Distributed Symbolic Reachability Analysis

Master Thesis

Author:
Wytse Oortwijn (s1247379)

Graduation Committee:
Prof. Dr. J.C. van de Pol
MSc. T. van Dijk
Dr. S.C.C. Blom

UNIVERSITY OF TWENTE.

Formal Methods and Tools (FMT)
University of Twente
July 31, 2015

Abstract

Model checking is an important tool in program verification and software validation. Model checkers generally examine the entire state space of a model to find behaviour that differs from a given formal specification. Most temporal safety properties can be verified via *reachability analysis*. A major limitation is the *state space explosion* problem, which occurs when the state space does not fit into the available memory.

State spaces can efficiently be stored as *Binary Decision Diagrams* (BDDs), a data structure used to efficiently represent sets. The BDD representations of state spaces can then be manipulated via BDD operations, which enables *symbolic reachability analysis*. Moreover, more hardware resources can be employed. By increasing processing units and available memory, larger state spaces can be stored and processed in less time.

This thesis focuses on symbolic reachability analysis, in addition to adding hardware resources by using a network of workstations. Our research goal is to implement distributed BDD operations that scale *efficiently* along all processing units and memory connected via a high-performance network.

Compared to existing work on distributed symbolic reachability, we use *modern* techniques and components, like Infiniband and RDMA, to reduce network latency. Furthermore, we use hierarchical private deque work stealing to implement efficient load-balancing. Finally, we attempt to use the idle-times of workers as efficient as possible by overlapping RDMA roundtrips as much as possible.

We designed an RDMA-based distributed hash table and private deque work stealing algorithms. Both components are micro-benchmarked in detail and used to implement the actual BDD operations for distributed symbolic reachability. The algorithms are evaluated by performing symbolic reachability over a number of BEEM models.

Instead of network latency, the limited throughput of the RDMA devices appears to be the performance bottleneck. As a result, we achieve good scalability along the number of machines added to the network, but scalability along the number of processes per machine remains limited. Compared to sequential runs, speedups up to 28 are observed. With respect to parallel symbolic reachability, by paying an *acceptable* performance price, the combined memory of a network of workstations can be used efficiently, and adding workstations does not decrease performance.

Contents

Chapter 1

Introduction

An important tool in program verification and software validation is *model checking*. Given a formal specification, model checking algorithms analyse program behaviour by exhaustively searching all reachable program states, which constitute a *state space graph*. State spaces often grow exponentially with the size of the program model, which may lead to well-known *state space explosions*. These exponential blow-ups occur when program states do not longer fit into the available memory. State space explosions may arise very quickly in practice, making them a major limitation in modern software verification.

Several attempts have been made to attack state space explosions. Those attempts include *reduction* techniques like Partial Order Reduction [57] and Bisimulation Minimization [30]. In addition, state spaces can be *compressed*, so that more states fit into the available memory. Binary Decision Diagrams (BDDs) can be used for such compression, as BDDs can efficiently represent sets. Moreover, state spaces can efficiently be manipulated by manipulating their BDD representatives, which enables *symbolic model checking*. BDDs can significantly improve the space complexity of state spaces compared to explicit representations [32], which motivates symbolic model checking. Alternative effective compression techniques include SAT-based approaches [11], inductive verification (e.g. IC3) [20], and other variants on decision diagrams (e.g. MDDs, LDDs, and ZDDs) [26, 31, 75], but this thesis focusses only on BDDs.

In addition to techniques for reduction and compression, more hardware resources can be employed [34, 51, 79, 80]. By increasing processing units and available memory, larger state spaces can be stored and processed in less time. Therefore, much research has been devoted to *parallel* algorithms for both symbolic and explicit model checking [41, 44, 75, 79, 80]. As a result, good speedups are obtained on many-core clusters. The parallel BDD package Sylvan [75], for example, reaches impressive speedups up to 38 with 48 cores.

Apart from parallel implementations, also *distributed* symbolic verification algorithms have been proposed [25, 26, 48, 56, 58, 68] (see Section 5.1 for more detail). Most approaches were motivated by the expensiveness and limitations of hardware scalability of many-core machines. Compared to sequential runs, no speedups were obtained by most implementations, but very large state spaces could be stored due to the large amount of available memory. Speedups remained low, since BDD operations perform only small amounts of computation per memory access. Distributed implementations thus require many remote memory accesses, which easily makes network latency a performance bottleneck.

Nowadays, high-performance networking components like Infiniband [1] are available. Their features include *Remote Direct Memory Access* (RDMA), which allows machines to access remote memory without invoking the CPUs on the targeted machines. Those operations are handled by

the RDMA devices instead, allowing CPUs to continue with other computational tasks. RDMA supports *zero-copy* data transfers, which means that no memory copies are performed when sending or receiving packets, unlike traditional protocols such as TCP [65]. Furthermore, RDMA also supports *kernel bypassing*, which avoids communication with the operating system kernel before handling packets. As a result, one-sided RDMA operations can be performed within $3\mu s$ on Infiniband hardware [3], compared to $60\mu s$ with TCP on Ethernet hardware [24]. The low latencies of RDMA and its CPU efficiency justify a renewed attempt in distributed symbolic verification.

Moreover, at the time of writing, Infiniband hardware is comparable in price to their Ethernet counterparts, which makes scaling along high-performance hardware just as expensive as scaling along standard Ethernet hardware. Some recent research, for example the RDMA-based key-value store Pilaf [24] (2014), was motivated by this cheap scalability. In addition, high-performance networks support an almost unlimited number of processing units and memory, in contrast to individual multi-core machines. Combining these reasons with the cheap scalability and low latencies of modern networking hardware motivates our research, and may allow more efficient distributed model checking.

1.1 Research Goals

In this thesis we present distributed algorithms for reachability analysis, based on BDDs. The algorithms are specialized for high-performance networks, like Infiniband, by making use of RDMA. To the best of our knowledge, no other attempts have been made to distribute symbolic verification with an RDMA-based approach. By experimental evaluation and reflection on the proposed algorithms, we answer the following main research question:

> **Research question (RQ):** *How efficient can RDMA-based distributed implementations of BDD operations scale along all processing units and available memory connected via a high-performance network?*

To determine *how* RDMA can most efficiently be used by BDD operations, we focused on three important design considerations, given in [32], for efficient distributed symbolic state-space generation. These considerations are: data distribution, load-balancing maintenance, and reduction of communication overhead and latency. Also exploiting data-locality is an important consideration, as part of data distribution [26]. To answer the main research question, we focused on all four considerations, resulting in several subquestions:

> **Subquestion 1 (SQ1):** *How can the storage and retrieval of data efficiently be managed to minimize their latencies?*

> **Subquestion 2 (SQ2):** *How can the total computational work be divided and distributed to maximize scalability along processors over a high-performance network?*

> **Subquestion 3 (SQ3):** *How can the idle-times of processes be minimized while performing network communication?*

Data distribution is covered in **SQ1**, load-balancing maintenance is covered by **SQ2**, and minimizing communication overhead is covered by **SQ3**.

1.2 Methods and Contributions

This section discusses the research methods used to answer the subquestions. In addition, our contributions and findings are given.

1.2.1 Data Distribution

To answer **SQ1**, we investigated existing high-performance distributed storage techniques, including hash tables and key-value stores. Existing work includes Pilaf [24], Nessie [67], HERD [10], and FaRM [5], which all use RDMA to reduce latency. With the exception of HERD, all these implementations either use Cuckoo hashing [52] or Hopscotch hashing [46] to resolve hash collisions. HERD only uses one-sided writes to minimize the required number of roundtrips, at the cost of CPU-efficiency, which we would like to retain. Cuckoo hashing uses $k \geq 2$ hash functions, which means k roundtrips when accessing remote buckets. Hopscotch hashing uses neighbourhoods, which can be retrieved with a single roundtrip, making Hopscotch arguably more efficient than Cuckoo. However, both hashing schemes require expensive relocation procedures when data can not be inserted. Those procedures may require many extra roundtrips and expensive distributed locking mechanisms. We argue that linear probing is more efficient, since it does not use such relocation procedures.

In Chapter 3 we present an RDMA-based distributed hash table that uses linear probing to reduce the number of roundtrips and thus reduces latency. The hash table supports one operation, namely `find-or-put`, that either adds data or gives a positive result. To minimize waiting-times, `find-or-put` overlaps roundtrips as much as possible, which further reduces latency. Because linear probing examines buckets that are consecutive in memory, a range of buckets, which we call *chunks*, can be obtained with a single roundtrip. By increasing chunk sizes, higher load-factors are supported and require fewer roundtrips, at the cost of slightly higher latencies.

A peak-throughput of 3.6×10^6 op/s is reached on an Infiniband network with 10 machines. On average, `find-or-put` finds intended buckets within $9.3\mu s$ with a load-factor of 0.9 when targeting remote memory. On the other hand, by adding processes, the throughput of the RDMA devices get saturated, which significantly affects the scalability of processes when also scaling along machines. Solving this limitation would require improving data-locality, as well as reducing the required number of roundtrips.

1.2.2 Load-Balancing Maintenance

Graph algorithms can effectively be parallelized by using *fine-grained task parallelism* [8], which includes operations on BDDs. Due to the success of the parallel symbolic BDD package Sylvan [75] by applying dynamic load-balancing via work stealing, we investigated hierarchical work-stealing approaches to answer **SQ2**. Existing approaches include Scioto [35] and HotSLAW [62], which both use split deques for storing tasks. Both approaches require expensive locking mechanisms when performing steals. Instead, we focus on private deque work stealing, similar to [50], due to the absence of locking and the decrease in roundtrips compared to split deques.

Min et al. [62] propose *hierarchical victim selection* to exploit locality, thereby reducing the average number of roundtrips. Also *hierarchical chunk selection* is proposed to adjust the number of tasks to steal, based on communication costs. Another technique that may increase performance is *leapfrogging* [77], that allows victims to steal back from their thiefs, thereby helping thiefs to complete parts of their original work.

In Chapter 4 we present lock-less hierarchical private-deque work stealing operations that steal more efficiently than current state-of-the-art implementations. Our implementation uses

the ideas of [50], but requires fewer roundtrips and supports hierarchical victim selection and leapfrogging. Due to time constraints, we were not able to also implement hierarchical chunk selection. Relative to HotSLAW, speedups up to 34.3 are reached for local steals and 9.3 for remote steals. On the other hand, HotSLAW generates less overhead when executing tasks. This is because HotSLAW supports tasks with variable-sized inputs and outputs, whereas our implementation does not. We solved this by using extra shared data structures, which are slightly less efficient than HotSLAW. Our implementation can likewise be optimized, but due to time constraints we were not able to do so. Implementing hierarchical chunk selection and reducing overhead would allow hierarchical work-stealing that is *more* efficient than the current state-of-the-art.

1.2.3 Efficient Communication

To answer **SQ3**, we investigated ways to most efficiently use RDMA in the various components of our BDD implementation. To minimize communication overhead we use *one-sided* RDMA, so that the CPUs on the targeted machines are not invoked. Wherever possible, we employ latency hiding via pre-fetching by overlapping roundtrips as much as possible. In the hash table design, multiple chunks are queried simultaneously, thereby reducing waiting times. Furthermore, hierarchical work stealing is used to exploit locality, reducing the overall idle-times of processes.

Chapter 5 presents the designs of our distributed BDD operations. We give asynchronous operations for consulting the shared memoization cache, enabling cache requests to be overlapped with other computational work. The BDD operations are implemented with UPC [27] and evaluated by performing reachability analysis over various BEEM models [53]. Compared to sequential runs, speedups are obtained by using a network of computers. Furthermore, scaling along computers does not decrease performance, and allows more memory to be used. Relative to two computers, each using 8 processes, speedups up to 5.18 are reached by scaling to 10 computers.

We observed that, rather than network latency, the main bottleneck was caused by the limited throughput of RDMA devices. To increase scalability, it is *key* to minimize the number of RDMA operations, as a *first* priority. We mainly focused on reducing waiting-times and minimizing communication overhead by employing RDMA, but scalability can still be improved. HERD, for example, is a distributed hash table that only uses one-sided RDMA writes as a message passing alternative, thereby achieving a minimal number of roundtrips, at the cost of CPU efficiency. By employing this technique, communication overhead is increased, but fewer RDMA roundtrips are generated, thereby allowing more processes to be used before the RDMA devices get saturated, which in turn increases both scalability and maximal achievable throughput.

1.3 Structure

Chapter 2 provides details on Infiniband network and gives various programming models usable in combination with RDMA. We chose to use a PGAS abstraction, whose operations are also explained. Chapter 3 presents our distributed hash table design that uses linear probing. Evaluational results of the latency and throughput of `find-or-put` are also given. Chapter 4 discusses work stealing techniques and present a hierarchical private-deque work stealing design. The implementation is evaluated by running several benchmarks and comparing the results with HotSLAW. Chapter 5 presents the design of our distributed BDD operations. The operations are evaluated by performing reachability analysis on several BEEM models. Finally, Chapter 6 summarizes our conclusions.

Chapter 2

Preliminaries on Partitioned Global Address Space Programming

2.1 Introduction

In this chapter, the Infiniband network architecture is discussed (Section 2.2). In particular, the Remote Direct Memory Access feature of Infiniband is discussed (Section 2.3), together with various programming models that support RDMA (Section 2.4). In our research, we use the Partitioned Global Address Space programming model, whose operations are discussed in Section 2.5. Finally, the performance of various RDMA operations are determined via micro-benchmarking (Section 2.6).

2.2 The Infiniband Architecture

Infiniband [1] is specialized hardware used to construct high-performance networks. Infiniband is manufactured by Microsoft and Mellanox. The specialized hardware includes switches, Network Interface Cards (NICs) [54], and interconnects that support bandwidths up to 300 GB/s. The interconnects use a bidirectional serial bus to keep communication costs low. Experimental evaluation of the algorithms proposed in this thesis are performed on a network of 10 machines, connected via a 20 GB/s Infiniband network.

At the time of writing, the prices of Infiniband hardware are comparable to their Ethernet counterparts [24]. As an effect, scaling along high-performance hardware is just as expensive as scaling along traditional Ethernet hardware. This motivated some recent research in the use of RDMA in non-performance computing [24].

In Infiniband terminology, the NICs are called *Channel Adapters (CAs)* and devices connected to an Infiniband network are called *nodes*. Infiniband uses a queue-based model. Communication between two nodes requires the use of a *Queuing Pair*, consisting of a *Send Queue* and a *Receive Queue*. Both queues reside in the memory of the Channel Adapters, instead of main-memory, allowing incoming messages to be handled by the CAs themselves, thereby bypassing the CPUs. Therefore, this queue-based model allows the use of *RDMA*, which is explained in the following section.

Operation	UD	UC	RC	RD
Send/Receive	X	X	X	X
RDMA write		X	X	
RDMA read			X	
Atomic operations			X	

Table 2.1: The different operations supported by the various service types used by Infiniband.

As an example, suppose that two nodes A and B need to communicate over an Infiniband network. Both nodes create a Queuing Pair, which they connect, so that messages placed in the Send Queue of A are transmitted to the Receive Queue of B, and vice versa. The Channel Adapters of A and B thus *forward* messages placed in the Send Queue and *process* messages placed in the Received Queue. In addition to a Queuing Pair, a *Completion Queue* can be used. After having processed a message from the Receive Queue, it can be pushed to the Completion Queue, thereby allowing the CPUs to check the status of messages.

2.3 Remote Direct Memory Access

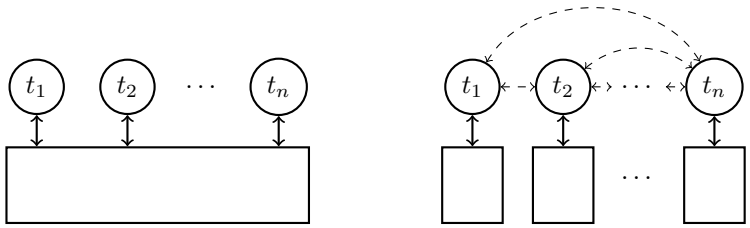
One of the features supported by Infiniband networks is *Remote Direct Memory Access (RDMA)*, which allows nodes to directly access the memory of other nodes, without necessarily invoking their CPUs. RDMA operations are handled by the Channel Adapters instead, which are able to access the main-memory of their host machine via the PCI-E bus. Using RDMA further reduces network latency, since RDMA supports *zero-copy* data transfers and *kernel bypassing*.

Traditional protocols, like TCP, require multiple memcopies when transferring data [65]. To illustrate, TCP packets are first copied from application memory to kernel memory, then send over the network, and finally copied from kernel memory to the application memory of the node that received the packet. As an effect, several context switches are performed. In RDMA, the network stack is completely *bypassed*, as messages are directly pushed on a Send Queue on the Channel Adapter. Furthermore, the Queuing Pairs reside in the memory of the Channel Adapters themselves, which prevents memcopies. As a result, RDMA operations can be performed within $3\mu s$ on Infiniband hardware [3], compared to $60\mu s$ with TCP on Ethernet hardware [24].

Sharing Memory. To support RDMA operations on remote memory, the Channel Adapters need to be aware of the blocks of memory exposed via RDMA. Therefore, a blocks of memory can be registered at the Channel Adapters as being *shared*. In Infiniband terminology, a consecutive block of shared memory is called a *Memory Region*. Every memory region can be accessed via local and remote addresses. For every Memory Region, the Channel Adapters hold a lookup table, so that remote addresses can be translated to local addresses. Prior to using RDMA, the remote addresses of the Memory Regions can be transmitted via MPI. In addition, permissions to RDMA-exposed memory can be specified via so-called *Protection Domains*.

Service Types. Various *service types* can be associated to Queuing Pairs, which determine how nodes respond to incoming messages. Figure 2.1 shows an overview of different service types, and the operations supported by them. These operations are discussed in the following Section.

The *Reliable Connection (RC)* service type guarantees a reliable data transfer between the two nodes, which means that data is guaranteed to arrive. After receiving a message, Ack and Nak packets are send, similar to traditional TCP. Packets are delivered in order and erroneous messages are automatically retransmitted. The *Unreliable Connection (UC)* specifies



(a) Shared Memory (b) Distributed Memory

Figure 2.1: Here n threads $t_1, \ldots, t_n$ are displayed in a *shared memory* and a *distributed memory* setting. Dashed edges represent communication links between threads.

a connection between two nodes in which data is not guaranteed to arrive, because Ack and Nak packets are not used. Compared to RC, less network traffic is generated, thereby allowing higher throughputs. In the *Unreliable Datagram (UD)* service type, no responses are expected, similar to traditional UDP. Therefore, RDMA is not supported. However, Grant et al. [61] propose support for RDMA over connectionless protocols, including UD, to improve performance even further. Finally, the *Reliable Datagram (RD)* service type can be used, which is similar to UD, but a reliable connection is guaranteed, since Ack and Nak packets are used upon message retrieval.

In our research, we use RC, since it allows a reliable connection, and therefore contributes to the correctness of the algorithms. However, some operations might be performed under unreliable service types, thereby increasing throughput. As future work, it would for example be interesting to use UD, together with [61], to further increase performance.

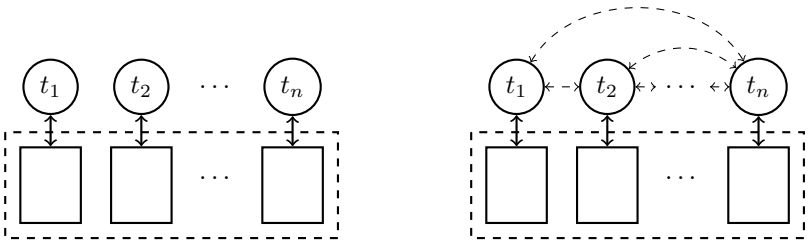
2.3.1 RDMA Operations

Infiniband supports various types of operations. Firstly, standard message passing is supported via `send` and `receive` operations. The `send` operation puts a message on the Receive Queue of the targeted node and `receive` polls the Receive Queue for incoming messages.

One-sided RDMA operations only use Send Queues and ignore Receive Queues. When a Channel Adapter receives a one-sided RDMA message, it directly executes its operation on the given location in main-memory. By not using the Received Queue, the CPUs of the targeted nodes are never invoked and are not aware of the one-sided operations executed by the Channel Adapters. One-sided `write` operations do not generate a response. One-sided `read` operations, on the other hand, require the targeted node to give a response, which is written to the Completion Queue of the node that initiated the operation.

Two-sided RDMA operations also use the Receive Queue, in addition to the Send Queues. The CPUs of the targeted nodes need to pop messages from the Receive Queue to process them. Two-sided operations can, for example, be used to support memory operations that are too complex for one-sided operations. Although the CPUs of both machines are involved, performance is still high, since the network stack is still bypassed and zero-copy data transfers are still used.

Also various *atomic* operations are supported, which are explained in Section 2.5.2.



(a) PGAS (b) Hybrid PGAS+MPI

Figure 2.2: Here n threads $t_1, \ldots, t_n$ are displayed in a *PGAS* setting. Every thread can access the entire combined memory, of which only a small portion might be local. The global address spaces are represented by dashed rectangles and the local address spaces by solid rectangles.

2.4 Parallel Programming Models

Several parallel programming models exist that expose memory in various ways. Figure 2.1 schematically shows the shared memory and distributed memory models.

Shared Memory. In a *Shared Memory Architecture* (Figure 2.1a), all threads share a single address space. Communication between threads is done implicitly, as threads may read changes made to the shared memory. This model simplifies programming, as the entire available memory is accessible via a shared address space. On the other hand, data locality is not taken into account, since threads are generally not aware where data resides.

An example of a shared memory architecture is *Non-Uniform Memory Access (NUMA)*, a model often used in computer systems having multiple CPUs. Every CPU has a local memory and is able to access the memories that are local to other CPUs via a shared address space. Different parts of the address space might use different buses, which causes non-uniform accesses.

Another example is the *Symmetric Multiprocessing (SMP)* architecture, in which all CPUs access memory via the same shared memory bus. This causes memory-intensive applications to generally perform better on NUMA architectures, as the shared bus can become a performance bottleneck.

Distributed Memory. In a *Distributed Memory Architecture* (Figure 2.1b) every thread has a local memory and threads are *not* able to access the memory local to another thread. Instead, threads have to perform *message passing*, represented by the dashed edges in Figure 2.1b. Distributed memory architectures are often used to create distributed applications, i.e. programs that run on a network or cluster of computers. Message passing is then performed over the network. A popular interface for message passing is the *Message Passing Interface (MPI)* [78]. An advantage of distributed memory is that data locality is fully exploited, since processes only access local memory. Also, programs written with MPI can easily be used on a scalable network of machines.

Partitioned Global Address Spaces. Figure 2.2 schematically shows the *Partitioned Global Address Space (PGAS)* model, which is a combination of the shared and distributed memory models. Every thread has a local memory, which are also combined into a global address space, accessible by every thread. In contrast to shared memory architectures, threads in the PGAS model know what parts of the total available memory are local for them, allowing them to exploit

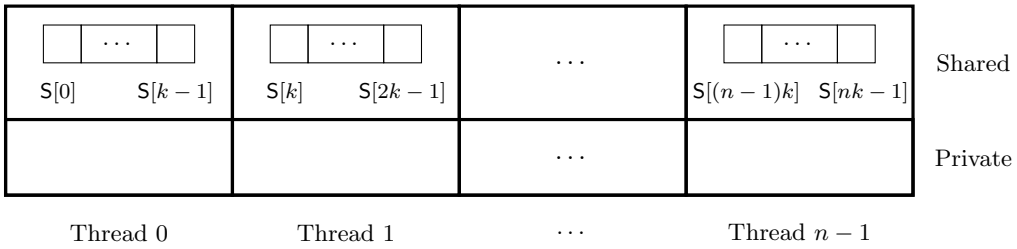


Figure 2.3: The memory layout of a PGAS setting, with a *shared* array $\mathsf{S}[0], \ldots, \mathsf{S}[nk-1]$, with n the number of threads and k the number of entries per thread.

data locality.

The PGAS model can be combined with MPI (Figure 2.2b), allowing them to perform message passing in addition to accessing the global address space. Similar to distributed memory architectures, the PGAS model allows easy scalability along a network of computers.

If PGAS is used in a distributed environment, it can be combined with RDMA, so that remote memory accesses are performed via one-sided RDMA operations. All local memories are then registered as Memory Regions at the Channel Adapters, so that the local memories receive remote addresses. These are used to constitute a global shared address space, making PGAS a high-performance parallel and distributed programming platform.

2.5 PGAS Implementations

Various PGAS implementations exist, including Unified Berkeley C (UPC) [27], Co-array Fortran (CAF) [49], Titanium [6], X10 [55], Chapel [19], and Open Shared Memory (OpenSHMEM) [14].

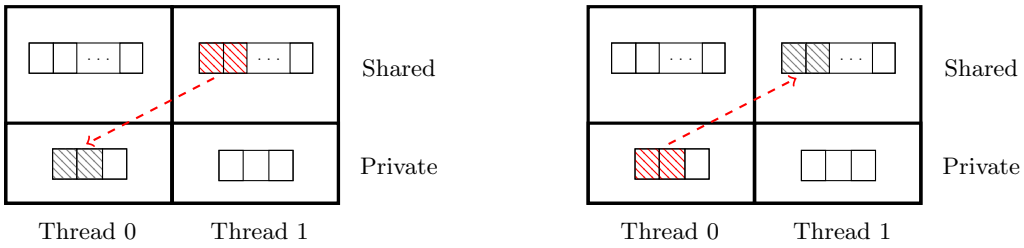
The languages UPC, CAF, and Titanium are extensions to C, Fortran, and Java, respectively. Chapel and X10 are both programming languages on themselves, which provide direct support for PGAS, task management, and dynamic task load-balancing. A disadvantage is that their structures differ from C, so all existing BDD operations written in C have to be rewritten entirely when using one of these languages. Also interoperability with existing tools, like LTSmin [64], could be problematic.

In our research, we chose to use Berkeley UPC, since it extends on C and supports asynchronous memory operations, where OpenSHMEM does not.

2.5.1 Memory Layout

Figure 2.3 shows the memory layout of a PGAS setting. Suppose that n threads are participating. Then every thread can have a portion of *shared* memory, i.e. memory registered as a Memory Region at the Channel Adapter. We denote all other memory as *private*. Threads thus only have access to their private memory and memory that is shared by other threads.

We assume a unified memory model, so that all threads allocate the same amount of resources, which simplifies programming. A *shared* array S is shown in Figure 2.3, which is partitioned over the n threads, so that each thread owns k entries of S. We refer to k as the *block size* of the shared array S. Therefore, thread i owns the region $\mathsf{S}[ik], \ldots, \mathsf{S}[(i+1)k-1]$.



(a) Schematic view of `memget`

(b) Schematic view of `memput`

Figure 2.4: Schematic overview of the `memget` and `memput` operations. In both cases, thread 0 accesses two shared elements owned by thread 1.

2.5.2 Memory Operations

We now discuss a number of operations supported by most PGAS implementations, starting with `memget` and `memput`. Figure 2.4 gives a schematic overview of these two operation. The `memget` operation transfers a block of *shared* memory into an equally-sized block of *private* memory (Figure 2.4a). Similarly, the `memput` operation transfers a block of *private* memory into an equally-sized block of *shared* memory (Figure 2.4b). Both operations are *blocking* operations, thereby preventing further execution of the thread until the operation (i.e. the roundtrip over the Infiniband network) has been completed.

Most PGAS implementations provide *asynchronous* versions of `memget` and `memput`, which we denote by `memget-async` and `memput-async`. Both functions return a *handle*, which can be used to *synchronize* on the roundtrip. Synchronization is performed with the `sync` operation, that takes a handle as parameter and blocks further execution of the thread until the corresponding asynchronous operation has been completed. Every asynchronous memory operation *must* be matched by a `sync`, otherwise handle leaks may occur, as the maximum number of handles that can be simultaneously opened is 65.536.

Finally, several *atomic* operations are supported, including `cas` and `fetch-and-add`. The `cas`(b, c, v) operation compares the content of a shared memory location b with a condition c. Only if the content of b matches c, the value v is written to b. After that, the *former* value of b is returned. The `fetch-and-add`(b, v) operations reads a shared memory location b and atomically adds v to its content. The *former* content of b is returned. Similar to `fetch-and-add`, most other bitwise operations are also supported, including `fetch-and-and`, `fetch-and-or`, `fetch-and-xor`, etcetera.

2.5.3 Challenges when using UPC

We encountered problems when scheduling the benchmarks on the cluster. The UPC runtime creates a number of UPC threads, distributed over a number of processes and machines via a scheduler. By default, MPI is used by the scheduler to prepare all participating machines and to set-up the UPC environment. In our case, the scheduler has no control over the actual process layout (which was also shown with a warning, given by the UPC runtime). As an effect, the scheduler could only handle a set-up when the number of machines and processes were powers of 2 (otherwise the scheduler could not set up the UPC environment at all). We solved this by using a SSH-based scheduler instead. Furthermore, we used `sshpass` to perform non-interactive authentication.

We also encountered problems when allocating large blocks of memory. It appears that, by default, Berkeley UPC does not support block sizes of more than 2^{20} bytes. By default, 64-bit integers are used to represent shared pointers, where 10 bits are used to denote the process ID and 20 bits to denote a block. Since buckets are 2^3 bytes in size, only $\frac{2^{20}}{2^3} = 2^{20-3} = 2^{17}$ bucket can be stored per process, which is by far not enough for a proper distributed hash table. This can be solved by configuring Berkeley UPC with the `--enable-sptr-struct` flag, that changes the pointer structure to a struct, which supports block sizes up to $2^{32} - 1$ bytes.

2.6 Performance of One-Sided RDMA

To determine the expected performance of network communication with one-sided RDMA, we measured its latency and throughput by using the OSU Micro-benchmark [2]. This benchmark tests the latency and throughput of several RDMA operations via OpenSHMEM, with various message sizes. Figures 2.5 and 2.6 show the latencies of one-sided `memget` and `memput` operations, respectively, and Figure 2.7 shows the latency of `memput`.

For small messages, i.e. 16 bytes or smaller, the local `memget` operations are about 76 times as fast as remote `memget` operations. Local `memget` operations are performed in $50ns$ on average, and remote `memget` operations in $3.87\mu s$. The local `memput` operations, on the other hand, are only 6 times as fast as remote `memput` operations. This is because the `memput` does not require a full roundtrip. We observe that the differences in latency gets smaller, both for `memget` and `memput`, when the message size increases. Local `memput` operations take $0.44\mu s$ on average, and remote `memput` operations take $2.68\mu s$ with message sizes up to 16 bytes.

For messages up to 16 bytes, the *throughput* of local `memput` operations is on average 48 times higher than remote `memput` operations (Figure 2.7). Similar to the latencies, the differences get lower when the message sizes increase.

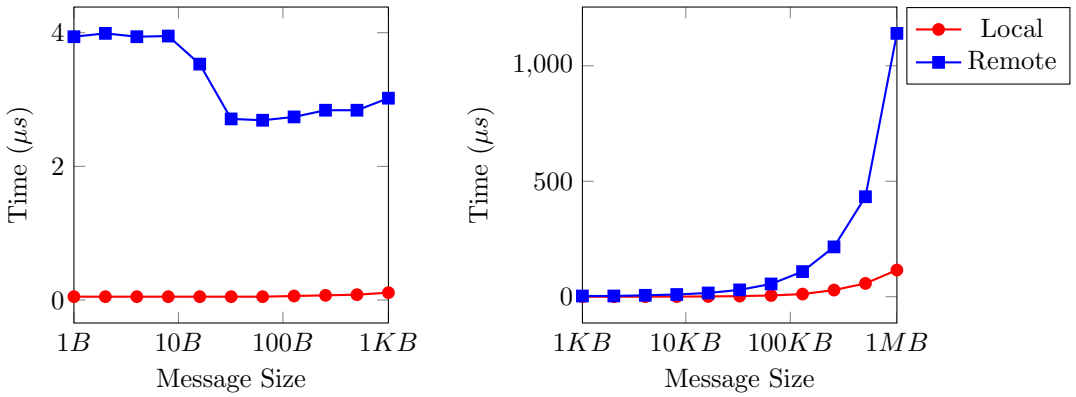


Figure 2.5: Latency of one-sided `memget` operations performed on both local and remote memory.

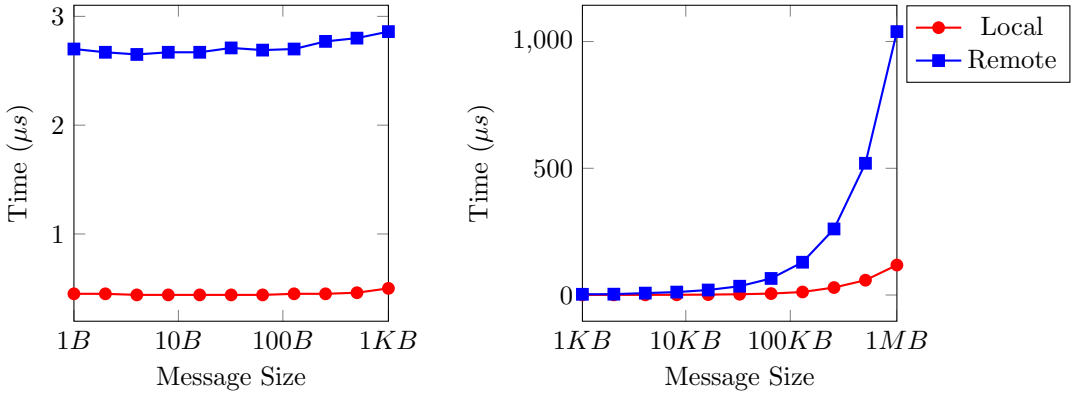


Figure 2.6: Latency of one-sided `memput` operations performed on both local and remote memory.

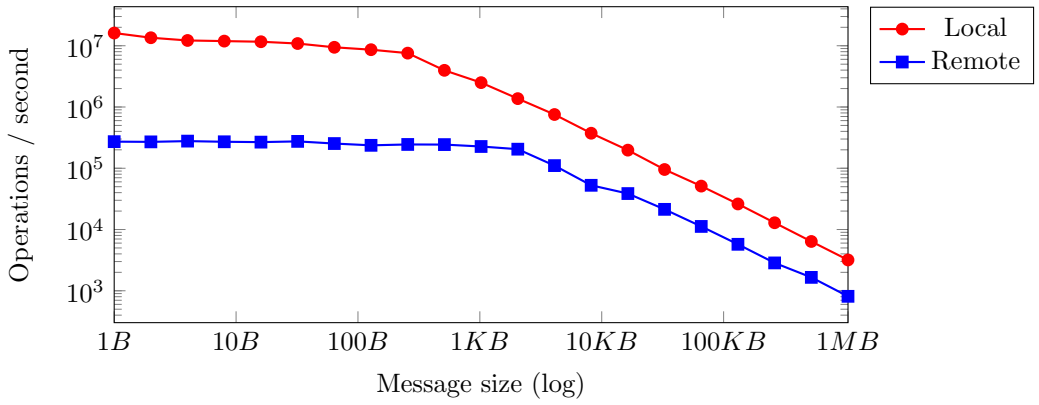


Figure 2.7: Throughput of the one-sided `memput` operation targeting both local and remote memory.

Chapter 3

Designing a Distributed Hash Table for Shared Memory

In this chapter different hashing schemes are compared and discussed. We argue that linear probing is expected to perform best because it requires less roundtrips than other hashing schemes. An analysis of linear probing is given and its expected performance is determined. After that, the design of `find-or-put` *is given and discussed in detail. A peak-throughput of* 3.6 *million op/s is reached by* `find-or-put` *on an Infiniband cluster.*

3.1 Introduction

Hash tables are popular data structures for storing maps and sets, since data can be retrieved and stored with time complexity $O(1)$ [69]. A *distributed* hash table is a hash table that is distributed over a number of computers. The advantage is that more memory is available, because the memory of every participating computer can be used. A disadvantage is that the performance of hash table operations might be lower due to the latency of the network and bandwidth limitations. Having a high-performance hash table is, however, essential for an efficient implementation of many distributed algorithms, like graph searching.

Our goal is to implement a distributed hash table using the PGAS programming model. The hash table supports only a single operation, namely `find-or-put`, that either inserts data when it is not already inserted or indices that the data element has already been added before. Secondly, the hash table must be applicable to any sort of memory-intensive high-performance distributed application that requires a hash table. Distributed graph searching or distributed symbolic reachability are merely examples. If needed, `find-or-put` can easily be split into two operations, `find` and `insert`. We did not implement a `delete` operation or garbage collection due to time constraints, but the hash table can easily be extended to support tombstones. Furthermore, the hash table should require minimal memory overhead, should be CPU-efficient (i.e. not be based on polling), and the latency of `find-or-put` should be minimal.

We tried to maximize the hash table performance by minimizing the number of roundtrips required by `find-or-put`, while keeping the hash table CPU-efficient. We compare the different hashing strategies used in previous work and argue that linear probing requires the least number of roundtrips. We show that many existing implementations either require more roundtrips or are CPU-intensive. Our implementation of `find-or-put` maintains CPU-efficiency by using one-sided RDMA operations, both for finding and inserting data. These operations are overlapped

as much as possible to minimize the waiting-times for roundtrips.

Previous work includes Pilaf [24], which is a key-value store that uses RDMA. Pilaf uses an optimized version of Cuckoo hashing to reduce the number of roundtrips in a worst case scenario. Lookups are performed by the clients, but inserts and deletes are performed by the server to simplify the synchronization of memory accesses. Nessie [67] is an RDMA-based hash table that also uses Cuckoo hashing. In Nessie, *all* hash table operations are performed by the clients, which makes its implementation more complex than Pilaf. HERD [10] is a key-value store that only uses one-sided RDMA writes and ignores the CPU bypassing features of RDMA to maximize the throughput. FaRM [5] is a distributed computing platform that exposes the memory of all machines in a cluster as a shared address space. A hash table is built on top of FaRM that uses a variant of Hopscotch hashing.

This chapter is structured as follows. Different hashing strategies are compared in Section 3.2. We argue that linear probing required the least number of roundtrips. In Section 3.3 the expected performance of linear probing in a distributed setting is analysed. Section 3.4 discusses the design of `find-or-put`. The experimental evaluation of `find-or-put` is given in Section 3.5. Finally, our conclusions are summarized in Section 3.6.

3.2 Preliminaries

It is critical to minimize the number of roundtrips required by `find-or-put` when accessing remote memory to achieve best performance. This is because hash table throughput is directly limited by the throughput of the RDMA devices. This is also shown in the experimental section. Furthermore, the waiting-times for roundtrips contribute to higher latencies of `find-or-put`. Minimizing the number of roundtrips reduces waiting-times and thus reduces latency. In this section some notation is given, followed by different techniques for solving hash collisions used in related work. After explaining those techniques, we argue that linear probing requires fewer roundtrips.

3.2.1 Notation

A *hash table* $T = \langle b_0, \ldots, b_{n-1} \rangle$ consists of a sequence of buckets b_i usually implemented as an array. We denote the *load-factor* of T by $\alpha = \frac{m}{n}$, where m is the number of elements inserted in T and n is the total number of buckets.

A hash function $h : U \to R$ maps data from some universe $U = \{0,1\}^w$ to a range of keys $R = \{0, \ldots, r-1\}$. We assume that every element $x \in U$ is a single machine word, so in case of a 64-bit architecture, $w = 64$ and U contains all binary-encoded 64-bit words.

Hash tables use hash functions to map words $x \in U$ to buckets $b_{h(x)}$ by letting $r < n$. Let $x \in U$ be a word. Then we write $x \in b_i$ if bucket b_i contains x, otherwise we write $x \notin b_i$. Let T be a hash table. We write $x \in T$ if there is some $1 \leq i \leq n-1$ for which $x \in b_i$, otherwise we write $x \notin T$.

For some $x, y \in U$ with $x \neq y$ it may happen that $h(x) = h(y)$. This is called a *hash collision*. A hash function h is called a *perfect hash function* if h is injective, meaning that h is perfect if $\forall x, y \in U. x \neq y \implies h(x) \neq h(y)$. A perfect hash function is thus collision-free. In practice, the domain U is unknown and often $|U| \gg |R|$, so usually h is not perfect [7].

A family $\{h_i : U \to R\}_{i \in I}$ of hash functions, indexed by some set I, is called *universal* if $\forall x, y \in U.\ x \neq y \Rightarrow\ Pr[h_i(x) = h_i(y)] \leq \frac{1}{|U|}$ for every $i \in I$. A hash function is called *universal* if it belongs to a universal family of hash functions [23]. In other words, $h : U \to R$ is called

universal if $Pr[h_i(x) = h_i(y)] \leq \frac{1}{|U|}$ for every $x, y \in U$. Universal hash functions have good theoretical properties, which are used to give theoretical expectations (in Section 3.3).

3.2.2 Hashing Strategies

In the ideal case, only a single roundtrip is ever needed by `find-or-put` to either find or insert data. This can, however, only be achieved when hash collisions do not occur. Unfortunately, they are very likely to occur in practice, since $|U|$ is much bigger than $|R|$ when a hash function $h : U \to R$ is used.

Our aim is to find a hashing strategy that is both CPU-efficient and requires a minimal number of roundtrips both for finding and inserting data. HERD only needs one roundtrip for every hash table operation, at the cost of CPU efficiency [10]. Only one-sided RDMA write operations are used by HERD to make requests to remote processes. These processes continuously poll memory locations to check for incoming requests. This greatly increases hash table throughput, but also limits HERD to be used in CPU-intensive algorithms. We aim to retain CPU-efficiency to keep the hash table usable in such algorithms.

Chained hashing. With chained hashing, every bucket is implemented as a linked list. Adding a data element $x \in U$ is performed by adding it to the linked list $b_{h(x)}$. Lookups and deletes can, however, take $\Theta(m)$ time in worst case, when all m data items are inserted into the same bucket. However, it can be shown that both successful and unsuccessful lookups require $\Theta(1+\alpha)$ time on average when a universal hash function is used [69], which is constant (as $\alpha \leq 1$). As a consequence, also deletes can be performed with a constant number of operations on average, since it uses `lookup`. Another consequence is that the number of RDMA operations required to perform remote finds and deletes is also constant. Because universal hash functions are very computational intensive, a more efficient hash function is often used in practice that achieves a suboptimal distribution. As an effect, it often happens that linked lists contain more than one element, thus also require more than one RDMA operation on average. Maintaining linked lists causes memory overhead due to storing pointers. Another disadvantage is that caching performance is poor due to the lack of data locality. Cache lines cannot effectively be used, because buckets are traversed via pointers.

Cuckoo hashing. *Cuckoo hashing* [52] is an open address hashing scheme that achieves constant lookup and deletion time and expected constant insertion time. The Cuckoo hashing scheme uses $k \geq 2$ independent hashing functions $h_1, \ldots, h_k : U \to R$ with $h_i \neq h_j$ for every $i \neq j$. With k-way Cuckoo hashing we refer to Cuckoo hashing with k hash functions. Pilaf uses 3-way Cuckoo hashing [24]. By increasing k, higher load-factors are supported, at the cost of memory accesses or network communication. The Cuckoo hashing scheme maintains the following invariant.

Invariant 1 (Cuckoo Invariant)**.** *Suppose that $k \geq 2$ independent hash functions $h_1, \ldots, h_k : U \to R$ are used by the Cuckoo hashing scheme. Then for every data item $x \in U$ it holds that either $x \notin T$ or $x \in b_{h_i(x)}$ for exactly one $1 \leq i \leq k$.*

Inserting a data element $x \in U$ involves inserting x in exactly one of the buckets $b_{h_1(x)}, \ldots, b_{h_k(x)}$. It may happen that all k buckets are occupied. In that case, the data element in one of the k buckets has to be relocated for x to be inserted. Suppose that $b_{h_i(x)}$ is chosen, for $1 \leq i \leq k$, and suppose that $y \in b_{h_i(x)}$. Then the relocation scheme removes y from $b_{h_i(x)}$, inserts x in $b_{h_i(x)}$, and inserts y by following the same procedure recursively. The recursion chain breaks when y can be inserted without requiring another relocation. Alternatively, the chain may also break when the recursion depth reaches a certain threshold or when the recursion chain enters a loop.

In both cases, a rehashing scheme must be applied, which not only may require a large number of roundtrips, but also a locking mechanism, which is particularly expensive in a distributed environment. By increasing k, the probability of having to perform a relocation decreases.

Lookups require at most k roundtrips, which can be overlapped to decrease waiting times and thus decrease latency. On the other hand, by increasing k, the number of RDMA operations also increases, which reduces the throughput of the hash table.

Bucketized Cuckoo hashing. A variant on Cuckoo hashing, called *Bucketized Cuckoo Hashing*, enables buckets to contain multiple data elements. Each bucket is subdivided into l slots, and bucketized Cuckoo hashing is then called (k, l)-*Cuckoo hashing*, where k is the number of hash functions used. The bucketized Cuckoo hashing scheme maintains the invariant that every inserted data element $x \in T$ is inserted into exactly one of the l slots, owned by one of the buckets $b_{h_1(x)}, \ldots, b_{h_k(x)}$.

Because every bucket contains multiple slots, it is more likely for a data item to be inserted without needing to relocate items. Furthermore, the number of roundtrips required by the hashing operations, as well as the length of relocation chains, is linearly reduced by l. Bucketized Cuckoo hashing performs efficient even when $\alpha > 0.9$ according to [15]. The designers of Pilaf pointed out that $(2, 4)$-Cuckoo hashing could be very efficient [24].

Hopscotch hashing. Another hashing scheme with constant lookup time and expected constant insertion time is *Hopscotch Hashing* [46]. The Hopscotch hashing scheme uses a single hash function $h : U \to R$ and every bucket is assigned to a fixed-sized *neighbourhood*. We denote the size of a neighbourhood by a constant $H \geq 1$. The neighbourhood of bucket b_i, which we denote by $N(b_i)$, is b_i itself, together with the next $H-1$ buckets modulo n, so $N(b_i) = \langle b_i, \ldots, b_j \rangle$ with $j = (i + H - 1) \mod n$. Hopscotch maintains the following invariant.

Invariant 2 (Hopscotch Invariant)**.** *Let $x \in U$ be some data item and let $N(b_{h(x)}) = \langle b_1, \ldots, b_H \rangle$. Then either $x \notin T$ or $x \in b_i$ for exactly one $1 \leq i \leq H$.*

Alternatively, for every $x \in U$ it holds that $x \notin T$ or appears exactly once in $N(b_{h(x)})$. As a consequence, a `lookup`(x) operation only needs to examine $N(b_{h(x)})$ in order to determine if $x \in N(b_{h(x)})$. Because neighbourhoods are consecutive blocks of memory, they can be obtained with a single RDMA roundtrip. Lookups can thus be performed with a single RDMA roundtrip. This is a better result than (Bucketized) Cuckoo hashing, which requires $k \geq 2$ roundtrips. A `delete`(x) operation simply searches for the bucket $b \in N(b_{h(x)})$ with $x \in b$ and empties b with a `cas` operation. Assuming that the `cas` succeeds, `deletes` require only two roundtrips.

The `insert`(x) operation examines the neighbourhood $N(b_{h(x)})$ and inserts x into the first empty bucket it encounters by performing a `cas` operation. If `cas` fails, the operation simply searches for the next empty bucket. If no such bucket exists, the `insert` operation tries to *relocate* one of the data items in $N(b_{h(x)})$ in order to make place for x, while still maintaining Invariant 2. This requires a relocation scheme similar to the one used in Cuckoo hashing. Such a scheme chooses some element $y \in N(b_{h(x)})$, reads the neighbourhood $N(b_{h(y)})$ of y and tries to insert y into that neighbourhood. If the insert fails, the relocation scheme is recursively applied to relocate y in $N(b_{h(y)})$, otherwise x can be swapped for y in $N(b_{h(x)})$. If the recursion depth reaches a certain threshold, or if none of the items in the neighbourhood can be relocated, a rehashing scheme must be applied. A relocation scheme is very expensive because it requires many roundtrips and a locking mechanism.

Linear probing. When finding or inserting some data element $x \in U$, the linear probing strategy examines a number of *consecutive* buckets $b_{h(x)+0}, b_{h(x)+1}, \ldots, b_{h(x)+t}$ until the intended

bucket has been found or some threshold $t \geq 0$ has been reached. Linear probing is very cacheline-efficient, because the examined buckets are consecutive in memory.For the same reason, a range of buckets can be obtained with a single RDMA roundtrip when using linear probing in a distributed environment, which reduces the number of roundtrips.

We expect linear probing to require fewer roundtrips than Hopscotch hashing due to the absence of relocation schemes, which require many roundtrips and a locking mechanism. The Hopscotch hashing scheme enabled lookups to be performed with only a single roundtrip, but that does not effectively apply to a `find-or-put` operation. Hopscotch hashing could be more efficient when the hash table is used under a read-intensive workload and when the load-factor is very high. On the other hand, inserting enough items to reach such a high load-factor could already make linear probing more efficient, because it does not need to relocate items. Furthermore, it may happen that relocations cannot be performed, which especially holds under high load-factors. We expect that maintaining Invariant 2 is too expensive for an efficient implementation of `find-or-put`.

A variant on linear probing is designed by Laarman et al. [7] in the context of NUMA machines. Here, linear probing is performed in a cache line, which the authors called *walking-the-line*, followed by double hashing to improve the distribution of data. This scheme obtains very high parallel speedups due to the efficient use of data locality. By walking over cache lines, the number of cache misses is minimized. Van Dijk et al. [75] uses this scheme to implement `find-or-put` for multi-core symbolic model checking.

Other open addressing schemes. Quadratic probing is similar to linear probing, only the hash sequence increases quadratically rather than linearly. More specifically, a quadratic hash function $h' : U \times \mathbb{N} \to R$ can be derived from $h : U \to R$ by choosing $c_1, c_2 \in \mathbb{N}^+$ and letting $h'(x, i) = h(x) + c_1 i + c_2 i^2 \mod r$. Then the buckets $b_{h'(x,0)}, \ldots, b_{h'(x,t)}$ can be tried until an empty bucket has been found or some threshold t has been reached. Quadratic probing gives a better data distribution than linear probing because it prevents clustering.

Double hashing uses two independent hash function $h_1, h_2 : U \to R$ to resolve hash collisions. A combined hash function $h' : U \times \mathbb{N} \to R$ can be constructed as $h'(x, i) = h_1(x) + i h_2(x) \mod r$. Similar to quadratic probing, h' can be used to examine a sequence of buckets. Since the interval depends on the actual data rather than a constant value, double hashing achieves an even better hash distribution than quadratic probing.

A disadvantage of both hashing strategies is that they examine buckets that are non-consecutive in memory. Because of that, data locality cannot be effectively exploited. As an effect, both strategies require more roundtrips than linear probing.

3.2.3 Conclusion

We expect Hopscotch hashing to perform better than Cuckoo hashing in a distributed environment due to block retrievals. In Hopscotch hashing, fixed-sized blocks of buckets are retrieved with a single roundtrips, whereas k-way Cuckoo hashing requires k roundtrips. Both hashing schemes need a relocation mechanism, which may require many extra roundtrips and a locking mechanism. Linear probing, on the other hand, also uses block retrievals and does not need a relocation mechanism. When a collision occurs, linear probing simply reads another block instead of relocating buckets, which we expect to be much cheaper. Our design of `find-or-put` uses linear probing for this reason.

3.3 Theoretical Expectations

To reduce the number of roundtrips, a fixed-sized range of buckets can be obtained with a single roundtrip, which we refer to as a *chunk*. We denote the chunk-size by $C \geq 1$. In this section we analyse the expected performance of linear probing with C-sized chunks. We also analyse the behaviour of linear probing under different values of C. The following Theorem shows the expected number of probes required by classical linear probing.

Theorem 1. *Assuming that a universal hash function is used, the expected number of buckets to examine until an empty bucket is found with linear probing is at most:*

$$g(\alpha) = \frac{1}{2}\Big(1 + \frac{1}{(1-\alpha)^2}\Big)$$

A detailed proof of 1 is given in [45]. Since $\alpha \leq 1$ we have $g(\alpha) \leq g(1)$, so the expected number of operations required by lookups, inserts, and deletes is constant. This implies that the expected number of roundtrips required by these operations is also constant. Since linear probing examines consecutive buckets by querying for C-sized chunks, the required number of roundtrips can be linearly reduced by C, which is shown in the next Collorary.

Corollary 1. *When using C-sized chunks with $C \geq 1$, the expected number of chunks to be inspected by linear probing until an empty bucket is found is at most:*

$$g(C, \alpha) = \frac{1}{2C}\Big(1 + \frac{1}{(1-\alpha)^2}\Big)$$

Proof. By Theorem 1, the expected number of buckets to examine until an empty bucket is found is at most $g(\alpha)$. Then the expected number of C-sized chunks to read until an empty bucket is found is at most:

$$g(C, \alpha) = \frac{g(\alpha)}{C} = \frac{1}{2C}\Big(1 + \frac{1}{(1-\alpha)^2}\Big)$$

□

Figure 3.1 shows the expected number of chunks $g(C, \alpha)$ to read with various chunk sizes. When the load-factor reaches 0.8, the value of $g(C, \alpha)$ grows vastly, no matter the value of C. Furthermore, the effects of increasing C gets smaller when the load-factor increases. The following Lemma gives a minimal value for C so that C-sized chunks are expected to contain an empty bucket.

Lemma 1. *A chunk of size at least $\frac{1+(1-\alpha)^2}{2(1-\alpha)^2}$ is expected to contain an empty bucket.*

Proof. Suppose that we have a hash table with load-factor α. We want to find a minimal chunk size C such that $g(C, \alpha) \leq 1$ is maximal, where $g(C, \alpha)$ is taken from Lemma 1. In other words, we want to find a term $f(\alpha)$ such that $g(f(\alpha), \alpha) = 1$. Thus:

$$1 = g(f(\alpha), \alpha) \Leftrightarrow 1 = \frac{1}{2f(\alpha)}\Big(1 + \frac{1}{(1-\alpha)^2}\Big) \Leftrightarrow 2f(\alpha) = 1 + \frac{1}{(1-\alpha)^2} \Leftrightarrow$$

$$f(\alpha) = \frac{1}{2} + \frac{1}{2(1-\alpha)^2} \Leftrightarrow f(\alpha) = \frac{1 + (1-\alpha)^2}{2(1-\alpha)^2}$$

Then a C-sized chunk is expected to contain an empty bucket if $C \geq f(\alpha)$. □

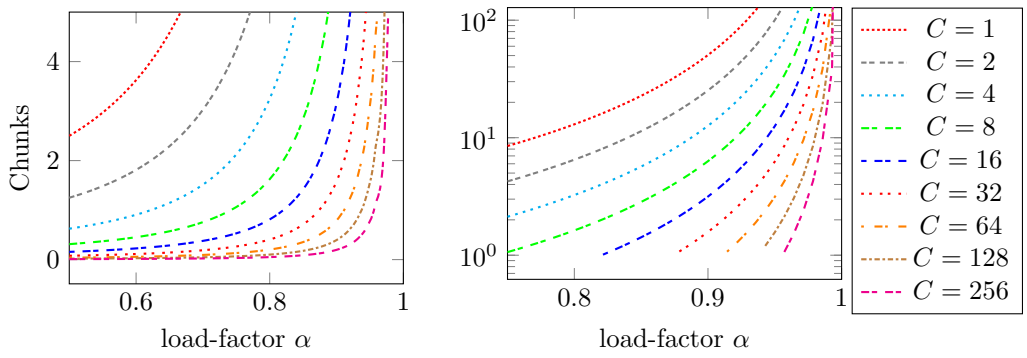


Figure 3.1: The expected number of chunks $g(C, \alpha)$ to read until an empty bucket has been found, with different chunk sizes C and load-factors α. The function $g(C, \alpha)$ is taken from Lemma 1. Observe that the value of $g(C, \alpha)$ increases vastly with $\alpha > 0.8$, no matter the value of C.

Lemma 1 shows the expected minimal chunk size that a hash table with load-factor α should have to find an empty bucket with one roundtrip. The effects of Lemma 1 are shown in Figure 3.2. With a load-factor $\alpha \leq \frac{1}{2}$, a chunk size of $C = 4$ can be used and we expect that no extra chunk has to be read when inserting a data item. When the load-factor grows bigger (i.e. $\alpha > \frac{1}{2}$), the suggested chunk size increases vastly. So a trade-off has to be made between the chunk size and the number of chunks to read. In other words, a trade-off between the size of the RDMA reads and the number of RDMA reads has to be made. The following Theorem gives an efficiency bound to the load-factor α.

Theorem 2 (Efficiency Bound)**.** *Assuming the use of a universal hash function, a chunk of size $C \geq 1$ is expected to contain an empty bucket if $\alpha \leq V(C)$, where:*

$$V(C) = 1 - \sqrt{\frac{1}{2C-1}}$$

Proof. By Lemma 1, a chunk of size C is expected to contain an empty bucket if $C \geq \frac{1+(1-\alpha)^2}{2(1-\alpha)^2}$, which can be rewritten to:

$$C \geq \frac{1+(1-\alpha)^2}{2(1-\alpha)^2} \Leftrightarrow 2C(1-\alpha)^2 \geq 1 + (1-\alpha)^2 \Leftrightarrow$$

$$(2C-1)\alpha^2 + (2-4C)\alpha + (2C-2) \geq 0$$

Solving this quadratic inequality yields two possibilities:

$$\alpha \geq \frac{(4C-2)+\sqrt{8C-4}}{2C-1} = 1 + \frac{\sqrt{2C-1}}{2C-1} = 1 + \sqrt{\frac{1}{2C-1}} \tag{3.1}$$

$$\alpha \leq \frac{(4C-2)-\sqrt{8C-4}}{2C-1} = 1 - \frac{\sqrt{2C-1}}{2C-1} = 1 - \sqrt{\frac{1}{2C-1}} \tag{3.2}$$

Since $\alpha \leq 1$, option 3.1 does not apply. Furthermore, $C \geq 1$, so $\sqrt{\frac{1}{2C-1}} \leq \sqrt{\frac{1}{2\times 1-1}} = \sqrt{1} = 1$, which gives a bound to option 3.2. Thus, an C-sized chunk with $C \geq 1$ is expected to have an

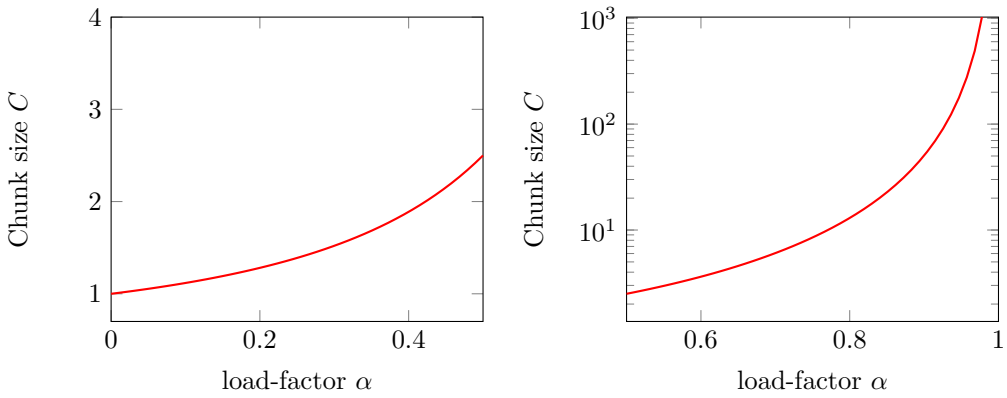


Figure 3.2: The chunk size C that is expected to contain an empty bucket in a hash table with load-factor α, which is the result of Lemma 1 and Theorem 2. Alternatively, this figure shows the *efficiency bound* $V(C)$ with chunk size $C \leq 1024$.

empty bucket for

$$0 \leq \alpha \leq 1 - \sqrt{\frac{1}{2C-1}} \leq 1$$

□

By performing a `find-or-put`(d) operation with $d \notin T$ and $\alpha \leq V(C)$, we expect that only two roundtrips are required, one to read the chunk and one to write d into an empty bucket. If $\alpha > V(C)$, we expect that $1 + g(C, \alpha)$ roundtrips are needed. We expect the hash table to perform *efficiently* when $\alpha \leq V(C)$. Figure 3.2 shows the efficiency bounds for a hash table under various load-factors with various chunk-sizes. The suggested chunk size increases vastly when the load factor gets bigger. For example, a hash table with chunk size 16 is expected to perform efficiently for $\alpha \leq 0.82$.

Corollary 2. *Let M be the maximum number of chunks to consider. Assuming the use of a universal hash function,* `find-or-put` *is expected to return* `full` *when:*

$$\alpha \geq 1 - \sqrt{\frac{1}{2CM-1}} = V(CM)$$

Corollary 2 follows from Theorem 2 and gives a bound to the load-factor supported by `find-or-put`. An universal hash function is assumed for this result. In practice, universal hash functions are very computational intensive, so high-performance algorithms often use cheaper hash functions with a poorer hash distribution. It would be interesting to determine the differences in distribution quality, which we could not do due to time constraints.

Corollary 2 and Corollary 1 also give a guideline to the choice of M, the maximum number of chunks to consider until `find-or-put` returns `full`. Every chunk is retrieved via a roundtrip, so taking a large value of M may result in a large number of roundtrips when the load-factor gets high. Choosing a small value of M and a large value for C might result in poor performance, because many buckets in a retrieved chunk may not be considered. To get optimal performance, a good trade-off between values M and C has to be made, which can now analytically be done.

Chunk Size C	**Efficiency Bound** $V(C)$	**Maximum Chunks** M	**Roundtrip Latency***
8	0.74	7	$2.69\mu s$
16	0.82	4	$2.74\mu s$
32	0.87	2	$2.84\mu s$
64	0.91	1	$2.84\mu s$
128	0.94	1	$3.02\mu s$
256	0.96	1	$3.13\mu s$

Figure 3.3: The expected performance for different values of C for `find-or-put` to support load-factors up to 0.9. The roundtrip latencies (*) are taken from the experimental results in Chapter 2.

3.3.1 Research Expectations

The size of a bucket in our hash table is 64 bits (i.e. 8 bytes), so a total of 8 buckets can be placed on a single cache line. To make optimal use of cache lines, we let C be a multiple of 8. If we choose to let $C = 8$, then linear probing is expected to perform efficiently until $\alpha \geq V(8) = 0.742$ by Theorem 2. If an additional cache line is used and $C = 16$, then the hashing scheme can efficiently be used until $\alpha \geq V(16) = 0.82$, which is a lot better than using only a single cache line (an improvement of 10.5%). Furthermore, we observed that one-sided RDMA reads performs better with 128-byte packets than with 64-byte packets, so we expect that $C = 16$ performs best. It is possible to further increase C to 24 and adding another cache line, but we expect this to have a negative impact on the performance of the remote operations. Also $V(24) = 0.854$, which is only a 4% improvement compared to $V(16)$.

As an example, suppose that we want the hash table to support load-factors up to 0.94, which is already a high load-factor in practice. Then Corollary 1 can be used to determine M by letting $M \geq g(C, 0.9)$, for any value of C. Furthermore, the efficiency bound of C can be determined by using Theorem 2. The results are given in Table 3.3. The roundtrips latencies are derived from the data presented in Figure 2.5.

3.4 Design and Implementation

In this section the design of `find-or-put` is explained and discussed. Linear probing is used, which could arguably be implemented to require fewer roundtrips than alternative hashing schemes. Furthermore, we tried to minimize the waiting times for roundtrips by overlapping them as much as possible. This is done by using asynchronous memory operations, which can be used in a number of PGAS implementations. Figure 3.5 shows the design of `find-or-put`. In this design, the value C is again used to denote the chunk size and the value M is used as a threshold on the maximum number of chunks to consider when searching for the intended bucket.

The `find-or-put` operation reads a chunk of buckets from the hash table via a one-sided RDMA read, and then iterates the chunk to search for the intended bucket. During the iteration, new chunks are already retrieved, so that the waiting times for roundtrips are reduced, should the previous chunk not contain the intended bucket. The remainder of this section discusses the memory layout of the shared hash table and its buckets, the asynchronous query operations, and the implementation of `find-or-put` with its design considerations.

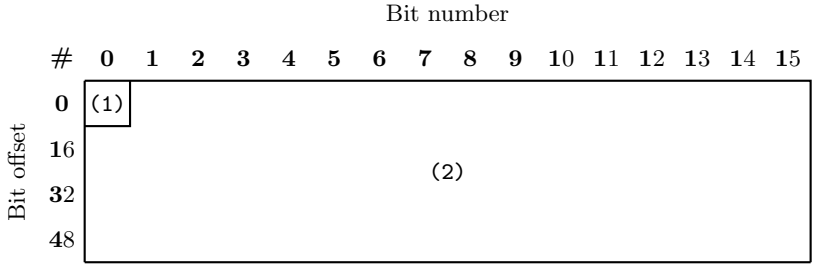


Figure 3.4: The memory layout of a 64-bit bucket, where `(1)` denotes the occupation bit and `(2)` is used to store the actual data.

3.4.1 Memory Layout

Figure 3.4 shows the memory layout of a bucket. Each bucket is 64 bits in size. The first bit is used as a flag to denote bucket occupation. The remaining 63 bits are used to store data. When data is inserted, the occupation bit is set via a `cas` operation to ensure data consistency and to prevent expensive distributed locking mechanisms. If the hash table needs to support the storage of data larger than 63 bits, a separate shared data array can be used, in which the actual data elements can be stored. The corresponding indices are stored in the 64-bit buckets.

The `cas` operation is explained in Section 2.5.2. The `data`(b) operation takes a bucket b as input and returns the data element stored in b, thus ignoring the occupation bit.

Shared and Private Arrays. Assuming that n processes $p_1, \ldots, p_n$ use the hash table, a *shared* table $\mathsf{B}[0] \ldots \mathsf{B}[kn-1]$ of buckets is allocated. Each worker owns (i.e. stores) k buckets. The `find-or-put` operation retrieves chunks by placing them into *private* memory, so that they can be iterated efficiently. For this, two-dimensional arrays $\mathsf{P}[0][0], \ldots, \mathsf{P}[M-1][C-1]$ are allocated in private memory on *every* process p_i. Every process aligns P along cache lines, thereby minimizing the number of cache misses when iterating over P. This reduces the number of fetches from main-memory and thus increases performance. However, we only expect performance increases when `find-or-put` targets local memory (i.e. memory owned by a local process). We expect that waiting times for roundtrips greatly outweigh the performance gains of cache line alignment. Cache lines are typically 64 bytes in size, so 8 buckets can be placed on a single cache line. For this reason, we let C be a multiple of 8.

3.4.2 Querying for Chunks

When a process queries a chunk, it transfers a consecutive block of C buckets from B into its private array P, so that the bucket may be examined locally. When `find-or-put` does not find the intended bucket, it may query for the next chunk. As a result, several *consecutive* chunks might be requested. The `query-chunk`(i, h) operation queries for the ith consecutive chunk, assuming $i \geq 0$, starting from bucket $\mathsf{B}[h \mod kn]$. The `query-chunk` operation returns a handle, which can be given to `sync` to synchronize on the query, which blocks further execution of the process until the roundtrip completes. By calling `query-chunk` multiple times before synchronizing on them, their queries overlap, thereby reducing the amortized waiting times.

The `query-chunk` operation starts by determining *start* and *end*, denoting the first and last index of the chunk to retrieve, respectively. When $start < end$, the chunk wraps around the kn-sized hash table, hence the check at line 5. In that case, the chunk consists of two smaller

chunks that are non-consecutive in memory. Two roundtrips are required to fetch them, which are performed by `split` (lines 9 and 10). The `memget`, `memput`, and `sync` operations are discussed in Section 2.5.2.

3.4.3 Design Considerations of `find-or-put`

The `find-or-put`(d) operation returns `found` when $d \in T$ (lines 17 and 19) before invoking the operation and returns `inserted` when $d \notin T$ (line 15) before invoking the operation. If $d \notin T$ and all MC consecutive buckets starting from $h(d)$ are already occupied, `full` is returned (line 20). In our implementation, the execution of the program simply terminates when `full` is returned, but it would also be possible to perform rehashing or resizing of the hash table. We did not implement either of the two schemes, since we allocate all available memory when using the hash table in the BDD implementation. This makes resizing superfluous. Furthermore, rehashing would only help slightly when the current hash function is replaced by a better, probably more computational intensive hash function. Figure 3.2 shows that the hash table is expected to easily handle load-factors up to 0.9 when a reasonable chunk size is used, which is enough for our use case.

Retrieving Chunks. The algorithm starts by requesting the first chunk (line 3) and, if needed, tries a maximum of $M - 1$ more chunks (line 4) to find the intended bucket. Before calling `sync`(s_i) on line 7, the next chunk is already requested by calling `query-chunk`($i+1, d$) on line 6. This call returns a handle s_{i+1} and before this handle is synchronized on, first a whole iteration of the for-loop at line 4 is performed. Such an iteration includes the synchronization of the previous query (with the exception of the first iteration) and the iteration over the chunk itself. This reduces overal waiting times. We actually expect queries to be already completed when synchronization is performed.

Iterating a Chunk. By iterating over a chunk in private memory, when some bucket $\mathsf{P}[i][j]$ is empty, `find-or-put` tries to insert *data* into that bucket with `cas` on line 13. $\mathsf{B}[a]$ is the bucket in shared memory corresponding to $\mathsf{P}[i][j]$ in private memory. The *former* value of $\mathsf{B}[a]$ is returned by `cas`, which is enough to check if the `cas` operation succeeded (line 14). In that case, `inserted` is returned (line 15). Otherwise, some other process claimed $\mathsf{B}[a]$ in the time between the calls to `query-chunk` and `cas`. It might happen that d is written to that bucket, hence the check at line 16. If not, `find-or-put` returns to line 8 and tries the next bucket. If $\mathsf{P}[i][j]$ is occupied, `find-or-put` checks if $d \in \mathsf{P}[i][j]$ at line 18. In that case, `found` is returned. If not, the next bucket is tried.

Interleaving Queries. Figure 3.6 shows the effect of performing asynchronous queries. The `find-or-put`(d) operation queries for the first and second chunk on lines 3 and 6, respectively. It then synchronizes on the first chunk at line 7, which causes the process to halt (shown by the gray bar at `sync`(s_0)). After that, the first chunk is iterated. If the intended bucket is not found in the first chunk, `find-or-put` starts a query to the third chunk and synchronizes on the second chunk. The `find-or-put` operation continues doing this until the intended bucket is found or M chunks have been tried. The effect of asynchronous queries is that the write rectangles in Figure 3.6 overlap with the gray ones, thus reducing the overall waiting times for roundtrips.

```
1  def find-or-put(data):
2    h ← hash(data)
3    s_0 ← query-chunk(0, h)
4    for i ← 0 to M − 1:
5      if i < M − 1
6        s_{i+1} ← query-chunk(i + 1, h)
7      sync(s_i)
8      for j ← 0 to C − 1:
9        if is-not-occupied(P[i][j])
10         ▷ Try to claim bucket B[a]
11         a ← (h + iC + j) mod kn
12         d ← data(data) | OCCUPIED
13         val ← cas(B[a], P[i][j], d)
14         if val = P[i][j]
15           return inserted
16         elif data(val) = data
17           return found
18       elif data(P[i][j]) = data
19         return found
20   return full

1  def data(b):
2    return (b & DATA)
```

```
1  def query-chunk(i, h):
2    ▷ Find start and end index
3    start ← (h + iC) mod kn
4    end ← (h + (i + 1)C − 1) mod kn
5    if end < start
6      return split(start, end)
7    else
8      ▷ Determine source/dest. blocks
9      S ← ⟨B[start], ..., B[end]⟩
10     P ← ⟨P[i][0], ..., P[i][C − 1]⟩
11     return memget-async(S, P)

1  def split(start, end):
2    ▷ Find the blocks in shared memory
3    S_1 ← ⟨B[start], ..., B[kn − 1]⟩
4    S_2 ← ⟨B[0], ..., B[end]⟩
5    ▷ Corresp. blocks in private memory
6    P_1 ← ⟨P[i][0], ..., P[i][|S_1| − 1]⟩
7    P_2 ← ⟨P[i][|S_1|], ..., P[i][C − 1]⟩
8    ▷ Retrieve the chunk
9    s_1 ← memget-async(S_1, P_1)
10   s_2 ← memget-async(S_2, P_2)
11   return ⟨s_1, s_2⟩
```

Figure 3.5: The design of `find-or-put` and `query-chunk`. The `query-chunk` operation queries the ith chunk and `sync` synchronizes on it. The 64-bit bitmask `OCCUPIED` denotes bucket occupation and `DATA` masks data in a bucket.

3.5 Experimental Evaluation

We implemented the design of `find-or-put` in Berkeley UPC, version 2.20.0. The implementation is evaluated by measuring latency, throughput, and the number of roundtrips required by `find-or-put` under various configurations. Furthermore, we compared the actual experimental outcomes with our research expectations, which are given in Section 3.3. The experiments have been performed on a cluster of 10 Dell M610 machines (the `m610` partition in the CTIT computing lab). Each machine has 8 CPU cores and 24 GB of internal memory. All machines run Ubuntu 14.04.2 LTS with kernel version 3.13.0 and are connected via a 20 GB/s Infiniband network. All experiments have been repeated at least three times, and the average measurements are considered.

All remote benchmarks (i.e. the benchmarks that involve at least two machines and the Infiniband network) are compiled with the command `upc -network=ibv`, so that UPC uses the Infiniband verbs library. All local benchmarks (i.e. the benchmarks that use only a single machine) are compiled with the command `upc -pthreads`. The `-pthreads` flag enables processes to create multiple UPC threads. The local benchmarks do not use the Infiniband Verbs library, otherwise all UPC operations invoked in `find-or-put` are performed via the RDMA device instead of via the CPU, which drastically decreases performance. The remote benchmarks were executed by running `upcrun -n` processes `-N` machines `-shared-heap 1100MB`, where we replaced processes with the total number of processes and machines with the number of partici-

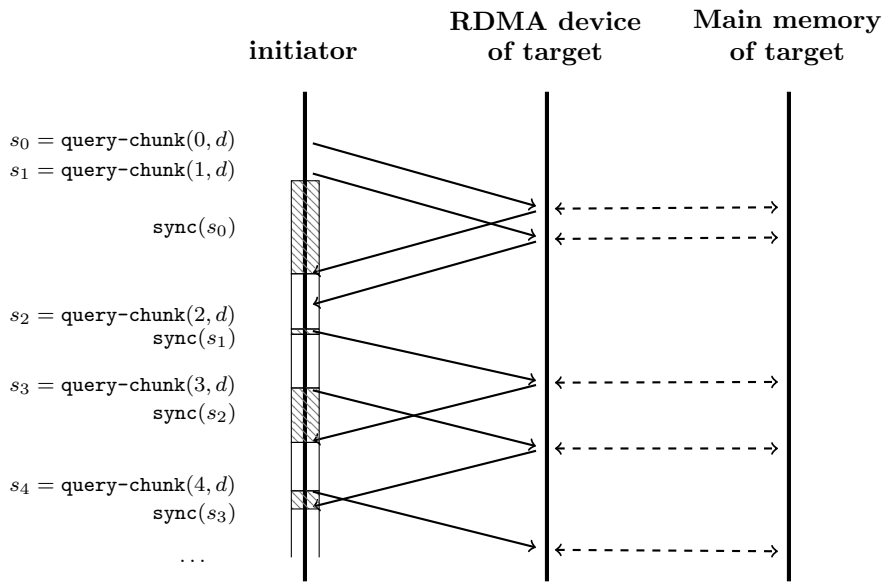


Figure 3.6: The effects of using `query-chunk` and `sync-chunk` to retrieve chunks of buckets asynchronously. The gray-striped rectangles denote the waiting time for the blocking `sync-chunk` operation to return. The white rectangles denote the time to iterate over the C buckets, obtained by the previous call to `query-chunk`.

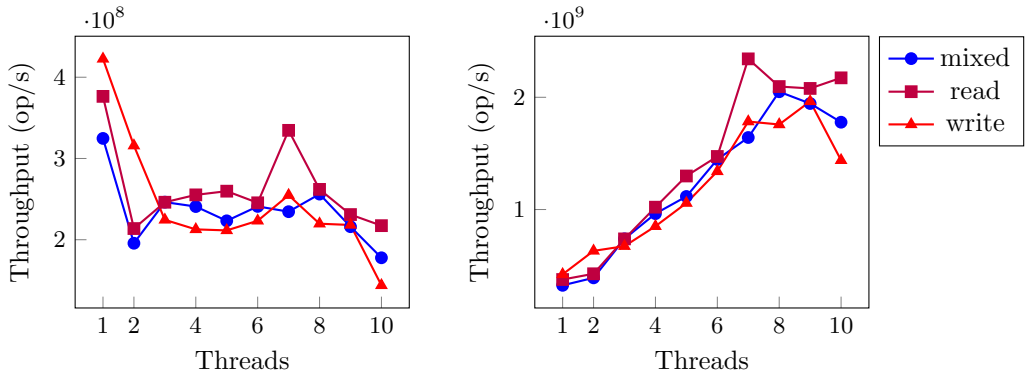
pating machines. The `-shared-heap` enables every process to allocate up to 1100 MB of shared memory. The local benchmarks were executed via `upcrun -n` threads `-N 1 -pthreads` threads `-bind-threads`, with theads the number of parallel UPC threads to use. The `-bind-threads` flag causes threads to be bound to CPU cores, which increases performance.

3.5.1 Throughput of `find-or-put`

We measured the throughput of `find-or-put` in terms of operations per second (op/s). This is done by creating a shared hash table where each participating process contributes a block of 1024 MB. Thus, the total hash table size is $p \times 2^{10}$ MB, with p the total number of processes. Every process performs 10^7 `find-or-put` operations with pseudo-randomized data.

Three different workloads have been taken into account when measuring throughput, namely a *mixed* workload (50% finds, 50% inserts), a *read-intensive* workload (80% finds, 20% inserts), and a *write-intensive* workload (20% finds, 80% inserts). For each workload, we used a different procedure to select the pseudo-randomized data given to each call to `find-or-put`. This procedure is able to introduce a skew in the workload if needed, while maintaining a proper distribution of data (i.e. the memory of every process is accessed equally often). Also, every process performs an approximately equal amount of finds and inserts.

Local Throughput. Figure 3.7 shows the throughput obtained with one machine, without the use of the Infiniband network. Here we used only one process, which spawns a various number of UPC threads. We scaled from 1 to 10 UPC threads, which we bound to separate CPU cores. Figure 3.7a shows that the average throughput per thread decreases when the number of threads increases. The biggest drop (54.9% on average) is observed when two threads are spawned, in



(a) Throughput per thread

(b) Total throughput

Figure 3.7: Throughput of `find-or-put` using one machine (i.e. without using RDMA). The throughput is measured under a mixed workload, a read-intensive workload, and a write-intensive workload. The total throughput is the sum of the throughputs of all threads.

Workload	**Base Throughput**		**Best Throughput**		**Speedup**
	Throughput	Procs.	Throughput	Procs.	
Mixed	324,676,333	1	2,049,388,900	8	6.31
Read-intensive	376,434,000	1	2,342,278,333	7	6.22
Write-intensive	422,593,000	1	1,963,570,267	9	4.65

Figure 3.8: The throughputs obtained when using one machine. The throughputs obtained with one process (base throughputs) and the best achieved throughputs are given.

comparison with one thread. In that case, shared memory is used (i.e. memory shared between two threads), which causes a performance reduction compared to one thread. When more than 8 threads are used, performance also drops. This is because the machines used in the experiments only have 8 physical CPU cores. When considering a mixed workload, a maximum speedup of 6.3 is reached when 8 threads are used (see also Figure 3.8). Surprisingly, the benchmarks with a mixed workload obtain higher speedups than the read-intensive workloads. We expect this to be caused by the overhead generated due to asynchronous queries. We may conclude that asynchronous queries are more beneficial for remote operations. The write-intensive benchmarks obtain lower speedup because those operations fill buckets, in addition to merely reading neighbourhoods.

Remote Throughput. Figure 3.10 shows the average throughputs when RDMA is used by the processes. In addition, Figure 3.12 shows the best achieved throughputs for the different workloads. Compared to the local throughputs, a performance drop of several orders of magnitude is observed. A peak-throughput of 2.05×10^9 op/s is reached locally under a mixed workload, compared to 3.6×10^6 remotely, which is 568 times lower. This difference is caused by the overhead generated by the RDMA devices. For every one-sided write, the RDMA device needs to read from main-memory, send a message to a remote RDMA device, which in turn needs to access remote memory. For a one-sided read, the remote RDMA device sends back a message in addition to the previous steps, and also causes an additional local memory access. Furthermore, all network accesses performed by RDMA devices are done via the PCI-E bus, instead of using

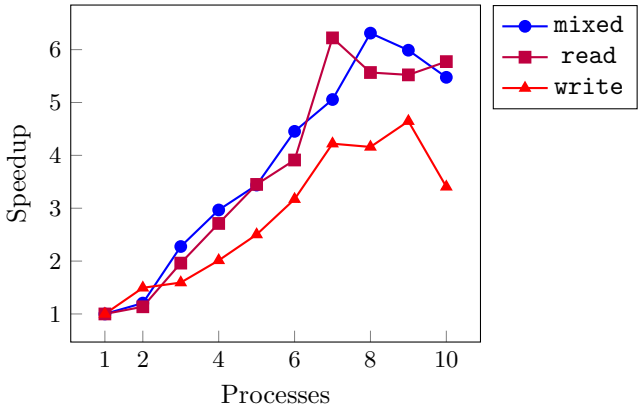


Figure 3.9: Speedups on throughput when using one machine (i.e. without using RDMA). A mixed workload (`mixed`), a read-intensive workload (`read`), and a write-intensive workload (`write`) is considered.

the north-bridge, which impacts performance.

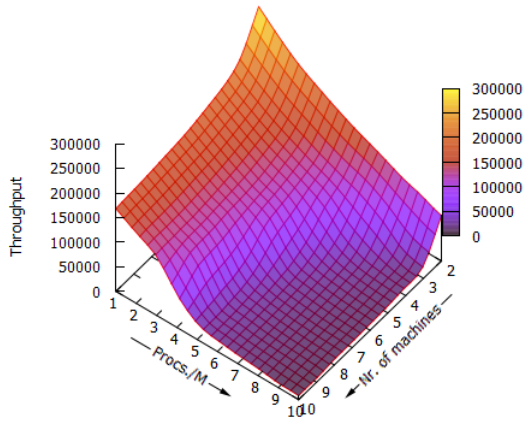
The average throughputs per process, shown in Figure 3.10, clearly shows the saturation points for the RDMA devices. When enough processes are added, the throughput per processes remains relatively linear. The relatively steep slope beforehand is caused by the many processes used per machine, which is a threshold for the RDMA devices to reach their maximum throughput. By default, Berkeley UPC has a fixed number of processes per machine, but we expect that this slope could be less steep when a *variable* number of processes per machine could be used instead. In addition, we expect that the maximum achievable throughput also increases by spawning processes dynamically. It would be interesting to try dynamic thread creation and apply algorithms that *learn* the effects on performance when threads are either added or removed.

The total throughputs, presented in the right plots in Figure 3.10, also show the saturation points of the RDMA devices. These results imply that scalability along CPUs decreases when the number of machines increases. Performance can only be improved by improving data-locality in some way, or by further minimizing the number of RDMA operations. Section 3.5.3 gives suggestions on how to potentially *double* the achievable throughput. By reducing the number of roundtrips, more processes are needed to saturate the RDMA devices, but scalability remains an issue. Therefore, improving data-locality is key to support networks of many-core machines.

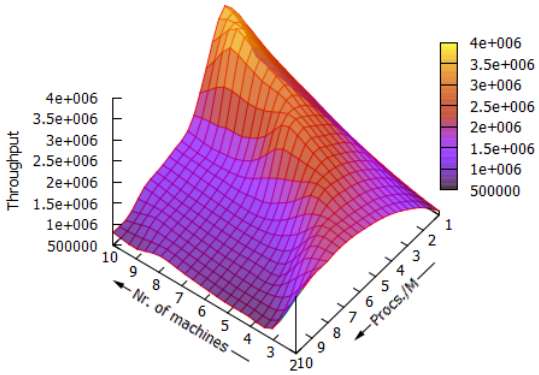
Figure 3.11 presents the speedups obtained by the remote throughput benchmarks, which look very similar to the graphs shown in Figure 3.10. The speedups are determined relative to two machines, each having one process. Again, we expect the steep slopes to become less steep when treads are dynamically created, adapting to performance differences.

3.5.2 Latency of `find-or-put`

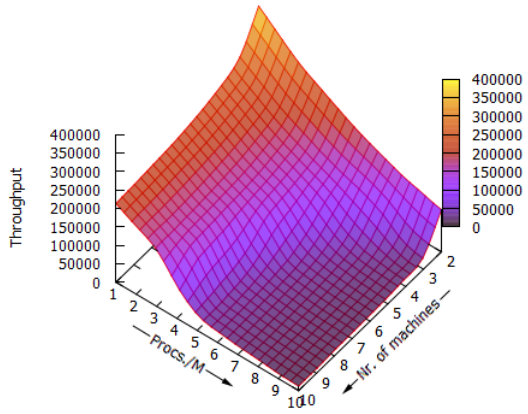
The latency of `find-or-put` is measured using various chunk sizes while increasing the load-factor α. This is done by using two processes on two machines, so that the Infiniband network is used. Each process owns a 1 GB portion of the hash table. One of the processes remains idle throughout the benchmark and the other process inserts a sequential range of integers until α reaches 0.92. This was the best we could do with the hash function we used, since `find-or-put` started to return `full` with small values of C and $\alpha > 0.92$. Hash functions with better distributions would support larger load-factors. The idle process simply exists to own a portion of remote memory,



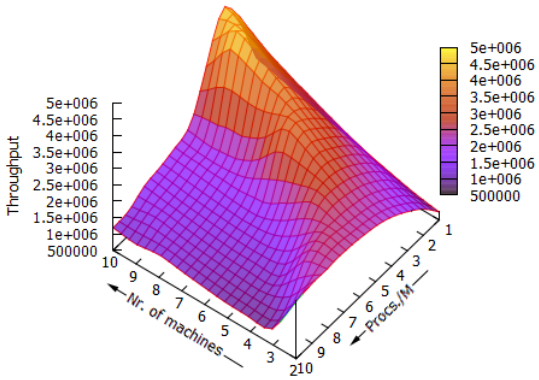
(a) Avg. throughput per process, `mixed`



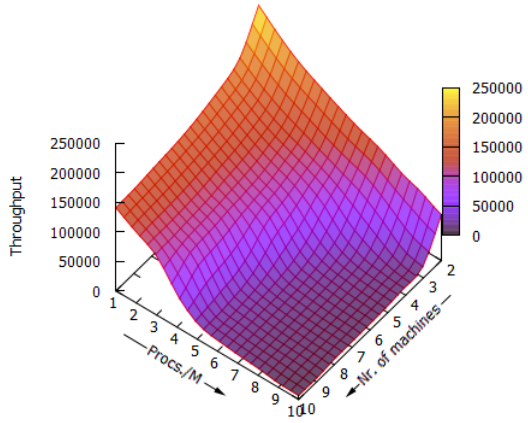
(b) Total throughput, `mixed`



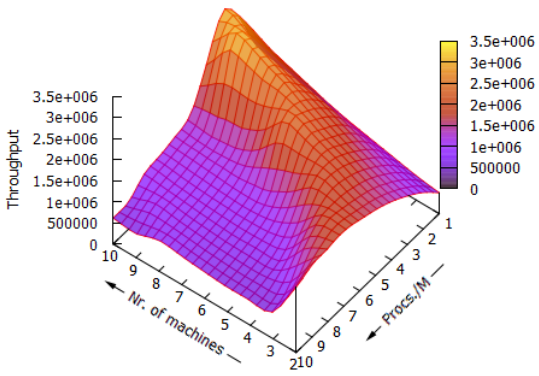
(c) Avg. throughput per process, `read-intensive`



(d) Total throughput, `read-intensive`

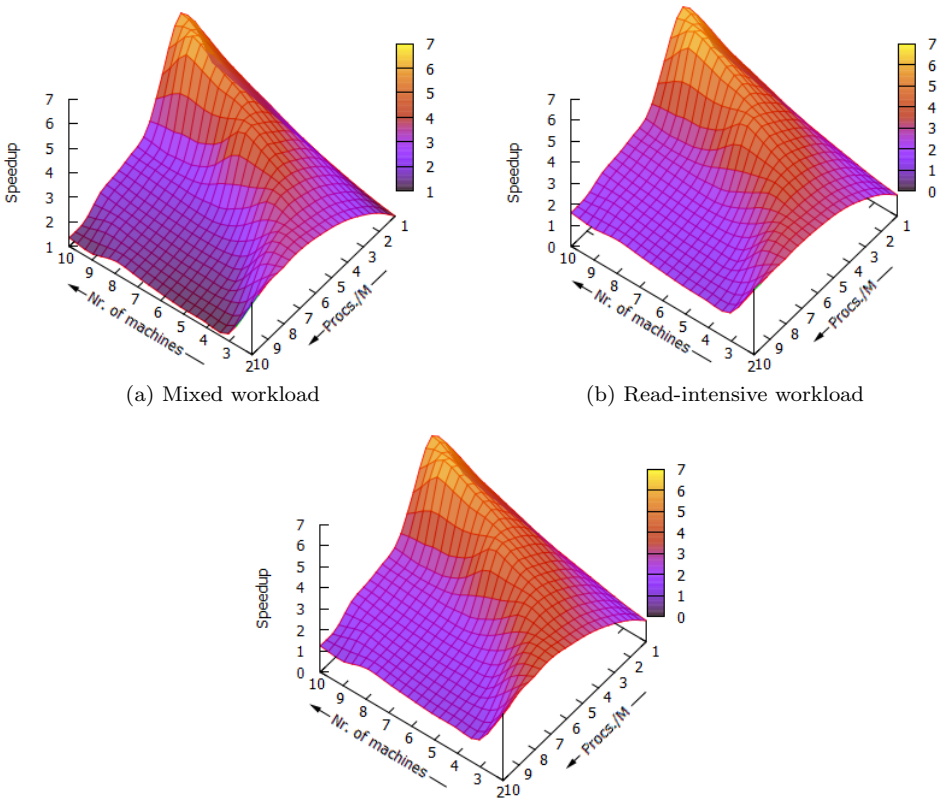


(e) Avg. throughput per process, `write-intensive`



(f) Total throughput, `write-intensive`

Figure 3.10: Average throughput per process and total throughput under mixed, read-intensive, and write-intensive workloads.



(a) Mixed workload

(b) Read-intensive workload

(c) Write-intensive workload

Figure 3.11: Speedups obtained in remote throughput under various workloads.

thus allowing RDMA to be used.

We benchmarked both local and remote latency. For the local latency benchmarks, we used only one machine, which prevents RDMA from being used.

Local Latency. Figure 3.13 shows the latency of `find-or-put` for different values of C, while increasing α. For $\alpha < 0.7$, the differences between latencies are already relatively large. Compared to 64-sized chunks, the latency is 63.5% higher on average than with 8-sized chunks. If RDMA is not used, UPC uses memcopies instead to transfer potions of local shared data in B to the private array P. By increasing C, the amount of memory to copy increases, which decreases performance.

The latency increases vastly when $\alpha > 0.7$, depending on the value C. The increase corresponds to the expected increase shown in Figure 3.1. By taking larger values of C, the average latency is higher, but the increase is lower when α gets big. Having a large value for C is beneficial when α is large. The latency of `find-or-put` on local memory is also presented in Figure 3.14 for some values of α. Also the remote latencies are given, together with the corresponding slowdowns with respect to local latencies.

Remote Latency. Figure 3.15 presents the remote latencies for `find-or-put`. For $\alpha \leq 0.5$ the differences between latencies are very small, no matter the value for C. The latency with

Workload	Base Throughput			Best Throughput			speedup
	TP.	M.	Procs./M.	TP.	M.	Procs./M.	
Mixed	592,929	2	1	3,607,003	10	3	6.08
Read	742,728	2	1	4,620,752	10	3	6.22
Write	495,370	2	1	2,999,234	10	3	6.05

Figure 3.12: The best achieved throughputs (TP.) when scaling along machines (M.) over a network and the number of processes per machine (Procs./M.).

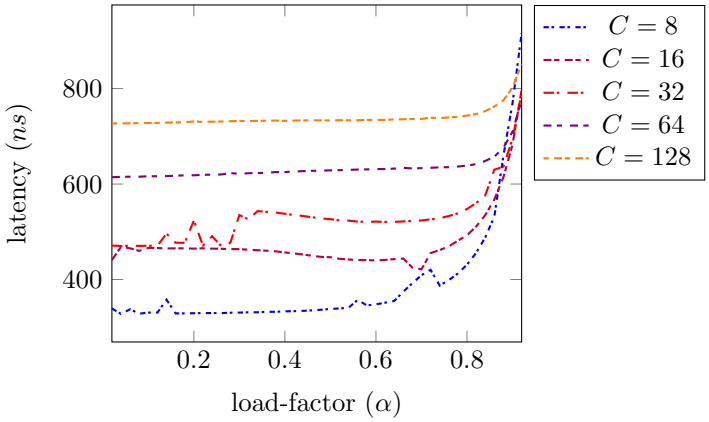


Figure 3.13: Latency of `find-or-put` *without* using RDMA (i.e. using only a single machine). Various chunk sizes C are used.

64-sized chunks is only 7.5% higher than 8-sized chunks, which shows that most overhead lies in network communication. However, when choosing large values for C, the average latencies increase vastly. To illustrate, 128-sized chunks already have 17.5% higher latency than 64-sized chunks.

When low chunk sizes are used, the average latencies increases vastly when $\alpha > 0.5$. By choosing large values for C, higher load-factors are supported, at the cost of higher latencies. Figure 3.16 shows the average number of roundtrips required to find a suitable bucket, which corresponds to Corollary 1. The required number of roundtrips directly influence the latency of `find-or-put`. The rightmost plot at Figure 3.15 shows a steeper increase than the roundtrips presented in Figure 3.16 due to asynchronous queries. When `find-or-put` is invoked, two chunks are being retrieved. As an effect, slightly higher load-factors are supported.

Figure 3.17 shows the required number of roundtrips and their expected values (based on Lemma 1) for finding suitable buckets.

3.5.3 Suggestions for Performance Improvements

Figure 3.6 shows that two queries are initially performed before the first chunk is even iterated. This causes two roundtrips (i.e. one-sided RDMA operations) to be performed, which are handled by the RDMA devices if they target remote memory. However, Figure 3.2 shows that, with a reasonable value for C, the first chunk often already contains the intended bucket. This especially holds when the load-factor is still relatively low (e.g. below 0.85, depending on the value C). We can thus argue that `find-or-put` very often performs queries that are never used. Furthermore,

Chunk size	Load-factor 0.5			Load-factor 0.8			Load-factor 0.9		
	Loc.	Rem.	SD.	Loc.	Rem.	SD.	Loc.	Rem.	SD.
8	338	7,981	23.61	431	9,602	22.28	769	15,873	20.64
16	447	8,046	18.00	493	8,746	17.74	682	11,653	17.09
32	526	8,217	15.62	548	8,446	15.41	695	9,928	14.28
64	629	8,565	13.62	639	8,645	13.53	710	9,304	13.10
128	733	9,383	12.80	743	9,413	12.67	802	9,708	12.10

Figure 3.14: The differences between local (Loc.) and remote (Rem.) latency (i.e. latencies when using one machines versus latencies over an Infiniband network). The latencies are given in nanoseconds (ns). A slowdown (SD.) of n means that the latency over an Infiniband network is n times lower than the latency using only one machine.

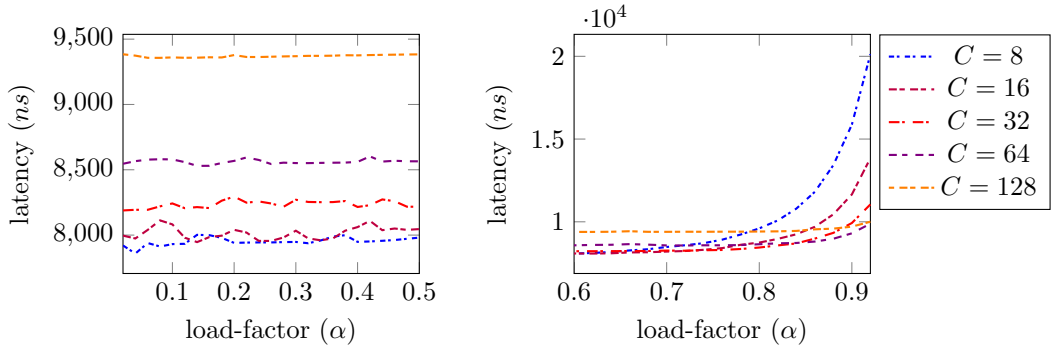


Figure 3.15: Latency of `find-or-put` over a network (i.e. using RDMA), while increasing the load-factor α. Various chunk sizes C are used.

when α gets high, the number of chunks to consider increases vastly. We argued in Section 3.5.1 that the hash table throughput is limited by the throughput of the RDMA devices, and that lowering the number of RDMA operations would increase hash table throughput.

As a suggestion to improve performance, we propose a research in adaptive chunk selection, in which the value of C is not constant, but determined dynamically. Lemma 1 gives an estimate to the value of C, that can be used if the value of α is known at runtime. This value can be approximated by all individual processes by counting the number of `inserted` returned by `find-or-put` and assuming that all other processes performed an equal number of inserts. This might be a valid assumption in many use cases, but also requires some research, since Lemma 1 assumes a universal hash function, which probably has a better distribution than most hash functions used in practice. Research has to be done in the quality of the hash distribution of those hash functions, and their deviations with respect to universal hash functions. By doing that, each worker can approximate the value of α, adjust it with the deviations found with respect to universal hash functions, and approximate a proper value for C.

We expect this suggestion to halve the number of roundtrips required by `find-or-put`, which potentially doubles the throughput of `find-or-put`. This could have a *significant* effect on the scalability of the BDD operations.

The key-value store HERD [10] only uses RDMA writes as an alternative to message passing. The CPUs continuously poll predetermined memory locations to read and handle incoming requests. This further reduces the number of RDMA operations, which results in higher through-

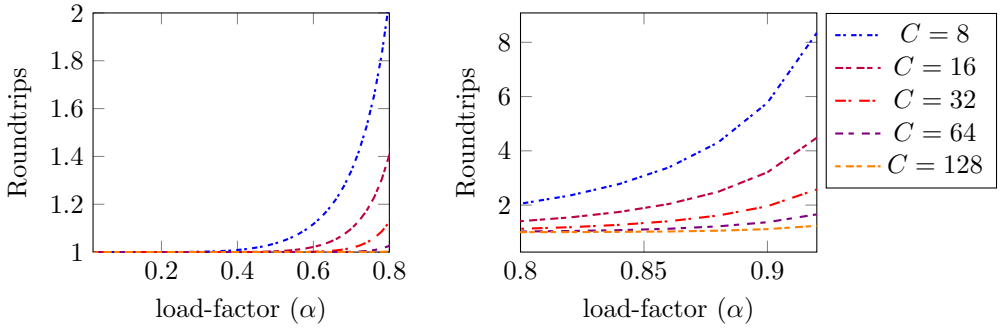


Figure 3.16: The average number of roundtrips needed by `find-or-put` to find a suitable bucket.

Chunk size	**Load-factor 0.5**		**Load-factor 0.8**		**Load-factor 0.9**	
	Expected	Actual	Expected	Actual	Expected	Actual
8	1	1.04	2	2.05	7	5.77
16	1	1.00	1	1.41	4	3.21
32	1	1.00	1	1.13	2	1.96
64	1	1.00	1	1.03	1	1.37
128	1	1.00	1	1.00	1	1.11

Figure 3.17: The expected and actual number of roundtrips performed by `find-or-put` under various load-factors.

puts, but requires a lot of CPU activity. Nonetheless, this technique might be beneficial when using fine-grained parallelism (explained in the next chapter), where relatively small computations are performed relative to the amount of memory accesses (or network communication). Since polling might be performed when waiting for roundtrips to complete, this technique might be more efficient than our technique. Furthermore, one-sided RDMA writes do not require a full roundtrip, since the remote RDMA device does not need to send replies. Also, although not supported by Berkeley UPC, the RDMA devices are able to handle RDMA operation with inline data, thus preventing the local RDMA device to perform a memory fetch over the PCI-E bus, which even further improves performance.

3.6 Conclusions

Minimizing the number of roundtrips is *critical* to increase hash table throughput. Overlapping queries via asynchronous remote memory operations reduces waiting times for roundtrips, and therefore reduces the overall latency of `find-or-put`.

Compared to techniques for resolving hash collisions used in existing work, linear probing requires *fewer* roundtrips. In addition, linear probing does not need any relocation or locking schemes. Finding an intended bucket with `find-or-put` takes $9.3\mu s$ on average with $\alpha = 0.9$ and $C = 64$. A peak-throughput of 3.6×10^6 op/s is achieved by using 10 machines. The hash table performance remains very high, even under very high loads.

Although performance does not degrade when adding machines to the network, we observed less scalability along the number of processes per machine. When adding enough processes, the RDMA devices get saturated, which *significantly* impacts performance. As future work, we

propose a number of ways to increase performance and scalability:

- Use Infiniband components that support higher bandwidths, use multiple Infiniband Channel Adapters, or both. Experimental evaluation has been performed on an 20 GB/s Infiniband network, but currently bandwidths up to 100 GB/s are supported. Also, using multiple Channel Adapters increases total throughput.
- Use adaptive chunk sizes, like described in Section 3.5.3. In most cases, `find-or-put` performs an extra roundtrip that is never actually used. Although it allows asynchronous querying, removing the need for this roundtrip reduces the number of generated RDMA operations, which increases scalability.
- Sacrificing CPU efficiency to reduce the number of roundtrips. HERD, for example, only uses one-sided RDMA writes to implement a message passing alternative, thereby minimizing the number of roundtrips. On the other hand, HERD requires continuous polling on memory locations, which means that its latency is determined on how fast the CPUs react on incoming messages. This may lead to higher latencies compared to CPU-efficient implementations. On the other hand, by minimizing the number of roundtrips, more processes per machine are supported, which in turn might lead to higher throughputs and better scalability.

Chapter 4

Hierarchical Lock-Less Private Deque Work Stealing

In this chapter, private deque work stealing is discussed and implemented. Private deques are used, as they minimize the number of roundtrips required for steals and do not use any locks or memory fences. Hierarchical work stealing is applied, so that workers prefer stealing local tasks over remote tasks, thereby reducing the number of RDMA roundtrips. The implementation is experimentally evaluated by applying it to a number of benchmarks. Moreover, the private deque implementation is compared to HotSLAW. We conclude that our implementation has more efficient steals than HotSLAW.

4.1 Introduction

The aim of *task-based parallelism* is dividing a computational problem into a number of smaller tasks, so that tasks can be performed in parallel by multiple threads [37]. By doing that, different parts of the computation can be performed simultaneously, hopefully resulting in speedup. A *task* is thus a basic unit of work and may create *subtasks* during its execution. Subtasks are created to further subdivide the computational problem and to allow parallelism. Tasks only depend on their intermediate subtasks for their execution. Tasks may synchronize on their subtasks and use their results. Every thread maintains a *task pool*. Threads do not communicate until their task pool gets empty, in which case they become *idle*. Idle threads obtain tasks from other task pools and write their status and results to predetermined memory locations, so that task owners can access them.

An advantage of task-based parallelism is that it provides the programmer with an interface to define tasks, without needing to explicitly take parallelism into account. Some other parallel programming approaches, like thread-centric programming, require the programmer to handle the complexity of performance scalability, as well as parallelism while implementing algorithms [12].

Load Balancing. The distribution of tasks to threads is called *load-balancing*. In the ideal case, the work can be split perfectly, so that all threads receive an equal amount of work. In that case, a potential speedup equal to the number of participating threads can be achieved. In practice, however, the problem size is often initially unknown and the computational work is often determined on-the-fly. During runtime, tasks generally do not know how many subtasks

they will generate, which makes it hard to partition the total work equally among threads.

By using task-based parallelism, programmers do not explicitly have to map computations to underlying hardware. Instead, tasks can be mapped dynamically, which allows dynamic workloads and support for heterogeneous hardware [12], which motivates the use of task-parallelism in our BDD operations.

Depending on the computational problem, a good speedup can be achieved by choosing a proper *task granularity*. The task granularity denotes the relation between the computational workload per task and the amount of communication required between threads. In *fine-grained* task parallelism, a large number of small tasks are generated. In *coarse-grained* task parallelism, a small number of large tasks are generated. In general, coarse-grained parallelism requires less communication, which may result in good performance. Especially when treads are distributed over a network and communication costs are high, this might be beneficial. On the other hand, fine-grained parallelism allows threads to be better utilized. By implementing cluster-based work stealing, a trade-off has to be made between computational work and the amount of communication over the network.

Work Stealing and Work Sharing. *Work stealing* is an efficient technique to implement fine-grained task parallelism. With work-stealing, idle threads try to *steal* work from task pools owned by other threads. The algorithm terminates when all threads become idle. We refer to the stealing thread as the *thief* and the targeted thread as the *victim*. Work-stealing is a form of dynamic load-balancing and is efficient since most of the computational overhead for stealing is given to the idle threads.

Work stealing is contrasted by *work sharing*. Instead of stealing work, idle threads communicate their status to other threads, so that non-idle threads can send parts of their work if needed. The advantage is that idle threads do not explicitly have to search for non-idle threads. On the other hand, the overhead for work sharing is given to non-idle threads, while the idle threads are merely waiting. Blumofe et al. [16] show that communication of work stealing is more efficient than work sharing and that work stealing has expected tight space bounds. However, work sharing might be more efficient when the number of non-idle threads is small. Wang et al. [81] describe an initial partitioning strategy in which work sharing is applied until all threads received work, followed by work stealing. When enough threads become idle again, the algorithm switches back to work sharing.

Contribution. In this chapter, we design and discuss efficient operations for cluster-based work stealing. To implement the thread pool, we use *private deques* (explained in Section 4.2), as they do not use any locks or memory fences and minimize the number of roundtrips required for steals. Victims are selected by considering the hierarchy of the network, so that thiefs prefer to steal from workers that are physically close to them. The implementation is benchmarks and the results are compared to HotSLAW.

This chapter is organized as follows. Section 4.2 gives preliminaries on deques and work stealing and lists related work. Section 4.3 presents the design of our private-deque work stealer. The design is implemented and evaluated with several common benchmarks. Section 4.4 discusses experimental results. Finally, Section 4.5 summarizes our conclusions.

4.2 Preliminaries

Many task-based parallel platforms, like Cilk [60], Wool [28], and Lace [74], use the operations **spawn**, **call**, and **sync** to spawn and execute new tasks and synchronize on tasks, respectively.

```
1 int fib(n):
2     if n < 2 return n
3     a ← fib(n − 1)
4     b ← fib(n − 2)
5     return a + b
```

```
1 int par-fib(n):
2     if n < 2 return n
3     spawn(par-fib, n − 1)
4     spawn(par-fib, n − 2)
5     r ← sync
6     return r + sync
```

Figure 4.1: A recursive implementation of Fibonacci (left) and a derived task-based implementation (right) using fine-grained parallelism.

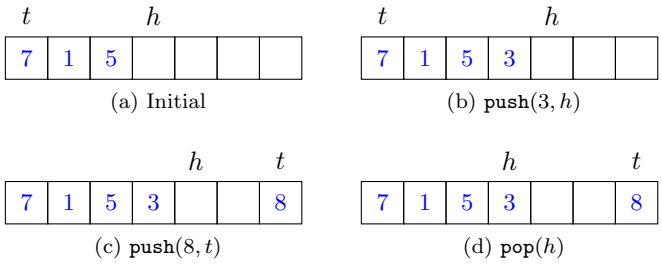


Figure 4.2: Several integers are popped and pushed on a fixed-size deque of size 7. From the starting configuration, first 3 is added to the head, then 8 is added to the tail, and finally 3 is popped of the head.

New tasks are created with the **spawn** operation. Every task must be matched with a **sync** operation, which returns the result of the matching task. Furthermore, **sync** is a blocking operation, as it blocks further execution of the worker until the corresponding task finishes. The **call** operation simply executes a task, without explicitly adding it to the deque.

Figure 4.1 shows an example of the usage of **spawn** and **sync**. A task-based parallel implementation of Fibonacci (right) is derived from the recursive implementation (left) by making tasks of every recursive call. Note that each call to **spawn** is matched by a call to **sync** like placed on a stack. So the **sync** at line 5 is matched by the **spawn** at line 4. Similarly, the **sync** at line 6 is matched by the **spawn** at line 3. Alternatively, the second **spawn** and **sync** (lines 4 and 5) could be replaced by **call** to prevent the task to be added to the queue, perhaps slightly increasing efficiency.

4.2.1 Split Deques

A *double ended queue (deque)* is similar to a normal queue, but has two ends, which we call *head* and *tail*. Items can be pushed and popped from both ends of the queue. Unlike normal queues, deques do not require a LIFO or FIFO ordering. Figure 4.2 shows an example, in which h denotes the head and t the tail. The deque is implemented with a fixed-sized array, and `push` wraps around the array if needed. Resizing would be possible when `push` finds the array full.

A split deque is a deque with a split point, which we denote by s. The split point determines what part of the deque belongs to the head and what part to the tail. Other than suggested in Figure 4.2, `pop`(p) operations are only successful when the part corresponding to p actually has items to pop. Figure 4.3 shows a split deque similar to the deque in Figure 4.2, but with a split point s. In addition, the split point s can be relocated, which is not shown in the figure.

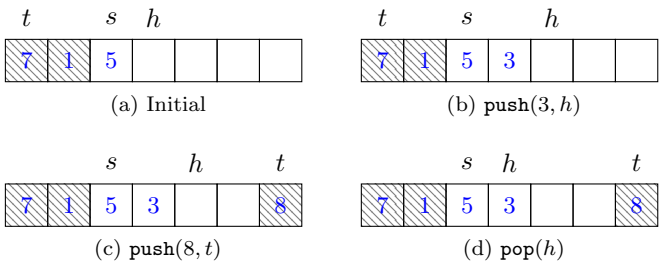


Figure 4.3: A split deque with operations similar to Figure 4.2, but with a split point s added. The dashed entries belong the tail of the split deque.

Many work stealers, including Lace, Wool, HotSLAW, and Scioto, use split deques, where the split point determines a *public* (or shared) and a *private* part of an array. Only tasks in the public region can be stolen and workers can influence the amount of stealable tasks by moving their split point, thereby making the public region either larger of smaller. For example, in Figure 4.3, the dashed entries would denote the *public* region. Work stealers do generally not push items to the tail, like done in 4.3c, but only to the head of the deque. The public region is then determined by moving s.

However, modifying the split point could conflict with steal operations performed concurrently by other workers. To solve this, locking procedures or memory fences are often used. Those operations are expensive, in particular when performed on distributed memory. In HotSLAW, all accesses to public regions are performed with locks [62], thereby resolving those conflicts. In Scioto, the whole queue is explicitly locked when attempting a steal [35] (the amount of locking is reduced in [36]). HotSLAW and Scioto are the two state-of-the-art hierarchical cluster-based work stealers, so by removing the need for locking, a considerable increase in performance can be achieved [66].

4.2.2 Private Deques

Several algorithms for private deque work stealing have been proposed by Acar et al. [71] to overcome some limitations of split deques. Private deques are used as stacks and do *not* have a public region. When stealing tasks, idle workers explicitly ask victims for work, which means that stealing is a process that involves active participation of both victim and thief. Alternatively, private deques can be seen as split deques, like in Figure 4.3, but without the dashed entries. Because the deques are entirely maintained in private memory, locking is not required. In a distributed setting, it is arguable that locking is more expensive than steals involving two workers. This is because those steals may not need to negatively affect latency, at the cost of CPU activity. When workers need to wait for roundtrips to finish, they can poll for incoming steal requests, thereby efficiently using their waiting times.

4.2.3 Selecting Victims

When stealing work, a victim must be selected. Random victim selection has stable performance [17] and tight expected space bounds [16], which makes it optimal for a large class of problems [36]. On the other hand, Dinan et al. argue that random victim selection may scale poorly on many-core clusters. Instead of performing local steals (i.e. steals from a worker on

the same machine), remote steals might be performed instead, which may unnecessarily introduce idle times. This effect increases when scaling along the number of machines, thus affecting scalability.

HotSLAW introduces *hierarchical* victim selection that considers the hardware topology as a hierarchical structure containing different domains. Workers first attempt to steal from the local domain. They perform a number of stealing attempts equal to the number of processors in that domain. If no task could be stolen, the worker moves up a domain in the hierarchy and follows the same procedure. Compared to existing implementations that use random victim selection, HotSLAW obtains better performance with hierarchical victim selection [62].

An additional strategy for victim selection is called *leapfrogging* [77], in which victims steal back from their thiefs. By doing that, the victims execute subtasks of work stolen from them, thereby helping thiefs to complete their original work. Furthermore, leapfrogging limits the space requirements of task pools to the size required for a sequential execution [29]. Wool [28] supports *transitive leapfrogging*, in which workers may steal from thiefs of their thiefs, thereby further helping thiefs to perform distributed parts of their original work. Van Dijk et al. [74] first attempts to steal from thiefs before performing random steals, which increases performance.

4.2.4 Related Work

Cilk [60] and Wool [28] are two task-based parallel frameworks that allow programs to be written in a style that is similar to sequential programs. Cilk is a compiler-based framework and Wool is library-based. Cilk attempts to steal work from the tail of the victims queue, hoping to maximize the probability of stealing the task with the maximum amount of work. Blumofe et al. [16] argue that tasks on the tail of the queue are likely to be of the largest granularity and contain half of the victims work. Both Cilk and Wool are designed for shared memory architectures.

Olivier et al. [50] (2008) present load-balancing operations for UPC which are evaluated with the UTS benchmark (created by Olivier and Dinan one year earlier). Tasks are stored on stacks. Several implementations for the steal operation are given. In one implementation, the victims stack is locked, so that a fixed-sized chunk of tasks can be claimed. The authors also described a lock-less implementation that is similar to our design. We improve on those ideas.

Scioto [35] (2008) is a task parallelism framework designed for PGAS. Scioto is implemented with ARMCI, which is an alternative PGAS implementation to OpenSHMEM and UPC. Scioto uses a locality-aware version of work stealing, in which locality is exploited by using a priority queue, so that tasks with a high local affinity are placed at the front of the queue. Steal attempts are then performed on the tail of the queue, which makes it likely for local tasks to be executed locally. One year later, Dinan et al [36] (2009) improved the split deque used in Scioto. In addition, the amount of locking on the critical path is reduced, contention is reduced, and the overhead of task creation is reduced. Furthermore, a lock-less design of the release operator on the deque is presented.

Ravichandran et al. [43] (2011) improves work stealing on multi-core clusters compared to [50] by combining the best aspects of intra-node parallelism (through shared memory) and inter-node parallelism (through distributed memory). Their algorithm increase efficiency and scalability of work stealing of large multi-core clusters.

Min et al. [62] (2011) present HotSLAW, a task library that implements hierarchical victim selection and hierarchical chunk selection. These mechanisms allow stealing various amounts of tasks, based on the distance between threads. HotSLAW is implemented in UPC.

Narang et al. [9] (2012) consider load-balancing mechanisms of existing task parallelism frameworks and present a new distributed scheduling algorithm that dynamically balances load across many-core clusters. Their algorithm achieves lower memory utilization than HotSLAW and bet-

ter performance and parallel efficiency than Charm++ [42], which is a machine-independent parallel programming model. The authors extended their work in 2013 [13] by implementing parameter tuning via machine learning. The parameters include: remote spawn rate, work-stealing rate, task granularity, and process grouping.

Ramesh et al. [59] (2014) reduces processing time and increases throughput of processors in a distributed environment with a new load-balancing mechanism. This mechanism balances load based on CPU utilization. Processors are placed in a priority queue based on their percentage of CPU utilization. These priorities are used for the victim selection protocol.

Palirria [76] (2014) is a work-stealing scheduling method for nested fork/join parallelism. Instead of random victim selection, Palirria uses alternative scheduling approaches to make efficient use of resources without degrading performance. Those approaches include a deterministic victim selection method and a resource estimation algorithm, which replaces the victim selection policy.

Lace [74] (2015) is a task-based parallel C library that uses non-blocking concurrent split deques, but also comes with a private deque implementation. Moreover, Lace uses leapfrogging, followed by random victim selection, which improves scalability. Like Cilk and Wool, also Lace is designed for shared memory architectures.

4.2.5 Motivation and Contribution

Our goal is to design a simple and efficient work stealing platform that minimizes the number of roundtrips, as well as blocking times due to synchronizations. The implementations of both Scioto and HotSLAW are based on deques, which require locking mechanisms. Especially in distributed environments, locking is very expensive. Our design is based on private deques, which are lock-less and thereby achieve better performance.

Moreover, implementing private deques with PGAS is arguably easier than implementing split deques. This is because private deques are maintained *solely* by their owners, which implicitly resolves concurrency issues. The public regions of split deques, on the other hand, are accessible to *any* worker. As an effect, split deques need to explicitly resolve concurrency issues, either via locking or with the use of atomic operations.

Acar et al. [71] shows that local portions of deques require memory fences and that steal-many policies are difficult to implement in deques. The authors solved those limitations by using private deques. Van Dijk et al. [74] presents concurrent deques that also solve those limitations. Because of time constraints, we did not attempt to implement the concurrent deque in PGAS. We designed distributed private deques, since they are relatively easy to implement, do not require locking, and potentially increases performance.

To the best of our knowledge, two other private deque work stealing platforms have been proposed. Firstly, the work stealing algorithms proposed by Acar et al. [71] use private deques to prevent locking, but are not designed nor optimized for cluster-based uses. Secondly, Olivier et al. [50] present a number of work stealing implementations, including a private deque variant, which is implemented in Berkeley UPC. Our implementation, however, requires fewer roundtrips. Finally, in contrast to our implementation, both platforms do not use hierarchical work stealing nor leapfrogging.

4.3 Design and Implementation

This section discusses the design of our work stealing framework, which uses private deques, hierarchical victim selection, and leapfrogging. The framework only supports a steal-1 policy. We attempted to implement both a steal-many strategy and hierarchical chunk selection [62] (i.e.

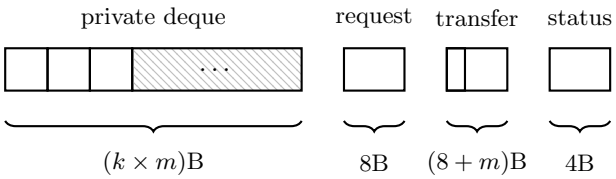


Figure 4.4: The memory layout (per process) of our implementation. The different memory blocks and their respective sizes (in Bytes) are shown. Here, k denotes the length of the private deques and m the size (in Bytes) of a single task.

a steal-many policy), but failed to complete it in the time available. Instead, we regard these extensions as future work. Related work shows that those extensions may significantly increase scalability and speedup, depending on the computational problem [35, 36, 50, 62]. In Section 4.4 shows the speedups obtained by HotSLAW with a Steal-10 policy, compared to Steal-1.

4.3.1 Memory Layout

In order to simplify our design, we assume that each task has a fixed number of parameters. In addition, we assume that each task returns a 64-bit integer. These assumptions are not enough to make a generic work stealing platform, like HotSLAW and Lace, but is enough to support BDD operations. As future work, we would like to extend our work stealer to enable tasks to have a various number of parameters and various types of return values. HotSLAW, for example, enables tasks to provide input and output via pointers, thereby supporting variable input and output sizes. Our design can be extended to support the same functionality.

Tasks are implemented with a `struct` and have three components. First, they have a 64-bit integer that stores a bit to denote whether the task is stolen, a task *descriptor*, and a task *owner*. The descriptor is used to find and invoke the function corresponding to the task. The task owner is used to implement leapfrogging, as it also stores the identifiers of thiefs. Secondly, the parameters of the task are stored, which requires a block of memory equal in size to the total size of the parameters. Finally, a 64-bit is reserved for the return value. When a task is stolen, the thief writes its result to this location.

4.3.2 The Stealing Procedure

In our design, every worker allocates a private deque, a request cell, a transfer cell, and a status cell in *shared* memory, so that every worker can access them. Figure 4.4 gives a schematic overview of this set-up. Tasks are stored in the private deques, supporting a maximum of k tasks. When a worker needs to perform a steal, it chooses a victim and writes its thread ID to the victims request cell by using `cas`.

The victim polls the request cell regularly (ideally, as often as possible to minimize latency). When it finds the request cell occupied, it checks if the private deque contains enough tasks to perform a steal. If so, it marks the task at the tail as stolen and writes the entire task *and* its index (i.e. the corresponding index in the deque) to the transfer cell of the thief. Figure 4.4 shows that the transfer cell contains 64-bits for a task index and m bits for the task itself.

After making a successful request with `cas`, the thief continuously polls the transfer cell for a response from the victim. If it receives a response, it executes the task and writes the result back to the victims private deque by using the received index.

Workers can have two statuses, namely IDLE and WORKING. When a worker runs out of tasks, it becomes IDLE. Workers maintain their status via the status cell, shown in Figure 4.4. Because this cell is placed in shared memory, other workers may access it, which is done to implement termination detection. When a worker becomes IDLE and performed enough failed steal attempts, it reads the status cells of all other workers via collective one-sided RDMA operations. When all cells contain the status IDLE, the worker terminates. Otherwise, when at least one worker is WORKING, it continues performing steal attempts.

While designing the work stealing operations, we did not consider to maintain any failed steal attempts, which might in fact be a good idea. Especially after a failed termination detection attempt, which is only invoked when enough workers are assumed to be IDLE, knowing what workers are non-idle might help with selecting victims. This may prevent additional failed steal attempts, thereby improving execution times and potentially speedups. This is something to consider in future work.

Required Number of Roundtrips. Steal attempts require a blocking remote `cas` operation to update the request cell. The victim writes a task to the thiefs transfer cell via a non-blocking one-sided write. After having completed a stolen task, the thief writes the result back to the thiefs private deque via a non-blocking one-sided write. To conclude, the whole procedure of stealing only requires one blocking remote memory operation, which is performed by the already idle worker. The non-blocking operations are handled by the RDMA devices (instead of the CPUs), which requires the victims to continue their work while performing them.

4.3.3 Algorithmic Design Considerations

We now explain the steps given in Section 4.3.2 in more detail and present the work stealing operations. When considering Figure 4.4, we denote the private deque by the two-dimensional shared array $\mathsf{deque}[0][0], \ldots, \mathsf{deque}[\texttt{THREADS} - 1][k-1]$. We assume that each worker i owns the block $\mathsf{deque}[i][0], \ldots, \mathsf{deque}[i][k-1]$, so memory operations performed on this block do not use RDMA, since it is local memory. Here THREADS denotes the number of participating threads.

We chose to use a consecutive two-dimensional array, because it makes the implementation easier. Every worker i has its own private deque $\mathsf{deque}[i]$, and when it completes a task stolen from worker j, it can write back its result to $\mathsf{deque}[j]$, as it is shared. In other words, the private deques are shared, but they allow other workers to write back the return values of stolen tasks.

The request cells are denoted by $\mathsf{request}[0], \ldots, \mathsf{request}[\texttt{THREADS} - 1]$. Similar to the deque, we let each worker i own the entry $\mathsf{request}[i]$, so that memory operations on that entry do not use RDMA. Request cells can have three different values, namely BLOCKING, EMPTY, or the identifier of a worker. When a cell is BLOCKING, no tasks can be stolen from that worker. When it is EMPTY, tasks can be stolen, but there are no pending requests. When it contains an identifier, the corresponding worker has requested a steal, which is not yet handled by the victim (i.e. pending).

The transfer cells are denoted by $\mathsf{transfer}[0], \ldots, \mathsf{transfer}[\texttt{THREADS} - 1]$. Each worker i owns the entry $\mathsf{transfer}[i]$, thereby holding it in local memory. Transfer cells can have two different values, namely EMPTY, or they contain a task.

Finally, the status cells are denoted by $\mathsf{status}[0], \ldots, \mathsf{status}[\texttt{THREADS} - 1]$. Again, each worker i owns the entry $\mathsf{status}[i]$ in local memory. Status cells can have two different values, namely IDLE and WORKING. During termination detection, a worker checks if all status cells are IDLE.

Handling Incoming Requests. Suppose that a worker performs a steal. Then it chooses a

```
1  def communicate():
2    if head − tail < 2
3      if request[MY-ID] ≠ BLOCKED
4        ▷ Not enough stealable tasks, block further requests
5        if request[MY-ID] ≠ EMPTY
6          reject-and-block()
7        elif cas(request[MY-ID], EMPTY, BLOCKED) ≠ EMPTY
8          reject-and-block()
9    elif request[MY-ID] = BLOCKED
10     ▷ Unblock requests, there are stealable tasks
11     request[MY-ID] ← EMPTY
12   elif request[MY-ID] ≠ EMPTY
13     ▷ Send a task to the thief
14     thief ← request[MY-ID]
15     request[MY-ID] ← EMPTY
16     ▷ Prepare task to be stolen
17     deque[MY-ID][tail].stolen ← true
18     deque[MY-ID][tail].owner ← thief
19     ▷ Construct the transfer message
20     msg ← new TransferMessage
21     msg.index ← tail
22     msg.task ← deque[MY-ID][tail]
23     ▷ Write message to the transfer cell of the thief
24     memput-async(transfer[thief], msg)
```

Figure 4.5: The implementation of `communicate`, which is responsible for maintaining the request cell of a worker.

victim and writes its ID (denoted by MY-ID) to the request cell of the victim. Figure 4.5 presents the `communicate` function that is responsible for handling incoming requests. The operation first checks if there are any tasks to steal (line 2). If the worker has less than two non-stolen tasks in its deque, it blocks further requests by writing BLOCKED to its request cell (lines 5 to 8). By blocking further requests, it may happen that some other worker requested work and is waiting for a response. In that case, `reject-and-block` is invoked, which is an operation that writes BLOCKED to request[MY-ID] and notifies the requesting worker that there is no work to steal. Its implementation is presented in Figure 4.6. If the request cell is empty, the worker attempts to write BLOCKED to it with `cas` (line 7).

If there are at least two stealable tasks, and the request cell contains BLOCKED, then the worker allows new requests by writing EMPTY to request[MY-ID] (line 11).

Otherwise, when there is an incoming request, the ID of the thief is obtained (line 14), the task at the tail of the deque is marked as stolen (lines 17 and 18), and the task is sent to the thief (lines 20 to 24).

Handling Steals. Figure 4.6 shows the implementation of `steal`, which takes the ID of a victim as argument. This operation performs a steal attempt, which returns `true` when the steal succeeded and `false` when it failed. After the transfer cell has been cleared (line 3), a request is sent via a `cas` operation (line 4). The `cas` operation tests if the request cell of the victim is EMPTY, in which case the ID of the thief is written to it. If the request cell was indeed empty (the

```
1  def steal(victim):
2    communicate()
3    transfer[MY-ID] ← EMPTY
4    res ← cas(request[victim], EMPTY, MY-ID)
5    if res = EMPTY
6      ▷ Wait for response from victim
7      while transfer[MY-ID] = EMPTY:
8        communicate()
9      if transfer[MY-ID] = EMPTY
10       return false
11     else
12       status[MY-ID] ← WORKING
13       i ← transfer[MY-ID].index
14       ▷ Execute the stolen task
15       task ← transfer[MY-ID].task
16       task.result ← call(task)
17       ▷ Write back the task result
18       task.completed ← true
19       memput-async(deque[victim][i], task)
20       return true
21   return false
```

```
1  def reject-and-block():
2    ▷ Block further requests
3    thief ← request[MY-ID]
4    request[MY-ID] ← BLOCKED
5    ▷ Send a negative response
6    msg ← new TransferMessage
7    msg.index ← 0
8    msg.task ← EMPTY
9    memput-async(transfer[thief], msg)
```

```
1  def termination-detection():
2    ▷ Send status requests
3    for i ← 0 to THREADS − 1:
4      s_i ← memget-async(
5          res[i], status[i])
6    ▷ Wait for status responses
7    for i ← 0 to THREADS − 1:
8      sync(s_i)
9      if res[i] = WORKING
10       return false
11   return true
```

Figure 4.6: Implementations of `steal`, `reject-and-block`, and `termination-detection`. The `reject-and-block` operation is used by `communicate` to give a thief a negative response (i.e. no tasks to steal) and block further steal attempts. The `steal` operation steals from a given victim by sending a steal request, waiting for a response, executing the received task, and writing back the result. The `termination-detection` operation returns `true` only if all workers have status `IDLE`.

check at line 5), it waits for a response (line 7), while handling incoming requests (line 8). When receiving a negative response (line 9), `false` is returned. When receiving a positive response, the worker updates its status to `WORKING` and executes the received task (lines 12 to 16). After having completed the task, its result is written back in a non-blocking manner (lines 18 to 20).

Compared to the private deque implementation of Acar et al. [71], our implementation is optimized for one-sided networking operations. For example, workers with no tasks to steal set their request cell to `EMPTY`, instead of maintaining a separate array for that. As a result, steals require fewer roundtrips. Furthermore, we store the identifiers of thiefs to perform leapfrogging.

Spawning and Calling Tasks. The implementations of **spawn** and **call** are given in Figure 4.7. These implementations are rather straightforward. By invoking **spawn**, a new task is created (lines 3 to 7) and inserted at the head of the deque (line 9). After that, the value of *head* is increased. By invoking **call**, first `communicate` is invoked (line 2), so that workers waiting for response do not have to wait for the function call to complete. After that, the function corresponding to the task is obtained (line 4) and invoked (line 6). This function is described with *desc*.

Synchronizing on Tasks. The implementation of **sync** is also given in Figure 4.7. This operation reads the topmost task on the deque (line 2). If that task is not stolen, **sync** executes it (line 21). Otherwise, it waits for the task to complete (lines 5 to 14). During that time, the

```
1  def sync():
2      task ← deque[MY-ID][head − 1]
3      if task.stolen
4          communicate()
5          while ¬task.completed:
6              ▷ Perform leapfrogging
7              if steal(task.owner) continue
8              if task.completed break
9              ▷ Perform hierarchical stealing
10             for i ← 0 to HIERARCHY-LVLS - 1:
11                 shuffle(domain[i])
12                 foreach victim ∈ domain[i] do
13                     if steal(victim) goto line 5
14                     if task.completed goto line 16
15         ▷ Return result from stolen task
16         head ← head − 1
17         tail ← tail − 1
18         return task.result
19     else
20         head ← head − 1
21         return call(task)
```

```
1  def spawn(desc, params):
2      ▷ Build a new task
3      task ← deque[MY-ID][head]
4      task.desc ← desc
5      task.stolen ← false
6      task.completed ← false
7      task.params ← params
8      ▷ Write new task to deque
9      deque[MY-ID][head] ← task
10     head ← head + 1
```

```
1  def call(desc, params):
2      communicate()
3      ▷ Find the intended function
4      func ← function-of(desc)
5      ▷ Invoke that function
6      return func(params)
```

Figure 4.7: Implementations of **spawn**, **call**, and **sync**. The **spawn** operator adds a new task to the deque, **sync** pops a task from the deque and executes it (or, when stolen, waits for its completion), and **call** executes a task without explicitly adding it to the deque.

worker tries to steal from other workers. By doing that, it first attempts leapfrogging (line 7), followed by hierarchical work stealing (lines 10 to 14). When one of the steals succeeds, the stealing process starts again by jumping to line 5. When the task completes, its result is returned (line 18).

We assume that *domain*[*i*] contains all workers on the *i*th level in the processing hierarchy. Berkeley UPC supports four of those levels, which they call `verynear`, `near`, `far`, and `veryfar`. We use these levels, which means that `HIERARCHY-LVLS` is 4. We use the `shuffle` function to randomly shuffle the workers in *domain*[*i*] in an efficient manner. If the number of workers gets *very* high (for example, 8192 or higher), it might be more efficient to simple pick workers randomly from *domain*[*i*]. Because **sync** is often invoked recursively, the order of workers *domain*[*i*] remains random. As an effect, workers attempt a number of random steals equal to the amount of workers on that level, like described by [62].

Starting a Computation. Computational tasks can be started by invoking `initiate` and `participate`, whose implementations are shown in Figure 4.8. One of the participating processes calls `initiate` and *all* others call `participate` as a collective process. This is also done by the collective function `compute`, where the process with `MY-ID` 0 initiates the computation and all others participate. Collective functions require the collaboration of all processes.

Both `initiate` and `participate` start by writing their process status (line 3) and waiting for all processes to have written their status (line 4). The initiating process executes the first task, so its status becomes `WORKING`. All other processes need to steal tasks before they can start working, which makes them `IDLE`. The `barrier` function is a collective operation used to

```
1  def initiate(desc, params):
2      ▷ Wait for all workers to start
3      status[MY-ID] ← WORKING
4      barrier()
5      ▷ Perform task
6      result ← call(desc, params)
7      ▷ Wait for all workers to complete
8      status[MY-ID] ← IDLE
9      barrier()
10     return result

1  def compute(desc, params):
2      if MY-ID = 0
3          initiate(desc, params)
4      else
5          participate()
```

```
1  def participate():
2      ▷ Wait for all workers to start
3      status[MY-ID] ← IDLE
4      barrier()
5      ▷ Perform hierarchical stealing
6      while true:
7          status[MY-ID] ← IDLE
8          for i ← 0 to HIERARCHY-LVLS - 1:
9              shuffle(domain[i])
10             foreach victim ∈ domain[i] do
11                 if steal(victim) goto line 5
12         ▷ No worker had tasks to steal..
13         if termination-detection()
14             break
15     barrier()
```

Figure 4.8: Implementations of `initiate`, `participate`, and `compute`. To start a computation, one workers calls `initiate` and all other workers call `participate`. Alternatively, a computation is started via the collective function `compute`.

synchronize on processes.

After having written the status cells, the initiating process **calls** the first task (line 6) and the participating processes start performing hierarchical work stealing (lines 6 to 11), very similar to the implementation of **sync** (Figure 4.7). When all steals failed, termination detection is invoked (line 13). If `termination-detection` returns a positive result, the while-loop at line 6 breaks and the process finishes by waiting for all other processes to complete (via a call to `barrier` at line 15).

Termination Detection. Figure 4.6 presents the implementation of `termination-detection`, which gives a positive result only if all processes are IDLE. The operation assumes that an array $res[0], \ldots, res[\texttt{THREADS} - 1]$ exists on every worker in private memory. Termination detection starts by asynchronously requesting the statuses of all processes (lines 3 to 5), where THREADS is a constant denoting the total number of processes. After that, the requests are synchronized on (line 8). If the received status equals WORKING, then at least one process is not IDLE and the operation returns a negative result (line 10).

4.4 Experimental Evaluation

We implemented the private deque work stealing algorithm in Berkeley UPC, version 2.20.0. The performance is evaluated by measuring scalability using various benchmarks. We compared our implementation with HotSLAW, which is a hierarchical task-based parallel framework also implemented and compiled in Berkeley UPC, version 2.20.0. Both frameworks have been compiled with the command `upcc -network=ibv`. All experiments have been performed in a cluster of 10 Dell M610 machines, each having 24 GB of internal memory and 8 CPU cores. All machines run Ubuntu 14.04.2 LTS with kernel version 3.13.0 and are connected via a 20 GB/s Infiniband cluster. All experiments have been repeated at least three times and the average measurements have been taken into account.

Benchmark	Nr. of Tasks	Avg. Task Time	Input/Output Size	Input/Output Size Hotslaw
`fib`(45)	3,672,623,805	0.154 μs	16/8 bytes	4/8 bytes
`nqueens`(15)	171,127,071	4.14 μs	20/8 bytes	28/8 bytes
`uts(T2L)`	96,793,510	0.986 μs	20/8 bytes	32/0 bytes
`uts(T3L)`	111,345,631	0.722 μs	20/8 bytes	32/0 bytes
`matmul`(512)	32,767	188.30 μs	20/8 bytes	28/0 bytes

Figure 4.9: Details of the models used to evaluate the work stealing implementation.

To make a fair comparison between our work stealing implementation and HotSLAW, the same benchmarking code is used for both implementations. HotSLAW has built-in support for pointers as arguments to tasks. When a task is stolen in HotSLAW, the pointers given to that task are updated accordingly. Our implementation does not support pointers. We solved this by giving each participating process a *shared stack*. Shared stacks are simple stacks implemented on top of shared memory. Tasks may then push data on their stack and use the corresponding shared pointers to that data. When the task is stolen, the shared pointers remain valid, even on another machine, and can be used to retrieve the corresponding data.

The benchmarks are executed by using `upcrun -n` processes `-N` machines `-shared-heap 128MB`, where we replaced processes with the total number of processes and machines with the total number of participating machines. For example, when using 7 machines, each having 4 processes, processes becomes 28 and machines becomes 7. In each experiment, we scaled the number of machines from 1 to 10 and the number of processes per machine from 1 to 10. So a total of 100 different configurations are benchmarked with a maximum of 100 processes. By using the `-shared-heap` flag, each process is allowed to allocate up to 128 MB of shared memory, which is more than enough for a work stealing queue.

4.4.1 Benchmarks

Figure 4.9 shows details of the models used in the evaluation of the work stealing implementations. During the evaluation, we did not apply a cut-off in the computation trees, as we are interested in the overhead generated by the work stealing algorithms. The `fib`(n) benchmark calculates the nth Fibonacci number by creating subtasks for `fib`($n-1$) and `fib`($n-2$) and synchronizing on their results. The computation tree of `fib` is skewed, which makes it non-trivial to parallelize.

The `nqueens`(n) benchmark calculates the number of unique solutions for placing n queens on an $n \times n$ chessboard, where n is a positive integer. Every task has a depth $j \leq n$ and tries to place a queen on every position at row i. If a queen can validly be placed, a subtask is generated for depth $i+1$. Then all valid boards on depth n are counted. The `nqueens`(15) problem, for example, has $2,279,184$ unique solutions.

The `uts` benchmark [66] is designed to evaluate the performance of parallel programs that require dynamic load-balancing. This is done by performing an exhaustive search on an unbalanced tree, which is generated on-the-fly with a random number generator. We used SHA-1 to generate the random numbers. UTS supports two types of unbalanced trees, namely geometric trees (e.g. `T2L`) and binomial trees (e.g. `T3L`). The geometric trees have a branching factor that follows a geometric distribution [66], which appears to be easy to balance in practice [74]. The binomial trees have parameters m and q, so that nodes have m children with probability q and no children with probability $1-q$ [66]. This causes trees to have unpredictable sub-tree sizes and depths, which makes them harder to balance compared to the geometric trees.

The `matmul(n)` benchmark calculates the product of two random $n \times n$ matrices, with n a

Benchmark	Sequential	Best Configuration			speedup
	Time	Time	Machines	Procs./M.	
`fib(45)`	563.87	8.85	10	8	63.69
`matmul(512)`	6.17	1.07	1	9	5.76
`uts(T2L)`	90.48	1.90	10	8	47.62
`uts(T3L)`	73.60	3.48	10	5	21.15
`nqueens(15)`	707.64	10.30	10	8	68.74

Figure 4.10: Sequential computation times of our work stealing implementation, together with the best achieved times and corresponding speedups.

positive integer. The two matrices, as well as the resulting matrix, must be placed in shared memory. Because of that, every task needs to perform a number of one-sided RDMA operations during its execution, which may negatively affect performance and scalability. Compared to the other benchmarks, `matmul` generates a relatively low number of tasks with a relatively large workload (coarse-grained).

4.4.2 Experimental Results

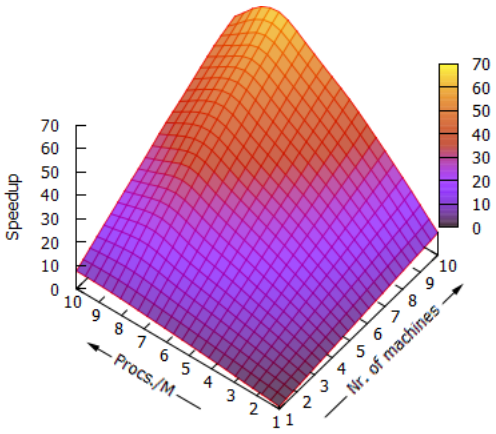
Figure 4.11 shows the speedups obtained in the different benchmarks. The sequential execution times and the best achieved parallel times are shown in Figure 4.10. Both `fib(45)` and `nqueens(15)` obtain high speedups, namely 63.69 (efficiency of 79.6%) and 68.74 (efficiency of 85.9%), respectively. When more than 8 processes per machine were used, however, the performance degraded. Recall that the machines used in the experiments only have 8 physical CPU cores. Since both benchmarks have a very large amount of very small tasks (Figure 4.9), and only require a relatively low number of steals (Figure 4.15), performance degrades when more than 8 processes are used.

The `uts(T3L)` benchmark achieves a maximum speedup of 21.76 (efficiency of 43.5%) when 50 processes are used. This speedup is less significant than `fib(45)` and `nqueens(15)`, because significantly more steals are required. Those steals all require multiple roundtrips, which affects scalability. In contrast to `fib(45)` and `nqueens(15)`, performance seems to slightly increase when more than 8 processes per machine are used, perhaps because the queues on the RDMA devices are less polluted.

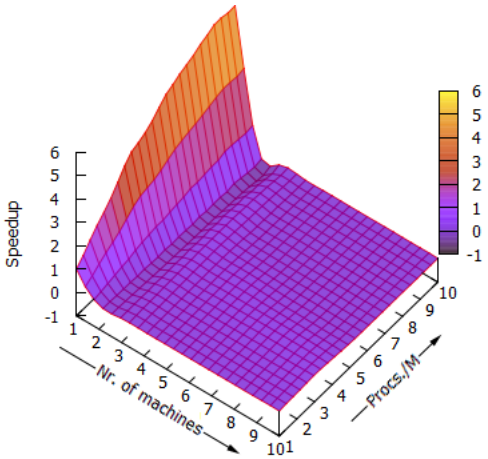
The `uts(T2L)` benchmark scales well even when more than 8 processes per machine are used. We expect this to be caused by the balance between task workload (about $1\mu s$ computation time per task) and the number of steals required (Figure 4.15). The benchmarks that require less steals (like `fib(45)` and `nqueens(15)`), allow more CPU activity, which introduces performance degradations with more than 8 Procs./M. The ones that require more steals (e.g. `uts(T3L)`) causes network latency to become the bottleneck.

The `matmul(512)` only achieves speedup when one machine is used. In the `matmul` benchmark, two random matrices are multiplied, which are placed in shared memory. When using one machine, both matrices and the resulting matrix all reside in local memory. When more than one machine is used, the matrices are partitioned over the machines. As an effect, every task performs many roundtrips to access the matrices, which results in serious performance degradations. No speedups are achieved when more than two machines are used. Therefore, `matmul` is not considered in the further evaluation.

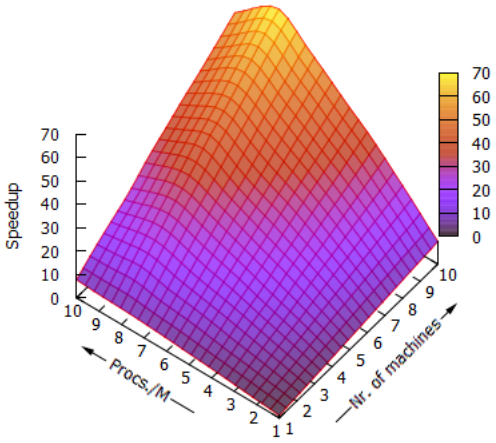
A Comparison with HotSLAW. We compared the computation times and the speedups of our work stealing implementation with HotSLAW. Figure 4.13 shows a comparison of compu-



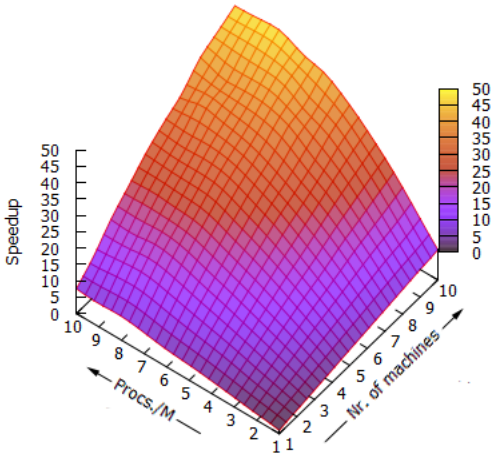
(a) Speedups of `fib(45)`



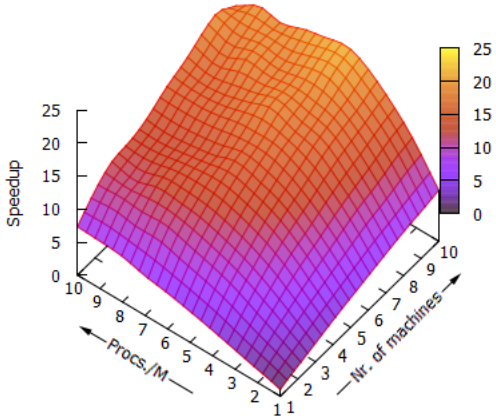
(b) Speedups of `matmul(512)`



(c) Speedups of `nqueens(15)`



(d) Speedups of `uts(T2L)`



(e) Speedups of `uts(T3L)`

Figure 4.11: Speedups obtained in various benchmarks by scaling along machines and the number of processes per machine (Procs./M).

Benchmark	Our Implementation			HotSLAW		
	Seq. Time	Best Time	Efficiency	Seq. Time	Best Time	Efficiency
`fib(45)`	563.87	8.85	79.6%	938.49	13.13	89.3%
`nqueens(15)`	707.64	10.29	86.0%	387.53	5.68	85.3%
`uts(T2L)`	90.48	1.90	59.5%	81.22	1.53	53.2%
`uts(T3L)`	73.60	3.48	42.3%	67.64	5.48	24.7%
`uts(T3L)`*	73.60	3.48	42.3%	51.69	1.32	49.0%

Figure 4.12: A comparison of total execution times between our implementation and HotSLAW. The sequential computation times and the best achieved computation times with their corresponding efficiencies are given. The HotSLAW implementations running the benchmarks with a * are using a Steal-10 policy.

tation times. A speedup of n means that our implementation performs n times faster (i.e. the computation time was n times less than HotSLAW). Figure 4.12 shows a corresponding table with best achieved execution times.

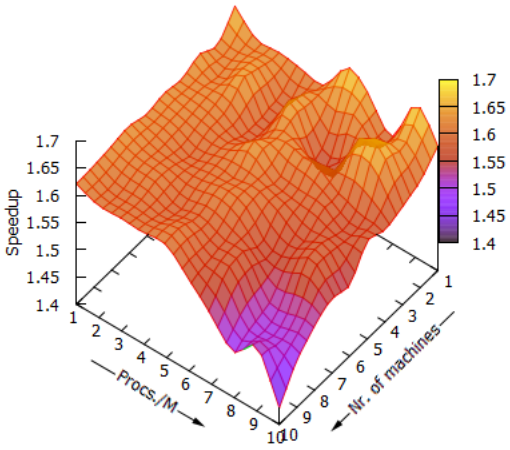
Both `fib(45)` and `uts(T3L)` (with Steal-1) perform better with our implementation. The two work stealers perform about equally well for `uts(T2L)`, and HotSLAW clearly performs better with `nqueens(15)` and `uts(T3L)` (Steal-10). Figure 4.12 shows that, with the exception of `fib(45)`, HotSLAW computes tasks more efficiently. This is because HotSLAW supports the use of pointers in tasks, which our implementation does not. Most benchmarks (with exception of `fib(45)`) require such pointers. We solved this by creating a *shared stack*, so that other processes can access the parameters of a task after stealing it, but this clearly is less efficient than HotSLAW. As future work, it would be interesting to improve the execution times of tasks by only using shared data structures when a task is stolen. This can even be done more efficiently than HotSLAW, because all steal requests are handled by victims, so victims have an influence in the data they share.

Figure 4.12 also shows the efficiencies of the work stealers under their fastest configurations (i.e. the settings in which they obtained best performance). The efficiency denotes the speedup relative to the number of processes used (i.e. a efficiency of 100% means a speedup of n when n processes are used). In three of five cases, our implementation obtains better efficiency. Figure 4.14 shows that `fib(45)` scales better in HotSLAW when a large number of processes are used (7 machines with 7 processes each, or more). Furthermore, `UTS(T3L)` with Steal-10 is more efficient under HotSLAW, compared to our implementation with Steal-1. We expect that Steal-10 simply works good with `UTS(T3L)`, which would explain the differences in efficiency.

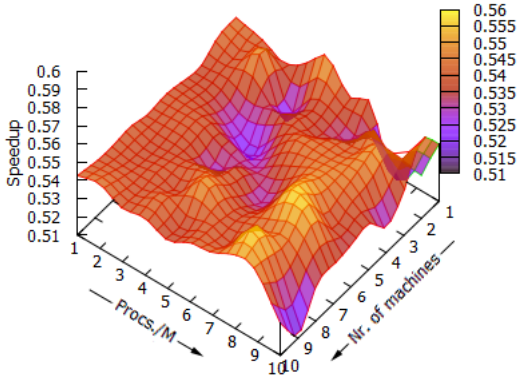
Figure 4.14 shows that our implementation scales about equally well compared to HotSLAW, with the exception of `UTS(T3L)`. When using Steal-1, `UTS(T3L)` scales better with our implementation, and HotSLAW scales better when Steal-10 is used.

The Latency of Steals. With *stealing latency* we mean the time *between* performing a steal request and getting a response from the (potential) thief. Figure 4.16 shows the latencies of stealing tasks, both locally and remotely. For these measurements, the `uts(T3L)` benchmark is used due to its many steals. When one machine is used, the latency of remote steals is obviously 0, since RDMA is not used. When the number of machines increases, the latency of remote steals also increases. When 100 processes are used, a peak-latency of $30,6\mu s$ is reached. When less than 45 processes are used, the latencies lie between $5,2\mu s$ and $8,0\mu s$.

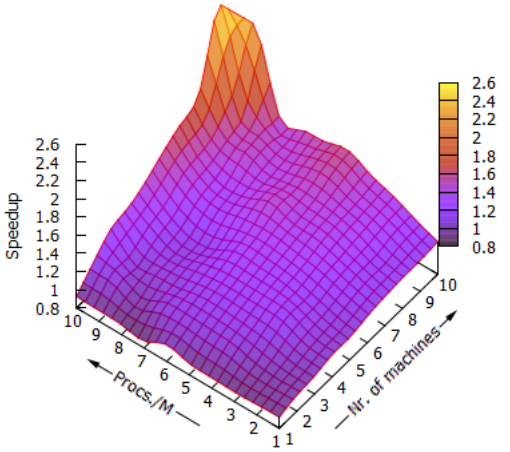
Local steals (shown in Figure 4.16c) are about 10 times faster than remote steals. When using one process per machine, only remote steals are performed, as a process does not steal



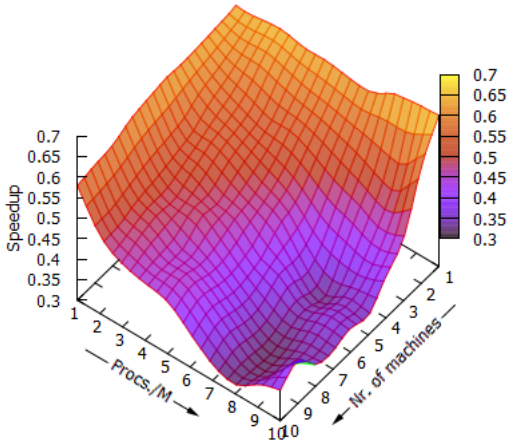
(a) Speedups of `fib(45)`



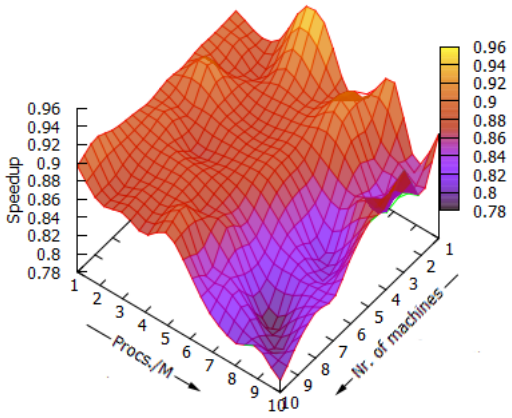
(b) Speedups of `nqueens(15)`



(c) Speedups of `uts(T3L)` using steal-1

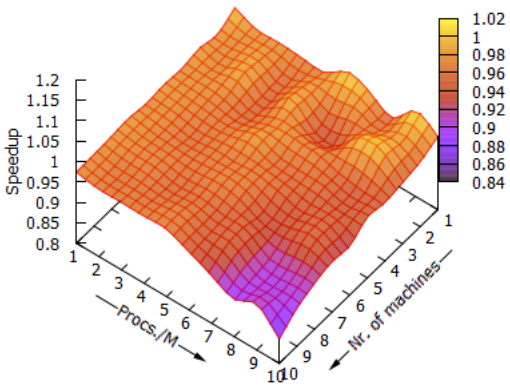


(d) Speedups of `uts(T3L)` using steal-10

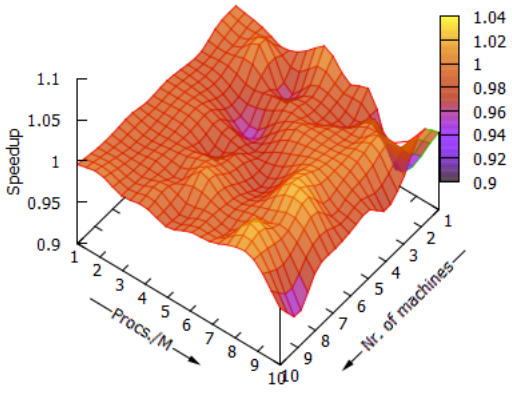


(e) Speedups of `uts(T2L)` using steal-1

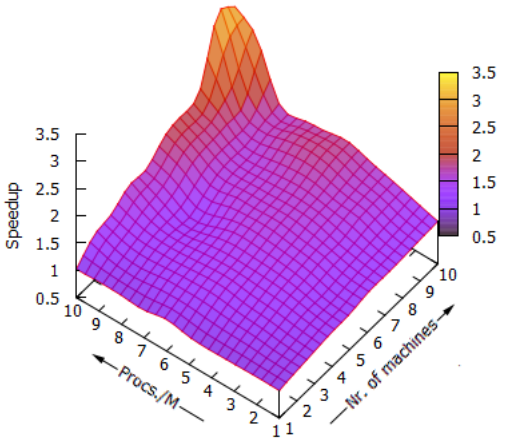
Figure 4.13: Comparing *computation times* between our implementation and HotSLAW. A speedup greater than one means that our implementation is faster.



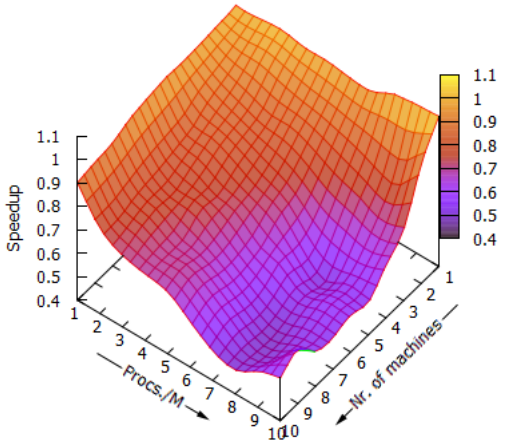
(a) Speedups of `fib(45)`



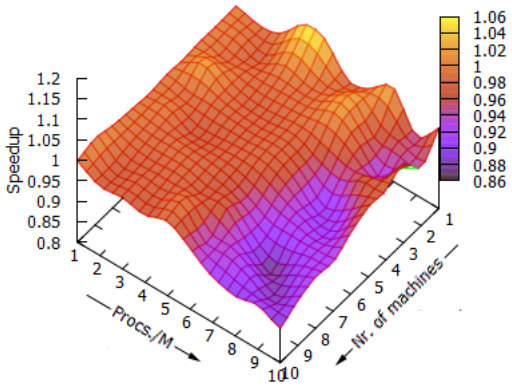
(b) Speedups of `nqueens(15)`



(c) Speedups of `uts(T3L)` using steal-1

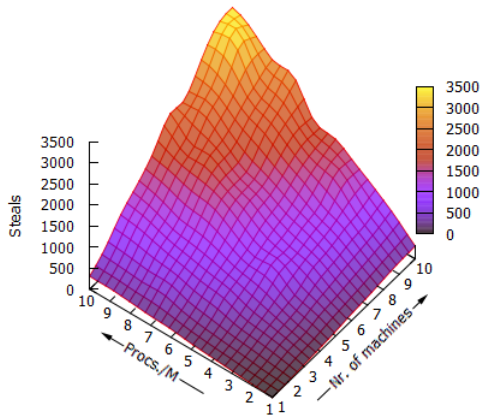


(d) Speedups of `uts(T3L)` using steal-10

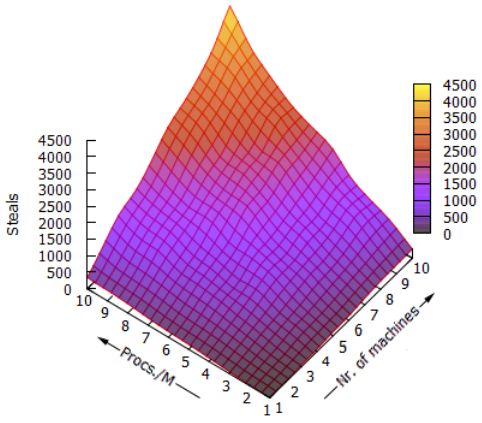


(e) Speedups of `uts(T2L)` using steal-1

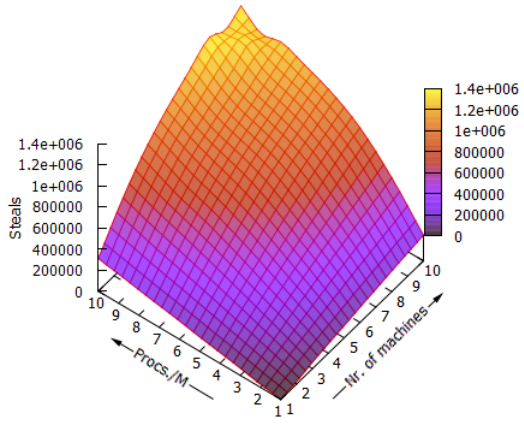
Figure 4.14: Comparing *scalability* between HotSLAW and our implementation (speedup of HotSLAW vs. speedup of our implementation). A speedup greater than one means that our implementation scales better.



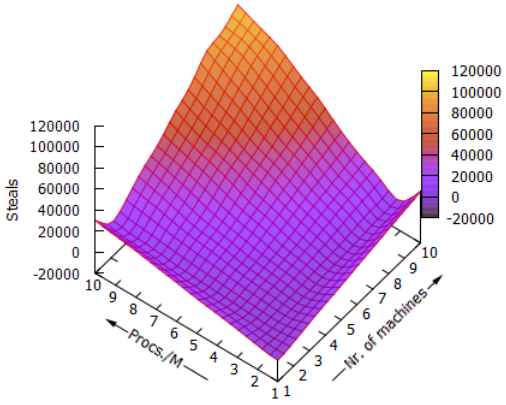
(a) Steals of `fib(45)`



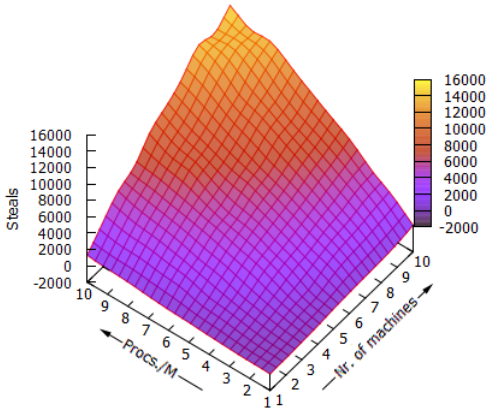
(b) Steals of `nqueens(15)`



(c) Steals of `uts(T3L)` using Steal-1

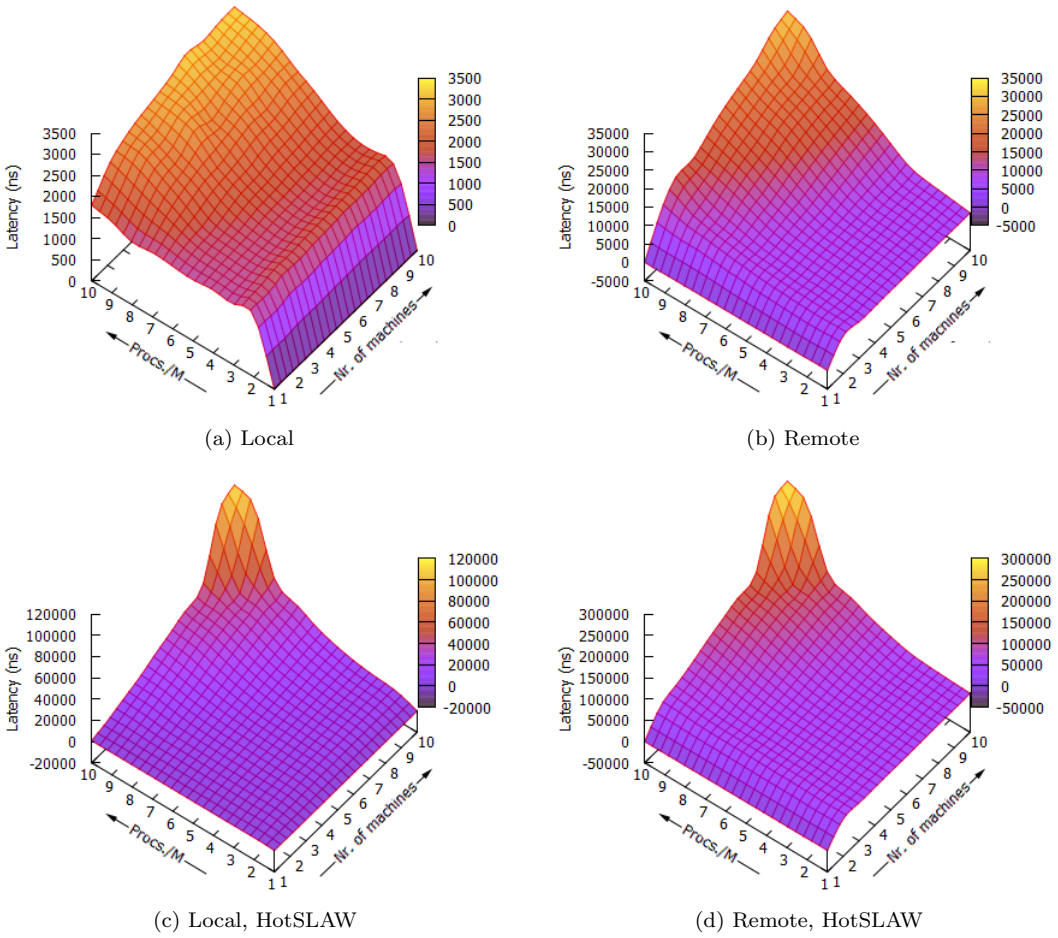


(d) Steals of `uts(T3L)` using Steal-10



(e) Steals of `uts(T2L)` using Steal-1

Figure 4.15: Average number of steals performed in the different benchmarks under various configurations. We distinct between Steal-1 and Steal-10 strategies in the `uts` benchmark.



(a) Local

(b) Remote

(c) Local, HotSLAW

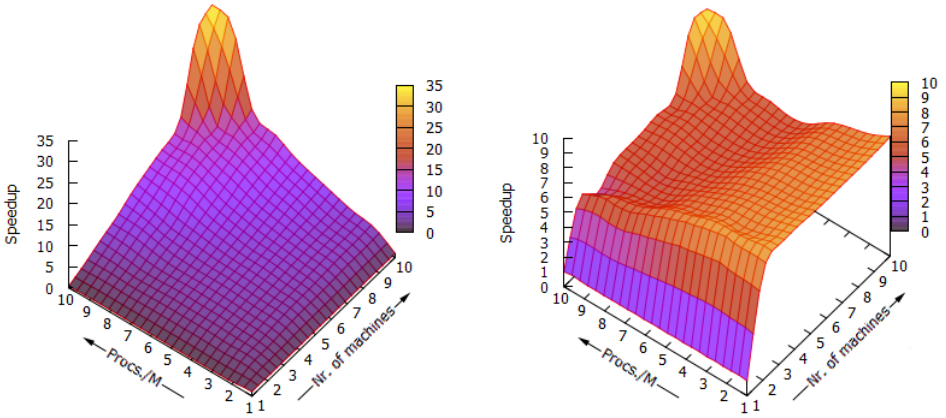
(d) Remote, HotSLAW

Figure 4.16: Latencies of the `steal` operation, both for HotSLAW and our implementation, when stealing a task either locally (i.e. from a process on the same machine) or remotely (i.e. from a remote machine using RDMA).

from itself. When 100 processes are used, local steals have a latency of $3,2\mu s$. Figure 4.16c shows an increase that is very similar to the increase in Figure 4.16d. By adding processes, the latency of steals increases vastly, both locally and remotely.

Figure 4.18 shows the average number of steal attempts performed per process under different machine configurations. The plots show that the number of steal attempts remains fairly stable when processes are added, although there are some peaks when roughly 50 processes are used. By adding processes, more RDMA roundtrips are performed, which increases the throughput of the RDMA devices. When the throughput increases, the steal latency also increases, since processes need to wait for RDMA devices to complete the operation. Also the initial phase and termination detection generate many RDMA operations, thus contributing to higher latencies.

Comparing Latencies with HotSLAW. Figure 4.16 shows the latencies of HotSLAW and Figure 4.17 gives a comparison of steal latencies between our implementation and HotSLAW.



(a) Local speedup

(b) Remote speedup

Figure 4.17: The speedups of local and remote steals of our implementation compared to HotSLAW.

A speedup of n means that our implementations performs steal operations n times faster than HotSLAW. For local steals, speedups up to 34.3 are reached. HotSLAW is about two times faster when only one machine is used. It could be that HotSLAW has an optimization for purely parallel executions. When more than one machine is used, however, our implementation steals at least double as fast. The speedup increases when the number of processes increases, especially when 90 or more processes were used. In those configurations, the speedup increased significantly. For some reason, the latencies of HotSLAW get very big under those configurations.

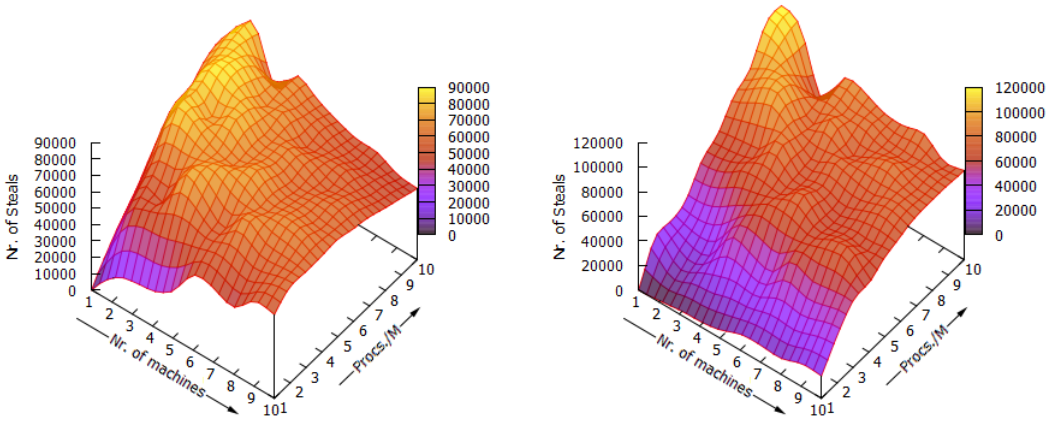
Not only local steals, but also remote steals are considerably faster with our implementation. Again, no remote steals are performed with one machine, which explains the slope in Figure 4.17b. The speedup remains constant when one process per machine is used, and slightly decreases when more processes are added. Similar to the local speedups, an increase is observed when 90 processes or more are used. Remote speedups up to 9.3 are observed when using 100 processes.

4.4.3 Suggestions for Performance Improvements

Lace uses concurrent deques, which allows stealing tasks without cooperation from the victim, in contrast to private deques [74]. Using concurrent deques might reduce the latency of steals if they not require more roundtrips than private deques. It would be interesting to perform research in efficient implementations of concurrent deques in PGAS, as they might increase performance. However, we expect their implementation to be more difficult than private deques.

Our design currently does not support steal-many strategies due to time constraints. Figures 4.13 and 4.14 show that steal-many significantly improves performance when there are many remote steals. Van Dijk et al. [74] mentioned that the performance of Lace does not improve when using steal-many, so performance may only improve for remote steals.

Our work stealer has a hierarchical victim selection protocol, like described in [36]. Paudel et al. [38] (2013) argue that, instead of focusing from which worker to steal, focusing on which *task* to steal makes more sense in distributed work stealing. This is because the costs of remote steals are partly determined by the size of the steal. Furthermore, the authors show that only stealing tasks that are not *locally critical* (i.e. tasks that have many dependencies with other local tasks) results in improved performance. These ideas can be implemented in our work stealer. The



(a) Successful steals

(b) Unsuccessful steals

Figure 4.18: The average number of successful and unsuccessful steals per process when scaling the number of participating processes.

implementation described by [38] uses a deque, so we expect that using either private deques or the concurrent deques of [74] might even further improve performance.

Although very dependent on the computational problem, applying a *coarser* task granularity may also increase performance and scalability. This is because fewer subtasks are generated, the amount of computation per worker increases relative to the amount of communication, which in turn reduces the overall number of roundtrips. In practice, however, updating the structure of tasks to have a coarser granularity may not be easy. Instead, Acar et al. [71] present *task coalescing*, in which a number of similar tasks can be combined. For example, in the `UTS` benchmark, every task explores only a single node in the tree. By using task coalescing, tasks can be combined, so that each task explores a *number* of nodes, instead of only a single node. For some problems, including graph searching, this might give performance improvements.

Finally, termination detection can be implemented more efficiently. Instead of having an explicit termination detection phase, the initiating thread may determine when termination occurs, since `initiate` returns immediately after the last task had been completed. Upon completion, the initiating thread simply broadcasts global termination, so that all other threads leave the stealing loop.

4.5 Conclusions

Compared to HotSLAW, our implementation steals more efficiently, as we observed speedups up to 34.3 for local steals and 9.3 for remote steals (both relative to HotSLAW). In most cases, our implementation also scales better, especially in the benchmarks that require many steals, like `uts(T3L)`, due to the efficiency of stealing.

On the other hand, HotSLAW generates less overhead when executing tasks. This is because HotSLAW supports tasks with *variable*-sized inputs and outputs, whereas our implementation only supports *fixed*-sized inputs and outputs. In addition, as a way to pass extra data to tasks, we may use *shared stacks* to store data and pass the corresponding indices as input parameters to tasks. In comparison with HotSLAW, this generates slightly more overhead, which makes HotSLAW more efficient in the benchmarks that rely heavily on the shared stacks.

By scaling along the number of processes, the number of steal attempts increases, which in turn increases the average steal latency. By using hierarchical work stealing, the number of generated RDMA roundtrips are reduced. Every remote steal only involves a single remote `cas` operation and two one-sided RDMA writes. We were not able to further reduce the number of RDMA operations. Performance and scalability can, however, still be improved by reducing the number of steal attempts. As future work, we would like to focus on the following improvements:

- Implement a steal-many strategy, as suggested in Section 4.4.3.
- Remove *explicit* termination detection with *implicit* termination detection (also explained in Section 4.4.3).
- Giving the request cells multiple *slots*, so that each slot may contain a pending request. The number of failed steal attempts may reduce by using multiple slots, without degrading performance when checking for pending requests.
- When starting a new task-based computation, using work sharing might be more beneficial than work stealing, as nearly all workers are then idle. Having an initial work sharing phase would therefore reduce the number of failed steal attempts.
- Implement task coalescing, discussed in Section 4.4.3, to make tasks somewhat coarser, depending on the amount of tasks to combine.

Chapter 5

Designing Distributed Binary Decision Diagram Operations

In this Chapter, RDMA-based designs of BDD operations are given and discussed. The operations make use of the distributed hash table and private deque work stealing operations discussed earlier. In addition, a shared memoization cache is given and discussed. The BDD operations are used to perform symbolic reachability over a number of BEEM models. In nearly all cases, distributed runs are faster than sequential runs, but parallel runs still scale better. Especially the larger BEEM models obtain good performances in distributed symbolic reachability, even in comparison with parallel runs.

5.1 Introduction

A Binary Decision Diagram (BDD) is a data structure used to efficiently represent Boolean functions. BDDs were introduced by [4] in 1978 to compress large digital functions. BDDs were further developed by Bryant [21, 22] in 1986 and 1992. They were used in the verification of hardware systems by performing symbolic model checking [40, 47]. Because BDDs can represent Boolean functions, they can also represent sets of states or relations. These sets can directly be manipulated by performing BDD operations on their representations, which makes symbolic model checking possible.

Alternatively to representing state spaces symbolically, they can also be represented explicitly. However, often the state space becomes so large that explicit data structures cannot cope [32]. This problem is known as the *state space explosion* problem, which is the biggest limitation in modern software verification. State space explosions arise because state spaces often grow exponentially with the size of the model (e.g. the number of variables used, as well as the size of the domain of those variables). By using a symbolic representation of the state space, the exponentially large domain may be stored in polynomial space [32]. Exponential blow-ups are, however, still possible with a symbolic representation and space efficiency is often reliant on the variable ordering. Finding an optimal variable ordering is NP-Complete [18], but finding a *good* ordering is often enough to significantly improve the space complexity of state spaces compared to explicit representations, which motivates symbolic verification. The following study is mostly based on a similar study in [72].

Early Work on Parallel BDD Manipulation. Several attempts have been made to speed up

BDD operations by performing parallel processing. Kimura et al. [44] (1990) views BDDs as automata. Parallelism is then achieved by calculating the products of automata in parallel, followed by reduction to keep the resulting BDDs small. Parasuram et al. [79] (1994) uses a distributed shared memory abstraction for sharing data and message passing for communication. The BDD algorithms are implemented with a Depth-First approach. Yang et al. [80] (1997) designed a parallel algorithm for BDD construction using a hybrid depth-first/breadth-first approach. The algorithm uses parallel Breadth-First expansion until a fixed threshold is reached. After that, the algorithm switches to Depth-First expansion to limit memory overhead.

Early Work on Distributed BDD Manipulation. Arunachalam et al. [56] (1996) exploit the available memory in a cluster of workstations to construct larger BDDs. This was done via distributed memory partitioning and a depth-first approach. As a result, larger BDDs could be constructed than was possible with a single machine. Sanghavi et al. [34] (1996) exploit memory hierarchy on a single machine by using a breadth-first approach to increase the locality of memory accesses. Compared to existing work, this resulted in a significant speedup for models that do not fit into main-memory. Due to this success, the authors expanded their algorithms to a network of workstations [58] (1996). BDD nodes are distributed based on their variable levels. Workers may then only perform work when their variable level has a turn, which makes the implementation sequential. As a result, no speedups were obtained, but very large BDDs could be constructed and manipulated. Milvang-Jensen et al. [48] (1998) presented the parallel BDD package BDDNOW. This package parallelizes BDD operations by striving for minimal communication overhead. Compared to existing work, BDDNOW was the first BDD implementation that achieved some speedup on a network of workstations, but only when the workstations ran out of memory and switched to disk instead.

Early Work on Symbolic Reachability. Heyman et al. [68] (2000) use BDDs for parallel symbolic reachability in a distributed-memory environment of workstations. The authors argued that network accesses are cheaper than disk accesses, so using a network of workstations is cheaper than external storage. Load-balancing is implemented by dynamically repartitioning the state space throughout the computation. Grumberg et al. [51] (2003) present a symbolic algorithm for reachability that is work-efficient. It allows dynamic allocation of hardware resources and good utilization of those resources via mechanisms to recover from local state space explosions. The authors concluded that a trade-off has to be made between work-efficiency (i.e. good utilization of the available hardware resources) and obtaining speedups. Chung et al. [25] (2004) implemented a symbolic algorithm for saturation in a distributed-memory environment. Although the implementation obtains no speedups, the runtimes of the distributed implementation come close to the sequential implementations. Later, Chung et al. [26] (2006) implemented distributed state space generation with MDDs. With respect to the best sequential implementation, speedups up to 17% were achieved on a network of workstations. As far as we know, this is the *best* result obtained with distributed symbolic algorithms.

More Recent Work on Symbolic Reachability. After 2006, most work was focussed on improving parallelizing symbolic algorithms (based on BDDs, but also LDDs and MDDs). Ezekiel et al. [41] (2007) tried to increase parallelism of [26] by improving locality on shared-memory architectures. Ciardo et al. [32] (2009) showed that, despite all previous attempts, obtaining linear speedups still remained a challenge. Van Dijk et al. [75] (2015) present Sylvan, a parallel BDD package that achieves parallelism by using fine-grained task parallelism via work-stealing. Sylvan obtains good scalability and speedups up to 38 by using 48 cores.

5.2 Challenges and Contributions

BDD operations are very memory-intensive, because they perform only a small amount of computation per memory access. This makes it very hard to distribute BDD operations. This is because only a small amount of computation is performed with respect to the amount of communication between computers, which is far more expensive than memory accesses.

The main considerations for an efficient distributed-memory implementation of BDD operations is the distribution of data, the maintenance of load-balancing, and the reduction of communication overhead and latency [32]. As part of data distribution, exploiting data-locality is also an important consideration [26].

This chapter describes a novel design of BDD operations that considers all these factors. We distribute data via a PGAS abstraction, which provides a shared distributed memory programming model that allows existing designs of BDD operations to be adopted relatively easily. We maintain load-balancing via cluster-based work stealing, due to the success of Sylvan [75]. We exploit data-locality via hierarchical work stealing, so that processes prefer stealing work from other processes located on the same machine. Finally, we perform communication via one-sided RDMA. Communication overhead is reduced via the *zero-copy* techniques used by RDMA and latency is reduced via *one-sided* RDMA operations.

The significant performance differences of one-sided RDMA in comparison with traditional TCP, in combination with the recent success of using fine-grained task parallelism in Sylvan was the main motivation for a renewed attempt to distribute BDD operations.

The rest of this chapter is structured as follows. Section 5.3 gives a formal definition of BDDs and gives preliminaries. Section 5.4 presents the design of the new BDD operations. These implementations have been implemented and evaluated. Section 5.5 discusses the experimental results. Finally, Section 5.6 summarizes our conclusions.

5.3 Preliminaries

5.3.1 Reduced Ordered Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are Directed Acyclic Graphs (DAGs) that express the Shannon decomposition of a Boolean function. The following definition of a BDD is taken from [75].

Definition 1. *An (ordered) Binary Decision Diagram is a DAG 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* $var(p) = x_i$ *and two outgoing edges, labelled* 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.* $\forall p \forall q.(lvl(p) = lvl(q) \wedge p[0] = q[0] \wedge p[1] = q[1]) \implies p = q$

Figure 5.1 illustrates an example in which four different simple Boolean function are represented as BDDs. The rectangle shaped nodes denote the terminal nodes 0 and 1 (or, alternatively, `false` and `true`). Every non-terminal node p has two edges, namely a thick edge representing $p[1]$ and a dashed edge representing $p[0]$. All paths in the BDD that lead to 1 makes the Boolean function represented by the BDD evaluate to `true`. For example, in Figure 5.1d, the truth assignments $(x_0 = 1)$ and $(x_0 = 0, x_1 = 1)$ both evaluate to `true`, which together is equivalent to $x_0 \vee x_1$.

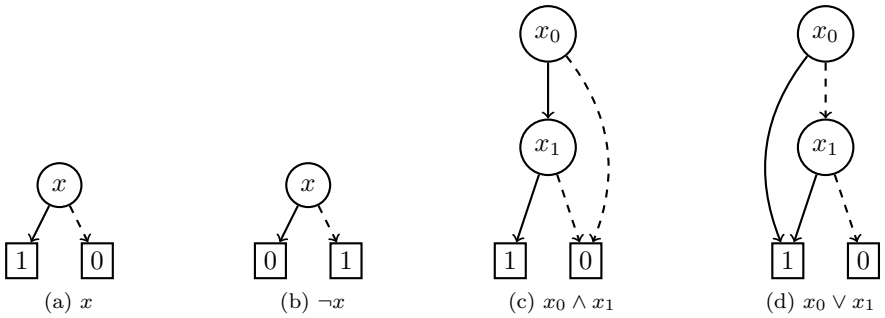


Figure 5.1: The BDD representation of four different simple Boolean functions.

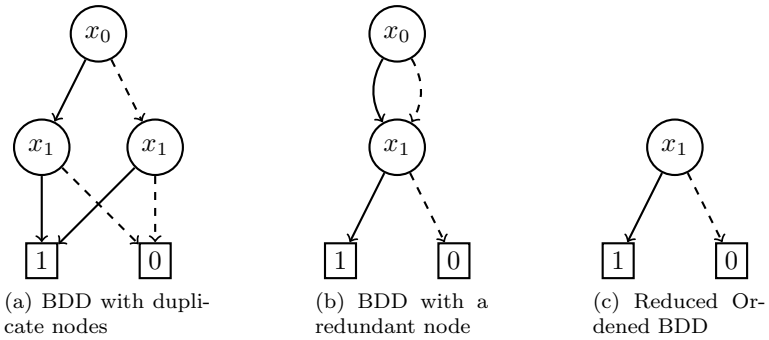


Figure 5.2: Three BDD representations of the Boolean function $(x_0 \wedge x_1) \vee (\neg x_0 \wedge x_1)$ (left representation), which is equivalent to x_1 (right representation). The rightmost BDD is obtained by removing duplicates and redundant nodes.

The total ordering $<$ enforced by property *3.* of Definition 1 ensures that the BDD is ordered, i.e., all variables on all paths from root to leaf satisfy the ordering $<$. The following definition, also taken from [75], introduces reduced BDDs.

Definition 2. *A reduced Binary Decision Diagram is a BDD without redundant nodes. A node p is redundant if $p[0] = p[1]$.*

A BDD is thus reduced if none of its nodes has identical child nodes and if there are no duplicate sub-graphs. Figure 5.2 shows examples of BDDs that have duplicate and redundant nodes. All three BDDs are identical, only the rightmost one is both reduced and ordered. In Figure 5.2a, both sub-graphs of node x_0 are identical. In Figure 5.2b, the node x_0 is redundant.

Let $\phi : \mathbb{B}^n \to \mathbb{B}$ be some Boolean function and let A be the reduced ordered BDD representing ϕ. Then the BDD $\overline{A}$ represents $\neg\phi$.

5.3.2 Defining `ITE` and `RelProd`

We now define two operations on BDDs, namely `ITE` and `Relprod`. Most other BDD operations can be expressed via `ITE` or `Relprod`. Both operations are recursively defined and are based on Shannon decomposition.

Definition 3 (Restriction). *Let $X = \{x_1, \ldots, x_n\}$ be a set of variables and $\phi(x_1, \ldots, x_n)$ a function $\phi : S^n \to \mathbb{B}$ on S. Then the restriction of $v \in S$ to $x_i \in X$ in ϕ, denoted by $\phi_{x_i=v}$, is defined as $\phi_{x_i=v} \equiv \phi(x_1, \ldots, x_{i-1}, v, x_{i+1}, \ldots, x_n)$.*

Theorem 3 (Shannon Decomposition). *Let $X = \{x_1, \ldots, x_n\}$ be a set of variables and $\phi(x_1, \ldots, x_n)$ a Boolean function $\phi : \mathbb{B}^n \to \mathbb{B}$. Then, for any $x_i \in X$, it holds that $\phi \equiv (x_i \wedge \phi_{x_i=1}) \vee (\neg x_i \wedge \phi_{x_i=0})$.*

The identity described in Theorem 3 is called the *Shannon decomposition* of ϕ along $x_i \in X$. The term $(x_i \wedge \phi_{x_i=1})$ is called the *positive Shannon cofactor* and $(\neg x_i \wedge \phi_{x_i=0})$ the *negative Shannon cofactor* of the Shannon expansion of ϕ along x_i. Many BDD operations, including `ITE`, are implemented recursively by choosing a variable and recursively calculating the results of its Shannon cofactors. The following operator denotes the cofactors of a BDD and is used in the implementations of `ITE` and `RelProd`.

Definition 4 (Cofactor). *Let $X = \{x_1, \ldots, x_n\}$ be a set of variables, $\phi(x_1, \ldots, x_n)$ a Boolean function $\phi : \mathbb{B}^n \to \mathbb{B}$, and A the reduced ordered BDD representation of ϕ. Let $x \in X$ be a variable from X. Then $\texttt{cofactor}_v(A, x)$ is the reduced ordered BDD representing $\phi_{x=v}$.*

To implement the *relational product* operation (`RelProd`), existential quantification and substitution is needed.

Definition 5 (Existential Quantification). *Let $X = \{x_1, \ldots, x_n\}$ be a set of variables, $\phi(x_1, \ldots, x_n)$ a Boolean function $\phi : \mathbb{B}^n \to \mathbb{B}$, and $x \in X$ be some variable from X. Then the existential quantification of x over ϕ, denoted by $\exists x\phi$, is defined as $\exists x\phi \equiv \phi_{x=0} \vee \phi_{x=1}$. Let $X' = \{x'_1, \ldots, x'_m\} \subseteq X$ be a subset of variables from X. Then $\exists X'\phi \equiv \exists x'_1\phi \ldots \exists x'_m\phi$.*

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

Definition 7 (If-then-else). *Let $\phi, \psi, \gamma : \mathbb{B}^n \to \mathbb{B}$ be three Boolean functions and let A, B, C be three reduced ordered BDDs representing ϕ, ψ, γ, respectively. Then the if-then-else operator, denoted by $\texttt{ITE}(A, B, C)$, is defined as the reduced ordered BDD representing $(\phi \wedge \psi) \vee (\neg\phi \wedge \gamma)$.*

Figure 5.1 shows a number of Boolean formulas and their expression with the `ITE` operator. BDDs can be used for symbolic model checking, and an important model checking algorithm is *reachability analysis*. Burch et al. [39] uses the relational product (`RelProd`) operation to implement *symbolic* reachability analysis. This operation is defined as follows.

Definition 8 (Relational Product). *Let $X = \{x_1, \ldots, x_n\}$ be a set of variables and $X' \subseteq X$ be a subset of X. Let $\phi(X)$ and $\psi(X)$ be two Boolean functions $\phi, \psi : \mathbb{B}^n \to \mathbb{B}$ and let A, B be their respective reduced ordered BDD representations. Then the relational product over A and B with respect to X', denoted by $\texttt{RelProd}(A, B, X')$, is defined as the reduced ordered BDD representing $\exists X'(\phi(X) \wedge \psi(X))$.*

Definition 9 (Rename). *Let $X = \{x_1, \ldots, x_n\}$ be a set of variables, $\phi : \mathbb{B}^n \to \mathbb{B}$ be a Boolean function and A be the reduced ordered BDD representing ϕ. Let $Y = \{y_1, \ldots, y_m\} \subseteq X$ and $Z = \{z_1, \ldots, z_m\} \subseteq X$ be two subsets of X. Then $\texttt{Rename}(A, Y, Z)$ is the reduced ordered BDD representing $\phi[Y \leftarrow Z]$.*

Boolean Formula	Representation with ITE
$\phi \wedge \psi$	$\texttt{ITE}(A, B, 0)$
$\neg\phi \wedge \psi$	$\texttt{ITE}(A, 0, B)$
$\phi \wedge \neg\psi$	$\texttt{ITE}(A, \overline{B}, 0)$
$\neg(\phi \wedge \psi)$	$\texttt{ITE}(A, \overline{B}, 1)$
$\phi \vee \psi$	$\texttt{ITE}(A, 1, B)$
$\neg(\phi \wedge \psi)$	$\texttt{ITE}(A, \overline{B}, 1)$
$\neg(\phi \vee \psi)$	$\texttt{ITE}(A, 0, \overline{B})$
$\phi \rightarrow \psi$	$\texttt{ITE}(A, B, 1)$
$\phi \leftarrow \psi$	$\texttt{ITE}(A, 1, \overline{B})$
$\phi \leftrightarrow \psi$	$\texttt{ITE}(A, B, \overline{B})$
$\phi \oplus \psi$	$\texttt{ITE}(A, \overline{B}, B)$

Table 5.1: Several Boolean formulas expressed with the ITE operation. The Boolean functions ϕ, ψ are represented by the reduced ordered BDDs A, B, respectively.

5.3.3 Symbolic Reachability Analysis

The verification of most temporal safety properties can be reduced to a reachability analysis problem [33], which is why reachability is an important component of model checking. Let $S = \mathbb{B}^n$ be the set of all states of n-sized Boolean vectors. The transitions between states can be represented as a binary relation $\rightarrow \subseteq S \times S$, which is called a *transition relation*. Then $(S_I, \rightarrow)$ is a transition system, where $S_I \subseteq S$ denotes the set of initial states. The set of reachable states is the reflexive, transitive closure of $\rightarrow$ applied to S_I [70]. Reachability algorithms calculate this closure to find all possible and reachable program states.

Reachability with Boolean Functions. Boolean functions can be used to represent sets of states. A subset $T \subseteq S$ can be represented by a Boolean function $\phi : \mathbb{B}^n \rightarrow \mathbb{B}$ such that $T = \{x \in S \mid \phi(x)\}$, or, alternatively, $x \in T \Leftrightarrow \phi(x)$ for every $x \in S$. The function ϕ is then called a *membership function* of T with respect to S. The transition relation $\rightarrow \subseteq S \times S$ can be represented by a Boolean function $\psi : \mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}$, such that $\forall x, y \in S.\ \psi(x, y) \Leftrightarrow (x, y) \in \rightarrow$.

Let $X = \{x_1, \ldots, x_n\}$ and $X' = \{x'_1, \ldots, x'_n\}$ be two sets of variable names. Then the ψ-successors of ϕ, which we may denote by ϕ', can be obtained by $\phi'(X) \equiv \exists X'(\phi(X') \wedge \psi(X', X))$ [70]. By repeatedly obtaining the ψ-successors, starting from the Boolean function representing the initial states S_I, the set of reachable states is computed. Formally, the set of reachable states is represented by the fixed point of the series [70]

$$\phi_{i+1}(X) \equiv \phi_i(X) \vee (\exists X'.\phi_i(X') \wedge \psi(X', X)) \tag{5.1}$$

Reachability with BDDs. Suppose that A, T are the BDD representations of $\phi_i(X)$ and $\psi(X, X')$, respectively. Then $\texttt{RelProd}(A, T, X')$ is the BDD representing $\phi_{i+1}(X') \equiv \exists X(\phi_i(X) \wedge \psi(X, X'))$. The set of next states from $\phi_i(X)$ is the BDD representation of $\phi_{i+1}(X) \equiv \phi_{i+1}(X')[X' \leftarrow X]$. The set of reachable states can be found symbolically by continuously calculating the next states, starting from the set of initial states, and taking their disjunctions, like shown in 5.1.

Figure 5.3 shows the basic symbolic reachability algorithm. Here I is the BDD representing the set of initial states and T the BDD representing the binary transition relation. The fixed point is calculated using the while-loop at line 4. In every iteration, RelProd is used to find successors (line 5) and Rename to obtain the next states (line 6). Finally, the disjunction of the states already found (line 8 and the next states is calculated by applying ITE

```
1 def Reachability(I, T, X, X'):
2     States ← I
3     Previous ← 0
4     while States ≠ Previous do
5         Successors ← RelProd(States, T, X)
6         Next ← Rename(Successors, X', X)
7         Previous ← States
8         States ← ITE(States, 1, Next)
9     return States
```

Figure 5.3: A basic reachability operation.

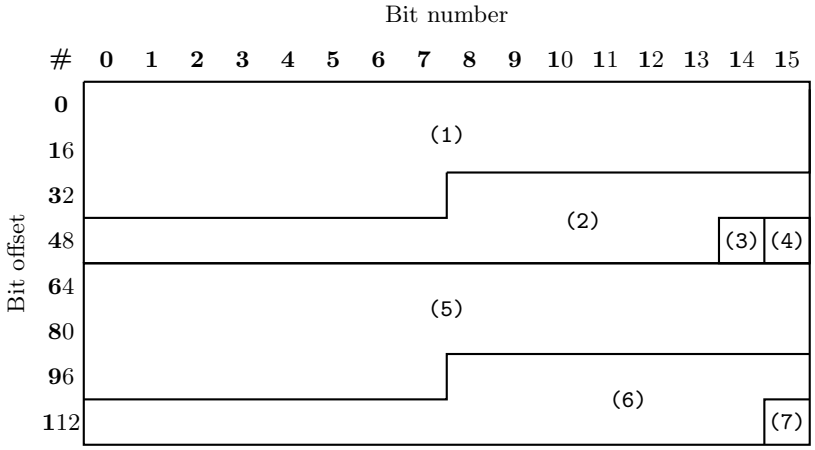


Figure 5.4: The memory layout of a 128-bit BDD node, where `(1)` and `(5)` represent the high and low edge, respectively. Furthermore, `(2)` represents the variable level, `(6)` represents data stored in the BDD, `(3)` is an occupation bit, used by the hash table, `(4)` is a locking bit, and `(7)` a complement bit.

5.4 Design and Implementation

5.4.1 Memory Layout

BDD nodes are 16 bytes (i.e. two 64-bit integers) in size. Figure 5.4 shows the memory layout of a BDD node. Sylvan uses two separate tables for storing buckets, namely an index table and a data table [75]. This is to make garbage collection more efficient. Due to time constraints, we chose *not* to implement garbage collection. Therefore, we use only one table. The BDD nodes are stored in the hash table described in Chapter 3, which we slightly adjusted to support 16-byte buckets instead of 8-byte buckets. By doing that, `find-or-put` can be implemented to require fewer roundtrips than with two separate tables, which makes the BDD operations more efficient.

Assuming that the hash table is used by n processes, we allocated a shared hash table $\mathsf{H}[0], \ldots, \mathsf{H}[mn-1]$ of BDD nodes, so that each process owns m entries, thus has a portion of $16 \times m$ bytes of the hash table. By default, we let $m = 2^{26}$, so that H is n GB in size.

Storing and Retrieving BDD Nodes. The `find-or-put` operation is adjusted in the following way. When `find-or-put` finds an empty bucket (i.e. a BDD node with its occupation bit set to 0), it first performs a `cas` operation on the first 8 bytes of the BDD node to set both its occupation bit and locking bit to 1. After that, it writes the whole bucket to the hash table with an one-sided RDMA write, thereby removing the lock. When `find-or-put` encounters a locked bucket, it polls for the (remote) memory location until the lock is removed. This ensures data consistency, although requiring extra roundtrips.

The high and low edges of BDDs are both 40 bits in size, which is enough to store indices of the hash table. This also implies that the hash table cannot contain more than 2^{40} entries, which is 16 Terabytes in memory. If larger hash tables need to be supported, the data portion of BDD nodes, represented by `(6)` in Figure 5.4, can be made smaller.

The complement bit, denoted by `(7)` in Figure 5.4, is used to denote complemented BDDs. If a BDD node is retrieved from the hash table with its complement bit set to 1, then the leaves of the BDD are switched.

Referencing BDD Nodes. We use 64-bit integers to refer to BDD nodes, where 40 bits are used to denote the index in the hash table and 1 bit for *complement edges* [63]. Suppose that the BDD A represents the Boolean function ϕ, so that $\overline{A}$ represents $\neg\phi$. Then A and $\overline{A}$ can be stored on the same location by using the complement bit, thereby reducing space requirements. This concept is called *complement edges*.

5.4.2 Designing a Shared Memoization Cache

The memoization cache is a simple cache used to make the recursive implementations of `ITE` and `RelProd` efficient. This cache is based on a simple hash table in which hash collisions are not resolved. Resolving collisions to store a value requires performing a number of probes, which are computationally more expensive and may require more RDMA operations than simply overwriting the existing entry with a given value, especially when targeting remote memory.

Cache entries are 32 bytes in size and have the form $\langle a, b, c, r\rangle$, where each component is 8 bytes in size. To ensure data consistency and prevent read-write races, we use `cas` operations on a to set a locking bit, which we denote via the bitmask `LOCK-MASK`. Read-write races may occur in a number of scenarios. They might occur when multiple RDMA devices are used and access the same memory location, when the CPU and RDMA device access the same memory location, or when multiple UPC threads access the same location.

Figure 5.5 shows the operations used to access the cache. We allocated a shared array $\mathsf{C}[0], \ldots \mathsf{C}[kn-1]$ with cache entries, assuming m participating processes. Each process owns k entries and thus contributes $32 \times k$ bytes to the shared array. By default, we let $k = 2^{23}$, so that C is $(256 \times n)$ MB in size.

Consulting the Cache. The `cache-request` operation is used to start a request to C, which is done asynchronously (i.e. the `memget` at line 4). A handle s and a private memory location R are returned by `cache-request`. The handle s can be given to `sync` to synchronize on the roundtrip. The location R can be given to `cache-check` to test if (a, b, c) is already cached. We chose to split cache reads into two operations, so that cache reads become asynchronous. During calls to `cache-request` and `cache-check`, work can then be performed.

The `cache-check` operation takes a memory location R as parameter and assumes that R is obtained via a call to `cache-request` and that the corresponding memory operation has been synchronized. The operation starts by testing if $R.a$ contains a lock. If so, `cache-check` waits for the lock to expire at line 3. After that, `cache-check` returns R if it contains (a, b, c), or

```
1  def cache-request(a, b, c):
2      h ← hash(a, b, c)
3      i ← h mod CACHE-SIZE
4      s ← memget-async(C[i], R)
5      return ⟨s, R⟩

1  def cache-check(a, b, c, R):
2      ▷ Wait for the lock to expire
3      while is-locked(R.a) do
4          h ← hash(a, b, c)
5          i ← h mod CACHE-SIZE
6          memget(C[i], R)
7      ▷ Compare R with a, b, c
8      if R.a = a ∧ R.b = b ∧ R.c = c
9          return R
10     else
11         return ⊥
```

```
1  def cache-put(a, b, c, r):
2      h ← hash(a, b, c)
3      i ← h mod CACHE-SIZE
4      ▷ Prepare a condition and value for
       the cas operation
5      cond ← a & ∼ LOCK-MASK
6      val ← a | LOCK-MASK
7      ▷ Perform the cas operation
8      if cond = cas(i, cond, val)
9          R ← ⟨cond, b, c, r⟩
10         memput-async(R, C[i])
```

Figure 5.5: The operations used to access the shared memoization cache. The `LOCK-MASK` bitmask is used to set the locking bit in a.

otherwise returns $\bot$. The value $\bot$ can then be used by the function that called `cache-check` to test whether (a, b, c) is in the cache or not.

Supplementing the Cache. The `cache-put` operation attempts to write r in the cache entry corresponding to (a, b, c). This is done by locking a via a `cas` operation (line 7). If the `cas` fails, then a is either changed or locked by some other process. In that case, r is not inserted in the cache table. If `cas` succeeds, a new entry is created (line 9) and inserted via an asynchronous write operation (line 10). This write also removes the lock from $\mathsf{C}[i].a$ by writing *cond* (which is a with the locking bit set to 0) to it (line 9).

5.4.3 Design Considerations of `ITE`

Figure 5.6 shows the implementation of `ITE`. We used the implementation of Sylvan [75], and modified it to support PGAS, cluster-based work stealing and efficient asynchronous remote memory operations. The implementation takes three references (a, b, c) to BDD nodes as parameters and starts by considering terminal cases (lines 3 to 11). Note that the `not` operator flips the complement bit of a given BDD reference, thereby using complement edges.

After considering terminal cases, a number of cache optimizations are performed, which are *not* shown in Figure 5.6. For those optimizations we refer to [73, 75].

We aimed to perform the asynchronous retrieval of BDD nodes (lines 13 to 15) as early as possible and use them as late as possible, so that work can be performed during the roundtrips. The BDDs cannot be retrieved before handling the terminal cases and are used shortly thereafter, so the only work that can be performed asynchronously with the roundtrips is sending a cache request (line 16). Nonetheless, the roundtrips themselves are performed asynchronously. After synchronizing on the queries that fetch the BDD nodes A, B, C, referred to by a, b, c (line 17), the top variable of (A, B, C) is determined. This is the *highest* variable in the total variable ordering. After that, we check if (a, b, c) is already in the cache at lines 20 to 22. We also considered to

```
1  bdd ITE(bdd a, bdd b, bdd c):
2      ▷ Terminal cases
3      if a = 1 return b
4      if a = 0 return c
5      if a = b then b ← 1
6      if a = not(b) then b ← 0
7      if a = c then c ← 0
8      if a = not(c) then c ← 1
9      if b = c return b
10     if b = 1 ∧ c = 0 return a
11     if b = 0 ∧ c = 1 return not(a)
12     ▷ Retrieve nodes asynchronously
13     s_a ← memget-async(H[a], A)
14     s_b ← memget-async(H[b], B)
15     s_c ← memget-async(H[c], C)
16     ⟨s_r, R⟩ ← cache-request(a, b, c)
17     sync(s_a), sync(s_b), sync(s_c)
18     x ← top-variable(A, B, C)
19     ▷ Consult the cache
20     sync(s_r)
21     R ← cache-check(a, b, c, R)
22     if R ≠⊥ return R
23     ▷ Get the cofactors
24     a_0 ← cofactor_0(A, x), b_0 ← cofactor_0(B, x), c_0 ← cofactor_0(C, x)
25     a_1 ← cofactor_1(A, x), b_1 ← cofactor_1(B, x), c_1 ← cofactor_1(C, x)
26     ▷ Calculate result recursively
27     if is-constant(a_1)
28         high ← if a_1 = 1 then b_1 else c_1
29         low ← call(ITE, a_0, b_0, c_0)
30         result ← make-node(x, high, low)
31     else
32         spawn(ITE, a_1, b_1, c_1)
33         low ← call(ITE, a_0, b_0, c_0)
34         high ← sync
35         result ← make-node(x, high, low)
36     ▷ Add result to cache and return
37     cache-put(a, b, c, result)
38     return result
```

Figure 5.6: The implementation of ITE.

consult the cache before retrieving the A, B, and C, but a cache request often results in a remote memory read. Performing such a read asynchronously with the other roundtrips is more efficient.

Finally, the cofactors of A, B, and C are retrieved (lines 24 to 25) and `ITE` is called recursively (lines 27 to 35) to enable task parallelism. If a_1 is constant (line 27), i.e. either 0 or 1, then the high edge of the resulting BDD can be determined without a performing a recursive call.

5.4.4 Design Considerations of `RelProd`

Figure 5.7 shows the implementation of `RelProd`. Similar to `ITE`, we used the `RelNext` operation implemented in Sylvan and modified it to support PGAS and distributed memory operations. The `RelProd` operation takes two BDD node references a, b and an integer v, used as index in the variable array.

In Sylvan, lists of variables are given to BDD operations as BDDs. For example, consider the set of variables $X = \{x_1, \ldots, x_n\}$. Then Sylvan uses the BDD representation of the membership function $x_1 \vee \cdots \vee x_n$ to store X. Subgraphs of that BDD are used to denote subsets of X. In our implementation, we used an array $vars[1], \ldots, vars[n]$ instead. This is because traversing a BDD requires multiple roundtrips and traversing an array does not.

The `RelProd` operation starts by considering terminal cases (lines 3 to 5). In line 5 `RelProd` tests if all variables have been checked. If so, the conjunction of a and b is returned.

In lines 12 to 15 variables are skipped, so that fewer tasks are generated and thus fewer roundtrips are required by `RelProd`. Sylvan implements this part by traversing through the BDD representing the set of variables, but this would considerably slow down execution in a distributed setting due to the generated roundtrips. We therefore use arrays instead.

After consulting the cache, `RelProd` retrieves the four cofactors of A, B (lines 22 to 22). If $x \neq vars[v]$ (line 25), the variable x can be skipped and `RelProd` is recursively applied to the cofactors of A and B (lines 41 to 44). Otherwise, Shannon decomposition along x is applied to the results of the recursive calls to `RelProd` (lines 27 to 39). Finally, the resulting BDD is constructed, stored in the cache, and returned.

5.5 Experimental Evaluation

We implemented the BDD operations in Berkeley UPC, version 2.20.2. The performance is evaluated by measuring the execution times and scalability of various common models from the BEEM database [53]. The implementation is compiled by using the Berkeley UPC compiler and gcc version 4.8.2, using the command `upcc -O -network=ibv`. The `-O` compiler option causes UPC to generate optimized executables, which improves performance. All experiments have been performed on a cluster of 10 Dell M610 machines running Ubuntu 14.04.2 LTS with kernel version 3.13.0, each having 8 CPU cores and 24 GB of internal memory. The machines are connected via a 20 GB/s Infiniband cluster. All experiments have been repeated at least five times and the average measurements have been taken into account.

The experiments were performed by spawning a number of processes on every machine. By default, every process contributes 1 GB to the hash table and 256 MB to the computation cache. For example, by using a configuration of 6 machines, each having 5 processes, the total hash table size is 30 GB and the total cache size is 7.5 GB. If needed, a custom memory layout can be used to make optimal use of the available memory. The benchmarks are executed by using the command `upcrun -n` processes `-N` machines `-shared-heap 1350MB`, where we replaced processes by the total number of processes (e.g. 30 in the example given above) and machines with the number of machines. Each process is allowed to allocate up to 1350 MB of shared memory by using the `shared-heap` flag.

```
1  bdd RelProd(bdd a, bdd b, int v):
2      ▷ Terminal cases
3      if a = 1 ∧ b = 1 return 1
4      if a = 0 ∨ b = 0 return 0
5      if v ≥ length(vars) return ITE(a, b, 0)
6      ▷ Retrieve nodes asynchronously
7      s_a ← memget-async(H[a], A)
8      s_b ← memget-async(H[b], B)
9      sync(s_a), sync(s_b)
10     x ← top-variable(A, B)
11     ▷ Skip variables
12     while 1 do
13         if x ≤ vars[v] break
14         v ← v + 1
15         if v ≥ length(vars) return ITE(a, b, 0)
16     ▷ Consult the cache
17     ⟨s_r, R⟩ ← cache-request(a, b, v)
18     sync(s_r)
19     R ← cache-check(a, b, v, R)
20     if R ≠⊥ return R
21     ▷ Get the cofactors of A and B
22     a_0 ← cofactor_0(A, x); b_0 ← cofactor_0(B, x)
23     a_1 ← cofactor_1(A, x); b_1 ← cofactor_1(B, x)
24     ▷ Recursively calculate result
25     if x = vars[v]
26         ▷ In this case, variable v is in vars. Retrieve nodes:
27         s_{b_0} ← memget-async(H[b_0], B_0)
28         s_{b_1} ← memget-async(H[b_1], B_1)
29         ▷ Get the cofactors of B_0 and B_1
30         sync(s_{b_0}); b_{0,0} ← cofactor_0(B_0, v); b_{0,1} ← cofactor_1(B_0, v)
31         sync(s_{b_1}); b_{1,0} ← cofactor_0(B_1, v); b_{1,1} ← cofactor_1(B_1, v)
32         ▷ Spawn new tasks
33         spawn(RelProd, a_0, b_{0,0}, v + 1); spawn(RelProd, a_1, b_{0,1}, v + 1)
34         spawn(RelProd, a_0, b_{1,0}, v + 1); spawn(RelProd, a_1, b_{1,1}, v + 1)
35         ▷ Spawn new tasks, which construct c ∨ d and e ∨ f
36         f ← sync; e ← sync; d ← sync; c ← sync
37         spawn(ITE, c, 1, d); spawn(ITE, e, 1, f)
38         d ← sync; c ← sync
39         result ← make-node(v, c, d)
40     else
41         ▷ In this case, variable v is not in vars
42         spawn(RelProd, a_0, b_0, v); spawn(RelProd, a_1, b_1, v)
43         r_1 ← sync; r_0 ← sync
44         result ← make-node(v, r_0, r_1)
45     ▷ Add result to cache and return
46     cache-put(a, b, v, result)
47     return result
```

Figure 5.7: Implementation of `RelProd`

While performing the experiments, we found out that our work stealer, in combination with the BDD implementation, contains a bug. This causes the BDD implementation to crash in about 50% of the runs when using small models (i.e. models with a relatively small state space), and even more often when larger models are used. This is because an erroneous interleaving of the execution is possible, which causes the program to crash or produce incorrect results. For this reason, continuing the evaluation with this bug was unpractical. Due to time constraints, we were also not able to either find or fix the bug in time. As an alternative, we used HotSLAW instead. Furthermore, we did not encounter such problems when evaluating the implementation of the work stealers. We assume that the bug was introduced after integrating our work stealer with the BDD implementation. We suspect the error to be caused by the stack sizes, as they might be too small, especially for larger models. For this reason, we expect the experimental evaluation of the work stealers to be valid, despite this bug.

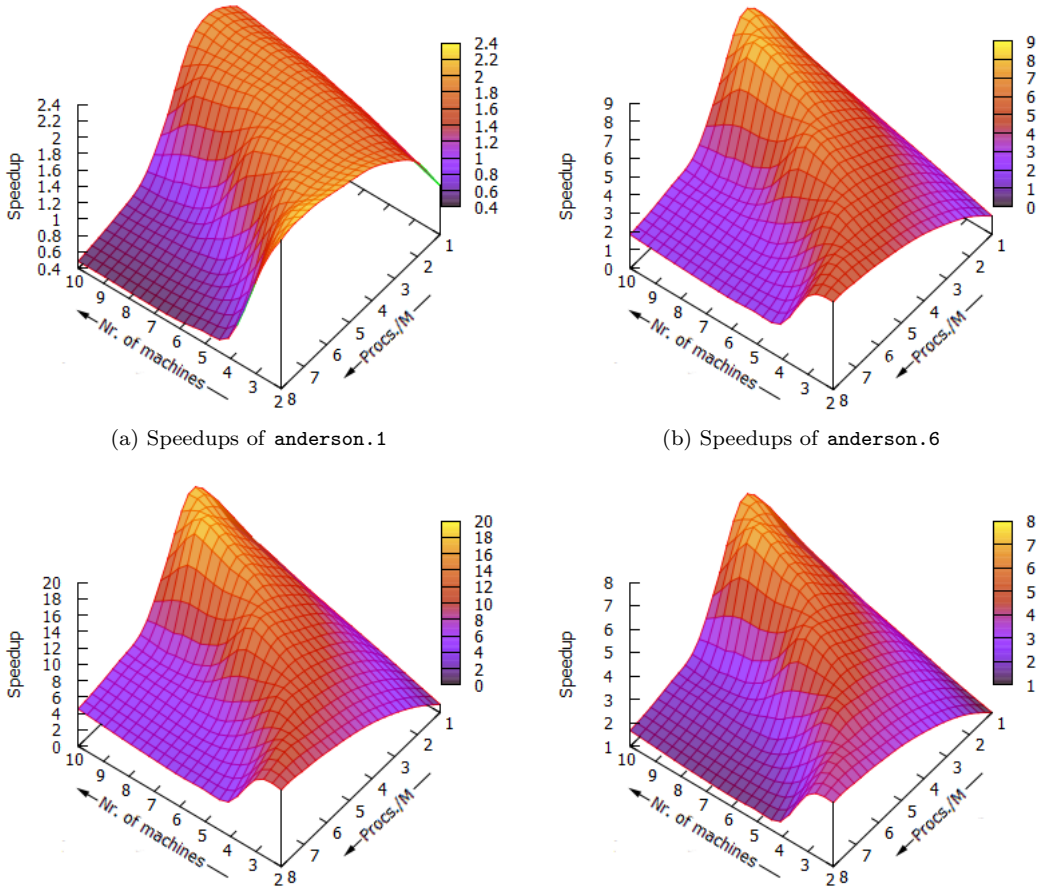
5.5.1 Distributed Scalability

We evaluated the scalability along the number of processes and machines in detail. The four smallest models of the BEEM database have been used, which are: `anderson.1`, `anderson.2`, `at.5`, and `schedule_world.2`. Evaluation is performed by scaling the number of machines from 2 to 10 and the number of processes per machine from 1 to 8 (which makes a total of 72 different configurations). The benchmarks have been repeated five times for every model and the average measurements have been taken into account.

We observed that benchmarking larger models in such detail is unpractical, because it would take too much time (benchmarking `at.5` five times already took more than 26 hours, benchmarking even larger models in such detail could take weeks). Instead of executing them 72 times, they are only executed 8 times by scaling only along the number of machines and choosing smart amounts of processes per machine. Details and experimental results of these benchmarks are given later in this chapter.

Figure 5.8 shows the relative speedup obtained when performing reachability over the four smallest BEEM models. In contrast to the evaluation of the work stealers, where we only used one process to determine relative speedup, here we use two processes. The tasks generated by the work stealing benchmarks do not perform RDMA operations, with the exception of `matmul`. This is the main reason for the good speedups achieved in `fib`, `uts`, and `nqueens` (shown in Figure 4.10). The tasks generated by the BDD operations, on the other hand, often perform multiple RDMA operations. When using only one machine, RDMA operations are not performed because all shared memory is local. This gives an unfair performance advantage when determining the distributed speedup when scaling along machines (the speedup graphs would then be very similar to the graph of `matmul(512)`, presented in Figure 4.10).

Similar to the results obtained in the evaluation of the hash table, scaling both along processes and machines results in a fairly low speedup. For example, the highest speedup obtained in `schedule_world.2` is 6.96 when using 30 processes (10 machines, each having 3 processes), which is an efficiency of only 23.2%. Another observation, which is also similar to the hash table behaviour, is that the throughput of the RDMA devices gets saturated from some point on, which results in a steep performance drop. This causes the speedup graphs of most of the models to show a waving pattern. The peaks in the speedup graph of `anderson.1` are constant (they are all about 1.9). This is because the computation depth of `anderson.1` is relatively large, and relatively little work is done on every depth, followed by termination detection. This has a negative impact on speedup. The `at.5` model achieves reasonable speedup, namely 17.9 with 30 processes (10 machines, each having 3 processes), which gives an efficiency of 67.5%. Table 5.9 shows the best achieved execution times and speedups.



(a) Speedups of `anderson.1`

(b) Speedups of `anderson.6`

(c) Speedups of `at.5`

(d) Speedups of `schedule_world.2`

Figure 5.8: The distributed speedup when scaling both along machines and the number of processes per machine (Procs./M).

Distributed Scalability on Larger Models

To perform symbolic reachability on the larger BEEM models, we scale only along the number of machines and choose an *optimal* number of processes per machine. The optimum is taken from the results presented in Figure 5.8 and shown in Figure 5.10. A surprising observation is that a configuration of 9 machines performs best when 36 processes are used, in contrast to 30 processes in a setting with 10 machines. It is reasonable to expect that a setting with 10 machines requires more processes to saturate the throughput of all RDMA devices than with 9 machines. When scaling from 3 to 4 processes per machine in a configuration with 10 machines, an average performance drop of 46% is observed. When scaling from 4 to 5 processes per machine in a setting with 9 machines, an average drop of 136% is observed, which is far more significant. Out of this, we expect that a configuration with 10 machines may even perform better when more than 30, but less than 40 processes are used in total. We did not verify this expectation due to time constraints and the fact that, by default, only a fixed amount of processes per machine can be used by UPC. It is, however, possible to either dynamically spawn additional processes or to

Model	Base configuration			Best configuration			Speedup
	Time	M.	P./M.	Time	M.	P./M.	
`anderson.1`	19.93	2	1	8.86	2	7	2.25
`anderson.6`	362.73	2	1	45.61	10	3	7.95
`at.5`	1,730.20	2	1	96.51	10	3	17.93
`schedule_world.2`	130.52	2	1	18.75	10	3	6.96

Figure 5.9: The best achieved execution times and speedups when scaling along both machines (M.) and processes (P./M.). Execution time is given in seconds. The speedup is calculated relative to the given base configurations.

Machines	Processes per machine	Total nr. of processes
2	8	16
3	7	21
4	6	24
5	5	25
6	4	24
7	4	28
8	4	32
9	4	36
10	3	30

Figure 5.10: The number of processes used in every configuration used in the evaluation of the distributed scalability with the larger models.

spawn extra processes and disable a few, but our implementation does not support that. The same observation applies to the configuration with 6 machines.

5.5.2 Parallel Scalability

Figure 5.13 shows the speedups of distributed reachability with respect to parallel reachability. By using one thread, the implementation seems to require some extra overhead, which leads to a slight decrease in performance. Furthermore, the speedup of `collision.5` could *not* be determined because its state space does not fit into the memory of a single machine. Finally, the computation time of `anderson.8` with 8 threads could not be determined, because the benchmark crashed. The crash was caused by a segmentation fault that occurred immediately after opening the BDD file.

Even when considering that single threaded runs require some extra overhead, in nearly all cases the distributed runs result in speedups. The `anderson.1` model obtains the *lowest* speedup, since it has a large computation depth and on every depth only a small number of computations are performed. We expect that, due to termination detection on each phase, speedup remains low. The *highest* speedups are obtained with the largest models (i.e. `at.7` and `anderson.8`). Although we could not determine the speedup of `collision.5`, we expect that this model may obtain the highest speedup.

On the other hand, Figure 5.13 also shows that, in a number of cases, the parallel runs are *faster* than the distributed runs. By having a parallel execution, data locality is better exploited, since all memory is local. We may thus conclude that distributed executions are generally faster than sequential executions, but not as fast as parallel executions. However, these performance

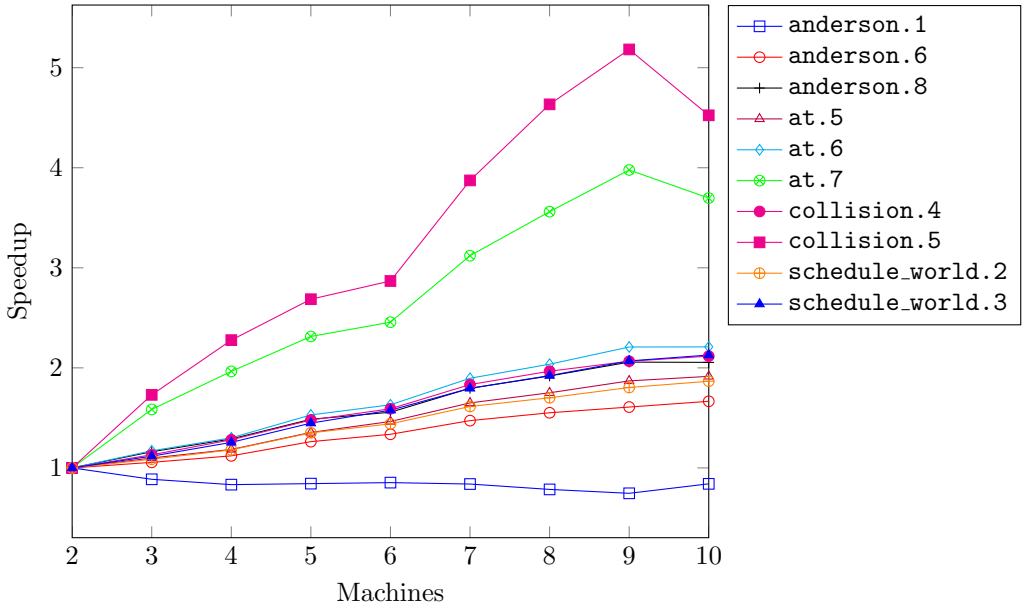


Figure 5.11: The relative speedup of the BDD implementation when scaling along the number of machines.

differences get smaller when the models get larger. For example, `at.5` still has a *much* better distributed computation time, and we highly expect that the distributed runs of `anderson.8` and `collision.5` are also much faster than 8-threaded runs.

5.5.3 Scalability with Private Deque Work Stealing

Figure 5.14 shows details of the models used to evaluate the BDD implementation. The models `anderson.1` and `schedule_world.2` were small enough to be executed with our private deque work stealer without crashing that often. Our work stealer crashed or produced incorrect results in 161 out of 288 runs for `anderson.1` (a failure-rate of 55%) and 144 out of 288 runs for `schedule_world.2` (a failure-rate of 50%).

We determined the relative speedup of the BDD operations implemented with our work stealer with respect to the HotSLAW implementation. We discarded the incorrect results produced by our work stealer. The speedup is determined by considering the lowest execution times achieved by the different implementations when scaling from 2 to 10 machines. Figure 5.15 presents the results of the comparison. Note that, due to the failure rate, these results are only indicative and may not accurately represent the performance of our work stealer with respect to HotSLAW. The `schedule_world.2` model only achieves a small speedup (up to 4.5%), but `anderson.1` achieves a significant speedup (up to 56.5%). This is because `anderson.1` requires more steals, since it has a large execution depth (namely 1293). On every depth, the work stealers need to perform termination detection, which requires many steal attempts. We already concluded that our work stealer has more efficient steals than HotSLAW, which explains the differences in performance. Every process performs termination detection separately, so the amount of work for termination detection increases when increasing the number of processes, which explains the speedup. Most models, however, have a relative small execution depth (very similar to `schedule_world.2`), so we expect our work stealer to perform only slightly faster than HotSLAW for most of the other

Model	2 Machines	Best Set-up		Speedup
		Machines	Time	
anderson.1	8.99	10	10.69	0.84
anderson.6	75.97	10	45.61	1.67
anderson.8	645.74	9	313.84	2.06
at.5	184.83	10	96.51	1.92
at.6	785.44	10	355.44	2.20
at.7	3,347.50	9	841.54	3.98
collision.4	779.89	10	368.38	2.12
collision.5	7,141.96	9	1,378.05	5.18
schedule_world.2	35.00	10	18.75	1.87
schedule_world.3	728.41	10	342.22	2.13

Figure 5.12: Average computation times (in seconds) of *distributed* reachability performed on the BEEM models.

Model	Parallel		Distributed		Speedup
	1 Thread	8 Threads	Machines	Time	
anderson.1	4.80	14.45	10	10.69	0.45
anderson.6	74.45	45.61	10	45.61	1.63
anderson.8	4,656.50	-	9	313.84	14.84
at.5	2,19.84	32.17	10	96.51	2.28
at.6	2,386.63	206.77	10	355.44	6.71
at.7	23,665.80	2,030.55	9	841.54	28.12
collision.4	2,500.06	197.78	10	368.38	6.79
collision.5	-	-	9	1,378.05	-
schedule_world.2	30.23	5.51	10	18.75	1.61
schedule_world.3	1,833.15	145.04	10	342.22	5.36

Figure 5.13: A comparison of execution times between parallel and distributed symbolic reachability, performed in several BEEM models.

models.

5.5.4 Suggestions for Performance Improvements

Figure 5.8 shows that the performance increases when adding processes, but to a certain extent. At some point, adding more processes per machine causes the throughput of the RDMA devices to be saturated, which has a negative impact on performance. The sweet spots are shown in Figure 5.10. We also argued that the scalability can be improved even further when slightly more processes are added. This may also be very dependent on the model, which is shown by the divergence of the speedup graph of `anderson.1` in Figure 5.8. We suggest to implement adaptive load-balancing, which disables further execution of a number of processes when the performance drops. This can be done by preventing steal operations on a number of processes, so that only a subset of the processes remain active. We expect that adaptive load-balancing increases scalability.

The `ITE` and `RelProd` operations often create new BDD nodes via `make-node` as a result of their computations. It would be possible for `make-node` to return the machine at which the node is stored (i.e. the *owner* of that BDD node). It would therefore also be possible to

Model	Depth	Groups	States	BDD nodes
anderson.1	1,293	6	352,664	22,221
anderson.6	181	18	18,206,917	75,220
anderson.8	246	21	538,699,029	285,148
at.5	51	26	31,999,440	156,785
at.6	95	26	160,589,600	420,526
at.7	57	31	819,243,816	986,322
collision.4	168	29	41,465,543	24,960
collision.5	182	29	431,965,993	29,537
schedule_world.2	18	26	1,570,340	18,779
schedule_world.3	23	34	166,649,331	28,500

Figure 5.14: Details of the BEEM models used to evaluate the implementation of the BDD operations.

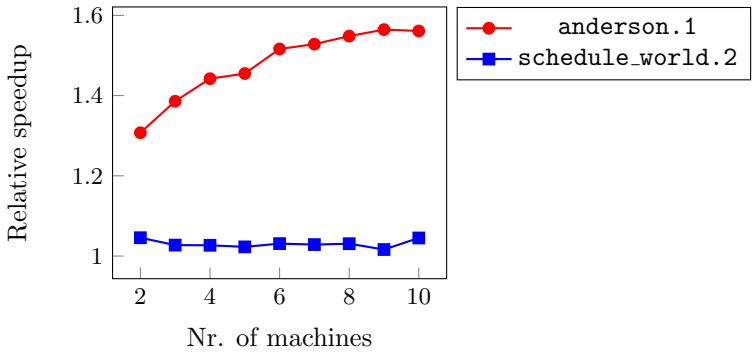


Figure 5.15: Relative speedups of our work stealing implementation with respect to HotSLAW. Only the smallest two BEEM models are used.

approximate the machine that would achieve best performance when exploring a BDD node. This approximation can be determined via the recursion trees of `ITE` and `RelProd`. We expect that steals become more efficient when using this approximation. This is because data-locality is then exploited to some extent, which is currently not done. By exploiting data-locality, less RDMA operations are performed, which increases total throughput and allows CPUs to be utilized more efficiently.

5.6 Conclusion

We have implemented several BDD operations that efficiently use the memory of all participating computers in a high-performance network. Hierarchical work stealing is used as a method to partition the computational work into a large number of small tasks that are dispatched to the participating processors. The BDD operations have been used to perform symbolic reachability over a number of BEEM models, with positive results.

When using a network of workstations, speedups up to 28 are obtained relative to single-threaded runs. Generally, the larger BEEM models achieve the highest speedups. Furthermore, adding computers to the network does *not* degrade performance, unless enough processes are used to saturate the RDMA devices. We thus conclude that the performance of BDD operations

scales *well* along the number of machines added to the network.

On the other hand, the implementation does not scale very well along the number of processes per machine, due to bandwidth limitations. This scalability can very well be improved by reducing the number of roundtrips, and we have suggested various ways to do that in the Sections: 3.5.3, 4.4.3, and 5.5.4 and in the conclusions of Chapters 3 and 4.

Compared to distributed scalability, the parallel executions achieve lower computation times in a number of cases. This is not surprising, since parallel runs have much better data locality. The differences in performance, however, get smaller when the BEEM models get larger. We therefore conclude that distributed runs are faster than sequential runs, but do not scale as well as parallel runs.

To conclude, by paying an *acceptable* performance price (compared to parallel runs), the total available memory of a network of workstations can be used *efficiently* for symbolic reachability, and adding workstations does not decrease performance. This makes distributed symbolic verification feasible and usable in practice. Considering the many performance improvements that can still be applied and the potential scalability of a network of workstations, performing further research in distributed (and even heterogeneous) verification algorithms could give *very* promising results.

Chapter 6

Conclusion

In this thesis, we described an RDMA-based distributed hash table and hierarchical private deque work stealing operations. Both parts have been evaluated in detail via micro-benchmarking in order to determine their behaviours. We used both parts to implement distributed BDD operations and evaluated its scalability by performing symbolic reachability over a number of BEEM models.

We expected roundtrip latency to be the bottleneck of distributed reachability analysis, but instead the throughput of the RDMA devices appeared to be the biggest limitation. When enough processes per machine are used, the throughput of the RDMA devices get saturated, which *drastically* degrades performance. As a result, we achieve good scalability along the number of machines added to the network, but the scalability along processes per machine remains limited. The scalability can be improved by reducing the number of roundtrips and by using hardware that supports higher bandwidths. Both suggestions are feasible, as the price of Infiniband hardware is comparable to standard Ethernet hardware, and we suggested various effective ways to reduce the number of roundtrips.

6.1 Efficient Distributed Symbolic Reachability

Chapter 1 mentions four design considerations that are important for efficient distributed symbolic state-space construction. In addition, we presented three subquestions that cover these four considerations. In this section, we discuss each of them.

6.1.1 Data Distribution

To answer **SQ1**, we looked at several existing RDMA-based distributed hash tables and discussed their efficiencies with respect to the number of required roundtrips. We argued that linear probing would be more efficient than the collision strategies used in existing work for an efficient implementation of `find-or-put`. We use chunk retrievals to fetch multiple buckets in a single roundtrip, as the buckets examined by linear probing are consecutive in memory. Moreover, we overlap roundtrips as much as possible to further reduce latency.

As a result, `find-or-put` only requires less than 2 roundtrips on average to find the intended bucket when a reasonable chunk size is chosen (i.e. 32 or 64). Furthermore, by using 64-sized chunks, `find-or-put` achieves a peak-throughput of 3.6×10^6 op/s and requires $9.3\mu s$ on average to find the intended bucket.

6.1.2 Load-balancing Maintenance

To answer **SQ2**, we investigated techniques for fine-grained task parallelism, in particular work stealing, as it has already been successfully used in parallel symbolic reachability [75]. Existing distributed task-based parallel frameworks consider the hierarchy of the network to achieve good scalability. Nearly all implementations use split deques to implement their task pools and require expensive locking structures when stealing work. We use private deque work stealing instead, which is lock-free and minimizes the number of roundtrips required for steals. A similar approach has been proposed in [50], but requires more roundtrips and does not consider the hierarchy of the network. Our implementation uses hierarchical victim selection, as proposed by HotSLAW, which gives performance improvements up to 52% compared to random victim selection [62]. Due to time constraints, we could not implement hierarchical chunk selection [36], which could also provide serious performance improvements (up to 122% in [62]).

Our private deque work stealing implementation achieves speedups up to 68.7 when using 80 threads. Compared to HotSLAW, the latency for remote `steal` operations are 9.3 times *lower* (and even 34.3 times lower for local steals). On the other hand, HotSLAW generates less overhead when executing tasks, which makes it more efficient in some cases. As future work, we would like reduce the overhead generated when executing tasks, remove explicit termination detection, and implement hierarchical chunk selection to further improve performance.

6.1.3 Communication Overhead

To answer **SQ3**, we tried to use asynchronous roundtrips as much as possible. We expected network latency to be the biggest bottleneck, so we attempted to perform as much work as possible *during* roundtrips to hide latency. As a result, we designed a querying system for the distributed hash table, so that consecutive chunks are retrieved while iterating over a chunk. This reduces waiting times, but also increases the number of RDMA operations required by the hash table. Ultimately, not the network latency, but the throughput of the RDMA devices appeared to be a bottleneck. We propose a strategy to determine *adaptive chunk sizes* to reduce the number of roundtrips, which increases performance and ultimately also reduces waiting times.

6.2 Scalability of Distributed Symbolic Reachability

To answer our main research question, we implemented distributed BDD operations by using hierarchical work stealing and our RDMA-based distributed hash table. We used these operations to implement distributed symbolic reachability. With experimental evaluation we conclude that the BDD operations scale well across a network of workstations. Compared to sequential executions, speedups up to 28 are reached. However, the implementation does not scale very well along the number of processes per machine, due to bandwidth limitations of the RDMA devices.

We also concluded that parallel scalability is better than distributed scalability, which is reasonable since parallel runs can better exploit data locality. However, the larger BEEM models used for evaluation still achieve very good distributed performance compared to the parallel runs.

To conclude, by paying an acceptable performance price (compared to *parallel* symbolic reachability), the total combined memory of a network of workstations can be efficiently used. Adding machines to the network does not decrease performance, which introduces scalability. Considering that performance can still be improved, distributed symbolic reachability analysis is feasible and usable in practice.

6.2.1 Future Work

To improve performance and scalability of distributed symbolic reachability, minimizing the number of RDMA operations is *key*. This thesis presents many ways to achieve this, but the most important considerations which we expect to yield the highest benefits are summarized in Sections 3.6 and 4.5.

Bibliography

[1] Infiniband trade association, 2008.

[2] Osu micro-benchmarks, 2015.

[3] Performance numbers of mvapich2 on intel ivybridge architecture with mellanox connectx-3, 2015.

[4] Sheldon B. Akers. Binary decision diagrams. 27(6):509–516, 1978.

[5] Orion Hodson Miguel Castro Aleksandar Dragojević, Aleksandar Narayanan. Farm: Fast remote memory. In *11th USENIX Conference on Networked Systems Design and Implementation, NSDI*, volume 14, 2014.

[6] David Gay Susan Graham Paul Hilfinger Arvind Krishnamurthy Ben Liblit Carleton Miyamoto Geoff Pike Luigi Semenzato Alex Aiken, Phil Colella et al. Titanium: A high-performance java dialect. *Concurrency: Practice and Experience*, 10:11–13, 1998.

[7] Jaco van de Pol Alfons Laarman and Michael Weber. Boosting multi-core reachability performance with shared hash tables. In *Conference on Formal Methods in Computer-Aided Design*, pages 247–256. FMCAD, 2010.

[8] Bruce Hendrickson Andrew Lumsdaine, Douglas Gregor and Jonathan Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[9] Ramnik Jain Ankur Narang, Abhinav Srivastava and R.K. Shyamasundar. Dynamic distributed scheduling algorithm for state space search. In *Parallel Processing*, pages 141–154. Springer, 2012.

[10] Michael Kaminsky Anuj Kalia and David G. Andersen. Using rdma efficiently for key-value services. In *ACM conference on SIGCOMM*, pages 295–306. ACM, 2014.

[11] Edmund M. Clarke Ofer Strichman Armin Biere, Alessandro Cimatti and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 58:117–148, 2003.

[12] Mats Brorsson Artur Podobas and Karl-Filip Faxén. A comparison of some recent task-based parallel programming models. 2010.

[13] Anurag Srivastava Arun Narang and R.K. Shyamasundar. High performance adaptive distributed scheduling algorithm. In *Parallel and Distributed Processing Symposium Workshops*, pages 1725–1734. IEEE, 2013.

[14] Swaroop Pophale Stephen Poole Jeff Kuehn Chuck Koelbel Barbara Chapman, Tony Curtis and Lauren Smith. Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, page 2. ACM, 2010.

[15] David G. Andersen Bin Fan and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Networked Systems Design and Implementation*, pages 371–384, 2013.

[16] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. 46(5):720–748, 1999.

[17] Robert D. Blumofe and Dionisios Papadopoulos. The performance of work stealing in multiprogrammed environments. In *Measurement and Modeling of Computer Systems*, volume 26, pages 266–267. ACM, 1998.

[18] Beate Bollig and Ingo Wegener. Improving the variable ordering of obdds is np-complete. 45(9):993–1002, 1996.

[19] Hans P. Zima Bradford L. Chamberlain, David Callahan. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[20] Aaron R. Bradley. Ic3 and beyond: Incremental, inductive verification. In *Computer Aided Verification*, page 4, 2012.

[21] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. 35(8):677–691, 1986.

[22] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. 24(3):293–318, 1992.

[23] Lawrence J. Carter and Mark N. Wegman. Universal classes of hash functions (extended abstract). In *Ninth Annual ACM Symposium on Theory of Computing*, pages 106–112. ACM, 1977.

[24] Yifeng Geng Christopher Mitchell and Jinyang Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114, 2013.

[25] Ming-Ying Chung and Gianfranco Ciardo. Saturation now. In *Quantitative Evaluation of Systems*, pages 272–281. IEEE, 2004.

[26] Ming-Ying Chung and Gianfranco Ciardo. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *Parallel and Distributed Processing Symposium*. IEEE, 2006.

[27] Tarek El-Ghazawi and Lauren Smith. Upc: Unified parallel c. In *ACM/IEEE Conference on Supercomputing*, page 27. ACM, 2006.

[28] Karl-Filip Faxén. Wool-a work stealing library. 36(5):93–100, 2009.

[29] Karl-Filip Faxen. Efficient work stealing for fine grained parallelism. In *Parallel Processing (ICPP)*, pages 313–322. IEEE, 2010.

[30] Kathi Fisler and Moshe Y. Vardi. Bisimulation and model checking. In *Correct Hardware Design and Verification Methods*, pages 338–342. Springer, 1999.

[31] Maryam Haji Ghasemi. Symbolic model checking using zero-suppressed decision diagrams, November 2014.

[32] Yang Zhao Gianfranco Ciardo and Xiaoqing Jin. Parallel symbolic state-space exploration is difficult, but what is the alternative? 2009.

[33] Shoham Ben-David Ilan Beer and Avner Landver. On-the-fly model checking of rctl formulas. In *Computer Aided Verification*, pages 184–194. Springer, 1998.

[34] Robert K. Brayton Jagesh V. Sanghavi, Rajeev K. Ranjan and Alberto L. Sangiovanni-Vincentelli. High performance bdd package by exploiting memory hierarchy. In *Design Automation Conference*, pages 635–640. ACM, 1996.

[35] Brian B. Larkins Jarek Nieplocha James Dinan, Sriram Krishnamoorthy and Ponnuswamy Sadayappan. Scioto: A framework for global-view task parallelism. In *Parallel Processing*, pages 586–593. IEEE, 2008.

[36] Ponnuswamy Sadayappan Sriram Krishnamoorthy James Dinan, Brian D. Larkins and Jarek Nieplocha. Scalable work stealing. In *High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.

[37] David R. O'hallaron Jaspal Subhlok, James M. Stichnoth and Thomas Gross. Exploiting task and data parallelism on a multicomputer. In *Principles and Practice of Parallel Programming*, volume 28, pages 13–22. ACM, 1993.

[38] Olivier Tardieu Jeeva Paudel and José N. Amaral. On the merits of distributed work-stealing on selective locality-aware tasks. In *Parallel Processing (ICPP)*, pages 100–109. IEEE, 2013.

[39] David E. Long Kenneth L. McMillan Jerry R. Burch, Edmund M. Clarke and David L. Dill. Symbolic model checking for sequential circuit verification. 13(4):401–424, 1994.

[40] Kenneth L. McMillan David L. Dill Jerry R. Burch, Edmund M. Clarke and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Logic in Computer Science*, pages 428–439. IEEE Computer Society, 1990.

[41] Gerald Lüttgen Jonathan Ezekiel and Gianfranco Ciardo. Parallelising symbolic state-space generators. In *Computer Aided Verification*, pages 268–280. Springer, 2007.

[42] Laxmikant V. Kale and Sanjeev Krishnan. *CHARM++: A Portable Concurrent Object Oriented System Based on C++*, volume 28. ACM, 1993.

[43] Sangho Lee Kaushik Ravichandran and Santosh Pande. Work stealing for multi-core hpc clusters. In *Parallel Processing*, pages 205–217. Springer, 2011.

[44] S. Kimura and E.M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Computer Design (ICCD): VLSI in Computers and Processors*, pages 220–223. IEEE, 1990.

[45] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.

[46] Nir Shavit Maurice Herlihy and Moran Tzafrir. Hopscotch hashing. In *Distributed Computing*, volume 5218, pages 350–364. Springer Berlin Heidelberg, 2008.

[47] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[48] Kim Milvang-Jensen and Alan J. Hu. Bddnow: a parallel bdd package. In *Formal Methods in Computer-Aided Design*, pages 501–507. Springer, 1998.

[49] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17, pages 1–31. ACM, 1998.

[50] Stephen Olivier and Jan Prins. Scalable dynamic load balancing using upc. In *Parallel Processing*, pages 123–131. IEEE, 2008.

[51] Tamir Heyman Orna Grumberg and Assaf Schuster. A work-efficient distributed algorithm for reachability analysis. In *Computer Aided Verification*, pages 54–66. Springer, 2003.

[52] Rasmus Pagh and Flemming F. Rodler. Cuckoo hashing. *Journal on Algorithms*, 51(2):122–144, 2004.

[53] Radek Pelánek. Beem: Benchmarks for explicit model checkers. In *Model Checking Software*, pages 263–267. Springer, 2007.

[54] Gregory F. Pfister. An introduction to the infiniband architecture. *High Performance Mass Storage and Parallel I/O*, 42:617–632, 2001.

[55] Vijay Saraswat Christopher Donawa Allan Kielstra Kemal Ebcioglu Christoph Von Praun Philippe Charles, Christian Grothoff and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *Acm Sigplan Notices*, 40(10):519–538, 2005.

[56] Craig M. Chase Prakash Arunachlam and Dinos Moundanos. Distributed binary decision diagrams for verification of large circuits. In *Proceedings of Computer Design (ICCD): VLSI in Computers and Processors*, pages 365–370. IEEE, 1996.

[57] Thomas A. Henzinger Shaz Qadeer Rajeev Alur, Robert K. Brayton and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351. Springer, 1997.

[58] Robert K. Brayton Rajeev K. Ranjan, Jagesh V. Sanghavi and Alberto L. Sangiovanni-Vincentelli. Binary decision diagrams on network of workstations. In *Computer Design: VLSI in Computers and Processors*, pages 358–364. IEEE, 1996.

[59] Dharavath Ramesh and Alok K. Pani. An incremental load balancing approach for heterogeneous distributed processing systems. 5, 2014.

[60] Bradley C. Kuszmaul Charles E. Leiserson Keith H. Randall Robert D. Blumofe, Christopher F. Joerg and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[61] Pavan Balaji Ryan E. Grant, Mohammad J. Rashti and Ahmad Afsahi. Scalable connectionless rdma over unreliable datagrams. *Parallel Computing*, 48:15–39, 2015.

[62] Costin Iancu Seung-Jai Min and Katherine Yelick. Hierarchical work stealing on manycore clusters. In *Partitioned Global Address Space Programming Models*, 2011.

[63] Nagisa Ishiura Shin-ichi Minato and Shuzo Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Design Automation Conference*, pages 52–57. IEEE, 1990.

[64] Jaco van de Pol Stefan Blom and Michael Weber. Ltsmin: Distributed and symbolic reachability. In *Computer Aided Verification*, pages 354–359. Springer, 2010.

[65] Ryan Stutsman Mendel Rosenblum Stephen M. Rumble, Diego Ongaro and John K. Ousterhout. Its time for low latency. In *USENIX Conference on Hot topics in Operating Systems*, page 11. USENIX Association, 2011.

[66] Jinze Liu Jan Prins James Dinan Ponnuswamy Sadayappan Stephen Olivier, Jun Huan and Chau-Wen Tseng. Uts: An unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing*, pages 235–250. Springer, 2007.

[67] B. Cassell T. Szepesi, B. Wong and T. Brecht. Designing a low-latency cuckoo hash table for write-intensive workloads using rdma. In *First International Workshop on Rack-scale*, 2014.

[68] Orna Grumberg Tamir Heyman, Danny Geist and Assaf Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *Computer Aided Verification*, pages 20–35. Springer, 2000.

[69] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[70] Alfons Laarman Tom van Dijk and Jaco van de Pol. Multi-core bdd operations for symbolic reachability. pages 127–143, 2012.

[71] Arthur Charguéraud Umut A. Acar and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *PPOPP*, volume 48, pages 219–228. ACM, 2013.

[72] Jaco van de Pol. *MaDriD Multi-Core Decision Diagrams.* 2010.

[73] Tom van Dijk. The parallelization of binary decision diagram operations for model checking. 2012.

[74] Tom van Dijk and Jaco van de Pol. Lace: non-blocking split deque for work-stealing. In *Euro-Par 2014: Parallel Processing Workshops*, pages 206–217. Springer, 2014.

[75] Tom van Dijk and Jaco van de Pol. Sylvan: Multi-core decision diagrams. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 677–691. Springer, 2015.

[76] Georgios Varisteas and Mats Brorsson. Palirria: Accurate on-line parallelism estimation for adaptive work-stealing. In *Programming Models and Applications on Multicores and Manycores*, page 120. ACM, 2014.

[77] David B. Wagner and Bradley G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *PPOPP.*, volume 28, pages 208–217. ACM, 1993.

[78] Nathan Doss William Gropp, Ewing Lusk and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789–828, 1996.

[79] E. Stabler Y. Parasuram and S.K. Chin. Parallel implementation of bdd algorithms using a distributed shared memory. In *27th Hawaii International Conference on System Sciences, Volume 1: Architecture*, pages 16–25. IEEE, 1994.

[80] Bwolen Yang and David R. O'Hallaron. Parallel breadth-first bdd construction. In *PPOPP*, volume 32, pages 145–156. ACM, 1997.

[81] Qi Zuo Yizhuo Wang, Weixing Ji and Feng Shi. A hierarchical work-stealing framework for multi-core clusters. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 350–355, Dec 2012.