes, i.e.,*
   $\forall p \forall q \cdot (lvl(p) = lvl(q) \wedge p[0] = q[0] \wedge p[1] = q[1]) \rightarrow p = q.$

The 0 and 1 edges are also referred to as high- and low-edges [31] and in diagrams they are indicated with dashed and solid lines respectively. A BDD that has all the properties that are mentioned in Definition 1 is called *quasi-reduced* (QROBDD). It does not contain duplicate nodes, but it may contain redundant nodes. These are nodes for which the node's high and low edge lead to the same child node. A BDD which also does not contain any redundant nodes is called a *reduced* OBDD (ROBDD). [32]

In [4] (fully-)reduced and quasi-reduced BDDs are defined as follows:

Definition 2 (Fully-reduced/Quasi-reduced BDD)**.** *Fully-reduced BDDs forbid redundant nodes, i.e. nodes with* $p[0] = p[1]$*. Quasi-reduced BDDs keep all redundant nodes, i.e., skipping levels is forbidden.*

The graphs shown in Figures 2.1, 2.2, and 2.3 all describe the following Boolean formula:

$$a \vee (b \wedge c) \tag{2.2}$$

Figure 2.1 shows the decision tree belonging to the formula without any reductions or ordering. In Figure 2.2 all duplicate nodes of the decision tree in Figure 2.1 are removed (i.e. the two b-nodes on the bottom right) and its variables are ordered. In Figure 2.3 also the redundant nodes are removed.

In the following we refer to ROBDDs simply as BDDs.

An important part of model checking with BDDs is that there must be a unique table of nodes which is used to guarantee that there are no duplicate nodes in a BDD. This prevents the creation of superfluous nodes which may lead to overhead. In many existing implementations a hash table is used for this.

In the following section the most important operations on BDDs will be explained.

2.2 BDD Operations

The most basic operation is restrict. It is an essential part of Shannon decomposition which is used to create BDDs from Boolean functions. The result of applying the restrict operator on a BDD is another BDD.

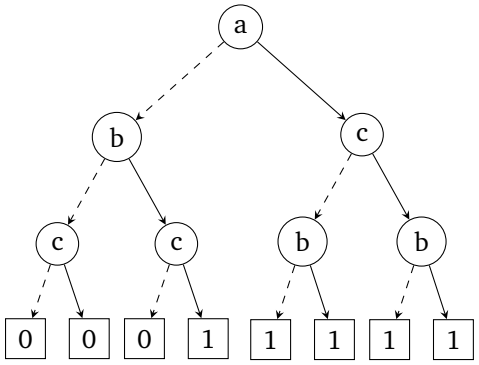


Figure 2.1: Binary Decision Tree

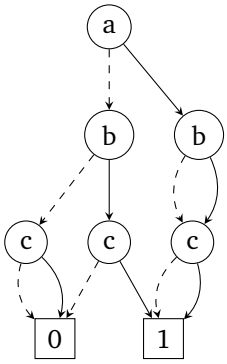


Figure 2.2: QROBDD

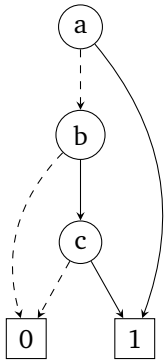


Figure 2.3: ROBDD

Definition 3 (Restriction (cofactor))**.** *Let $f(x_0,\ldots,x_n)$ be a BDD representing a Boolean function and $0 \leq i \leq n$. Then:*

$$\begin{aligned} restrict(f,x_i,1) &= f_{x_i=1}(x_0,\ldots,x_i,\ldots,x_n) \\ &= \quad f(x_0,\ldots,1,\ldots,x_n) \\ restrict(f,x_i,0) &= f_{x_i=0}(x_0,\ldots,x_i,\ldots,x_n) \\ &= \quad f(x_0,\ldots,0,\ldots,x_n) \end{aligned}$$

are the positive *and* negative restrictions *(or* cofactors*) of f with respect to x_i.*

Multiple BDDs can be combined in certain ways using Boolean operators like conjunction ($\wedge$), disjunction ($\vee$), implication ($\rightarrow$), exclusive or ($\oplus$) and more. Given two BDDs ϕ and ψ and a Boolean operator $\langle op \rangle$ a new BDD $\phi \langle op \rangle \psi$ can be constructed which is defined as follows:

$$\phi \langle op \rangle \psi = x(\phi_x \langle op \rangle \psi_x) + x'(\phi_{x'} \langle op \rangle \psi_{x'}) \tag{2.3}$$

The algorithm that computes $\phi \langle op \rangle \psi$ is based on Shannon decomposition (see Equation (2.1)) and is commonly called *apply*. [33]

The operations computed by apply can also be expressed in an if-then-else (ITE) structure. This ITE operation is defined as follows (definition from [4]).

Definition 4 (If-Then-Else (ITE))**.** *Let $ITE_{x=v}$ be shorthand for the result of $ITE(\phi_{x=v},\psi_{x=v},\chi_{x=v})$ and let x be the top variable of ϕ, ψ and χ. Then ITE is defined as follows:*

$$ITE(\phi,\psi,\chi) = \begin{cases} \psi & \phi = 1 \\ \chi & \phi = 0 \\ MK(x, ITE_{x=1}, ITE_{x=0}) & otherwise \end{cases}$$

Boolean operator	ITE
$f \wedge g$	$ITE(\phi, \psi, 0)$
$\neg f \wedge g$	$ITE(\phi, 0, \psi)$
$f \wedge \neg g$	$ITE(\phi, \neg\psi, 0)$
$f \vee g$	$ITE(\phi, 1, \psi)$
$\neg(f \wedge g)$	$ITE(\phi, \neg\psi, 1)$
$\neg(f \vee g)$	$ITE(\phi, 0, \neg\psi)$
$f \rightarrow g$	$ITE(\phi, \psi, 1)$
$f \leftarrow g$	$ITE(\phi, 1, \neg\psi)$
$f \leftrightarrow g$	$ITE(\phi, \psi, \neg\psi)$
$f \oplus g$	$ITE(\phi, \neg\psi, \psi)$

Table 2.1: Boolean operators and their ITE representation, where ϕ and ψ are the unique BDD representatives of f and g, respectively.

$MK(x, T, F)$ in Definition 4 is used to create a new BDD node with variable x. The outgoing high-edge of x goes to the BDD T and its low edge to the BDD F.

Table 2.1 shows Boolean operators and their ITE equivalents.

For a reachability analysis with BDDs there are four operations which are necessary to calculate the reachable states. These are $\wedge$, $\vee$, $\exists$ and substitution. The Boolean operators are covered by the apply operation explained above. Furthermore $\wedge$ and $\exists$ are commonly combined in a single algorithm. [4]

Listing 2.1 shows an outline of a reachability analysis, where *Initial* is an initial BDD, T is a BDD representing a transition relation from one state to another and X and X' are sets of states.

```
1 BDD Reachability(BDD Initial, BDD T, Set X, Set X')
2 BDD Reachable := Initial, Previous := 0
3 while Reachable != Previous
4     BDD Next := (∃X · (Reachable ∧ T))[X'/X]
5     Previous := Reachable
6     Reachable := Reachable ∨ Next
7 return Reachable
```

Listing 2.1: Reachability algorithm from [4]

In line 4 of the algorithm $\exists X \cdot (Reachable \wedge T))[X'/X]$ computes the set of reachable states from state `Reachable` using the transition relation T. After applying $\exists$ only the successors remain. This combination of $\exists$ and $\wedge$ is called the relational product (RelProd).

The formal specification of RelProd is given by the following definition [33].

Definition 5 (RelProd). *Let $X = \{x_1, \ldots, x_n\}$ and $X' = \{x'_1, \ldots, x'_n\}$ be two sets of variables, $f : \mathbb{B}^n \to \mathbb{B}$ a Boolean function and $g : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}$ a Boolean relation. Let ϕ and ψ be the respective BDD representations of the functions f and g. The relational product over ϕ and ψ with respect to X', denoted by $RelProd(\phi, \psi, X, X')$ is the BDD representing $\exists X \cdot (f(X) \wedge g(X, X'))$.*

An example algorithm of `RelProd` is shown in Listing 2.2 [4].

```
1  BDD RelProd( BDD A, BDD B, Set X)
2
3      // (1) Terminating cases
4      if A = 1 ∧ B = 1 then return 1
5      if A = 0 ∨ B = 0 then return 0
6
7      // (2) Check cache, hash table for constant time access on
8      // intermediate results of BDD operations
9      if inCache(A, B, X) then return result
10
11     // (3) Calculate top variable and cofactors
12     x := topVariable(x_A, x_B)
13     A_0 := cofactor0(A, x) B_0 := cofactor0(B, x)
14     A_1 := cofactor1(A, x) B_1 := cofactor1(B, x)
15
16     if x ∈ X
17
18         // (4) Calculate subproblems and result when x ∈ X
19         R_0 := RelProd(A_0, B_0, X)
20         if R_0 = 1 then result := 1 // Because 1 ∨ R1 = 1
21         else
22             R1 := RelProd(A_1, B_1, X)
23             result := ITE(R_0, 1, R_1) // Calculate R_0 ∨ R_1
24     else
25
26         // (5) Calculate subproblems and result when x ∉ X
27         R_0 := RelProd(A_0, B_0, X)
28         R_1 := RelProd(A_1, B_1, X)
29         result := MK(x, R_1, R_0)
30
31     // (6) Store result in cache
32     putInCache(A, B, X, result)
33
34     // (7) Return result
35     return result
```

Listing 2.2: `RelProd` algorithm

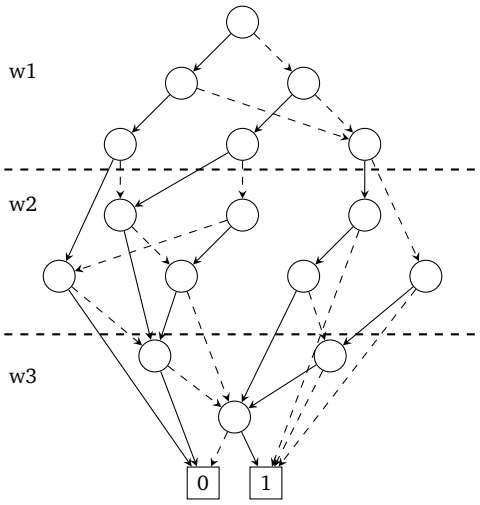
The last BDD operation we discuss is *substitution* which is used to rename one variable to another.

Definition 6 (Substitution)**.** *Let* $X = \{x_1, \ldots, x_2\}$ *be a set of variables and* $f : \mathbb{B}^n \to \mathbb{B}$ *be a Boolean function. Let* $x_1, y \in X$ *be two variables from X. Then the substitution of* x_i *by* y*, denoted by* $f[x_i \leftarrow y]$*, is defined as* $f[x_i \leftarrow y] \equiv f(x_1, \ldots, x_{i-1}, y, x_{i+1}, \ldots, x_n)$*. Let* $Y = \{y_1, \ldots, y_m\} \subseteq X$ *and* $Z = \{z_1, \ldots, z_m\} \subseteq X$ *be two subsets of X. Then* $\phi[Y \leftarrow Z] \equiv (((f[y_1 \leftarrow z_1])[y_2 \leftarrow z_2]) \ldots)[y_m \leftarrow z_m]$.

2.3 BDD Partitioning

2.3.1 Horizontally Partitioned BDDs

In horizontal partitioning [6, 9] the levels of a BDD are distributed over machines and each level is assigned to a single machine. This means that all nodes that belong to a certain level are located on the same machine. Figure 2.4 visualizes this partitioning strategy.



An advantage of vertical partitioning is that no duplicate nodes are created. This approach focuses on increasing the available space (by utilizing more machines) and not on decreasing computation time. Due to more resources larger models can be processed. Through an approach called “speculative firing” also faster computations (compared to linear analyses) could be achieved. However, vertical partitioning (see section 2.3.2) achieved better results with respect to computation time.

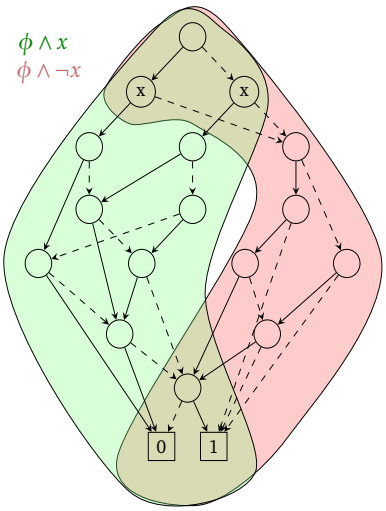


Figure 2.4: BDD ϕ with two partitions

2.3.2 Vertically Partitioned BDDs

In [22] partitioned-ROBDDs were introduced which can be exponentially smaller than other BDDs. With partitioned-BDDs not the entire Boolean space is represented as a whole, but it is divided in multiple partitions and each of them is represented by one ROBDD. The division is done using one or more "windowing functions" w. A windowing function can be a Boolean variable or a Boolean formula, which represents a part of the BDD's Boolean space. In Definition 7 which is taken from [22] partitioned-ROBDDs are described formally.

Definition 7 (Partitioned-ROBDDs)**.** *Given a Boolean function $f : B^n \rightarrow B$ defined over X_n, a partitioned-ROBDD representation x_f of f is a set of k function pairs, $x_f = \{(w_1, \tilde{f_1}), \ldots, (w_k, \tilde{f_k})\}$ where $w_i : B^n \rightarrow B$ and $\tilde{f_i} : B^n \rightarrow B$ for $1 \leq i \leq k$, are also defined over X_n and satisfy the following conditions:*

1. *w_i and $\tilde{f_i}$ are represented as ROBDDs with the variable ordering π_i, for $1 \leq i \leq k$.*
2. *$w_1 \vee w_2 \vee \cdots \vee w_k \equiv True$*
3. *$\tilde{f_i} \equiv w_i \wedge f$, for $1 \leq i \leq k$*

According to Definition 7 partitions in partitioned-ROBDDs do not need to be disjoint. This means that duplicate nodes can be created during partitioning.

An advantage of partitioned-ROBDDs, which can make the representation even smaller, is that each partition can have its own variable ordering (see condition 1 of Definition 7).

Research in [8,17,34] also uses partitioned-ROBDDs. Based on Definition 7 they developed algorithms to find suitable windowing functions for partitioning a BDD. Figure 2.4 visualizes a BDD after partitioning with windowing function $w = x$. We base this research on the findings in [34].

Chapter 3

Method

In the following we will describe how we try to answer the main research question and the three subquestions stated in section 1.3.

To answer *research question 1* we use the work on partitioned-ROBDDs done in [34] as a starting point and try to implement their partitioning algorithm, which divides Boolean functions into multiple BDDs. These newly created BDDs can be processed locally by an existing model checker. We will use Sylvan for this purpose since this tool has shown significant speedup on multi-core machines compared to other tools. In this way we hope to achieve also good speedups in a distributed environment. The goal is to extend Sylvan's functionality in order to obtain communication between multiple machines within reachability analysis. This leads to several additional challenges that have to be solved like synchronization of local analyses and avoidance of unnecessary communication overhead. For communication between machines/processes we will introduce Open MPI [35], an open source implementation of the Message Passing Interface (MPI) standard [36].

Research question 2 aims at the influence of both the moment when a BDD is partitioned during reachability and the number of partitions that are created when a BDD needs to be divided. By *the moment of partitioning* we mean that we will set a certain maximum BDD partition size (number of nodes). When exceeding this threshold the BDD will be split into more partitions. We will evaluate the influence of both configuration parameters by repeatedly analyzing several models of different sizes. By a model's size we mean the approximate size it reaches during an analysis and not its initial size. We will use well-known BEEM [37] and Petri-net [38] models for this purpose. Between runs, we vary the parameter values while keeping the number of machines constant. Here, we focus on finding suitable configurations for a given model. Therefore, we

are interested in differences in performance (execution time) between multiple analyses of the same model with varying initial settings of the parameters mentioned above.

In order to find an answer to *research question 3* and be able to make assumptions over the scalability of the system, the proposed partitioning and communication method, we will evaluate the system's performance depending on the maximum size of the input model and the utilized hardware resources. In contrast to research question 2, we want to gain insights into the performance of our implementation when the initial configurations are kept the same, but the size of the input model grows. Furthermore, we are interested in how efficiently additional hardware resources can be used.

3.1 Validation

We will validate the implementation by executing multiple reachability analyses on several models. We do this with varying parameters/settings of our implementation (like the moment when a BDD is partitioned and the number of partitions that are created during partitioning). We will compare the results with those of other implementations (DistDD). Hereby we use the number of reachable states as an indication of correct calculations.

3.2 Performance Measurement

When it comes to the general evaluation of the invented method we want to be able to draw conclusions regarding its contribution with respect to the overall computation time.

To set our approach into relation with existing ones we will record and measure the following factors:

- total computation time
- size of the generated state space
- number of machines that are not idle at the end of the computation (see chapter 4)
- used settings for the analysis (moment that a BDD is partitioned and number of partitions that are created during partitioning)

By setting the computation time in relation to the other values, we will try to find bottlenecks of the system.

We will compare the execution times to those obtained with (1) the distributed system DistBDD, (2) the not distributed but parallel system Sylvan and (3) the linear execution in order to be able to draw conclusions about the competitiveness of the system.

Chapter 4

Designing and Implementing Distribution and Communication Algorithm

4.1 Algorithm Overview

During reachability analysis of large models the BDD of that model might become too large to process on a single machine. In this case, the process can split its BDD and send one or more of its partitions to processes on other machines, if available. In this section we will give a short overview of the reachability algorithm we designed and implemented in this work. In the following sections, we will go into more detail.

One essential part of our design is that each analysis using our algorithm consists of at least two processes of which one is called the *master process* and all other are *worker processes*. The master process handles part of the communication between processes and provides information about the progress of the analysis. The worker processes, on the other hand, are responsible for the actual reachability analysis.

When an analysis is started, all processes (one master and one or multiple workers) are immediately initialized. They all get the transition relation to perform the reachability analysis, but only one worker process gets the initial BDD. All other processes get an empty BDD (=false) and are initially idle. Furthermore, every worker has a list of *split variables* or *split functions*, which are necessary when more than one worker process is involved in the computation. Split variables and functions are essentially the same as ”windowing functions”

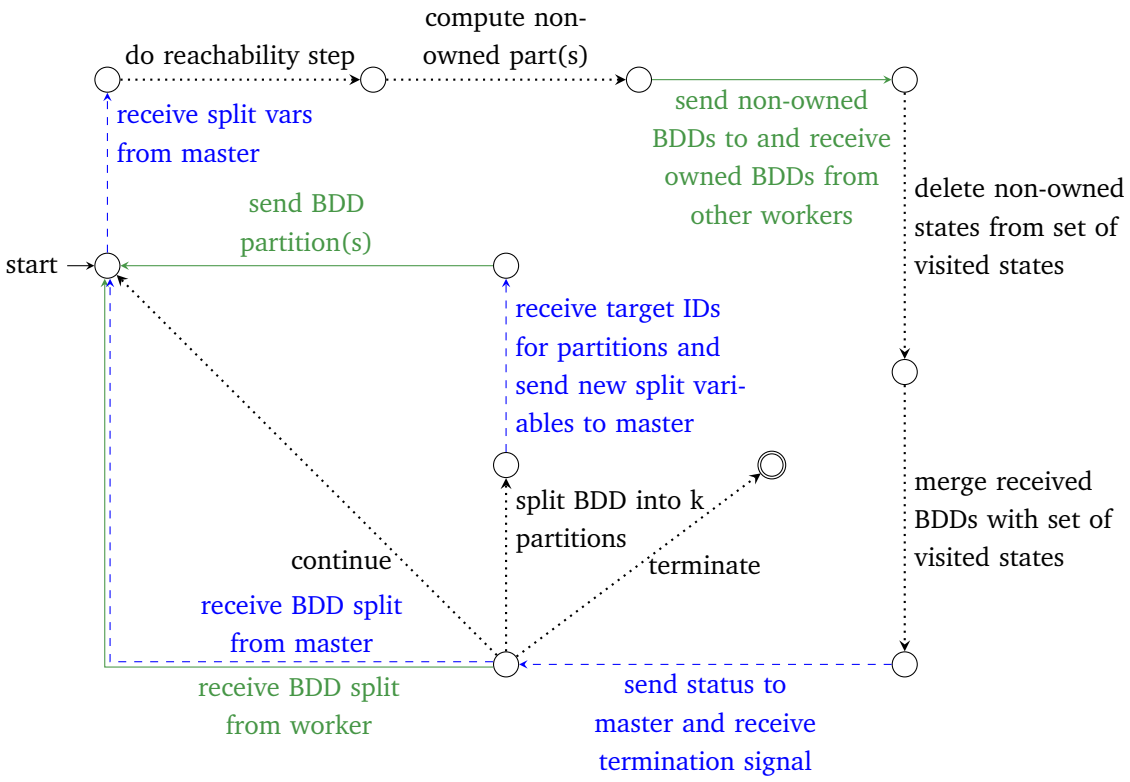


Figure 4.1: Flow chart of the process from the workers' view. Dotted (black) arrows: local processing steps, dashed (blue) arrows: communication between a worker and the master, solid (green) arrows: communication between workers.

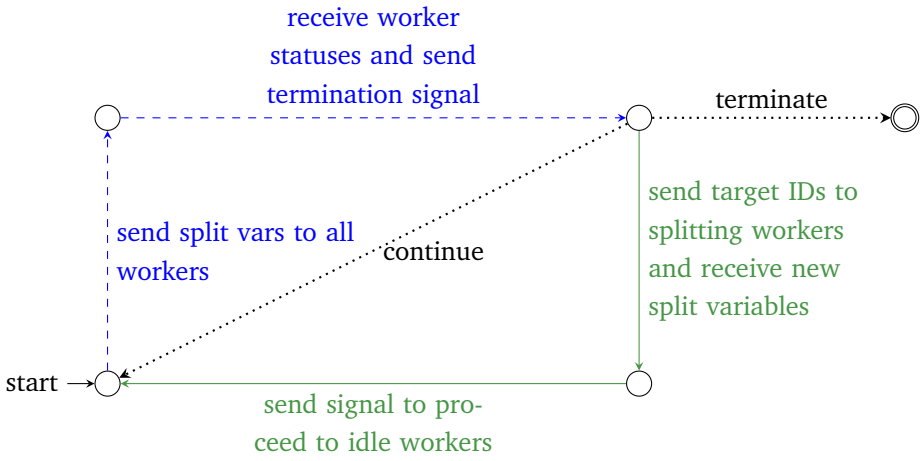


Figure 4.2: Flow chart of the process from the master's view. Dotted (black) arrows: local processing steps, dashed (blue) arrows: communication between master and all workers, solid (green) arrows: communication between master and one/some workers.

from section 2.3.

Figures 4.1 and 4.2 show an overview of all the steps that each process, worker and master, perform multiple times during the reachability analysis. When the processes begin the analysis (see start node in the figures), the initial setup is already done (initializing processes, reading transition relations and initial BDD from file). The following steps are performed in each iteration:

1. Updating the list of split variables. The master process keeps track of all changes and notifies the worker processes (see section 4.5 for details).

2. Each worker process performs one level of the reachability analysis (computes the set of next states using the transition relation). Note that also the idle states (their local BDD is still empty, or *false*) do this step. Their set of next states will be empty.

3. Worker processes exchange non-owned states (see section 4.4). States that do not belong to the BDD partition of a process will be removed and states that have been received from other processes will be merged with the local BDD partition.

4. All worker processes merge their set of next states (including the received states from other processes) with all reachable states that have been explored locally.

5. Each worker process determines a status (“idle”, “out of work”, “in progress”, “needs to split”) and sends it to the master process. The master process receives these messages and determines how to proceed. If all workers send the message “out of work”, the master will notify the workers to terminate. If at least one of the workers sent ”needs to split”, the master will handle the splitting procedure in the next step.

6. If no signal to terminate has been received, worker processes may receive or send BDD partitions. This depends on the processes’ status:

   - “in progress” or “out of work”: The worker process will do nothing and continue with the next step.
   - “idle”: The worker process waits for a BDD from another worker process or the master process.
   - “needs to split”: The worker process determines a split variable to split its BDD into one or multiple partitions. Then it receives one or

multiple target processes from the master and sends the partitions to these targets. Finally it notifies the master about the used split variables.

7. The master sends empty BDDs to every process that is still idle.

Listing 4.1 gives the pseudocode to the steps described above, that runs on each worker process. Note that it is similar to the reachability algorithm in Listing 2.1 with some additional steps. On lines 7 and 16 the actual reachability analysis happens. First, the set of reachable states (*Next*) from the current set of already explored states (*Reachable*) is calculated using the relational product of the current state space and the transition relation T. Then, the states in *Next* are added to *Reachable*. While the algorithm in Listing 2.1 tests whether *Reachable* changes with respect to the previous iteration to terminate the while loop, our algorithm terminates, when all processes are finished. More specifically, they terminate, when the master process sends a termination signal.
On lines 9 to 15 the exchange of *non-owned states* happens. Each process sends states that it is not supposed to keep to other processes and may receive states from other processes. We explain non-owned states and their exchange in more detail in section 4.4.
The following lines of code (18 to 22) determine, if a process can receive or must send a BDD partition, does not need to do anything or has to terminate (in this case the reachability analysis is finished).
Receiving and sending of BDD partitions happens on lines 24 to 30, if any process has to split its BDD or can receive one.
Finally, each process receives updates about used split variables from the master.

Primitive	**Description**
BDD	datatype from Sylvan representing BDDs
BDDVAR	datatype from Sylvan representing a single variable in a BDD
(target)ID, (target)rank	integer representing an ID of an MPI process
MPI_ANY_SOURCE	MPI constant that can be used when no specific rank is necessary
status	integer representing the current status of a process
WORKER_STATUS_SPLIT	integer representing the status *split* of a process
WORKER_STATUS_IDLE	integer representing the status *idle* of a process
WORKER_STATUS_PROGRESS	integer representing the status *in progress* of a process

Figure 4.3: Primitives commonly used in the following pseudo codes of this chapter

In the next sections, we will describe each step of the algorithm in more detail.

```
1  // Initial BDD, transition relation T, variable sets X, X'
2  BDD Reachability(BDD Initial, BDD T, Set X, Set X')
3      BDD Reachable := Initial
4      BDD Split_vars[number_of_processes]
5      int q  // id of this process
6      while not all processes finished
7          BDD Next := (∃X · (Reachable ∧ T))[X'/X]
8
9          for each process p, p ≠ q
10             BDD next_split := Next ∧ split_vars[p]
11             send_to(p, next_split)
12         for each process p, p ≠ q
13             Next := Next ∨ receive_from(p)
14
15         Next := Next ∧ Split_vars[q]
16         Reachable := Reachable ∨ Next
17
18         Status := determine_status()
19         send_to_master(Status)
20         if receive_termination_signal()
21             break
22
23         if Status == SPLIT
24             int n := 0
25             while n < k:       // k = number of partitions
26                 Reachable := handle_split(Reachable)
27                 n := n + 1
28         if Status == IDLE
29             Reachable := receive_from()
30
31         receive_and_update(Split_vars)
32     return Reachable
```

Listing 4.1: Customized reachability algorithm

4.2 Splitting and Sending a BDD

The essence of our approach is that large BDDs are partitioned into multiple BDDs which will be processed by different workers. In our design we assume the workers to all have the same amount of memory available. Small adaptations of the algorithm might be sufficient to make it suitable for heterogeneous system memory. Next to designing the splitting procedure itself, it is important to make decisions about when to split a BDD and how many partitions to create. We will first explain what the latter two factors indicate and will then proceed

with the algorithm itself. Note, that we will provide details about BDD partitions and their creation in section 4.3. In this section we will keep this concept on a higher level. At this point it is sufficient to keep in mind, that each BDD can be split into multiple partitions. These partitions can "overlap" (are usually not disjoint) and their conjunction gives the initial BDD.

By *when to split* we mean: "How large is the state space of one process allowed to be?". As a measure of the state space we use the number of nodes of the BDD. We can either choose a large number of nodes as limit, such that a BDD does just fit in the working memory of a machine. Or we can set the limit lower and split the BDD when it is smaller. The advantage of a high value is that the reachability analysis is kept on as few machines as possible. The fewer machines involved in the computation, the less time-consuming communication between them is necessary. Only when the working memory of the utilized machines does not suffice, we split and use more machines. However, splitting and sending large BDDs might consume more time compared with splitting and sending smaller BDDs. If we set the limit lower, the BDD will be split earlier. The advantage of this approach is that more machines may get involved in the computation (e.g. a BDD will be split 3 times instead of once), which can process the partitions in parallel. When a good partitioning can be found (see section 4.3) it is possible that splitting the BDD early decreases the total number of nodes (removal of redundant states). This could have an impact on the computation time. Also, splitting and sending partitions might happen faster. The disadvantage is that more communication between machines will be necessary (when exchanging non-owned states) and more duplicate nodes might be created compared with an execution with a higher maximum number of nodes.

How many partitions means: "Into how many parts should a process split its BDD when it exceeds the maximum number of allowed nodes?". If a BDD exceeds the maximum number of allowed nodes, we can split it into at least two parts, but it is also possible to create more partitions. When we follow the approach to utilize as few machines as possible, we can set the number of partitions (*split count*) low (2), possibly combined with a high maximum number of nodes. In this way, when a BDD needs to be split, one partition will be kept by the process and one additional process will get the other partition. On the other hand, we can set the number of partitions to a higher value, which means that more machines will be used for the computation. The advantages and disadvantages are similar to the ones of a low and a high maximum number

of nodes. When the number of partitions is low, less machines are needed, which leads to less communication between machines. But a low split count also means that the created partitions are likely to consist each of more nodes than a partition in the case that the split count is high, because all the states of the initial BDD will be distributed over less partitions. At the same time, more partitions might lead to more duplicate nodes. We explain duplicate nodes in section 4.3. One advantage of more splits is that it might take more time until the next process needs to split, because the partitions are initially smaller. However, at the same time, creating more partitions might take longer and the communication between processes might increases.

In our implementation we use fixed numbers for the maximum number of nodes and the number of partitions per run of the algorithm. So for every partition and every split that a partition performs during the entire analysis, the same numbers are used.

Listings 4.2 and 4.3 give the pseudocode that belongs to the splitting procedure, the first from a worker's point of view, the second from the master's. As seen in Listing 2.1 on lines 26 to 28, each worker that has to split its BDD will call `handle_split()` $k-1$ times, where k is the number of partitions that we want the worker to create.

In each iteration, the worker process will first search a suitable split variable (see section 4.3). In the next step, two partitions will be created from the BDD that contains all explored states so far. The split variable is used to generate these partitions. After this, on line 11 of Listing 4.2 and line 14 of Listing 4.3 the worker will receive a target process from the master. For this, the master iterates through all workers and chooses the first "idle" process. The worker can now send the smaller one of the created partitions to the target process which is already waiting to receive a BDD (see lines 29 and 30 in Listing 2.1). It is important that the worker keeps the larger partition, because we might want to split this one again, if the process has not yet split $k-1$ times. In this way we try to achieve an even distribution of states over all created partitions. The master's next step is to set the current status of the target process to "in progress". This is done to avoid sending multiple partitions to that process.

After a partition has been sent to another process and the target process' status has been updated, the splitting worker sends the used split variables, including its own and the used target ID to the master. The master will store all process IDs and their belonging split variables and will use them when he notifies all workers about changes.

Finally, `handle_split` returns the partition that has not been sent and the worker process will set its BDD of reachable states to this partition.

```
1  BDD handle_split(BDD reachable)
2
3      // determine the best variable to split over
4      BDDVAR splitvar := select_splitvar(reachable)
5
6      // perform the actual split and return both partitions
7      // where left is the larger part, if not equal
8      BDD right, left = decompose(reachable, splitvar)
9
10     // receive a "target" process from master
11     int target_rank := receive_target_rank()
12
13     // sending smaller part of the split to "target_rank"
14     bdd_bsendto(right, target_rank)
15
16     // send the split variables and process IDs to the master process
17     send_new_split_vars(target_rank, splitvar, worker_id, neg(splitvar))
18
19     // Return the other part of the splitted BDD
20     return left
```

Listing 4.2: Pseudocode for `handle_split(BDD next)`

```
1  void handle_communication_BDD_split()
2
3      for (int j = 1; j < split_count; j++)
4
5          // loop through worker processes
6          for each process p
7
8              // if p has sent worker status ``has to split''
9              if current_status[p] == WORKER_STATUS_SPLIT
10                 // find a worker with status ``idle'', abort otherwise
11                 int target_id := find_first_idle_process_or_abort();
12
13                 // send target id to process p
14                 send_target_id(p, target_id)
15
16                 // update status of target process from ``idle'' to ``in progress''
17                 current_status[target_id] := WORKER_STATUS_PROGRESS
18
19                 // receive new split variables from p
20                 receive_splitvars(p)
```

Listing 4.3: Pseudocode for `handle_communication_BDD_split()` from the master's perspective

4.3 Finding the Split Variable

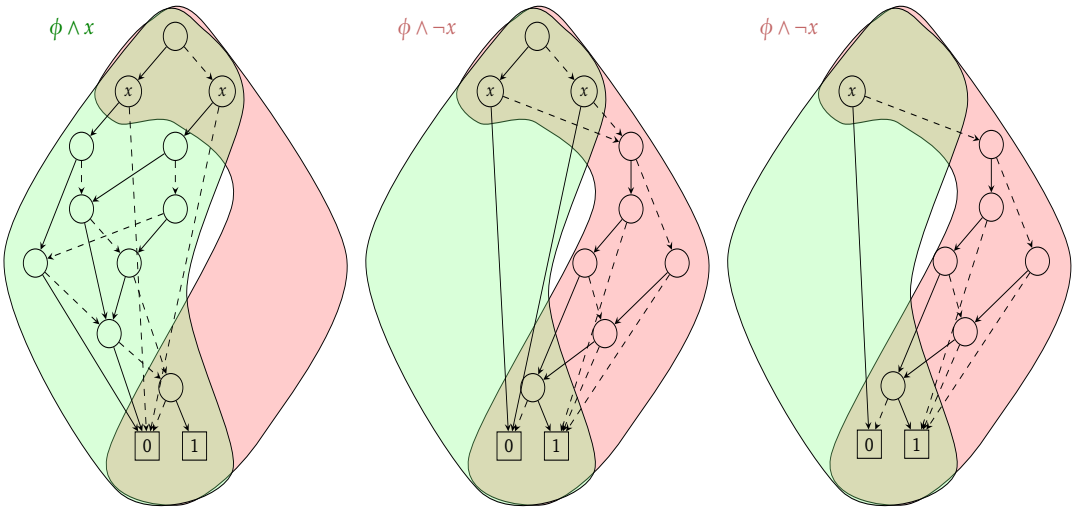
Whenever a worker needs to split its BDD (because it became too large), one or multiple split variables must be determined to divide the BDD into partitions. In order to generate k partitions, $k-1$ split variables are necessary. When k partitions are created, the splitting worker keeps one partition and sends $k-1$ partitions to other workers. However, more than two BDD partitions are never created in a single step, but to create multiple partitions, the BDD ϕ (and its partitions) will be divided in two parts multiple times. For instance, when three partitions must be generated, first the initial BDD will be split along variable x. For this split variable x, one partition contains all nodes that are reachable via x and the other partition contains all nodes that are reachable via $\neg x$. In the next step, one of these partitions (e.g. the $\neg x$-part) will again be split into two parts along variable y. This means that we now have three partitions: $\phi_1 = \phi \wedge x$, $\phi_2 = \phi \wedge \neg x \wedge y$ and $\phi_3 = \phi \wedge \neg x \wedge \neg y$, where each has its own *split function*. Figure 4.4 shows how two BDD partitions of the BDD given in figure 2.4 might look like.

Figure 4.4a shows the partition $\phi \wedge x$, which contains all BDD nodes of the initial BDD ϕ that are reachable via high (positive) edges of split variable x. Low edges of x in the initial BDD now lead to 0. Figures 4.4b and 4.4c show the counterpart of Figure 4.4a. Here, all high edges of variable x lead to 0 and the BDDs contain all nodes that were reachable via low edges of x in the initial BDD ϕ. The BDD of Figure 4.4b is still unreduced, while in Figure 4.4c all redundant nodes are removed. In the unreduced version, both nodes on level x have high and low edges leading to the same nodes, so they can be merged. This leads to the fact that the top node becomes redundant, because both its edges lead to the same node. This unreduced version shown in Figure 4.4b is actually never created and is included here for illustration purposes only. When BDD operations are performed on BDDs with Sylvan, the resulting BDD is kept reduced at all time and duplicate nodes are never created.

Given Figure 4.4 (especially Figures 4.4a and 4.4c) we can observe that there are many possible ways to split the initial BDD, leading to different results with respect to (1) node distribution, (2) duplicate nodes and (3) node reduction:

(1) The left partition contains 11 nodes, while the right one contains 7 nodes.

(2) There is one duplicate node at the top of the BDDs (left x node) and one



(a) Left partition ($\phi \wedge x$) *(b)* Right partition ($\phi \wedge \neg x$) *(c)* Reduced right partition

Figure 4.4: Partitioned BDD ϕ with split variable x

at the bottom.

(3) The right partition could be reduced by two nodes.

Finding a suitable split variable is essential for the overall performance of the reachability analysis using vertical partitioning. An optimal split variable splits the BDD into two partitions where

1. one partition has approximately $\frac{|\phi|}{k}$ and the other has approximately $|\phi| - \frac{|\phi|}{k}$ nodes, where $|\phi|$ is the number of nodes in the initial BDD and k is the number of desired partitions,
2. the number of duplicate nodes (redundancy) is as low as possible and
3. the partitions can be reduced, such that the total number of nodes decreases.

In [8] an algorithm has been developed to find a good split variable with respect to the above criteria (see Listing 4.4). This algorithm uses the following function to determine the cost of a given partitioning of BDD ϕ along variable v.

$$cost(\phi, v, \alpha) = \alpha * \frac{MAX(|\phi_v|, |\phi_{\neg v}|)}{|\phi|} + (1 - \alpha) * \frac{|\phi_v| + |\phi_{\neg v}|}{|\phi|} \tag{4.1}$$

The first part, $\frac{MAX(|\phi_v|,|\phi_{\neg v}|)}{|\phi|}$, gives a measure of the reduction achieved by the partition (criterium 3), while the second part, $\frac{|\phi_v|+|\phi_{\neg v}|}{|\phi|}$, gives a measure of the

number of shared BDD nodes (redundancy) in ϕ_v and $\phi_{\neg v}$ (criterium 2). The weight of both parts of the function depends on the value of α, which has to be between 0 and 1. A low α results in a low weight of the first (reduction) part and a high weight of the second (redundancy) part. As α increases, the weight of the first part increases and the weight of the second part decreases. The value of α is reset after each search for a split variable.

Equation (4.1) only effects criterium 2 and 3, the redundancy and the reduction of the partitioning. The partitioning algorithm (Listing 4.4), however, also takes criterium 1, the node distribution into account. First, the value of α (which is needed in the cost function) and a *step variable* $\Delta\alpha$ will be set to $min(0.1, \frac{1}{k})$, where k is the number of partitions we want to achieve. Then, on line 2, the best split variable $best\_var$ of all variables $v \in \phi$ with the given BDD ϕ and α is determined. Because α is initially low, the cost function will give most weight to the number of duplicate/shared nodes of the partitions.

At this point it may be that $|\phi_1| \approx |\phi|$ and $|\phi_2| \approx 0$. Therefore, the split variable will be improved – if possible – in the following while loop (rows 3 to 5) to achieve a more balanced split. Because we want the two partitions to be of size $\frac{|\phi|}{k}$ and $|\phi| - \frac{|\phi|}{k}$ (see criterium 1), a threshold variable δ is set to $\frac{|\phi|}{k}$ and will be used in the while condition. According to [8], when the redundancy of the split is small and one of the partitions is of size δ, then the other partition is very likely of size $|\phi| - \delta$. From this follows that we want to find a split variable for which the larger partition of the split is smaller or equal to $|\phi| - \delta$. The first part of the while condition, $max(|\phi \wedge best\_var|, |\phi \wedge \neg best\_var|) > |\phi| - \delta$, checks if this property is fulfilled. When no such split can be found, α will be increased by $\Delta\alpha$ which equals $min(0.1, \frac{0.1}{k})$ and therefore the reduction part's weight of the cost function increases. This will be repeated until either a suitable split variable is found or alpha becomes 1, which means that the weight of the reduction factor is maximal.

Note that although processing the while loop may take up to $max(k, 9)$ iterations with each $|V|$ processing steps to calculate the minimum cost for each variable, where $|V|$ is the number of variables in ϕ, the total processing time of *select_var()* does not increase significantly due to the while loop, because $|\phi \wedge best\_var|$ and $|\phi \wedge \neg best\_var|$ are only computed once.

```
1 α , Δα := min(0.1,1/k)
2 δ := |φ|/k
3 best_var = the variable v with minimal cost(φ,v,α)
4 while ((max(|φ ∧ best_var|,|φ ∧ ¬best_var|) > |φ| − δ) ∧ (α ≤ 1))
5     α := α + Δα
6     best_var := the variable v with minimal cost(φ,v,α)
7 return best_var
```

Listing 4.4: Pseudo code for `select_splitvar(`ϕ`)` which searches for a split variable `best_var` that can be used to divide the given BDD into two partitions with desired sizes, minimal redundancy, and maximal reduction

4.4 Exchanging Non-Owned States

During reachability analysis it is very likely that a process will encounter *non-owned states*. These are states that belong to one or more other partitions. To define these states, we introduce *split functions* which are essentially a single or a conjunction of multiple split variables. Every partition ϕ' of an initial BDD ϕ has its own unique split function X such that the conjunction of ϕ and X gives the partition ϕ'. On the other hand, the union of all partitions except ϕ' equals the conjunction of ϕ and the negation of X. A more formal definition of split functions is given in Definition 8.

Definition 8 (Split function)**.** *A split function $X : \mathbb{B}^n \to \mathbb{B}$ of BDD (sub-)partition ϕ_i of an initial BDD ϕ is a Boolean function over the set of variables $\{x_1, \ldots, x_n\}$ of BDD ϕ, for which*

$$\phi_i \;=\; \phi \wedge X$$

$$\textit{and} \quad \bigvee_{\substack{1 \le j \le n, \\ j \neq i}} \phi_j \;=\; \phi \wedge \neg X$$

Additionally, for every BDD ϕ with split function X the split functions of its partitions $\phi_1 = \phi \wedge x$ and $\phi_2 = \phi \wedge \neg x$ with split variable x are $X'_1 = X \wedge x$ and $X'_2 = X \wedge \neg x$ respectively.

Now that we have defined split functions, we will describe non-owned states more precisely and explain why it is important that other partitions know about these states. Figure 4.5 shows an example of how the BDD from Figure 4.4c (in the following referred to as ϕ') might look like after one reachability step. Initially, ϕ' equals $\phi \wedge X$, where ϕ is the BDD from 4.4 and $X = \neg x$ is a split function to create ϕ' from ϕ. At this point, $\phi' \wedge \neg X = \mathit{false}$.
After one reachability step (the resulting BDD will be referred to as ϕ'_{R1}) using the transition relation ψ, for some of the successors of ϕ' the negation of the

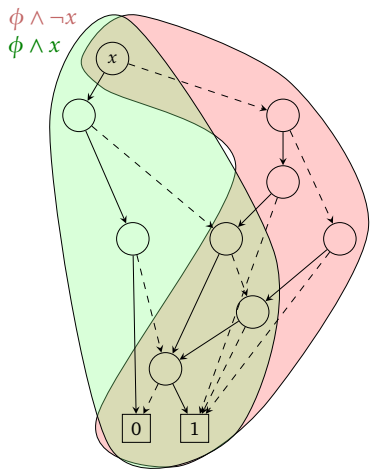


Figure 4.5: Resulting BDD ϕ'_{R1} after one reachability step on the partition $\phi' = \phi \wedge \neg x$

split function, $\neg X$, is true (see the green, left part of Figure 4.5). These successor states are called *non-owned states* of ϕ'_{R1}, while states for which X is true are called *owned states*.

Definition 9 (Non-owned states)**.** *Let ϕ' be a BDD partition $\phi \wedge X$ of BDD ϕ with split function $X : \mathbb{B}^n \to \mathbb{B}$, then the non-owned states of ϕ' are all states of ϕ' for which $\neg X$ is true. Therefore, $\phi' \wedge \neg X$ gives the BDD that contains all non-owned states of ϕ'.*

With the owned states of ϕ'_{R1} nothing needs to happen. However, the non-owned states need to be sent to other partitions of the initial BDD, because it might be that these states could only be explored via the states in partition ϕ'. On the other hand, partition ϕ'_{R1} might receive states for which X is true from other partitions. After all non-owned states have been exchanged, every partition will only keep their owned states and merge its received states with its own state space. For ϕ'_{R1} this means that $\phi'_{R1} := (\phi'_{R1} \wedge X) \vee R$, where R is the BDD of received states. After this, $\phi'_{R1} \wedge \neg X = \mathit{false}$ and $\phi'_{R1} \wedge X = \phi'_{R1}$. How the exchanging procedure works will be explained in the following. Note, that the set of non-owned states only needs to be computed for the newly discovered states during this iteration of reachability analysis (variable $Next$ in Listing 2.1) and not for the entire state space of ϕ'_{R1}. Initially the BDD of all reachable states ($Reachable$) does only contain owned states. After this, only owned states will be added to this BDD, because non-owned states are removed before merging $Next$ with $Reachable$.

We implemented the calculation and exchange of non-owned states as shown in Listing 4.5.

```
1  exchange_non_owned(BDD next) {
2      BDD result = false;
3
4      // send non-owned states
5      for each process p, p ≠ q
6          BDD other;
7
8          // determine parts owned by worker p by using the split
9          // function of p
10         if (split_functions[p] == true)
11             other = false;
12         else
13             other = next ∧ split_functions[p]
14
15         // send non-owned parts to process p without waiting
16         // for p to receive the message
17         bdd_ibsendto(other, p)
18
19     // receive owned states from other workers
20     for each process p, p ≠ q
21         BDD tmp = bdd_brecvfrom(ANY_SOURCE);
22         result = result ∨ tmp;
23
24     // wait until all non-owned states have been
25     // received by other processes
26     for each process p, p ≠ q
27         wait_until_recv(p);
28
29     return result;
```

Listing 4.5: Pseudocode for `exchange_nonowned(BDD next)`

4.5 Updating the List of Split Variables

Every process that is involved in the reachability analysis has a local list of split variables or split functions, stored as BDDs. For each entry in this list, also the ID of the process is stored which a specific split function belongs to.

Every time when new BDD partitions have been created, this list of split variables has to be updated. In order to minimize the communication that is needed to update all lists, the processes are only notified about the variables that have changed, or more specific, have been added to a split function. This happens according to a specific protocol.

In Listing 4.2 on line 17 a splitting worker sends the split variable it used

to generate the partitions and also its own and the target process ID to the master. The master receives a list of four entries (target ID, split variable target ID, worker ID, split variable worker ID – in this order) for every split that a process does. It stores all lists that it receives during one reachability step in a larger list. In the end of each reachability step the master sends this list to each process (see line 33 of Listing 4.2). Figure 4.6 visualizes the design of the list that the master sends. The gray boxes on top of the list indicate which entries belong to one split and which process sent the information. These are not stored by the master but are only shown for clarification purposes.

process 1								process 2				
split 1				split 2				split 3				
t_id	t_sv	s_id	s_sv	t_id	t_sv	s_id	s_sv	t_id	t_sv	s_id	s_sv	...

Figure 4.6: List of new split variables

The length of the list that the master sends is always a multiple of four. The processes receive this list and add the new split variables to their local list of split variables (L) as follows:

For each set of four elements

- Read the target process ID (t_id)
- Read the target split variable (t_sv) and transform it into a BDD (target BDD).
- Merge the target BDD with the current entry for the target ID in L and store the result as the new split BDD of the target process (L[t_id] = L[t_id] $\wedge$ target BDD).
- Read the source process ID (s_id).
- Merge L[t_id] with the current entry for the source ID in L and store the result as the new split BDD of the target process (L[t_id] = L[t_id] $\wedge$ L[s_id]).
- Read the source split variable (s_sv) and transform it into a BDD (source BDD).

- Merge the source BDD with the current entry for the source ID in L and store the result as the new split BDD of the source process (L[s_id] = L[s_id] $\wedge$ source BDD).

Due to the fact that the process with the target ID has received a partition from the process with the source ID, its received partition will only contain states that have been in the BDD of the target process before the split. These are states for which L[s_id] is true. This is why the split function of the target process (before the split) has to be merged with the split variable (or split BDD) that is used to create the target's partition.

4.6 Determining the Status and Termination

In each iteration the master process has to determine if all processes are finished with their calculation, if some are still running or if a BDD needs to be partitioned. To be able to do this, every worker sends his status to the master once in each iteration. We chose to use four statuses for this purpose:

1. idle
2. out of work
3. in progress
4. needs to split

The first two statuses, idle and out of work, essentially mean the same: A process with this status did not explore new states during the last iteration of reachability analysis. Neither did it extend its state space itself, nor has it received states from other processes during the exchange of non-owned states. The difference is that an idle process has *never* received a BDD (its BDD is still empty or *false*), while a process that is out of work has a BDD, but did not discover more states. An idle process can only receive states/a BDD when another process has sent a partition of its BDD. A process that is out of work, on the other hand, might receive states in a following iteration, when other processes send their non-owned states. We distinguish between these two, because in our implementation every process can receive only one BDD during the entire computation. We chose for this implementation, because we expect a processes' working memory is likely to be not large enough for receiving more than one

BDD partition, given that the split happens only when a BDD does not fit in a single machine's working memory any more.

The third status, in progress, means that a process has explored new states and its state space is still smaller then the maximum number of allowed nodes.

When a worker has the fourth status, needs to split, he has discovered new states and the size of its BDD exceeds the maximum number of allowed nodes. This status signals the master that it has to perform additional actions in the next step such as sending target process IDs for BDD partitions to the worker.

After all processes have sent their statuses, the master determines an overall status and sends a signal to continue or terminate back to the processes. There are three outcomes:

- At least one process sent needs to split: Notify workers to continue. Handle splitting procedure in the next step. Abort if no idle process is left. Otherwise, if

- at least one process sent in progress: Notify workers to continue. Analysis is not done, yet. Otherwise,

- all processes sent either idle or out of work: Notify workers to terminate. The analysis is done.

4.7 Implementation Details

In this section we will provide some information about our implementation of the algorithm described above.

In our implementation we combined locally running versions of Sylvan with a BDD partitioning approach. For this, we tried to reuse as much code from Sylvan as possible. With respect to the splitting procedure there was no existing code that we could base our implementation on. This is why we have implemented this part from scratch. As the communication interface between processes we used MPI, the Message Passing Interface. One advantage of MPI is that it can handle communication between processes on one machine as well as communication between processes on different machines without the need to change any of the code.

We used C++ for our implementation since it is fast and compatible with both Sylvan and MPI.

4.7.1 Functionality from Sylvan

In our implementation several features of Sylvan have been used. The goal was to minimize effort of reprogramming and to take advantage of the achievements of Sylvan. As two basic features we took the cache and unique nodes table. These functionalities provide fast access and computation on BDDs. Furthermore, the unique table supports garbage collection, which releases us of tedious memory management during BDD operations. For a closer look on the details of Sylvan we refer to [4].

Next to the cache and unique table, we could reuse Sylvan's implementations of BDD operations and node creation. Common commands of Sylvan that we adopted in our work are *and*, *or*, *not* and *RelProd* (also see [4]).

Two more functions that are important are *satcount* and *nodecount*. The former counts the number of satisfying variable assignments of a given BDD, the latter counts all nodes in it. While satcount is usually only called after the analysis, this is not true for nodecount. In our design, we need to know the current size of every BDD partition in each iteration of the analysis. This means that every worker calls this function at least once per reachability step. In addition to this, if a worker needs to split its BDD, the splitting procedure will be initialized, which bases its cost function on the sizes of both resulting BDD partitions. Every time a worker searches for a split variable, the nodecount function may be called as many times as there are levels in a given BDD. In other words, as the nodecount function is likely to be called very often during an analysis run, it is vital for its algorithm to be as performant as possible. In vanilla Sylvan this function could not be executed in parallel. This is why we implemented our own version to be able to calculate the size of two BDDs in parallel. The reason why Sylvan can count the nodes of only one BDD at a time is that there exists only one marker type for visited nodes. By adding an extra marker type the algorithm can now distinguish which nodes for which count run are already visited.

4.7.2 Sending and Receiving BDDs with MPI

For the communication within the network we used Open MPI, an open source implementation of the Message Passing Interface. This API bundles a tool set of common purpose high-speed data transmission operations. One special property of MPI is that it can create processes and it does not matter how these are distributed over machines. The software frees the developer from concerns

regarding the underlying design and design changes of the network. MPI provides an abstraction layer which eliminates the need for the programmer to distinguish between processes on the same machine and processes on different machines.

Open MPI has many built-in functions of which *MPI_Send*, *MPI_Recv*, *MPI_Isend*, *MPI_Wait* and *MPI_Bcast* have been used in our implementation. Further details on these functions can be found on [35, 36].

Chapter 5

Experimental Evaluation

In order to evaluate the performance of the introduced algorithms, we will run our tests on models of the BEEM database and well-known Petri Nets. We chose these models, because they are often used in related work, which makes it easier to compare the performance of our approach with others. In this chapter we will present and discuss the benchmark results of our implementation. First the results will be reviewed in relation to existing work. Then, we look at performance and performance variation of our implementation in more detail. From this point on we will use the name *PartBDD* for our implementation using partitioned BDDs.

All test runs have been done on the DAS-5 compute cluster (Distributed ASCI Supercomputer 5) [39]. The cluster consists of 68 dual-eight-core compute nodes (ASUS nodes with Intel® Xeon® E5-2630v3 CPUs, 2.4 GHz). Each node has 64 GB working memory and the operating system is CentOS Linux, release 7.2. The nodes are linked via Infiniband for high-speed interconnection.

During all test runs we utilized 16 cores on each machine of the cluster if not stated otherwise. Since we assigned exactly one process to each machine, this also means that each process had 16 threads. We compare our results with documented test runs of Sylvan and DistBDD. All analyses with Sylvan are done on a single machine with one or 16 threads. We refer to the single-core execution with *Sylvan linear* or just *linear* and with *Sylvan* to its 16-core execution. When we compare our results to those of DistBDD, we always refer to analyses with 16 threads and a varying number of machines.

The results show that the choice of split parameters – split size and split count – is crucial for the performance of our implementation. These parameters determine at which size a BDD will be divided and – when reaching this limit – into how many parts the BDD is split. The choice of both values can influence

the total execution time of reachability analysis significantly.

In many cases computations with our program were slower than those done with Sylvan or DistBDD. However, for some models our implementation reached speedups of up to 5.3 compared to DistBDD and 2.5 compared with Sylvan with specific configurations of split size and split count. For the comparison with DistBDD we only used computations where our approach used at least two worker processes – executions with a single worker process do not involve partitioning, which is the essential feature of our algorithm. We also compared our results with a linear analysis (using Sylvan on one machine with a single thread). In this case speedups of up to 29 could be observed.

The communication overhead between idle machines/processes is negligible as we will see in greater detail in section 5.3. We define idle machines/processes as those that did not receive a BDD to analyze according to the description in section 4.1. As opposed to this, active machines or (worker) processes are those with a status different from idle. The terms non-idle and active processes are used interchangeably in the following sections.

In general, the number of total nodes at the end of an analysis does not correlate with the number of active workers. The increase or achieved reduction seems to depend on the structure of the model instead of the number of generated partitions. For some models a significant reduction of nodes could be achieved compared to the number of final nodes in a single machine computation. Our results show a maximum reduction of nearly 40%, from 1593166 nodes with a single active worker to 957244 nodes with 7 active workers and a split count of 4.

The tests also show that it is not beneficial to partition very small BDDs. By very small we mean that reachability analyses on those models finish within a few milliseconds to seconds. In these cases splitting the BDD and exchanging partitions takes more time than an analysis on a single machine would take.

The main reason for unsuccessful reachability analyses was an unsuitable combination of split size and split count. When the split size is chosen too small, the split count too large or both, there are no machines left where BDD partitions could be send to. Another reason for unsuccessful runs were timeouts. For most analyses we set the timeout to 5 hours. Especially the larger models did not even split, before this time limit had been reached. On the other hand, when they did split (smaller split size), in most cases they had to split very often, outrunning the number of available machines/worker processes such that the program stopped.

5.1 Overall performance

Compared to a linear analysis PartBDD reached significant speedups of up to 29. The best achieved results, compared to a linear execution, are shown in Figure 5.1. The figure plots execution times against the number of utilized machines for Sylvan (1 in this case). For PartBDD, the x-axis indicates the number of worker processes that are involved in the computation and not the number of machines that are available. We will explain this in section 5.3. Every analysis with n non-idle/active worker processes can be executed on $n+1$ or more processes (n worker processes and one master process). Since we chose to assign one process to each machine, "n non-idle worker processes" means, that an analysis was done on at least $n+1$ machines. Our implementation reached its minimum execution time for model collision4 with 14 active worker processes.

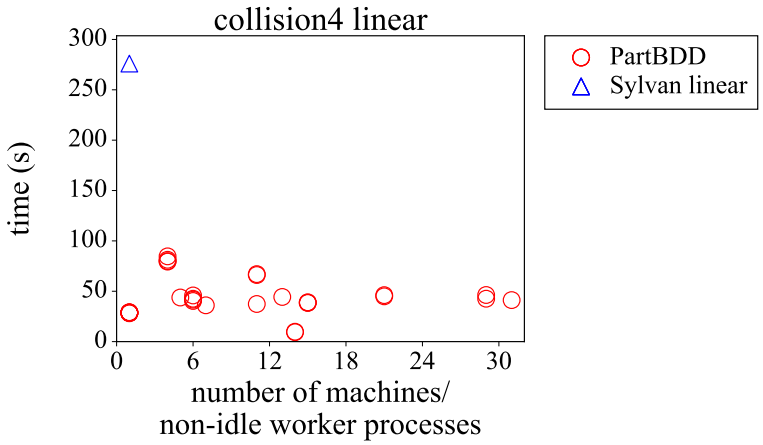


Figure 5.1: Model (collision4) with the best achieved speedup with respect to linear analysis. Linear analysis means that Sylvan performed the computation on a single machine and a single core. PartBDD utilized 16 cores on each machine. The model was analyzed several times with different configurations of split size and split count.

In general, PartBDD was significantly faster than the linear execution. For every analyzed model there was at least one run with our implementation which had been faster. However, especially for small models, some of PartBDD's execution times were higher than the linear one's due to unsuitable split sizes or split counts.

The best speedups that our analyses reached compared to Sylvan and DistBDD were 5.3 and 2.5 respectively. For most models the execution times using DistBDD decrease as the number of machines increases. An exception is the computation on one machine, which is often faster than the distributed analy-

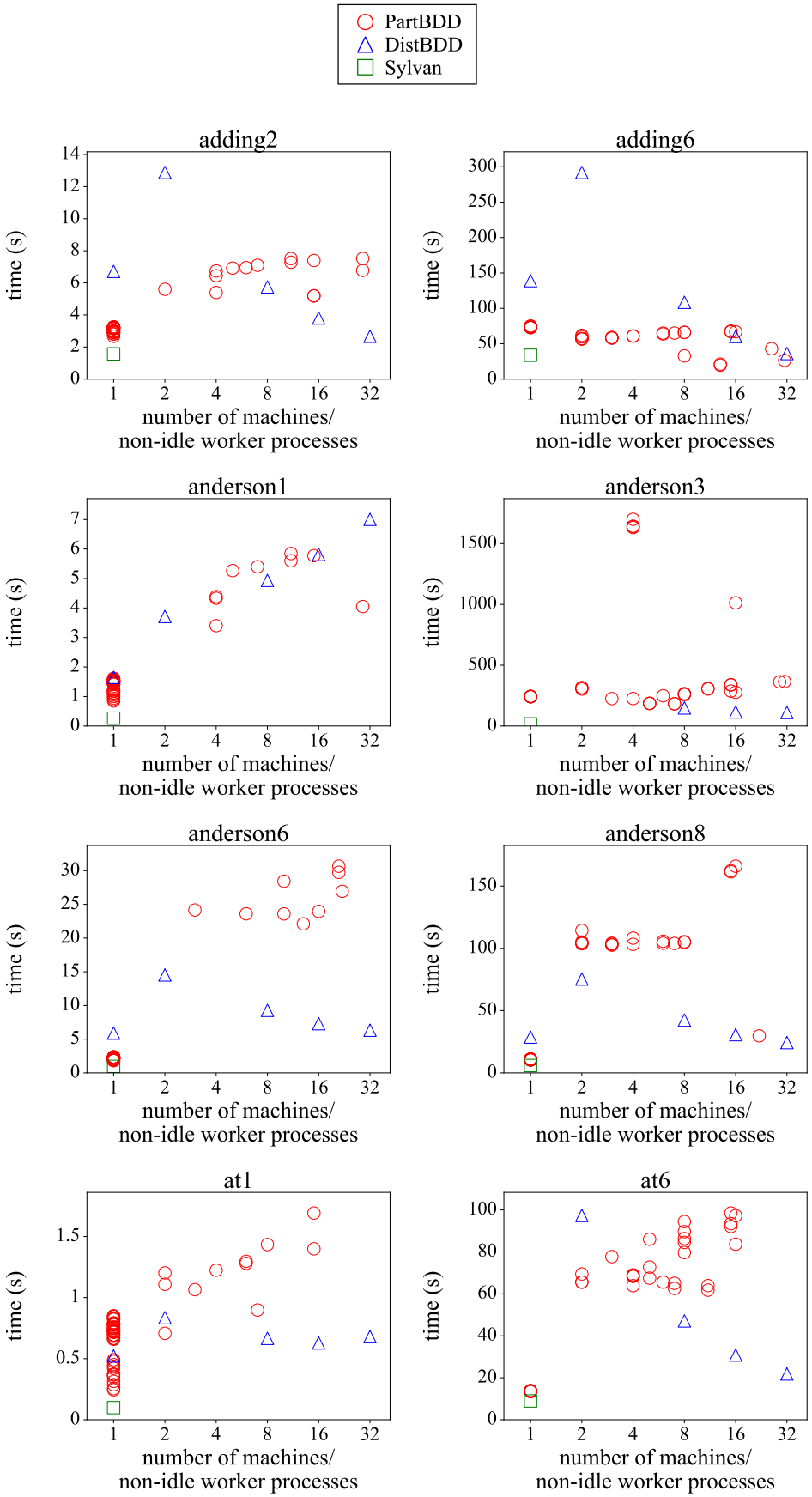


Figure 5.2: Benchmark results compared to Sylvan and DistBDD. Each plot shows multiple analyses (with 16 threads per machine) of a single model. For Sylvan and DistBDD the x-axis gives the number of utilized machines, for PartBDD it indicates the number of active worker processes at the end of an analysis.

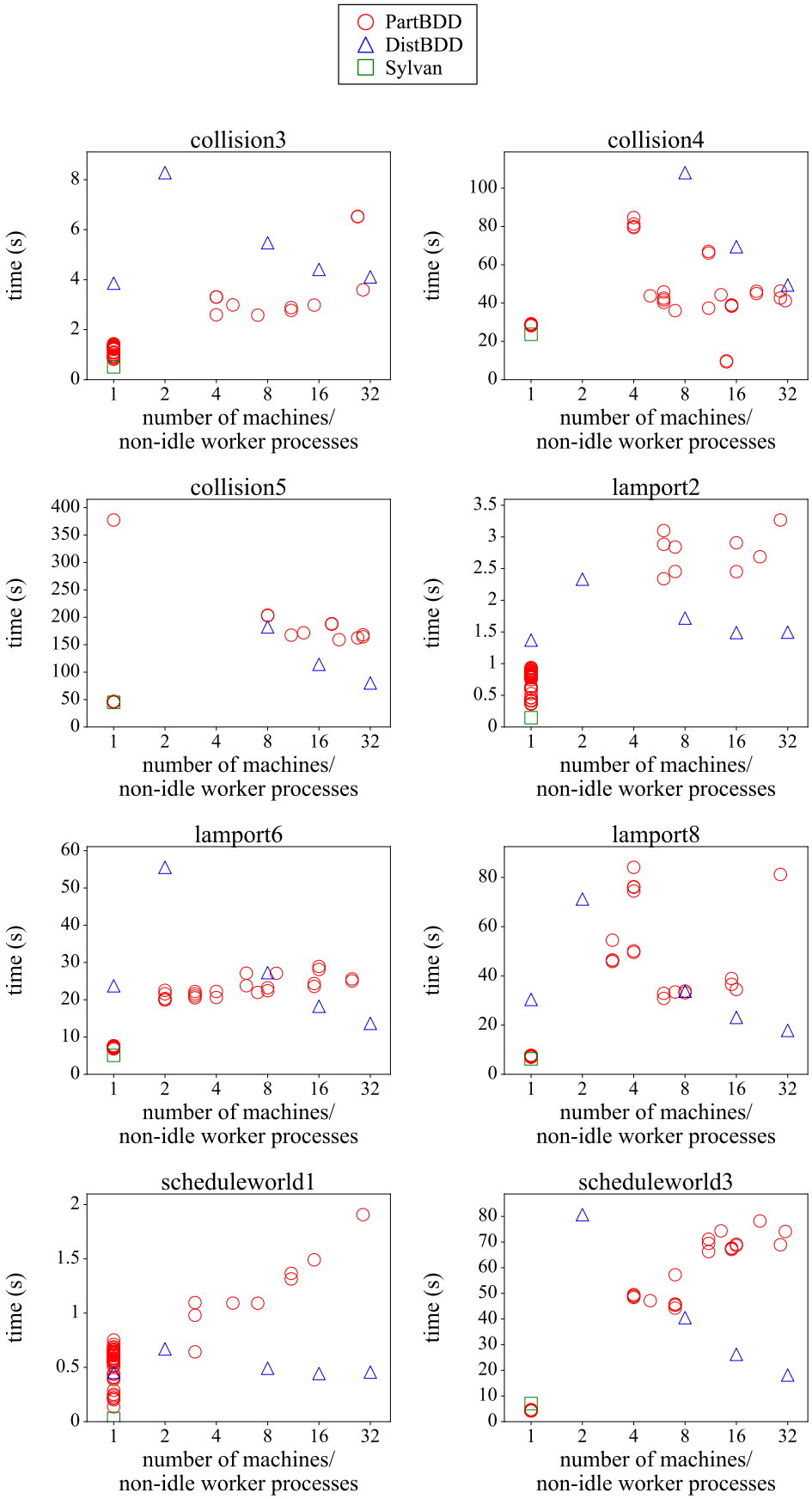


Figure 5.2: Benchmark results compared to Sylvan and DistBDD. (continued)

ses. In contrast, we could not observe a correlation between the execution time and the number of active worker processes using PartBDD. Figure 5.2 visualizes a selection of our benchmark results and compares them to DistBDD and Sylvan. The performance of PartBDD compared to DistBDD and Sylvan varies per model and configuration. We based the selection of results on different aspects such as the size of the model and the relation between analysis configurations and execution time. In this way, we try to cover all different effects and observations like positive, negative or no correlation between parameters.

Most of the time PartBDD was slower than Sylvan with 16 threads, which does not surprise, since Sylvan is used for the reachability analyses and PartBDD adds additional communication between processes. However, we were able to find configurations (split size and split count) for which PartBDD was faster than Sylvan. This is the case for models adding6 and collision4.

The best results compared to DistBDD were achieved for models adding6, anderson1, collision3, and collision4. The poorest performance can be observed for models anderson6, at1, lamport2, scheduleworld1 and scheduleworld3. Anderson8 is one of the models for which PartBDD was slower than DistBDD for most runs. Nevertheless, with a good configuration, results could be achieved that came close to the results of DistBDD (see Figure 5.2, anderson8, 22 non-idle worker processes). We could not relate the results to properties of the mentioned models like their size of number of final states.

Our benchmark results also show that the analyses on one machine – this means without partitioning – were often the fastest. In many cases there is a steep rise in execution time when going from a single worker to multiple workers. The time difference between single and multiple worker executions is often larger than the variation within the multiple worker executions (e.g. anderson1 and anderson6). However, this is not true for all models (e.g. anderson3 and collision4).

5.2 Number of Final Nodes

When partitioning a BDD, our algorithm searches for the best split variable to create a partition. The choice of this split variable, but also the general structure of a BDD influences the total number of nodes that are needed to represent all partitions. For some models partitioning might not be possible without creating many duplicate nodes. In order to draw conclusions about our approach we

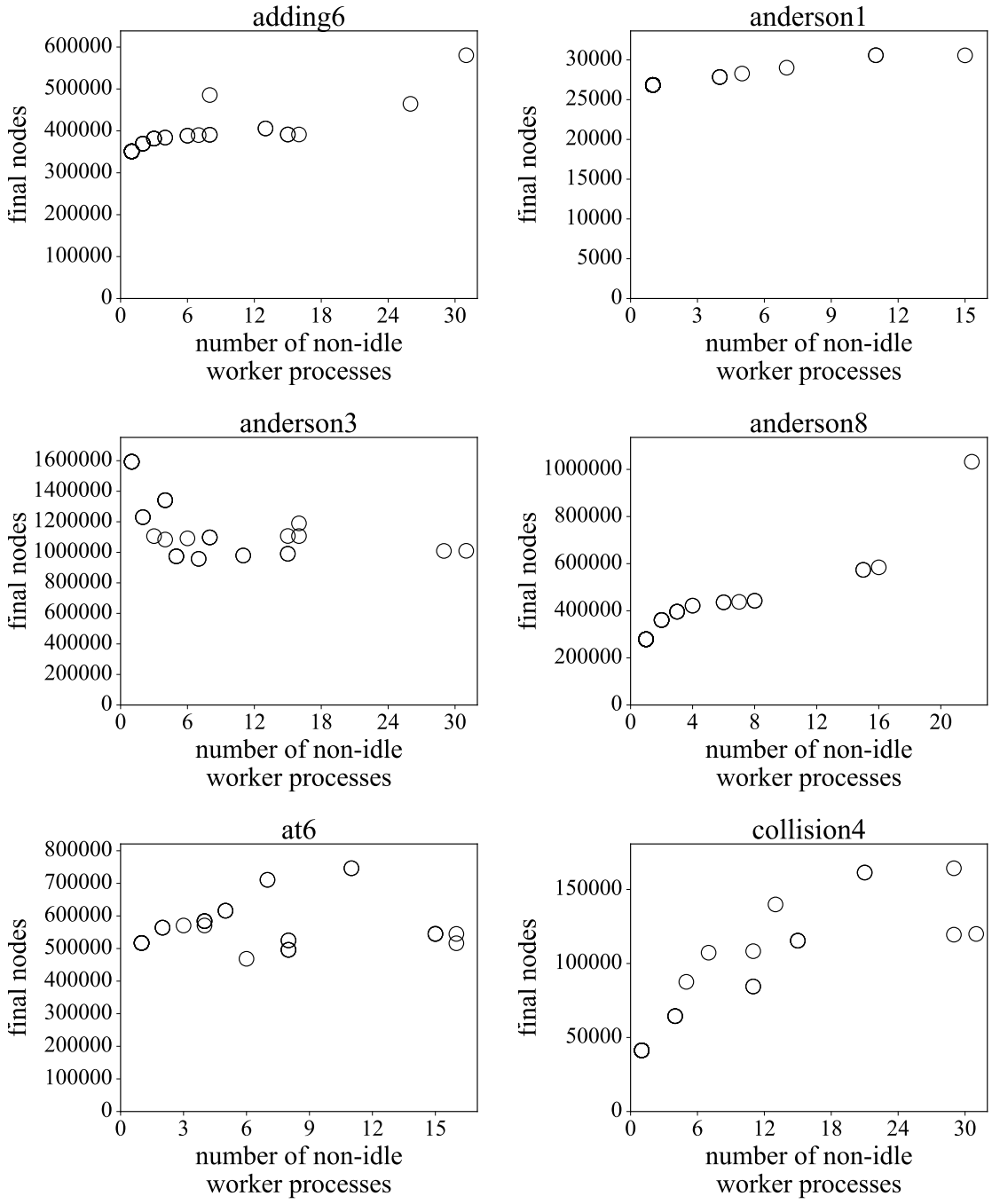


Figure 5.3: Plots of final number of nodes against the number of active worker processes. The final number of nodes is the sum of all worker's BDD nodes at the end of an analysis.

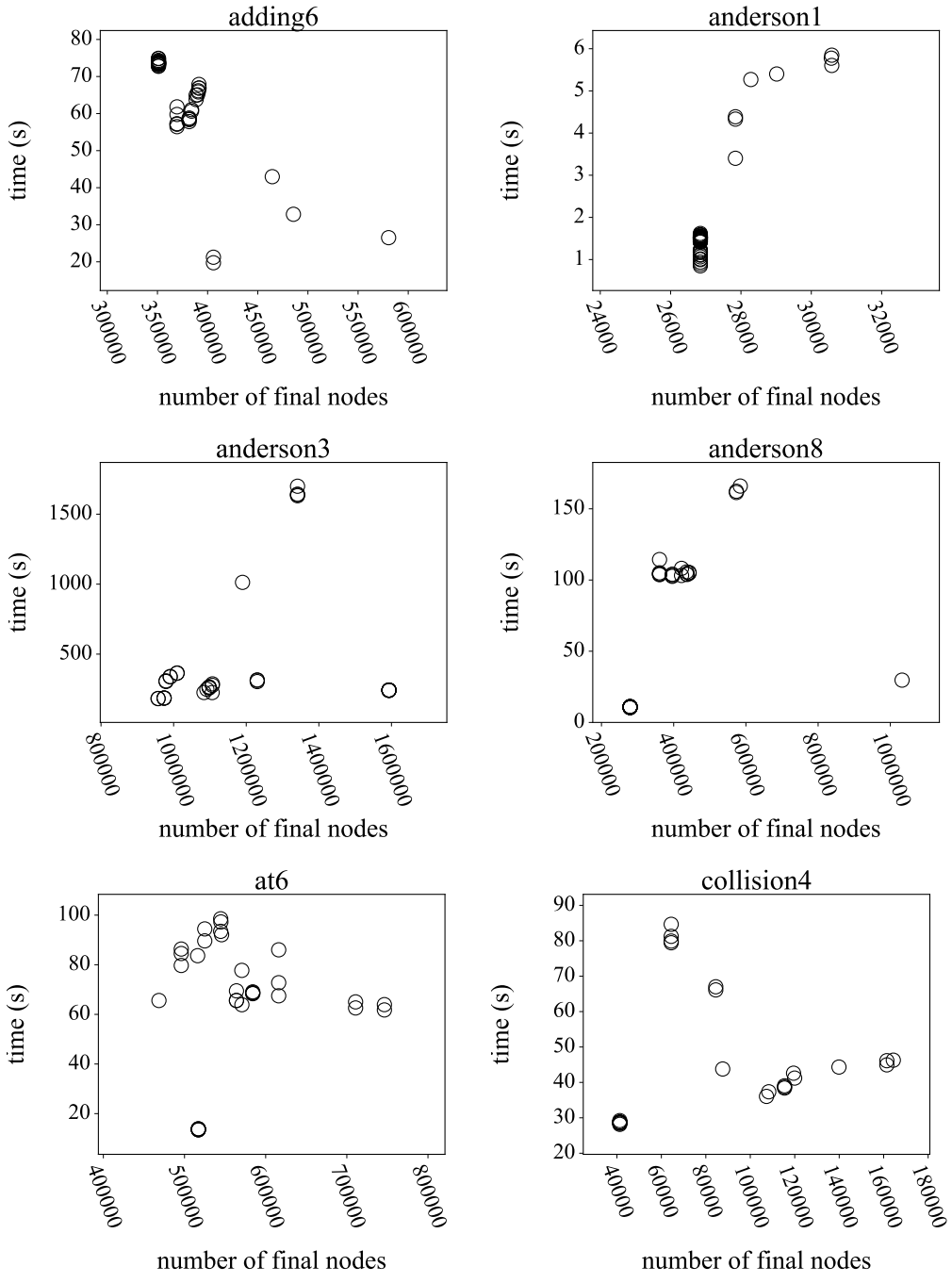


Figure 5.4: Plots of execution times against the final number of nodes. The final number of nodes is the sum of all worker's BDD nodes at the end of an analysis.

measured the number of final nodes of each execution (this is the sum of the final nodes of each partition) and looked at two relations:

1. The number of nodes depending on the number of partitions.
2. The execution time depending on the number of final nodes.

These relations are visualized in Figures 5.3 and 5.4 for a selection of models.

Regarding the correlation of the final number of nodes and the number of active worker processes/BDD partitions (Figure 5.3), it is not possible to generalize any conclusions for all models. Model anderson1 shows a slight increase of nodes for more partitions, while the increase of nodes is stronger for model collision4. The number of nodes in adding6 and anderson8 rises from 1 to 4 partitions, stays nearly constant for a medium number of partitions and rises again for more than 13 to 18 partitions. There is no tendency of an in- or decreasing number of nodes for model at6. Finally, the number of final nodes does even decrease for anderson3 for the first partitions and stays constant after that.

Figure 5.4 shows that there is also no clear correlation between final number of nodes and execution time. For instance, analyses on adding6 performed better when more nodes were created (in this case this also means more partitions, see Figure 5.3). The execution time of anderson1 significantly rises with the number of nodes in the range of 26000 to 31000 nodes. Considering the total execution time of at most 6 seconds and the increasing number of nodes with the number of partitions as shown in Figure 5.3, this difference could be caused by the splitting time instead of number of nodes. Additionally the execution time with less than 18000 nodes is not slower or faster than the executions with more nodes. Also models anderson3 and at6 show that the number of nodes does not necessarily influence the execution time, while collision4 indicates that a (nearly) constant number of nodes can still result in varying execution times.

5.3 Communication Overhead

5.3.1 Network Traffic Caused by Idle Workers

In our implementation all initialized worker processes need to communicate with each other and with the master process a few times in each iteration of the analysis. This is necessary even if workers are still idle. We measured the communication overhead induced by idle processes to find out if the total execution

time is influenced by the number of initialized worker processes, especially in the case that not all processes are needed due to a low number of splits.

We measured this by repeatedly executing the same analysis of the same model with a high split size (such that no partitions would be created) and varying number of machines/worker processes. Since no partitioning takes place due to the high split size the value of the split count parameter is not important in this case. A selection of the results is shown in Figure 5.5. The observed execution times indicate that about one second difference can be attributed to the number of idle workers, regardless of the size of the model. For very small models (see lamport2) this can cause a significant increase of execution time, since the entire analysis on one machine takes less than 0.4 seconds. For larger models this variation can be neglected since it only makes up a small fraction of the total execution time (see adding4). Even if the number of machines is increased even more and the communication time caused by idle processes grows as well, this amount will not significantly change the total execution time.

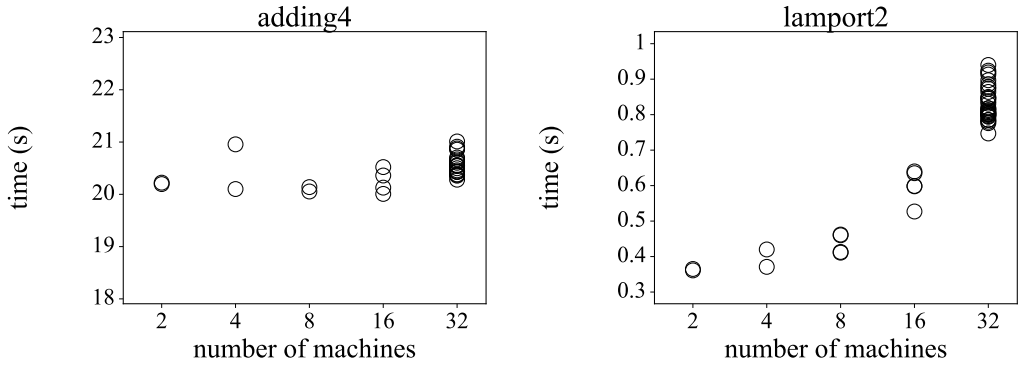


Figure 5.5: Execution times plotted against the number of machines for runs with a single active worker process. This visualizes the communication overhead caused by idle workers. Each dot represents a single run with x machines. Since at least one worker and one master are needed for every analysis, the minimum number of machines is 2. This means that an analysis using x machines was done by one master process/machine, one active worker process/machine and $max(0, x-2)$ idle workers.

5.3.2 Network Traffic Caused by Active Workers

The approach mentioned above analyzes the network traffic caused by idle processes. We also tried to measure latency of the network and message sizes caused by active workers (e.g. due to exchanging non-owned states). For this

purpose we chose five models and six different configurations and we analyzed each model ten times with each configuration. This is 60 runs per model in total. For the chosen models we knew in advance which split size would result in which number of splits. Note that this is the only time that we used a number of threads per process different from 16. The six configurations differ in (1) network design, (2) split size and (3) split count.

(1) Network design:

   (a) Analysis on 1 machine with 4 processes (1 master, 3 workers) and 4 threads per process (16 total)

   (b) Analysis on 4 machines (1 master, 3 workers) with each 1 process and 4 threads per process (16 total)

(2) Split size:

   (a) High split size such that the BDD will not be partitioned

   (b) Lower split size such that the BDD will be partitioned exactly once

(3) Split count:

   (a) Split count 2 such that two worker processes are active after one split

   (b) Split count 3 such that all three worker processes are active after one split

To distinguish the different designs, we will use the notation $T(m,n)$ in the following, where m indicates the number of machines and n the number of active worker processes. The six combinations of configurations are as follows:

In combination with different split sizes and split counts both network designs produce one, two and three BDD partitions (and active worker processes) for different runs of each model.

Using all six setups we can compare execution times of both network designs for each combination of split size and split count independently to see if communication between machines is more expensive than communication on one machine. Furthermore we can compare both designs with respect to the changes of computation time depending on the number of partitions.

The benchmark results of the approach described above are shown in Figures 5.6 and 5.7. The first figure combines the result with the corresponding

network design	**split size**	**split count**	$T(m,n)$
(1a) 4 processes on a single machine, 4 cores per process	(2a) 50.000.000	– ((3a) 2, does not split)	$T(1,1)$
	(2b) 500.000	(3a) 2	$T(1,2)$
		(3b) 3	$T(1,3)$
(1b) 4 processes distributed over 4 machines, 4 cores per process	(2a) 50.000.000	– ((3a) 2, does not split)	$T(4,1)$
	(2b) 500.000	(3a) 2	$T(4,2)$
		(3b) 3	$T(4,3)$

final node counts. The second shows the execution times together with the time that was needed to partition the BDD. This includes selecting one or two split variables – depending on the split count – and partitioning the BDD using the determined variable(s).

There are several aspects we can observe. First of all, model adding6 is the only one that shows a decrease in execution time when the number of partitions increases. All other model's executions were fastest with a single worker process ($T(x,1)$). For those models the execution time increases significantly as we compare the analyses of one worker with those of multiple active workers. The difference between two and three active workers is a lot smaller – if there is any at all.

The analyses on four machines ($T(4,x)$) were almost always faster than those on a single machine ($T(1,x)$) regardless of the fact that the processes in both designs had the same resources available (4 cores). The differences in performance are the least without partitioning ($T(x,1)$) and increase for two and three partitions ($T(x,2)$ and $T(x,3)$) (except for adding6). We explain the better overall performance of the design with four machines with a better work load balancing. When we assume that idle workers do only minimal work, a computation with one active worker process uses the capacity of up to 8 cores (4 for the active worker and 4 for the master). In our case this is a half of the available cores on a single machine ($T(1,1)$). If distributed over multiple machines, only 4 cores would be occupied ($T(4,1)$). When the number of active workers is three, all cores of the machine of the computation with only one machine ($T(1,3)$) are utilized. As opposed to this, in the other design ($T(4,3)$) only a quarter of each machine's cores is used. Because the processing speed (especially memory access speed) does not scale linearly with the number of

utilized cores on a machine, this could explain the faster executions on four machines although communication overhead rises with increasing number of machines.

The number of final nodes increased for all models when more partitions were created. While the number of final nodes of model adding6 only changed by a factor of less than 1.1, the final node count of models anderson8 and lamport7 is nearly 1.5 times as much for three partitions ($T(x,3)$) as for a single partition ($T(x,1)$). Also creating three partitions requires more time than creating two partitions. This splitting time, however, does not increase linearly, but splitting a BDD into two partitions takes approximately $\frac{2}{3}$ of the time it takes to create three partitions. No significant difference can be observed between both network designs. The splitting procedures took up to 12 seconds depending on the model and split count. For some models this is a large proportion of the time an analysis without splitting lasts in total.

If we assume the communication, especially the exchange of non-owned states to be responsible for the increase of processing time, we would expect a correlation between the number of final nodes and the execution time. Of course, the maximum number of nodes during the analysis (peek) would be a more accurate value, but since we did not measure BDD sizes in each iteration of the calculation, we use the final number of nodes for this purpose. The higher the increase of final nodes, the more duplicate nodes were created through partitioning and the more communication between active workers is expected to be necessary. More specifically, if network latency is the bottleneck, we also expect the communication *between* machines (in the 4 machine design, $T(4,x)$) to be more expensive than the communication *within* a machine (1 machine version, $T(1,x)$), which would cause a faster increase of execution time in the design with four machines when more partitions are created. As we compare the one and the two worker computations in relation to those factors (processing time and number of final nodes) we observe that the number of nodes as well as the execution time of most models increases. However, while the difference in execution time between both network designs is little or not available when no partitioning is done, the analyses on four machines finished significantly earlier than those on a single machine for the runs with two active workers ($T(4,2)$ and $T(1,2)$ respectively). This is exactly the opposite of what we would expect to observe, if the communication between machines via the network would be the bottleneck. Even if the network latency attributes to the increase of execution time, this is compensated by a better load balancing in the case of four

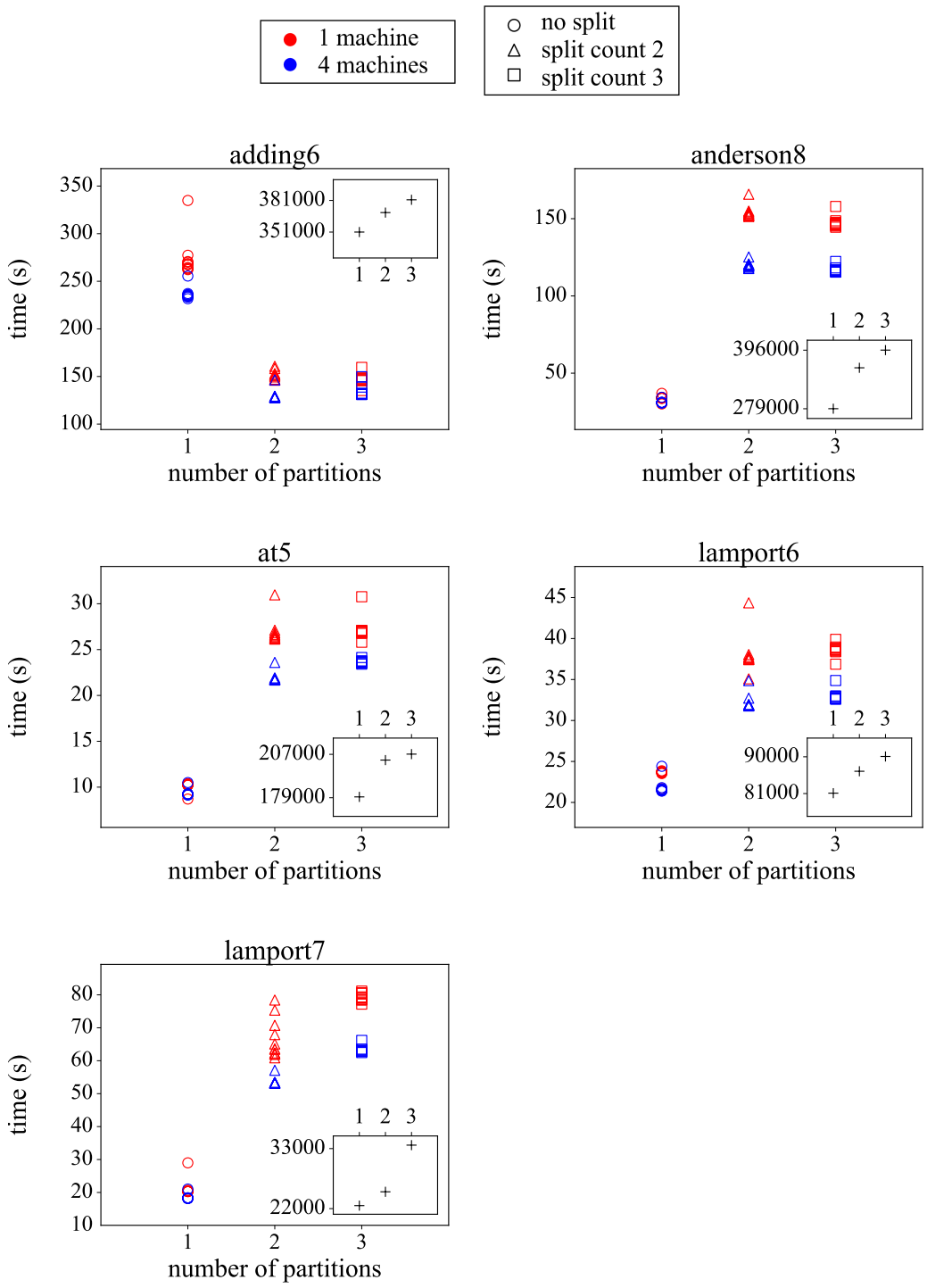


Figure 5.6: Comparision of analyses on one and four machines. The plots show the execution time (large plots) and corresponding final number of nodes (small plots) against the number of partitions for each model. The different markers indicate the network design and split count. The split count equals the number of partitions and active worker processes. Each setting was repeated ten times for every model.

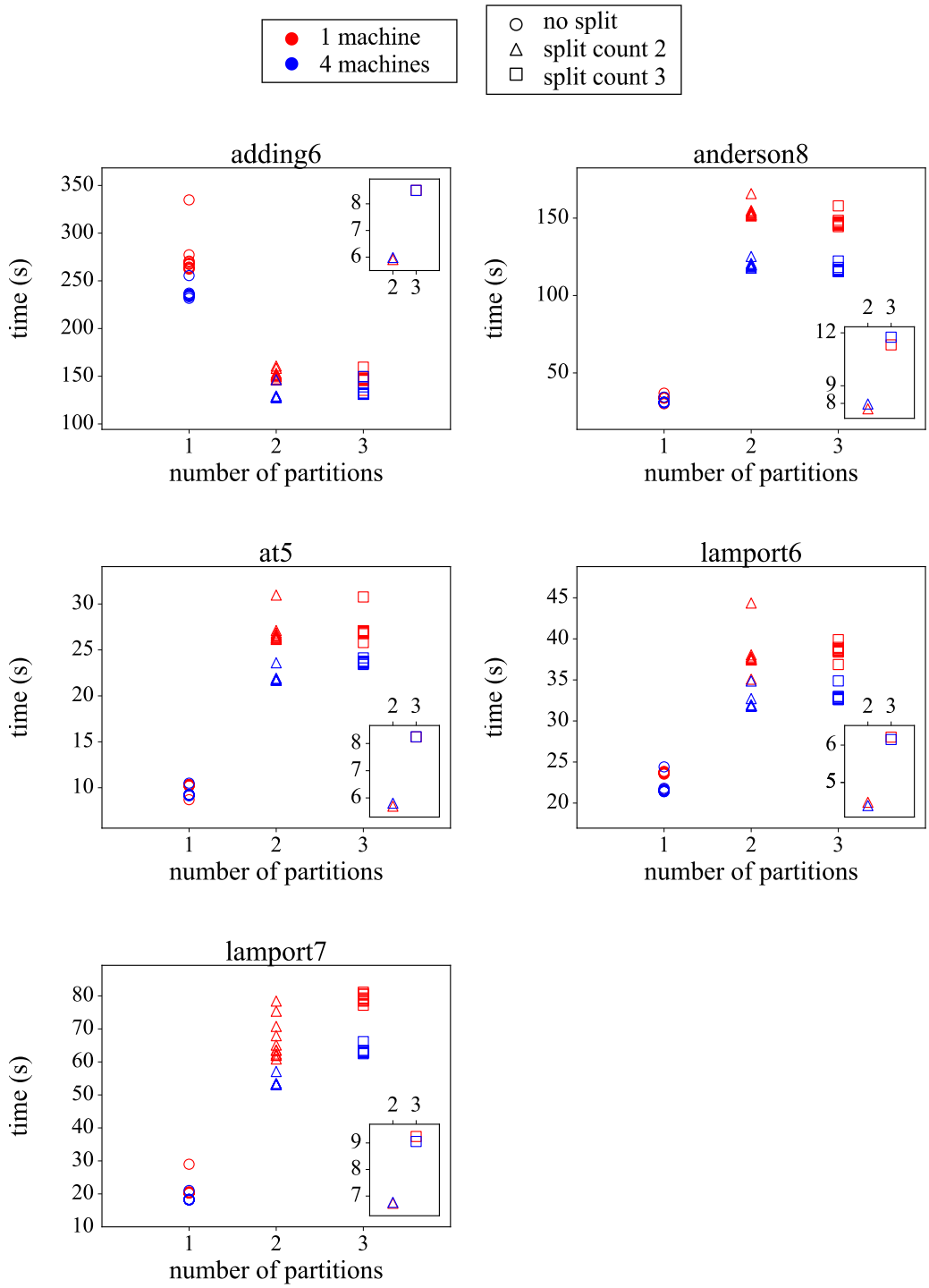


Figure 5.7: Comparision of analyses on one and four machines. The plots show the execution time (large plots) and the corresponding split time (small plots) against the number of partitions for each model. The different markers indicate the network design and split count. The split count equals the number of partitions and active worker processes. The split time is the time that the first active worker needed to select a split variable and partition its BDD.

utilized machines. We see similar results for the runs with tree partitions. The four machine design is still faster than the single machine design and the time difference between both does not decrease with more partitions.

The increase of execution time could be caused by combination of the partitioning procedure (finding split variables and splitting the BDD), duplicate work that has to be done due to additional nodes through partitioning, and idle times of active workers.

5.4 Influence of Split Size and Split Count

As mentioned earlier, the choice of split size and split count has a great influence on the performance of our implementation. In this section we will discuss some of our benchmark results in detail with respect to these two parameters. In order to minimize variation caused by other factors than split size and split count, all reachability analyses that are used in this section are executed on 32 machines with 16 threads on each machine. We will show some selected results in this section. More plots can be found in appendix A.

Over all reachability analyses there was no clear correlation between the split size and execution time. For some models it seems to be beneficial to split early, for others a high split size resulted in better performance. For most models similar execution times were possible with different split sizes, provided that the split sizes were small enough such that the partitioning procedure would be carried out at least once. There is also no significant correlation between split count and execution time. However, some conclusions about the interaction of split size and split count can be drawn.

In Figure 5.8 we plotted the execution times of two models against the split count. For both models the split size is constant for all runs of that model (1,000,000 for anderson3 and 500,000 for at5). These split sizes caused each analysis to partition exactly once. The results of many other models are similar to these ones. For every model and constant split size there seems to be one best split count. To the left and to the right of this best number the execution times increase. For instance, a split count of 3 would be optimal for anderson3 with the given split size, while at5 performs better with split size 4. We explain the decrease of execution time (from split count 2 to the optimal split count) by a better distribution of work. Also the partitioning/splitting time does not increase linearly with the number of partitions to be created (split count), but

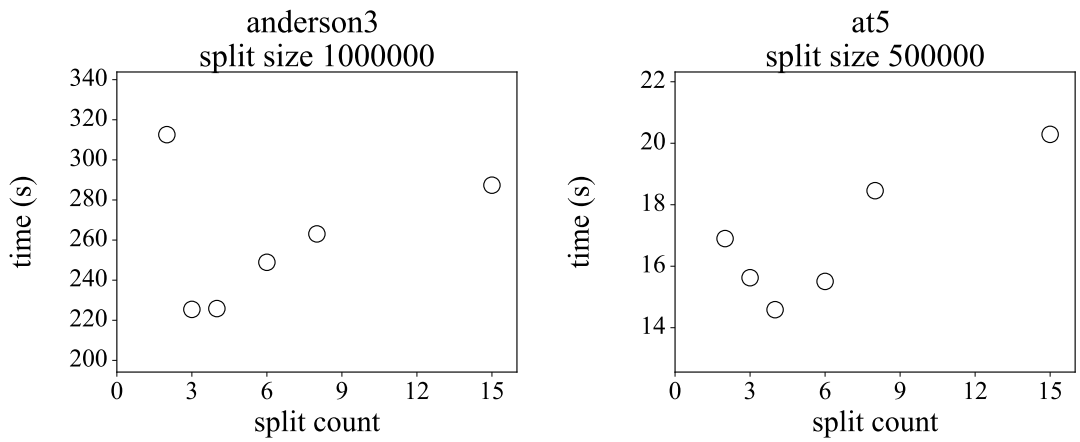


Figure 5.8: Execution times plotted against the split count with a constant split size of 1,000,000 and 500,000 for models anderson3 and at5 respectively. In every run of both models the split size was exceeded exactly once during each execution.

slower. Therefore, creating more partitions might be beneficial. The increase of execution time for split counts higher than the optimal one could be explained by additional splitting time on the one hand and the amount of duplicate work and additional communication on the other hand. Also idle times might become a problem. If too many partitions are created, the proportion of time for the actual reachability analysis might decrease as compared to the amount of time that the synchronization, communication with other workers and the duplicate work takes.

Figure 5.9 shows results of analyses of model at7. The execution time is plotted against the number of splits and the split count. Furthermore, two plots show results with a split size of 500,000, the other ones with a split size of 1,000,000. Markers give information about split count and number of splits, depending on which variable is used on the x-axis. The number of splits gives the number of times that a BDD needed to be partitioned (exceeded the split size). This means that in a run with e.g. 2 splits and a split count of 3, four split variables had to be found during the entire analysis.

When the split size is 500,000, the execution time increases with the number of splits and decreases as the split count increases. In this case, a higher split count also results in more splits. From this we can conclude that a high split count can be beneficial, if the partitions that are created are small or at least do not grow fast during following iterations such that the high split count results in less splits.

With a higher split size (1,000,000) at7 splits at most three times. The

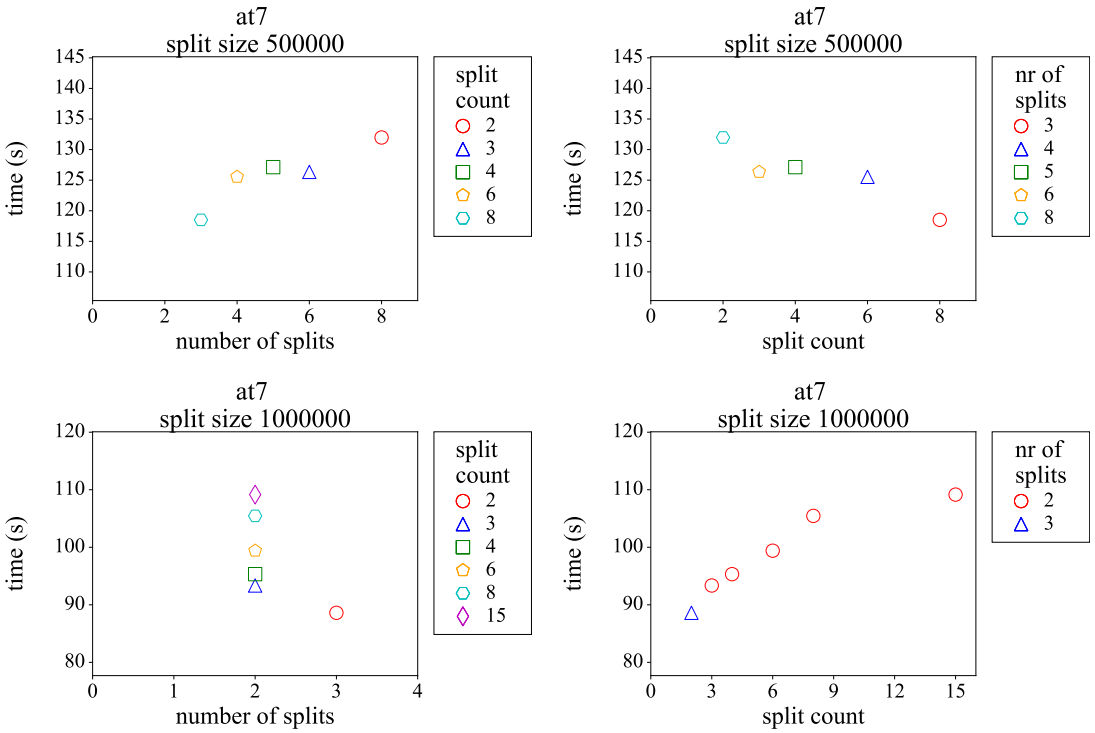


Figure 5.9: Execution times plotted against the number of splits and split count with split size 500,000 and 1,000,000. The markers indicate the corresponding split count or number of splits.

fastest computation is the one with a split count of 2 and 3 splits. Runs with a higher split count had to partition two times and execution times increase with the split count. The fact that the execution with split count 2 is fastest – although it had to split three instead of two times – could be explained by a better timing of the partitioning procedure. In this case, the first time the BDD's size reaches the split size only two partitions are created. This means that only two workers are active, which need to exchange nodes and have to wait for each other to finish an iteration (all workers need to synchronize, but it is unlikely that idle workers finish after active workers). Only when a partition again exceeds the size of 1,000,000 nodes, a third partition is created and an additional worker will be active.

We can also observe that the analyses with a higher split size are faster for this model.

For model collision4 (Figure 5.10) the differences between execution times of runs with different split sizes are smaller than those of at7. As opposed to model at7, here, a split count of 4 is the best choice for both split sizes. With

split size 500,000 split count 3 increases the execution time significantly. The same is true for split count 2 when the split size is 1,000,000. Although no significant correlation between final node count and execution time could be found in general, in this case the high execution time could still be caused by an increase of nodes. For most runs of collision4 the final number of nodes is less than 200,000. When split size and split count are chosen to be 1,000,000 and 2, this number is more than 1,100,000.

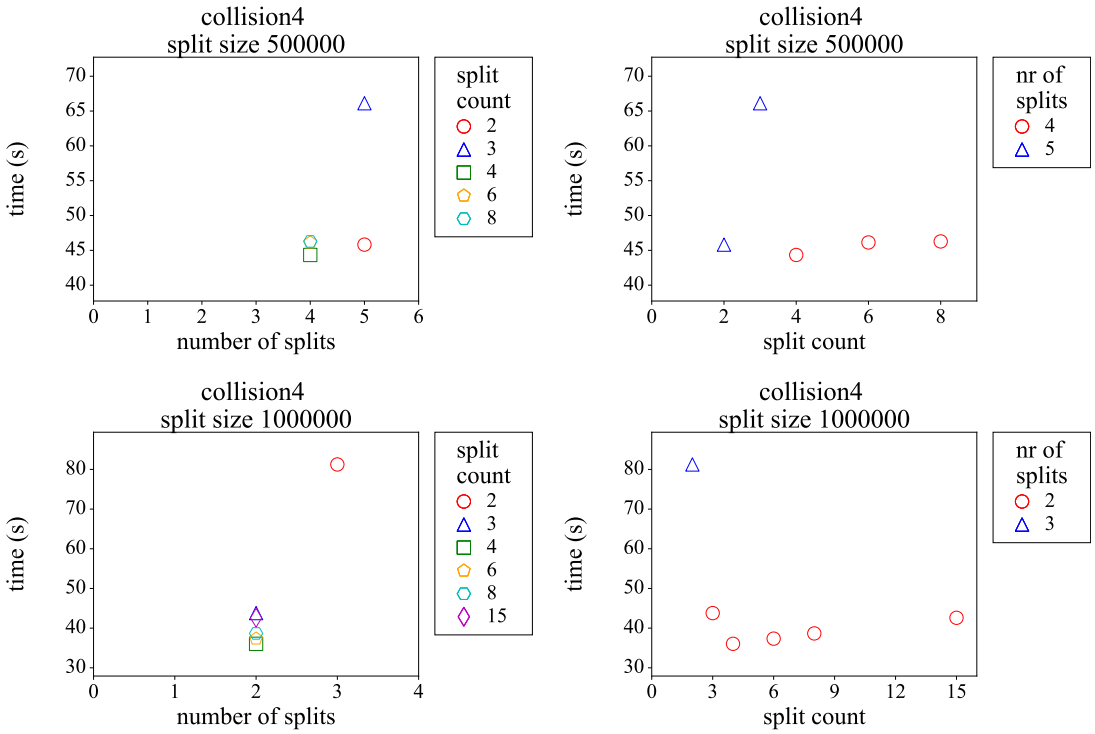


Figure 5.10: Execution times plotted against the number of splits and split count with split size 500,000 and 1,000,000. The markers indicate the corresponding split count or number of splits. The number of splits indicates the number of times that a partition's BDD exceeded the split size and the partitioning procedure was initialized.

To conclude, there is no general best combination of split size and split count for all models based on the variables we monitored. The underlying structure of the model also plays an important role.

5.5 Validation

As a measurement of the validity of our implementation we used the final numbers of states that were computed when performing reachability analyses on the

models. We compared these values with the reported results from [10, 11, 40]. For most executions using our implementation the state counts were equal to those calculated with DistBDD or Sylvan. However, some runs resulted in state counts that deviated by a few states (usually less than ten) from the reported correct numbers. This happened sometimes, when many partitions were created during the analysis due to a high split count. We suspect that this is caused by synchronization issues during the exchange of non-owned states. However, due to the fact that this happens only in a few cases, we observed this issue only after finishing the benchmarks. Therefore, it is not yet resolved in the implementation.

Chapter 6

Conclusion and recommendations

6.1 Conclusion

6.1.1 How can principles of vertical partitioning be combined with existing multi-core model checking solutions?

In our work we combined the multi-core symbolic model checker Sylvan with the vertical partitioning approach for distributed reachability analysis of Grumberg et al. As there was no implementation of their vertical partitioning approach that we know of, we based our design on the algorithms described in [34], including the cost function which is used for finding suitable split variables. The local reachability analyses are done by a modified version of Sylvan. We added functionality to synchronize and exchange data between processes and machines (via MPI).

Parallelization was one of the bottlenecks of [34]. Grumberg et al. parallelized some parts of the function to find a split variable, but this was done on multiple single core machines and not by multiple cores on a single machine. By using Sylvan and multi-core hardware we added the possibility to parallelize on a single machine. Our multi-core solution enables us to keep some parallel computations local and thereby avoid additional network traffic.

Compared to DistBDD, network latency is not the bottleneck of our final design. In each iteration of reachability analysis processes need to synchronize a few times (see Figures 4.1 and 4.2) to exchange data. This means that in our approach larger chunks are sent at once, but also that during the calculation of next states no communication takes place. The drawback of this design is that all processes have to synchronize for exchanging non-owned states. This

results in idle times during which the machine's capacities are not fully utilized. Another difference compared to DistBDD is, that in our design an analysis is kept on a single or few machines as long as possible (depending on the split size), whereas DistBDD uses all available machines right from the beginning. The main advantage of keeping analyses on one or few machines is that there is no or little network traffic. Of course, this way also less hardware recourses are utilized.

The greatest shortcoming of our current design and implementation is the fact that each worker process can only receive one BDD partition during an entire reachability analysis. As a consequence, low split sizes can cause runs to terminate before finishing the reachability analysis, because no worker processes are left to receive partitions. Especially analyses of large models might benefit from lower split sizes, because this could result in better load balancing. As seen in section 5.3 it can be profitable to not occupy the entire capacity of one machine but to distribute work among machines. Even if this means that more work needs to be done in total and more communication is necessary, memory access might be faster, which can decrease the total computation time. The implementation of this feature should be simple. One option is to merge new partitions and their corresponding split functions with a worker's current BDD partition and split function – provided that there is still enough memory available. After this, the worker could proceed as if he had received only one partition in total. Another – or additional – option is to increase the split size during runtime, when too many partitions are created.

As described in chapter 5 our design was able to achieve good results, even compared with existing approaches. The benchmarks show that vertical partitioning is suitable for application in a multi-core distributed environment.

6.1.2 How do different configurations regarding the partitioning policy affect the overall performance of the resulting system?

The execution of analyses using our design depends on two parameters, split size and split count. The former indicates when a worker will split its BDD (number of nodes), while the latter determines how many partitions will be created while splitting.

The analysis of our benchmark results clearly shows that the choice of appropriate split sizes and split counts is the challenge of our algorithm. When

taking the results of all analyzed models into account, no correlation between both variables and the total execution time could be observed. For some models either high or low split sizes resulted in better results, for others there was no significant difference at all. The same is true for the value for the split count. As described in section 6.1.1 executions with a small split size and a large split count at the same time could not always be finished due to the mentioned limitations of our implementation. The best choice of both variables is different for each model. We could not find a relation between the size of the model and both factors.

The total execution time may depend on the number of duplicate nodes after partitioning, which may correlate with the number of non-owned states that need to be exchanged afterwards. Also the size of partitions on different machines may vary considerably such that bad load balancing and idle times may cause longer execution times.

In our benchmarks we did not monitor any values during the analyses but only final results like total execution time, number of nodes and number of states. Because no definitive conclusions can be drawn about when to use which parameter values, more research on this would be valuable. This could include measurements of e.g. the total number of nodes in each iteration (the sum of all machines), the split, idle, and communication times as well as message sizes and the overall network traffic (more about recommendations in section 6.2). Based on this, split size and split count could (dynamically) be adjusted.

6.1.3 How does the proposed method scale with the size of the graph and the number of machines used?

To test the scalability of our design, we performed several analyses using models of different sizes. We repeated the analyses while changing the number of available machines and keeping the configurations regarding split count and split size the same.

In general, our results show that the number of involved machines alone has nearly no influence on the total computation time. Although many machines may be available, not all of them need to be utilized for a computation. Some of them may synchronize with others, but the worker processes on those machines may never receive a BDD. These worker processes stay idle. As discussed in section 5.3 only up to a second of the total execution time is caused by idle workers. E.g. when ten workers are active at the end of an analysis, it does not

matter for the execution time whether ten machines were available or twenty. Therefore, not the number of machines but the number of active workers needs to be observed.

For very small models we can observe from our benchmarks that it is not advisable to analyze these in a distributed environment. By "very small" we mean models that can be analyzed on a single machine within fractions of a second. The overhead generated by partitioning and synchronizing between machines is too large compared to the total execution time. Larger model, however, can benefit from partitioning.

As stated earlier, we cannot make general assumptions about the correlation between execution time and number of active workers. Some models could be analyzed faster when more partitions were created and hence more machines were utilized. For other models it was the other way round. Often, however, we could not observe any correlation between both factors.

Our design does not utilize the involved machine's capacities as well as DistBDD. Results of DistBDD often show good results on a single machine and a peek when using two machines. After that the execution times decrease as more machines are involved. Similar to DistBDD computations using our design often achieved the fastest results on a single worker machine. Analyses on two machines (plus one master machine) took longer in most of the cases, but on average the time difference is smaller as compared to DistBDD. This indicates that communication overhead is less in our design. When using more than two worker machines, there is no clear tendency, while this is the case for DistBDD. The performance of PartBDD depends highly on the defined split size and split count as mentioned earlier and due to the fact that a worker process could only receive one BDD during an execution, very large models (mostly Petri Nets) could not be analyzed. When this limitation was removed, a better utilization of hardware capacity could be achieved.

6.2 Future Work

Next to the improvements mentioned above there are a few aspects we would recommend for future research.

The first topic regards the splitting procedure. We did not monitor the split variables that were chosen with respect to the underlying variable ordering. It might be possible that the variable is likely to be in the "upper" half or $\frac{2}{3}$ of all

split variables, because variables from the other half or third would result in many duplicate nodes. If this is the case, the splitting time could be decreased by taking only a part of the set of variables into account when searching for a suitable split variable. Furthermore, our implementation parallelizes the computation of $\phi \wedge x$ and $\phi \wedge \neg x$ (where ϕ is a BDD and x is a split candidate), but it is not possible to do this also in parallel for different split variable x. This computation of partitions for multiple split variables at the same time was done in [8]. They managed to do this by distributing the work over multiple machines.

Another question which would be interesting to answer is, if it is possible to gain insights into the structure of a model's BDD and to automate the selection of split size and split count, based on the findings. For this, it would be useful to monitor some selected models in more detail and to compare them with each other with respect to e.g.:

- idle, splitting and communication times
- message size
- duplicate nodes and number of nodes after each iteration (per machine and/or totals)
- size of the transition relation

In our experiments some analyses needed many short iterations to finish, for others there were less iterations, but each one took longer. If might be beneficial to partition a BDD with grows fast earlier than one that grows slowly. If predictions about the growth of a BDD during reachability analysis could be made, this algorithm might compete more successfully with other approaches.

Although no best configuration of our program could be determined, which is suited for each model, our final recommendation is to use a split size of 500,000 and a split count of 2. Over all models, PartBDD achieved the best results with these values (see an overview of several configurations in Figure 6.1).

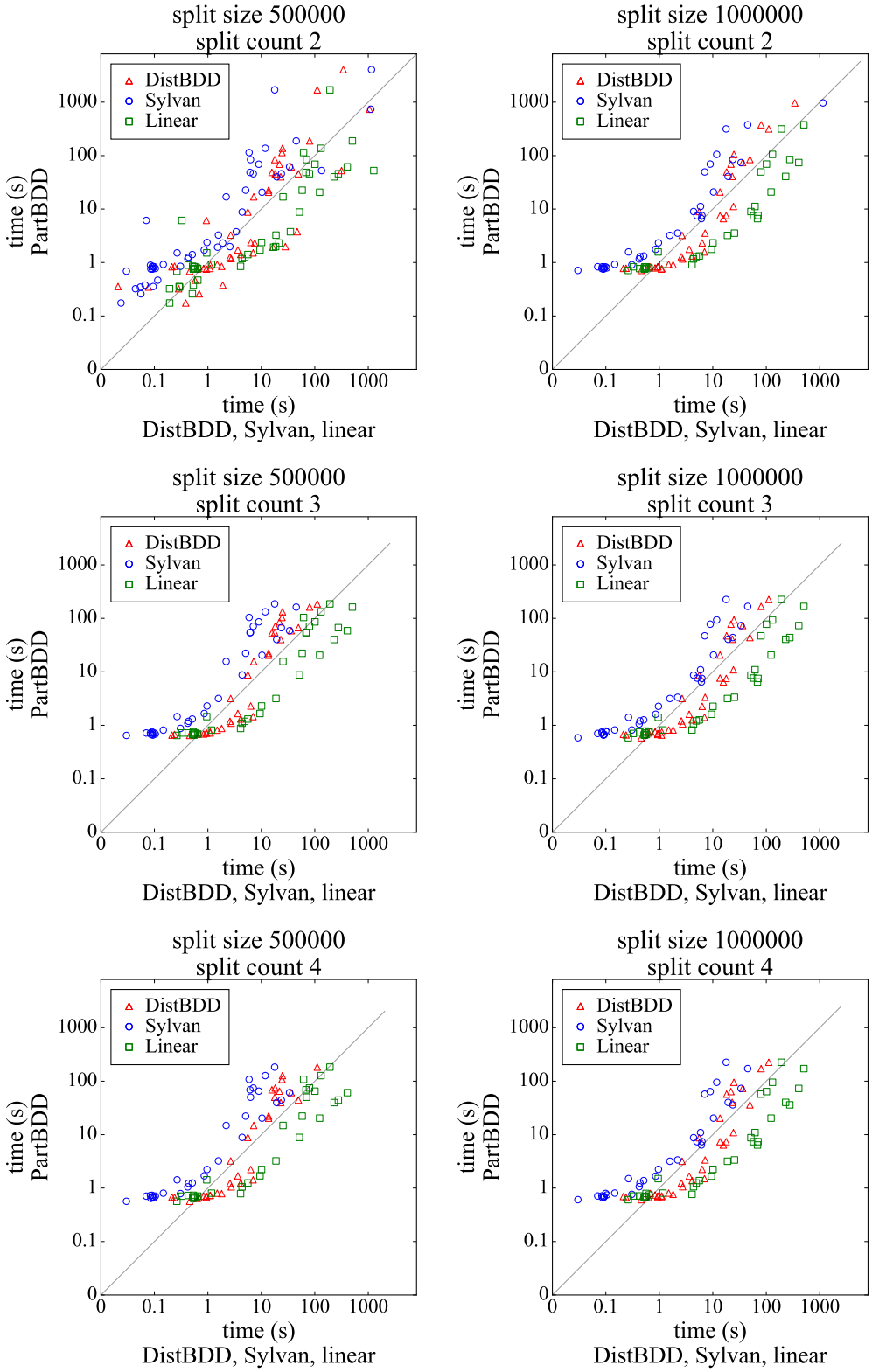


Figure 6.1: Execution times of PartBDD plotted against execution times of Sylvan, DistBDD and linear execution. Each plot shows the execution times of models with PartBDD for a specific split size and split count. The configurations of each program were as follows. **PartBDD**: 32 machines, 1 process per machine, 16 threads per process. **DistBDD**: 32 machines, 16 threads per machine. **Sylvan**: 1 machine with 16 threads. **Linear**: 1 machine with 1 thread. The best mean speedup was achieved with split size 500,000 and split count 2 (*speedup = time PartBDD/time DistBDD|Sylvan|Linear*).

Bibliography

[1] D. Lam and B. Cozzarin, "The joint strike fighter / f-35 program a canadian technology policy perspective," vol. 28, p. 45, 03 2014.

[2] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia, *Petri net analysis using boolean manipulation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 416–435. [Online]. Available: http://dx.doi.org/10.1007/3-540-58152-9_23

[3] C. Dong and P. Molitor, "What graphs can be efficiently represented by bdds?" in *Proceedings of the International Conference on Computing: Theory and Applications*, ser. ICCTA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 128–134. [Online]. Available: http://dx.doi.org/10.1109/ICCTA.2007.133

[4] T. Dijk and J. Pol, "Sylvan: Multi-core decision diagrams," in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 677–691. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-46681-0_60

[5] K. Milvang-Jensen and A. J. Hu, *BDDNOW: A Parallel BDD Package*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 501–507. [Online]. Available: http://dx.doi.org/10.1007/3-540-49519-3_32

[6] M. Chung and G. Ciardo, "Saturation NOW," in *1st International Conference on Quantitative Evaluation of Systems (QEST 2004), 27-30 September 2004, Enschede, The Netherlands*, 2004, pp. 272–281. [Online]. Available: http://dx.doi.org/10.1109/QEST.2004.1348041

[7] M.-Y. Chung and G. Ciardo, "A dynamic firing speculation to speedup distributed symbolic state-space generation," in *Proceedings 20th IEEE In-*

ternational Parallel Distributed Processing Symposium, April 2006, pp. 10 pp.–.

[8] T. Heyman, D. Geist, O. Grumberg, and A. Schuster, "Achieving Scalability in Parallel Reachability Analysis of Very Large Circuits," in *Proceedings of the 12th International Conference on Computer Aided Verification*, ser. CAV '00. London, UK, UK: Springer-Verlag, 2000, pp. 20–35. [Online]. Available: http://dl.acm.org/citation.cfm?id=647769.734107

[9] M.-Y. Chung and G. Ciardo, "A pattern recognition approach for speculative firing prediction in distributed saturation state-space generation," *Electron. Notes Theor. Comput. Sci.*, vol. 135, no. 2, pp. 65–80, Feb. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2005.10.019

[10] W. H. M. Oortwijn, T. van Dijk, and J. C. van de Pol, "A Distributed Hash Table for Shared Memory," in *Proceedings of the 11th International Conference on Parallel Processing and Applied Mathematics (PPAM 2016), Revised Selected Papers., Krakow, Poland*, ser. Lecture Notes in Computer Science, vol. 9574. London: Springer Verlag, September 2015, pp. 15–24.

[11] W. Oortwijn, T. v. Dijk, and J. v. d. Pol, "Distributed binary decision diagrams for symbolic reachability," in *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, ser. SPIN 2017. New York, NY, USA: ACM, 2017, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/3092282.3092284

[12] K. Andreev and H. Räcke, "Balanced Graph Partitioning," in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '04. New York, NY, USA: ACM, 2004, pp. 120–124. [Online]. Available: http://doi.acm.org/10.1145/1007912.1007931

[13] G. Ciardo, Y. Zhao, and X. Jin, "Parallel symbolic state-space exploration is difficult, but what is the alternative?" in *Proceedings 8th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2009, Eindhoven, The Netherlands, 4th November 2009.*, 2009, pp. 1–17. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.14.1

[14] S. Orzan, J. van de Pol, and M. Valero Espada, "A State Space Distribution Policy Based on Abstract Interpretation," *Electron. Notes Theor. Comput.*

Sci., vol. 128, no. 3, pp. 35–45, Apr. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2004.10.017

[15] G. Karypis and V. Kumar, "METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0," Tech. Rep., 1995.

[16] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," *CoRR*, vol. abs/1503.00626, 2015. [Online]. Available: http://arxiv.org/abs/1503.00626

[17] O. Grumberg, T. Heyman, and A. Schuster, "A Work-Efficient Distributed Algorithm for Reachability Analysis," in *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, 2003, pp. 54–66. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-45069-6_5

[18] InfiniBand Trade Association, "Architecture Specification, Release 1.3," 2015. [Online]. Available: http://www.infinibandta.org

[19] S. Hong, S. Depner, T. Manhardt, J. Van Der Lugt, M. Verstraaten, and H. Chafi, "PGX.D: A fast distributed graph processing engine," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York, NY, USA: ACM, 2015, pp. 58:1–58:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807620

[20] Y. Guo, A. L. Varbanescu, D. Epema, and A. Iosup, "Design and experimental evaluation of distributed heterogeneous graph-processing systems," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 203–212.

[21] C. P. Inggs and H. Barringer, "Effective state exploration for model checking on a shared memory architecture," *Electronic Notes in Theoretical Computer Science*, vol. 68, no. 4, pp. 605–620, 10 2002.

[22] A. Narayan, J. Jain, M. Fujita, and A. Sangiovanni-Vincentelli, "Partitioned ROBDDs – a Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions," in *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ser. ICCAD

'96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 547–554. [Online]. Available: http://dl.acm.org/citation.cfm?id=244522.244894

[23] S. Edelkamp, P. Kissmann, and Álvaro Torralba, "Lex-partitioning: A new option for BDD search," in *Proceedings First Workshop on GRAPH Inspection and Traversal Engineering, GRAPHITE 2012, Tallinn, Estonia, 1st April 2012.*, 2012, pp. 66–82. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.99.8

[24] J. V. Sanghavi, R. K. Ranjan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Binary decision diagrams on network of workstations," EECS Department, University of California, Berkeley, Tech. Rep. UCB/ERL M96/9, 1996. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1996/2969.html

[25] W. Townsend and M. Thornton, "Partial binary decision diagrams," in *SOUTHEASTERN SYMPOSIUM ON SYSTEM THEORY*, vol. 34, 2002, pp. 422–425.

[26] S. Iyer, D. Sahoo, C. Stangier, A. Narayan, and J. Jain, *Improved Symbolic Verification Using Partitioning Techniques*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 410–424. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-39724-3_35

[27] C. Y. Lee, "Representation of switching circuits by binary-decision programs," *Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959. [Online]. Available: http://dx.doi.org/10.1002/j.1538-7305.1959.tb01585.x

[28] C. E. Shannon, "The synthesis of two-terminal switching circuits," *The Bell System Technical Journal*, vol. 28, no. 1, pp. 59–98, Jan 1949.

[29] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986. [Online]. Available: http://dx.doi.org/10.1109/TC.1986.1676819

[30] R. R. Tucci, "Binary Decision Diagrams are a Subset of Bayesian Nets," Tech. Rep. quant-ph/0209009, Sep 2002. [Online]. Available: http://cds.cern.ch/record/578844

[31] H. R. Andersen, "An introduction to binary decision diagrams," *Lecture notes, available online, IT University of Copenhagen*, 1997.

[32] C. Baier, B. R. Haverkort, H. Hermanns, J. P. Katoen, and M. Siegle, *Validation of Stochastic Systems: A Guide to Current Research*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004. [Online]. Available: https://books.google.nl/books?id=2xmiaQt_POoC

[33] F. Somenzi, "Binary decision diagrams," in *Calculational System Design, volume 173 of NATO Science Series F: Computer and Systems Sciences*. IOS Press, 1999, pp. 303–366.

[34] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster, *Achieving Speedups in Distributed Symbolic Reachability Analysis Through Asynchronous Computation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 129–145. [Online]. Available: http://dx.doi.org/10.1007/11560548_12

[35] "Open MPI: Open Source High Performance Computing," 2017. [Online]. Available: https://www.open-mpi.org/

[36] "The Message Passing Interface (MPI) standard." [Online]. Available: http://www.mcs.anl.gov/research/projects/mpi/

[37] R. Pelánek, "Beem: Benchmarks for explicit model checkers," *Model Checking Software*, pp. 263–267, 2007.

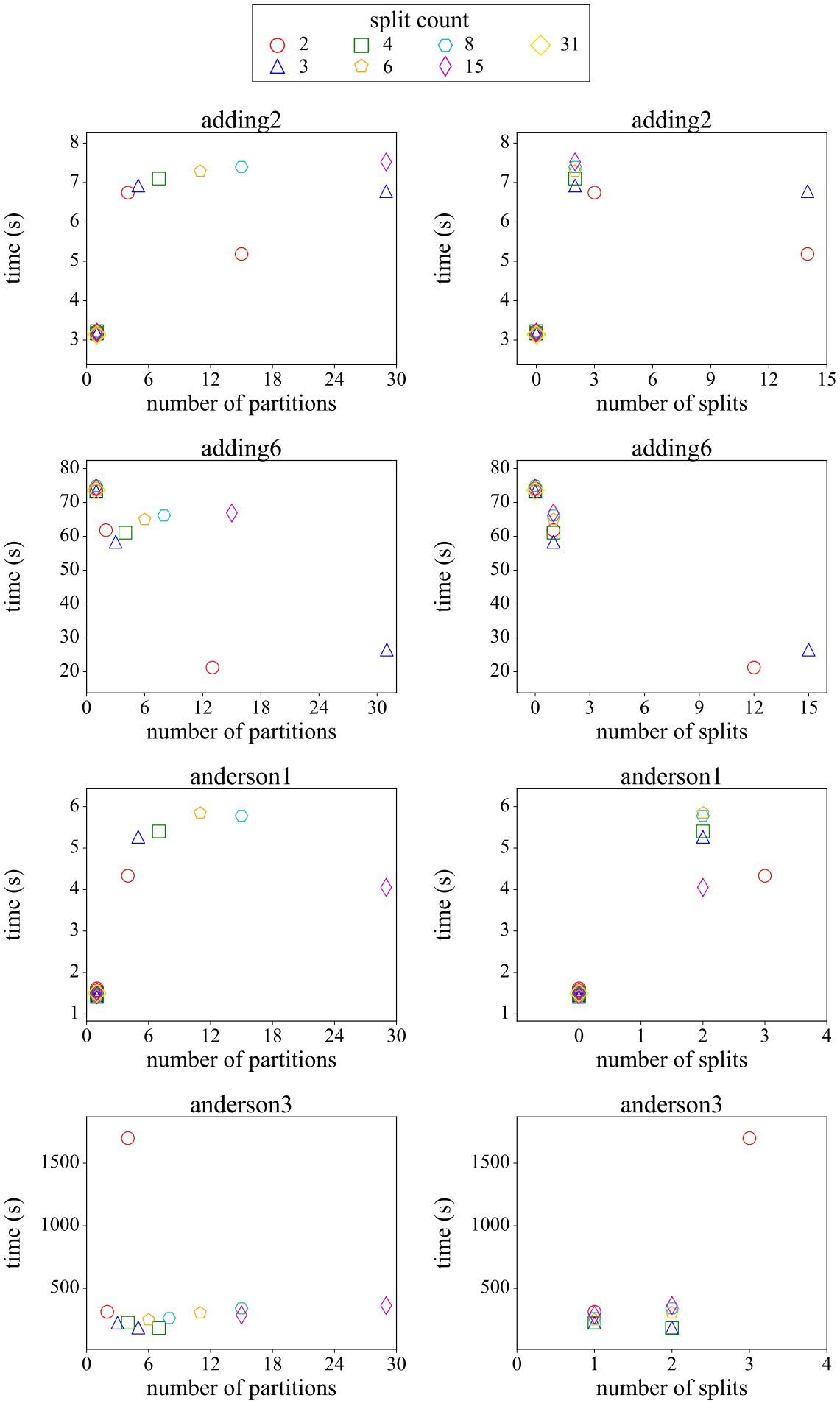
[38] "Model Checking Contest @ Petri Nets 2015," https://mcc.lip6.fr/2015/models.php, 2015. [Online]. Available: https://mcc.lip6.fr/2015/models.php

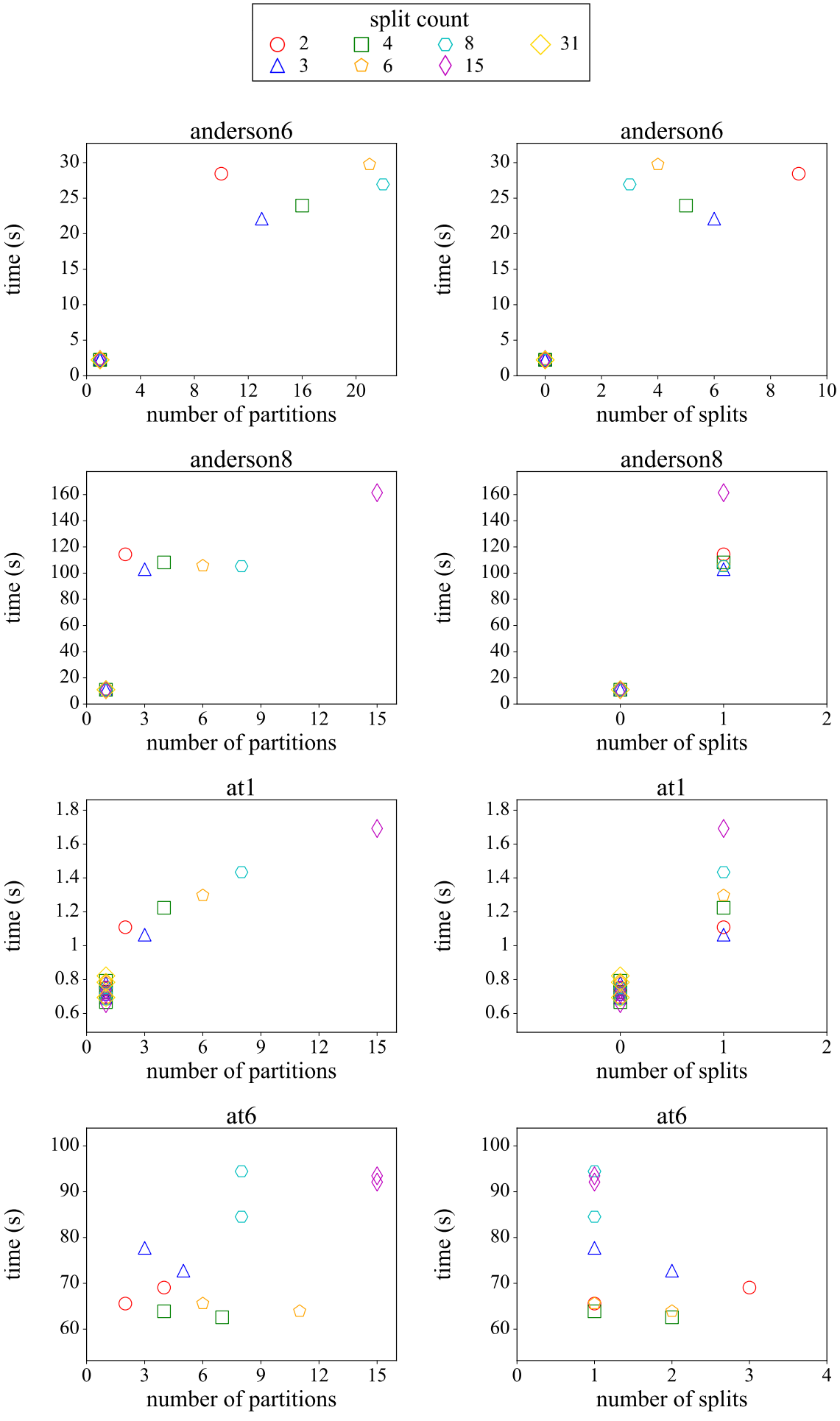
[39] "The Distributed ASCI Supercomputer 5," 2015. [Online]. Available: http://www.cs.vu.nl/das5

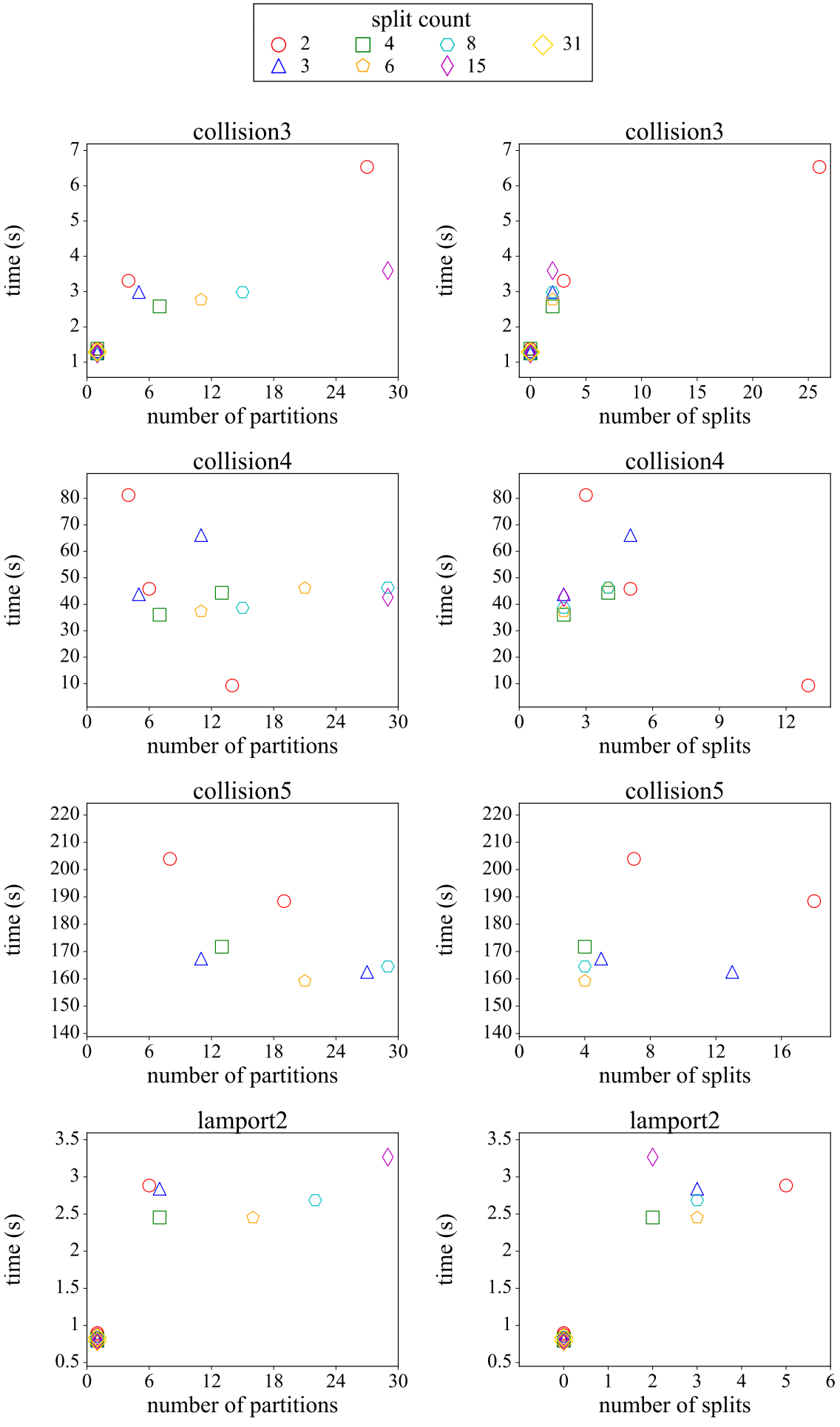
[40] T. van Dijk, A. Laarman, and J. van de Pol, *Multi-Core BDD Operations for Symbolic Reachability*, ser. Electronic Notes in Theoretical Computer Science. ELSEVIER, 9 2012, pp. 127–143.

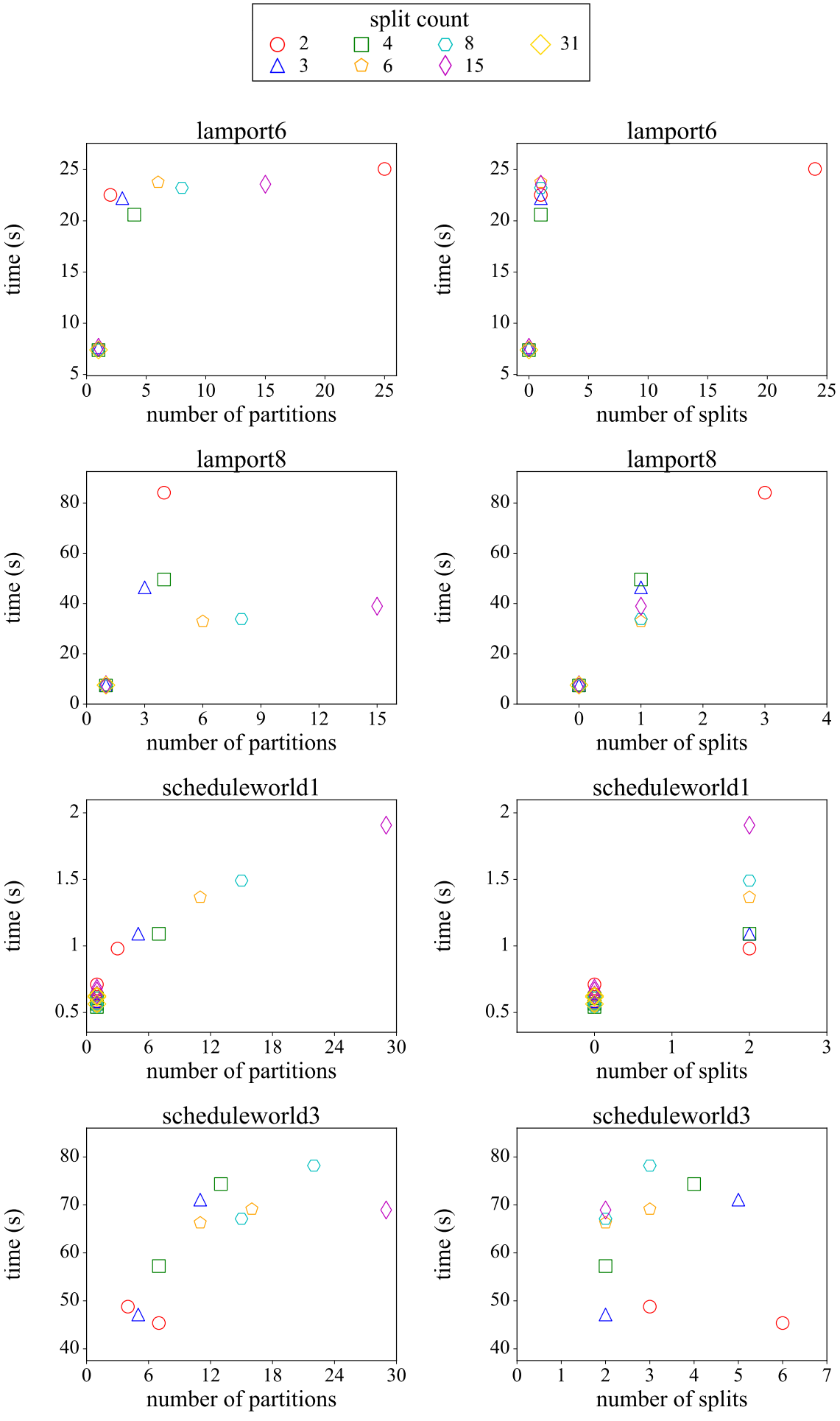
Appendix A

Split Size and Split Count


split count
2
3
4
6
8
15
31
adding2
time (s)
number of partitions
adding2
time (s)
number of splits
adding6
time (s)
number of partitions
adding6
time (s)
number of splits
anderson1
time (s)
number of partitions
anderson1
time (s)
number of splits
anderson3
time (s)
number of partitions
anderson3
time (s)
number of splits


split count
2
3
4
6
8
15
31
anderson6
time (s)
number of partitions
anderson6
time (s)
number of splits
anderson8
time (s)
number of partitions
anderson8
time (s)
number of splits
at1
time (s)
number of partitions
at1
time (s)
number of splits
at6
time (s)
number of partitions
at6
time (s)
number of splits


split count
2
3
4
6
8
15
31
collision3
time (s)
number of partitions
collision3
time (s)
number of splits
collision4
time (s)
number of partitions
collision4
time (s)
number of splits
collision5
time (s)
number of partitions
collision5
time (s)
number of splits
lamport2
time (s)
number of partitions
lamport2
time (s)
number of splits


split count
2
3
4
6
8
15
31
lamport6
time (s)
number of partitions
lamport6
time (s)
number of splits
lamport8
time (s)
number of partitions
lamport8
time (s)
number of splits
scheduleworld1
time (s)
number of partitions
scheduleworld1
time (s)
number of splits
scheduleworld3
time (s)
number of partitions
scheduleworld3
time (s)
number of splits