Dresden University of Technology

Faculty of Computer Science

Institute of Theoretical Computer Science

Chair for Algebraic and Logical Foundations of Computer Science

Minor Thesis

Development and Analysis of Barrier Protocols

Author: Ronny Brendel
Responsible University Professor: Prof. Dr. Christel Baier
Supervisor: Dr. Sascha Klüppelholz

Vienna, 5th December, 2013

Task

1. Literature research and detailed survey of currently used barrier protocols as well as implementation possibilities for barriers.
2. Analysis and improvement of Nicolas Mc Guire's proposed barrier.
3. Identification and formalisation of key functional properties concerning correctness as well as quantitative aspects for determining performance of barrier protocols with regard to energy consumption and speed.
4. Modelling and quantitative analysis:
   - Modelling a) a prominent representative of barrier protocols for distributed memory, b) a prominent representative of barrier protocols for shared memory, c) the barrier protocol based on Mc Guire's idea, in each case using the probabilistic model checker PRISM.
   - Analysis and comparison of the three above-mentioned models, regarding the functional as well as quantitative properties identified in 3., using PRISM. If possible: complement the performance evaluation of the three barriers using measurement-based methods.
5. Summary and outlook: Discussion of the insights gained in 4.

Statement of Academic Integrity

I hereby declare that I prepared this thesis independently and without use of tools other than specified. Foreign thoughts, taken literally or in spirit, are marked as such. I also declare that I have not filed the present work at any other location or will submit it.

Contents

1 Introduction **3**

2 Background **5**
2.1 Means to Implement Barriers 5
2.1.1 Shared Memory 5
2.1.2 Distributed Memory 6
2.2 Current Barriers 9
2.2.1 Shared Memory 10
2.2.2 Distributed Memory 11
2.3 Barrier Building Blocks 13

3 Innovative Barriers **15**
3.1 MGB Barrier 15
3.2 B1 Barrier 16
3.3 B2 Barrier 17

4 Analysis and Comparison **21**
4.1 General Properties 21
4.2 Model Checking 24
4.2.1 Preliminaries 25
4.2.2 Modelling 26
4.2.3 Functional Properties 31
4.2.4 Quantitative Properties 32
4.3 Discussion 43

5 Conclusion and Future Work **45**

Bibliography **47**

1 Introduction

Synchronisation is a central element of parallel programs. Various synchronisation mechanisms with different goals have been devised. One such mechanism is the barrier. If a group of threads or processes executes a barrier operation, each of them must wait at this point until all other threads/processes arrive at the barrier.

Many synchronisation operations imply a barrier. For example the parallel for loop of the OpenMP [39] programming interface makes sure that no thread continues program execution until all iterations are completed, since the following operations might rely on the results of the loop. Likewise in MPI [36], global operations like reduce, scatter and gather imply a barrier synchronisation.

Many parallel programs use a lock-step approach to interleave computation with communication phases. For example a program simulating weather divides the map into a grid. Each square in this grid is assigned to one process, so that the weather progression of the next few minutes, for all squares, can be simulated concurrently. Changes in weather condition at the borders of a square influence the next simulation step of neighbouring squares. Therefore, before entering the next simulation phase, information, like temperature, humidity or cloud movement, about these regions is exchanged. Before beginning this exchange the program needs to make sure that each square has completed the current simulation step. To achieve that, they collectively invoke a barrier operation. Once it is completed, data is exchanged and the next simulation step starts.

The barrier is a commonly used synchronisation operation. According to a survey conducted at the High Performance Computing Center Stuttgart [44] in 2000, 5.3% of all time spent synchronising and 0.7% of the total program execution time is spent in barriers.

Today's barrier implementations use protocols that are fairly old. One example is the Dissemination Barrier [19], presented in 1988. This suggests that, due to the rapidly changing computer science and engineering landscape, one can improve upon existing barriers.

At a recent workshop, Nicholas Mc Guire proposed a new low-level synchronisation scheme, called probabilistic Write/Copy-Select (short pW/CS) lock [31]. It aims to improve the performance of exclusive access to shared memory. The core ideas, though, apply to synchronisation in general:

- Concurrent algorithms work, mostly, in a deterministic fashion. That is the arrangement of synchronisation is predetermined. The program is executed in a strict form. Faulty behaviour is avoided at great cost.

- One can improve performance by relieving this constraint. Tolerate inconsistencies, make errors detectable and/or ignorable.

- Because of the inherent randomness [32], induced by caching, scheduling, memory access latency differences, or short: the complexity of today's computer systems, one can view concurrent memory access as genuinely random. This enables the use of probabilistic algorithms for development, and the tools of probability theory for analysis.

One wants to design algorithms that have a sufficiently high probability of success, tolerate faulty behaviour and, on average, perform better than their deter-

ministic counterparts. We believe that synchronisation primitives, in particular barriers, can be improved by applying these principles.

To gain confidence that a program works properly, i.e. satisfies certain properties, one normally uses testing and benchmarks. Testing, as well as measurement, is usually deterministic. When testing probabilistic algorithms deterministically, there is always a chance of errors not being revealed. Randomising tests only gives a certain degree of confidence as well. Both methods do not guarantee full coverage. Two more drawbacks of measurement are that the granularity of the measurement scale is limited, and that measuring itself can influence the outcome. This makes analysing low-level synchronisation primitives, and small program pieces in general, hard and sometimes impossible. Model checking avoids these three problems. It is exhaustive, arbitrarily granular and, since it simulates the protocol, involves no measurement overhead. But model checking has, of course, its own set of shortcomings. First of all, the available tools are hard to use. Second, the accuracy of results depends on the quality of the model. Third, exhaustive analysis needs lots of hardware resources, and can therefore be infeasible even in relatively confined scenarios. Model checking, thus, complements measurement-based analysis.

In this thesis, we devise a set of novel pW/CS-influenced barrier protocols and compare them to state-of-the-art protocols using model checking techniques. One of these new barriers performs better than existing ones, regarding execution time and energy consumption.

The structure of this work is as follows. First, we survey the tools developers have at their disposal to implement barriers. Second, we present today's mainly used barrier algorithms. To conclude the background study, we give an account of the basic anatomy of barriers. We then move on to present three newly devised barrier protocols. The largest part of the thesis concerns the analysis and comparison of two known and two new barriers. We analyse them by means of an informal scrutiny combined with functional and probabilistic model checking. The final section summarises the presented work and gives an outlook on possible directions for future research.

2 Background

In this section we will first take look at the tools at our disposal to implement barriers, on both shared memory and distributed memory architectures, followed by an overview of which barrier algorithms are used in today's parallel programming frameworks. Finally we present a view of the barrier as a modularised algorithm consisting of various independent components.

2.1 Means to Implement Barriers

A shared memory system consists of a number of processors which are connected to one chunk of memory. Information can be exchanged between processors by simply reading from and writing to the same memory location, since they all share the same address space. Desktops, laptops and small to medium servers are typical examples of shared memory systems.

Many shared memory systems (called nodes) can be connected via a network to compose a distributed memory system. Such systems do not have one global memory, instead each of the nodes has its own private memory. Computation is done on each node using only local memory. In order to share information between nodes, messages via network need to be exchanged. For example large servers and high-performance computers are oftentimes distributed memory systems.

Programming shared memory systems is very different from developing for distributed memory architectures. Therefore, we will subsequently differentiate between these two worlds.

2.1.1 Shared Memory

A parallel program for shared memory systems consists of multiple threads of execution – short *threads*. The straightforward way to share information between threads is to *load* from and *store* to the same memory address.

Furthermore, most modern processors provide means to execute a small series of computations on a date while making the memory unavailable to other threads. They are called *atomic operations*. Two examples are the *test-and-set* and *add-fetch* instruction. The former stores a value to an address, if a given condition is fulfilled, the latter loads a date, adds a given value to it and stores it back.

Without these operations parallel programming is much more difficult and error-prone, since concurrently issuing normal load and store operations can easily cause *race conditions*. This convenience, on the other hand, comes at the cost of additional management overhead for the processor. To date shared memory systems usually have 64 threads or fewer. Some high-performance computing implementations reach 512 concurrent threads. In comparison distributed memory systems can have over a million processing units.

The intuitive ordering of concurrent memory accesses is *sequentially consistent*.

> *The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.* (Lamport [29])

Enforcing this intuitive semantic comes at a cost. Modern processors buffer memory operations on many levels. To enforce sequential consistency, mechanisms to control and circumvent this buffering have to be introduced, resulting in higher memory access latency and more transistors in the processor.

If an algorithm is carefully designed, one can weaken the consistency model to gain better performance. One prominent way of such a model is *release consistency* [13]. Memory access appears unordered except for so called *acquire* and *release* operations. No memory access that has been issued after an acquire operation is permitted to finish before this acquire operation. No memory access is allowed to finish later than the next release operation. Acquire and release encapsulate the memory accesses in between.

Aside from implementing barriers in software, vendors can wire a barrier implementation into the system's hardware. This is almost never done on shared memory architectures. One example for hardware barrier support is the SGI UV 2000 [45], a high-performance shared memory computer.

2.1.2 Distributed Memory

A parallel program for a distributed memory architecture consists of multiple *processes*. Each process has its own address space and is therefore unable to directly share memory with other processes.

First, we will give an overview of general ways to implement synchronisation primitives on distributed memory systems followed by an analysis of one specific distributed memory programming standard.

One way to exchange information between two processes is via *synchronous message passing*. Two processes meet at a point in time, exchange information, then part and continue their computational tasks. This method requires the involved peers to wait for each other, buffer and queue information and send it via network.

Another way to approach this is to wait as few as possible, since there is, for example, no need for a sending entity to wait until a receiver arrives. The sender might as well give its information to the communication framework and continue its computational work while the framework delivers the message as soon as the receiver is ready. This method is called *asynchronous message passing*. Aside from less waiting time, asynchronous message passing incurs similar overhead to synchronous message passing, because on the communication framework level the same work has to be done. The framework establishes network channels, buffers and queues data, and waits for the receiver to arrive.

A comparatively recent development is the move to *remote memory access* (RMA). Remote memory access makes a portion of main memory available via network, so that others can load from it and store to it without actively involving the memory's owning process. It is therefore sometimes called *one-sided communication*. Implementations of RMA oftentimes support atomic operations. In order for RMA to be available, the network hardware has to support it. For example InfiniBand [24] has RMA facilities built in. Due to its one-sided nature RMA incurs less overhead than the two previous techniques, but still buffers data. At very high networking bandwidths buffering data becomes a serious problem. To mitigate this, vendors implement so called *zero-copy* techniques.

Hardware support for barriers in distributed memory systems is widespread especially in high-performance computing. Examples include the Earth Simu-

lator [18], IBM Blue Gene/L [11], IBM Blue Gene/Q [7] and a low-cost FPGA-based system [21]

Another interesting and seemingly untested approach is to use lossy communication channels for developing synchronisation protocols. Hardware is always erroneous. For example network packages conflict, packages drop, packages are corrupted or connections drop all the time. Reliability-increasing techniques and protocols such as the Transmission Control Protocol (TCP), that are used to mitigate these errors, incur a great performance penalty for transmitting short messages. TCP for example creates comparatively huge packages for small amounts of data, and each transmitted package has to be acknowledged by the receiver. Checking for flipped bits in the package adds further latency. If one would instead craft algorithms so that they do not require 100% reliable connections, further performance improvements are conceivable. Two ways one could implement such algorithms are via the User Datagram Protocol (UDP) [42] and InfiniBand's unreliable connections.

We will now investigate the facilities the Message Passing Interface (MPI) [34], the most prominent high-performance distributed programming interface, provides to implement barrier protocols. MPI is a standardised, low-level, language-independent and portable interface specification. It is very customisable and supports various modes of operation, in order to support many different hardware configurations and allow the user to achieve maximum performance. MPI is widely adopted in high-performance computing and scientific computing [16, 41, 46] and there exist a variety of implementations in all major programming languages.

In our analysis we will only scratch the surface of what the standard offers, since diving deep involves lots of details, complicated semantics, and many of the details do not matter for our purpose.

MPI supports synchronous message passing via *send* and *receive*. Additionally, you can issue a send operation that immediately returns, if no receiver is already in place waiting for the sender.

Furthermore, MPI facilitates asynchronous message passing. One can send and receive messages without blocking, and one can test if a message has been successfully transmitted. Synchronous and asynchronous message passing can be mixed. That means one is able to, for example, issue a non-blocking send that is matched by a blocking receive operation.

Since version 3.0 the MPI standard supports *non-blocking collectives*. "Collectives" is a collective name for all communication operations where there is a variable number of communicating peers, like barrier, broadcast, reduce and more. Non-blocking collectives allow, in the example of the barrier, to not block at the operation, but to register at the barrier upon arrival and subsequently test the barrier for completion, doing computational work meanwhile. This facility is probably unsuited for building new barriers on top of it, but is nevertheless an interesting recent development.

MPI also supports RMA. Although some MPI implementations, for example Open MPI [40], do not yet support version 3.0 of the interface specification, we will restrict ourselves to analysing it, because it subsumes the features of version 2.2 [33], it is already a year old and supported by other prominent implementations such as MPICH 3.0 [35] and its derivatives MVAPICH2 [38] and Cray MPT [8].

The definition of RMA according to the standard is rather involved, and we

have to go into more detail to highlight important points.

To initiate sharing remote memory each participating process issues a collective call to create a *window* of memory to be shared. A window is associated with the group of processes that created it, and through it each participating process exposes a portion of local memory to the other peers. Upon creating a window, a number of restrictions regarding the use of the window can be established, in order to gain better performance. One can for example lower the memory consistency requirements or guarantee to not use certain operations.

Depending on the underlying hardware and MPI implementation there are two different memory models. A window may either have the *separate* or the *unified* model. The separate model states that the memory that is visible to the remote peers and the memory that is visible to the local process is not the same and might therefore not automatically update between these two copies. That means a change by a peer to one process' memory might not be made visible. This process has to manually update its local copy in order to see the change. In contrast the unified model makes sure that the two copies are eventually updated. Therefore, manual refreshing is unnecessary. This differentiation is a direct result of trying to support many different hardware configurations.

Before going into the details of window synchronisation we have to introduce the concept of *epochs*. An epoch is a period between two window synchronisation calls. RMA operations like *put*, *get*, *accumulate* may only be used inside an epoch. Epochs are used to accumulate multiple operations to one larger remote transfer, in order to achieve better efficiency.

MPI differentiates between *active* and *passive* target communication. In active target communication all peers are actively involved in window synchronisation, whereas in passive target communication only one side's activity is required.

One way to synchronise a window is via the *fence* operation. A fence is a collective operation among all window group members that makes sure all outstanding RMA requests on this window are finished before the operation returns. It implies a barrier synchronisation.

A more fine-grained instance of active target communication is via the operations *start*, *complete*, *post* and *wait*. Before issuing any remote access calls, the issuing process first has to advertise its intention through the start operation. Once it is finished it will invoke the complete operation to announce that its transmission is complete. The receiving end similarly issues post and wait to expose and synchronise its memory. It is allowed to access its window's local memory meanwhile. In contrast to a fence operation, one can restrict such synchronisation to arbitrary processes of the window.

The *lock* and *unlock* operations facilitate one way of passive target communication. Through these two functions one can block all access to a chunk of remote memory for all other window members. Between lock and unlock, the process can then issue remote memory access operations, without being interfered.

Another method to synchronise memory passively is through *flush*. A flush makes sure all outstanding RMA calls, issued by the caller, are completed before returning. One can choose if the operations have to be completed only at the issuer's side or on the receiver's side as well.

Window synchronisation can be optimised through giving the implementation information about which operations are not used during the epoch and

what happens before and after this synchronisation.

Inside an epoch one may issue remote memory access operations. The two simple ones are *put* and *get*. Memory access through put and get appears unordered inside an epoch. Furthermore, one can use atomic operations like *accumulate*, *fetch-and-op* or *compare-and-swap*, with the intuitive semantic and where *op* is a placeholder for various operations. The default global visibility order of atomic operations is sequentially consistent. When creating a window one can weaken this by, for example, allowing remote write operations to overtake remote read operations. Any combination of overtaking between reads, writes or between equal operations can be allowed.

Remote memory access in MPI is subject to a number of restrictions:

- The buffer supplied to a put call should not be written to until the operation is finished.
- The buffer supplied to a get call should not be accessed until the operation is finished.
- The outcome of concurrent remote memory access to the same location inside an epoch is undefined.
- The outcome of concurrent local and remote access to the same location inside an epoch is undefined.
- The outcome of a single process issuing multiple put operations to the same location inside an epoch is undefined.

Therefore, the only way to make sure that data is correctly written or read is through atomic operations or isolating access to the same location with costly window synchronisation operations. Normally a processor commits 32 or 64 bits to main memory atomically, and in the remainder of this thesis we will assume this is the case. Therefore, these restrictions make sense for buffers at sizes above this, because concurrent memory access of interfering operations might overlap and the outcome is then indeed undefined. For amounts of memory below this threshold we wish to have a semantic that is more specific. For example if two puts of at most 64 bits are concurrently issued to the same location, the outcome is that of the last put committed. Since this is not the case, MPI's restrictive RMA semantic makes it unsuitable to implement barrier algorithms based on the ideas of the pW/CS lock on top of it.

Shared memory barriers usually rely on atomic operations. Most distributed memory barriers use synchronous message passing. To improve efficiency, these protocols have also been implemented using RMA, e.g. in [17, 22, 30].

2.2 Current Barriers

Having surveyed the tools for implementing synchronisation primitives we will now examine the barrier protocols which power today's implementations for both shared and distributed memory systems.

2.2.1 Shared Memory

Many prominent implementations of shared memory concurrent programming facilities are part of closed source commercial products, like for example Intel Composer and Microsoft Visual Studio, and can therefore not be analysed. So we restrict ourselves to examining the GNU C Library [14] and GNU OpenMP [15].

The GNU OpenMP implementation, included in the GNU Compiler Collection (GCC) version 4.8.0, implements a *Central Counter Barrier*. Figure 1 describes the algorithm in pseudocode, which is executed on each thread taking part in the barrier synchronisation. The Central Counter Barrier uses the variable `barrier` to hold the number of threads that do not yet have arrived at the barrier. Initially it is assigned the number of threads. Upon entering the barrier each thread atomically decrements the variable by one and then waits until it reaches zero. After it reaches zero, all threads eventually leave the barrier.

```
shared variables: integer barrier := threadCount
---------------------------------------------------

atomic{barrier := barrier - 1}

wait until barrier = 0
```

Figure 1: Pseudocode for the Central Counter Barrier

The protocol can be separated into two phases (Figure 2). During the first phase each thread decrements the variable, and in the second reads the variable until it reaches the desired value.

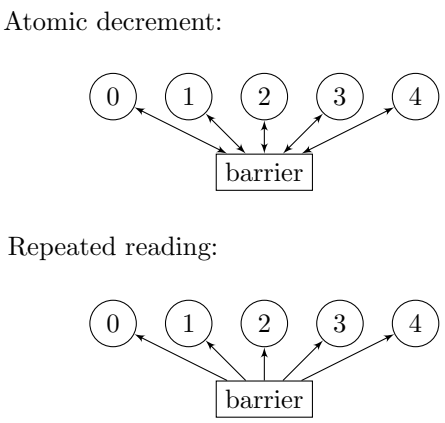

Figure 2: The phases of the Central Counter Barrier for five threads

The actual implementation in GNU OpenMP is more involved than the pseudocode in Figure 1 suggests. For example the algorithm shown here is destructive in the sense that it cannot be repeated, because if the variable is zero at the end, repetition would mean that the variable is negative in the next

round, which breaks the protocol. To mitigate this in the implementation, the variable is copied, by each thread, upon first reading it and the last thread to enter resets the shared variable to its initial value.

The loop additionally keeps track of the number of unsuccessful reads. If this number reaches a certain amount, an operating system mechanism, called futex [10], is employed to suspend the thread until a later time when the other threads hopefully have arrived. Through this back-off mechanism, the system avoids unnecessary busy waiting which would negatively impact power consumption. The threshold of this counter is architecture specific and set to an equivalent of about 3 milliseconds of waiting on GNU/Linux. In case the number of threads is larger than the number of physical cores, i.e. the processor is oversubscribed, the counter is set to a much lower value. This way the operating system scheduler can switch between threads more quickly so that threads that need time for computation are not blocked by busy waiting ones.

Throughout this thesis we will assume that the thread and process count is at most as high as the number of physical processor cores in a system, because oversubscription is not useful inside a single program and therefore rarely happens.

The GNU POSIX Threads implementation, called Native POSIX Threads Library (NPTL), is part of the The GNU C Library. The basic barrier algorithm used in it is the same as for GNU OpenMP. One main differences is that it does not use atomic operations to decrement the variable, but explicitly locks it to guard from interfering access. Second, there is no busy waiting, but upon arriving at the barrier the thread will immediately suspend, via a futex, and wait to be woken up by the last thread arriving. The third difference is that to reset the barrier each thread atomically increments the barrier variable by one upon leaving.

2.2.2 Distributed Memory

In this section we present the barrier algorithms that are used in the two major open source MPI implementations, namely Open MPI [40] and MPICH [35]. We restrict our analysis to MPI for the same reasons we only considered MPI in Section 2.1.2.

Open MPI's barrier implementation distinguishes between process counts that are powers of two and which are not. If it is a power of two, a *recursive doubling* barrier is used. A detailed explanation of this protocol can be found in [20]. For process counts that are not a power of two Open MPI uses the *Dissemination Barrier*, introduced by Hensgen, Finkel and Manber in 1988 [19].

Figure 3 presents the pseudocode of this algorithm, which is executed on each process individually. A process is identified by its *process index*, which is a number between 0 and $processCount - 1$. For readability reasons we have chosen a representation of the protocol where read/write operations use local or remote memory depending on where the accessed array element is located. The protocol can similarly be implemented using message passing techniques. In contrast to the Central Counter Barrier, the Dissemination Barrier uses not only one variable but an array of boolean variables per process involved. Each of these arrays is located in the local memory of its owning process and is initially set to false. The protocol is organised in rounds. Each round the distance (variable `dist`) to the next two partners, to read from and write to, doubles.

Process indices are always calculated modulo the total number of processes to achieve a wraparound. Figure 4 illustrates this communication pattern.

```
constants:          me := processIndex
shared variables:   boolean
                    barrier[processCount][processCount]
local variables:    integer dist, to, from
initialisation:     barrier[*][*] := false
---------------------------------------------------------

dist := 1
while dist < processCount {
    to   := (me + dist) (mod processCount)
    from := (me - dist) (mod processCount)

    barrier[to][me] := true

    wait until barrier[me][from] = true

    dist := dist · 2
}
```

Figure 3: Pseudocode for the Dissemination Barrier

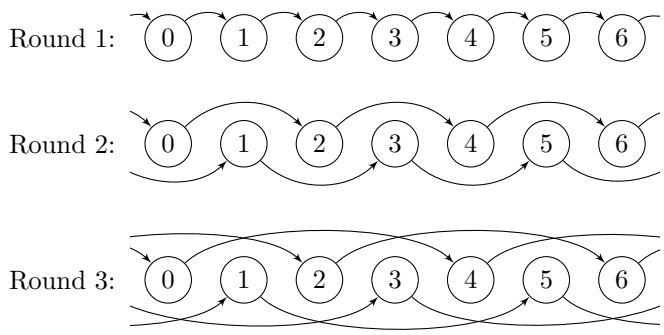

Figure 4: Communication pattern of the Dissemination Barrier

At first a process notifies the next one that is has arrived at the barrier by setting the array element at index `me` on the next process to true. Then it waits until the previous one notifies its arrival. When that happens the waiting process enters the next round. It now knows that itself, and the previous process have arrived at the barrier and notifies the process at double the previous distance (2) of its knowledge, by setting its array element at the target process to true. Through this communication pattern the number of processes, that each process knows have arrived, doubles each round. Thus after $\lceil \log_2 \mathit{processCount} \rceil$ rounds

every process knows of the arrival of all others and can therefore leave the barrier.

Notifying the next process is done via writing to remote memory, and waiting for notification via busy waiting on local memory. The number of remote access operations per round is the same as the number of processes involved.

The actual implementation in Open MPI uses MPI's *send-receive* operation instead of our version's remote write and local read. This operation sends a message and waits for another at the same time. The protocol is not destructive because send and receive do not change the state of the protocol. Therefore, no resetting mechanism, as used by GNU OpenMP for the Central Counter Barrier, is needed. Open MPI is able to replace this implementation with hardware specific ones, if it detects supported hardware. For example InfiniBand uses a variation of the Dissemination Barrier based on RMA [22].

MPICH version 3.0, which forms the basis of many other prominent MPI implementations like MVAPICH2 and Cray MPT, implements the same send-receive-based Dissemination Barrier as is used in Open MPI, but disregards the number of processes.

Aside from distributed memory architectures it is possible to implement these protocols for shared memory systems, as has been demonstrated for the Intel XEON Phi [12].

2.3 Barrier Building Blocks

During our analysis of general implementation facilities and concrete barrier implementations we observed that one can assemble barriers from orthogonal components.

One independent piece is the employed way of reinitialising the barrier after use. As we have seen for the Central Counter Barrier (Section 2.2.1) resetting might be achievable through slight modification of the algorithm. This does not work for all protocols, though.

Another possibility is to replace boolean variables, as used in the Dissemination Barrier in Figure 3, with a repetition counter. The remote write and local spin-waiting condition are then replaced with atomic increment and a test for the correct number of repetitions. The additional amount of memory needed is linear in the number of processes. The added computational overhead is negligible.

A way of resetting that works in all circumstances is to use three distinct barriers and switch cleverly between them. The proposed procedure, to be executed by each process individually, is illustrated in Figure 5. In the beginning the first two barriers are initialised, whereas the third one is set to the state of a completed barrier. Upon entering the routine one uses the barrier which comes after the currently finished barrier and resets the one which has been finished before. This approach is provably correct, uses triple the amount of memory of a single barrier, and ads minimal computational overhead.

In order to lower power consumption limiting the amount of busy waiting is important for barrier implementations. Choosing a back-off strategy can be done independently of the used barrier protocol. In almost all cases one wants to take apart the waiting loop and insert a suspension mechanism.

During the first initialisation of the barrier an implementation can decide how to handle the case where the number of threads or processes exceeds the

```
shared state:    barrier1, barrier2, barrier3
initialisation:  barrier1 := initialised
                 barrier2 := initialised
                 barrier3 := finished
---------------------------------------------

if barrier3 is finished {
    use barrier1
    reset barrier3
} else if barrier1 is finished {
    use barrier2
    reset barrier1
} else if barrier2 is finished {
    use barrier3
    reset barrier2
}
```

Figure 5: Pseudocode for the triple barrier reset method

number of physical processor cores. Therefore, implementing a solution to this scenario is orthogonal to the other algorithm components.

Another useful approach, which has been implemented in [17], is to assemble multiple different barrier protocols to create a new one. For example assume you have a hardware topology of shared memory nodes which are connected via network. It is conceivable to construct a barrier that executes a Central Counter Barrier on each node, and then one process of each node takes part in a Dissemination Barrier. After the Dissemination Barrier is completed the intra-node processes are notified and the assembled barrier completes. This way one can benefit from the best performing barrier on each level of the topology. This might also make sense if barrier algorithms vary in performance for different numbers of threads or processes.

3 Innovative Barriers

In this section we first present a new variation on an already presented barrier protocol of ours. The second algorithm reverses the approach of the Central Counter Barrier. Third we demonstrate a pW/CS-influenced modification of the second barrier.

3.1 MGB Barrier

The original idea of the following algorithm is part of a private communique between Nicholas Mc Guire, the pW/CS lock inventor, and us. Figure 6 presents the new barrier, which is a variation of the protocol first mentioned in [6]. It is intended for shared memory architectures.

```
constants:          me:=2^threadIndex , full:=Σ_{i=0}^{threadCount-1} 2^i
local variables:    integer copy
shared variables:   integer first , second , current
initialisation:     first := 0, second := 0, current := 1
---------------------------------------------------------------
if current = 1 {

    do {
        copy := (copy & ~me) | first
        if copy & me = 0 {
            copy  := copy | me
            first := copy
        }
    } while copy ≠ full and current = 1

    current := 2
    second  := 0

} else if current = 2 {

    do {
        copy := (copy & ~me) | second
        if copy & me = 0 {
            copy := copy | me
            second := copy
        }
    } while copy ≠ full and current = 2

    current := 1
    first   := 0
}
```

Figure 6: Pseudocode for the MGB Barrier

Before going into detail, we first have to repeat some of the notation used in the pseudocode. It borrows from the C programming language. &, | and ∼ are the bitwise *and*, *or* and *not* operator. The constant `threadIndex` is a thread identifying number between 0 and $threadCount - 1$.

The protocol consists of two symmetric sub-barriers. It uses `current`, a variable which holds the number of the sub-barrier that is currently used, to switch between them. Through `first` and `second` the barrier keeps track of which threads have arrived. The variable `copy` is a local copy of either `first` or `second`. A thread commits itself to `first` or `second` by setting its respective bit to 1. In the pseudocode we implement this bit set behaviour through integers and working with powers of two. If all threads committed themselves to one of the sub-barriers, it is a full bit set equal to the constant `full`. All threads participating in the barrier synchronisation enter the same of the two parts. Inside a loop participants try to synchronise. When successful, the barrier is reset through the two subsequent assignments of `current` and either `first` or `second`.

Initially `first` and `second` are zero, and `current` is one. When a thread arrives at the barrier, it will enter the upper if branch. It then asks which threads have already arrived, adds itself to the acquired bit set, and stores it back to the variable `first`. He then repeatedly polls `first` in order to find out if its commitment attempt has been overwritten, i.e. its bit is not set anymore, by another thread's concurrently issued write operation, and secondly to determine whether the barrier is completed. The upper barrier is completed, if either `first` equals `full` or `current` is 2. The first thread to realise that `first` has been set to `full` will exit the loop, set `current` to 2 and continue its computational work. Subsequently all other threads eventually leave the barrier. When entered again the same procedure is executed in the lower if branch using the variable `second` instead of `first`. A third repetition utilises the upper branch again.

The presented protocol reflects the ideas of the pW/CS lock in that it loosely organises communication between threads and is less enforcing in the sense that threads may overwrite each others' commitment attempts.

In comparison to the Central Counter Barrier this approach does not rely on the existence atomic operations.

The number of threads this protocol supports is limited by the number of bits a processor can atomically commit, which is usually 64 bits. To alleviate this problem one can implement an array-based variant of the protocol. Instead of reading `first` and `second` one copies the array element by element. Comparing is done element by element as well. Instead of setting the variable one writes back only the one element where the thread's bit resides on. This way writing still requires only one atomic 64-bit commitment.

3.2 B1 Barrier

The following barrier protocol reverses the idea of the Central Counter Barrier (Section 2.2.1) to use a single global counter that all threads atomically access once and read repeatedly. Instead an array of boolean variables (one element for each thread), which is also accessed differently, is employed. The algorithm's pseudocode, executed individually on each thread taking part, is depicted in Figure 7.

```
shared variables: boolean barrier[threadCount]
local variables:  integer i
initialisation:   barrier[*] := false
---------------------------------------------

barrier[threadIndex] := true

i := 0
while i < threadCount {
    if barrier[i] = false {
        i := -1
    }
    i = i + 1
}
```

Figure 7: Pseudocode for the B1 Barrier

Initially each array element is false. Upon arriving at the barrier a thread registers itself by setting its array element to true. It subsequently queries all other threads for arrival. As soon as it finds one that has not yet arrived, it resets the loop counter and begins querying all others anew. Once all threads arrive, eventually each of them will leave, thus completing the barrier synchronisation.

Similar to the Central Counter Barrier, the protocol can be split into two phases (Figure 2). The difference between the two protocols is that, because each thread writes only to its own array element, they do not require atomic access to register themselves at the barrier, and during the reading phase each thread has to read each other thread's variable instead of polling just a single global one. The threads may concurrently write, but have to spend more time reading.

The Barrier cannot be repeated as is. A way to add resetting to the protocol is by using the repetition counter approach introduced in Section 2.3. In order to avoid false sharing [5], which would majorly degrade performance, an actual implementation of the protocol allocates one cache line for each array element. Cache lines are usually 64 bytes large. Therefore, transforming the array into an array of counters does not require more memory.

A small improvement to the shown algorithm is to keep track of the first few threads that have successfully arrived. The loop counter is then reset to this number instead of 0.

Since we do not use any central information, this protocol can be implemented for both shared and distributed memory architectures.

3.3 B2 Barrier

The B2 Barrier is a variation on the previous barrier in which less communication is traded for more computation.

In principle the following protocol can be implemented on both shared and distributed memory architectures. We now switch to a distributed setting, because preliminary analysis showed that the saved memory access cost in com-

parison to the additional computational overhead favours its use in a distributed scenario.

Figure 8 presents the B2 Barrier's pseudocode. The code uses the same remote memory access style as the Dissemination Barrier (Figure 3 in Section 2.2.2), i.e. read/write operations use local or remote memory depending on where the accessed array element is located. The notation used in this listing is explained in Section 3.1. Each array element of `barrier` is located in the local

```
constants:          me := processIndex, meBit := 2^me,
                    full := Σ_{i=0}^{processCount-1} 2^i
shared variables:   integer barrier[processCount]
local variables:    integer i
initialisation:     barrier[*] := 0
                    i := processIndex
---------------------------------------------------------

barrier[me] := meBit

do {
    do {
        i := i + 1 (mod processCount)
    } while barrier[me] & 2^i = 1

    barrier[me] := barrier[me] | barrier[i]
} while barrier[me] ≠ full
```

Figure 8: Pseudocode for the B2 Barrier

memory of the process with the matching index. These elements are bit sets representing the knowledge of the owning process about the arrival of others.

When the first process arrives at the barrier it registers itself by setting its bit in its local array element and then continues to query the remote processes. It only acquires a remote array element, if it did not yet successfully do so, i.e. the remote process' bit is not set in the local bit set. If this is the case, it fetches the remote element and adds it to its own, thus adding the arrival of all processes the remote end knows about to its own bit set. Querying the other processes is repeated until the bit set indicates that all processes arrived.

Comparing this behaviour to the B1 Barrier, a process now locally keeps track of the processes that have arrived and only queries those where the owning process did not yet get an affirmative answer. The protocol does not maintain the differentiation between reading and writing phase since it keeps updating its local variable during the loop.

This protocol writes only locally, and reads only remotely. It does not rely on the existence of atomic operations.

In comparison to other distributed memory barrier protocols, e.g. the Dissemination Barrier (Section 2.2.2), the communication pattern is largely random. The order of remote access depends on remote memory access latency, local computation and memory access speed, the exact time of arrival of each

process and operating system influences like scheduling. The effect of these factors combined makes the order of communication practically random.

Like the B1 Barrier this protocol is destructive. One can add resetting via the triple barrier method presented in Section 2.3.

Like the MGB Barrier this protocol is restricted to 64 processes. An extension to arrays of bit sets can be implemented in the same way as described in Section 3.1.

4 Analysis and Comparison

In this section we will evaluate one commonly used barrier for each shared and distributed memory architectures. In both categories we compare the chosen protocol to one of our newly presented ones.

Early evaluations have shown the MGB Barrier to perform worse than the Central Counter Barrier, whereas the B1 and B2 Barrier show promise. Therefore, we restrict our attention to these two.

In the shared memory setting the Central Counter Barrier is pitted against the B1 Barrier. In the distributed memory case we compare the Dissemination Barrier to the B2 Barrier.

In our evaluation we assume no back-off strategy is used. This means if a variable is repeatedly requested inside a loop, it is fetched as often as program execution speed allows. The topic of back-off strategies is a subject in its own right and goes beyond the scope of this thesis.

The analysis is split into two parts. First, we present properties, perceivable without measurement and modelling, that are inherent to the protocols. Second, we employ model checking to quantify our observations and, thus, gain a deeper understanding of the chosen barriers.

4.1 General Properties

This section is divided into two parts: shared and distributed memory barriers. In each we present the memory requirement of the algorithms. For the Central Counter and B1 Barrier we additionally consider properties of the reading and writing phase. In the distributed memory scenario we are also interested in the remote access behaviour of both chosen protocols.

Shared Memory Barriers The Central Counter Barrier (Figure 1 in Section 2.2.1) and the B1 Barrier (Figure 7 in Section 3.2) are both divided into two phases: the reading and the writing phase.

In the writing phase of the Central Counter Barrier, concurrently issued add-fetch operations are serialised, because atomic operations on a single variable cannot be executed in parallel. On the other hand, the B1 Barrier allocates one array element per thread. Therefore, all write operations can be performed concurrently. If every thread arrives in the same instant, the commitment of all threads takes only as long as one write operation does, therefore speeding up this process by a factor of n, where n is the number of threads participating. In a more balanced scenario, i.e. processes arrive in intervals, this advantage over the Central Counter Barrier diminishes.

During the reading phase of the Central Counter Barrier a thread fetches a single variable repeatedly until it has the desired value. A thread in the B1 Barrier needs to read between 1 and n array elements before it can leave the loop.

The B1 Barrier has the edge in the writing phase, whereas the Central Counter Barrier performs well during reading. We quantify the effect of this behaviour in Section 4.2.

The memory required by the Central Counter Barrier is logarithmic in the number of threads. With a 64-bit counter up to 2^{64} threads can be synchronised. The B1 Barrier uses a linear amount of memory – $threadCount \cdot 64$ bytes,

assuming a cache line is 64 bytes large and disregarding the local variable. This might seem like a lot, but since shared memory systems rarely exceed 512 cores, a maximum of only 32 kibibytes is used.

Distributed Memory Barriers In this section we assume n to be the number of processes participating in the barrier synchronisation.

The Dissemination Barrier (Figure 3 and 4 in Section 2.2.2) and the B2 Barrier (Figure 8 in Section 3.3) communicate very differently.

The minimum number of remote access operations required to perform a barrier synchronisation is $2 \cdot (n-1)$. Figure 9 illustrates a simple gather and broadcast barrier (process zero gathers all arrival notifications and broadcasts to all processes when the barrier is complete) using this number of operations. It does so in $2 \cdot \lceil \log_2 n \rceil$ rounds. All operations inside a round can be executed in parallel. However rounds themselves are completed serially. This means the minimum execution time of this barrier is $2 \cdot \lceil \log_2 n \rceil$ times the duration of a remote write operation. In order to lower the number of rounds to $\lceil \log_2 n \rceil$ the Dissemination Barrier issues $n \cdot \lceil \log_2 n \rceil$, more than the minimum needed, remote access operations. The added communication overhead is traded for a decrease in overall execution time.

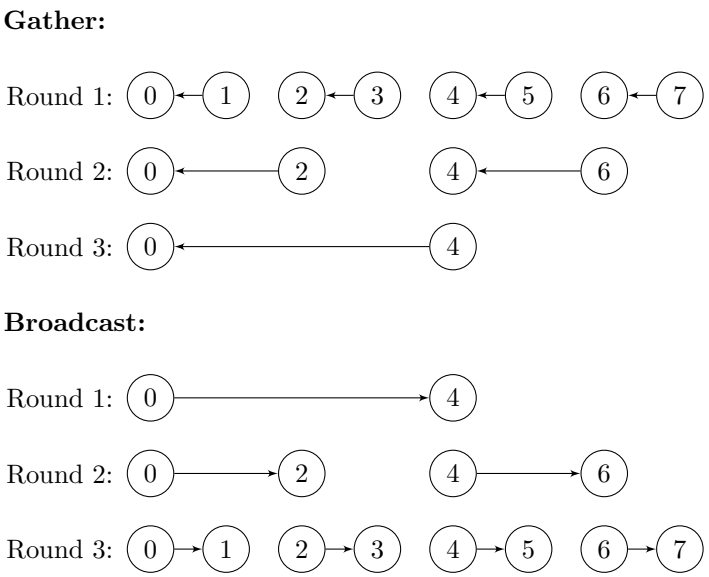

Figure 9: Communication pattern of the gather and broadcast barrier

Because of its random communication pattern, there is no fixed number of remote accesses issued by the B2 Barrier. The number of successful, non-zero, remote reads, lies between $2 \cdot (n-1)$ and $n \cdot (n-1)$. The amount of unsuccessful reads is not limited, since the loop remotely polls other processes until all arrive.

Where a process in the Dissemination Barrier waits for one designated communication partner each round, the B2 Barrier does not wait for a single process but instead continues to poll the next one.

Requiring $\lceil \log_2 n \rceil$ rounds is obviously worse than $\log_2 n$. Especially from n, where n is a power of two, to $n+1$ processes the number of rounds increases by

one. Therefore, the runtime of the Dissemination Barrier is expected to show a stair-esque behaviour. Every time the number of rounds increases the duration of the barrier increases by a large step, whereas adding more processes to the same number of rounds adds little overhead. In contrast to this the execution time of the B2 Barrier is distributed evenly.

Synchronisation using the Dissemination and B2 Barrier requires the same amount of memory. In principle $2 \cdot \lceil \log_2 \ n \rceil$ bits per process suffice for both algorithms. The Dissemination variant shown in Figure 3 uses a total of $n \cdot (n + \lceil \log_2 \ n \rceil)$ bits, but, since each process receives from only $\lceil \log_2 \ n \rceil$ other processes, a modified array index calculation would allow the protocol to work with $n \cdot 2 \cdot \lceil \log_2 \ n \rceil$ bits.

Assuming all processes arrive at a similar time, Figure 4 in Section 2.2.2 illustrates the progression through the rounds of the Dissemination Barrier. More often than not, processes arrive in large intervals, compared to the execution time of a barrier itself. Therefore, some processes advance through the rounds before every process arrived. But how far exactly can the protocol progress without all processes being present?

Suppose every process but the fourth has entered the barrier. Figure 10 then shows the advance of the processes through the rounds.

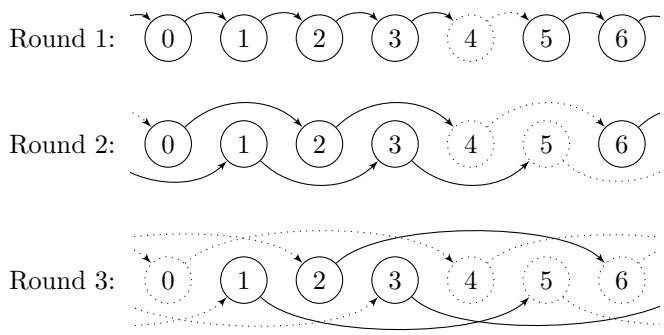

Figure 10: Progress through the rounds of the Dissemination Barrier with process four not having entered

Every dotted line indicates a remote write operation that is not executed. Processes with dotted circles have not reached the current round. Since process four does not notify process five of its arrival, in the second round process four and five do not advertise their arrival to process six and zero. The number of stuck processes doubles each round. Once process four enters, the missing remote operations in all three rounds are still to be completed until the processes may leave the barrier.

Table 1 illustrates which process knows of which other's arrival through the rounds of the Dissemination Barrier and for the B2 Barrier. The formula below each subtable title indicates the number of processes that are known to have arrived.

Because the number of blocked processes doubles each round the ratio of stuck to not-stuck processes is worse for non-power-of-two than it is for power-of-two process counts.

Table 1: Which process knows of which other's arrival with process four not having entered

Diss. after round 1:
$(n-1)+(n-1)=14$

	0	1	2	3	4	5	6	7
0	×							×
1	×	×						
2		×	×					
3			×	×				
4				×				
5						×		
6						×	×	
7							×	×

Diss. after round 2:
$14+6\cdot 2=26$

	0	1	2	3	4	5	6	7
0	×					×	×	×
1	×	×					×	×
2	×	×	×					×
3	×	×	×	×				
4		×	×	×				
5			×	×		×		
6						×	×	
7							×	×

Diss. after round 3:
$26+4\cdot 4=44$

	0	1	2	3	4	5	6	7
0	×					×	×	×
1	×	×					×	×
2	×	×	×					×
3	×	×	×	×				
4	×	×	×	×		×	×	×
5	×	×	×	×		×	×	×
6	×	×	×			×	×	×
7	×	×	×	×			×	×

B2 Barrier:
$(n-1)\cdot(n-1)=49$

	0	1	2	3	4	5	6	7
0	×	×	×	×		×	×	×
1	×	×	×	×		×	×	×
2	×	×	×	×		×	×	×
3	×	×	×	×		×	×	×
4								
5	×	×	×	×		×	×	×
6	×	×	×	×		×	×	×
7	×	×	×	×		×	×	×

The B2 Barrier does not work in rounds and does not have designated communication partners. It therefore fetches all other process' arrival notifications while one process is missing.

When the last process arrives, the Dissemination Barrier requires at least as many serial remote writes as there are rounds, whereas the B2 Barrier issues its remaining remote read operations concurrently. The number of remaining remote access operations is $2^{\lceil \log_2 \ n \rceil}-1$ for the Dissemination Barrier and n for the B2 Barrier.

We attempt to quantify the influence of this behaviour on the execution time and energy consumption in the following section.

4.2 Model Checking

The formal properties of the chosen protocols can be divided into two broad categories: functional and quantitative properties. Functional properties describe the supposed behaviour of the barriers, whereas quantitative properties express conditions that rely on probabilistic aspects of the protocols, e.g. execution time.

To check correctness of the barriers we model every instance where execution is random through non-deterministic choice. This yields a Markov decision process, which we then use a basis for automated analysis with the SPIN model checker.

To model the inherent randomness of our algorithms we use exponential distributions. This way we obtain continuous-time Markov chains (CTMCs) which serve as a basis for formal quantitative analysis using the PRISM model

checker. Models are obtained in a compositional way, using action labels and global state to synchronise.

4.2.1 Preliminaries

In this section we repeat principles of continuous-time Markov chains that are of interest to our analysis. Further details can be found in textbooks on Markov chains, e.g. [25, 26]. Some of this Section borrows from [2].

If S is a finite set, then a distribution on S is a function $\nu : S \to [0, 1]$ with $\sum_{s \in S} \nu(s) = 1$.

A CTMC $\mathcal{M}$ is a tuple (S, Act, R, μ) where S is a finite set of states, Act a finite set of action names and R a function of type $S \times Act \times S \to \mathbb{R}_{\geq 0}$, called the rate matrix of $\mathcal{M}$. μ is a distribution on S specifying the probabilities for the initial states.

If $R(s, \alpha, s') = \lambda$ (short $s \xrightarrow{\lambda:\alpha} s'$) and $\lambda > 0$ then $\mathcal{M}$ has a transition from s to s' with action label α and rate λ. λ specifies the rate of the exponential distribution. That means the probability for the transition $s \xrightarrow{\lambda:\alpha} s'$ to be ready for firing some time in the interval $[0, t]$ is $1 - e^{-\lambda t}$. The average delay of this transition is $\frac{1}{\lambda}$. If $R(s, \alpha, s') = 0$ then there is no transition in $\mathcal{M}$ from s to s' via α. The choice between multiple enabled transitions is made by the following rule. The probability to trigger a particular transition $s \xrightarrow{\lambda:\alpha} s'$ is $P(s, \alpha, s') = \frac{\lambda}{E(s)}$ where $E(s)$ is the exit rate of state s, i.e. the sum of the rates of all outgoing transitions of state s. The probability that $s \xrightarrow{\lambda:\alpha} s'$ will fire within t time units is then $P(s, \alpha, s') \cdot (1 - e^{-E(s) \cdot t})$.

Paths in a CTMC are sequences of consecutive transitions augmented by the time points when they are taken. We employ the logic CSL [1, 3, 27] to analyse such transition systems. To specify sets of infinite paths, we will use an LTL-like notation, such as $\Diamond T$ ("eventually T") and $\Box T$ ("always T") where T is a set of states. Instead of naming states, we oftentimes use state predicates, combined using propositional formulas, to describe sets of states. Their meaning will be obvious from the context.

We will also study reward based properties formalised using the logic CSRL [3, 27]. This requires the extension of $\mathcal{M}$ by two reward functions. The state reward function $srew : S \to \mathbb{R}$ specifies the reward earned per time unit while being in state s. The transition reward function $trew : S \times S \to \mathbb{R}$ assigns a reward to each transition between two states regardless of action labels. For finite paths one can then reason about accumulated reward and reachability reward. Suppose π is a finite path where the underlying state sequence is $s_1 s_2 ... s_n$ and let $t_0 = 0$ and t_i the time point where π takes the i-th transition. The accumulated reward of π is then

$$Rew(\pi) = \sum_{i=0}^{n-1} \big((t_{i+1} - t_i) \cdot srew(s_i) + trew(s_i, s_{i+1})\big)$$

It is the sum, over all states but the last, of the state rewards multiplied by the time spent in each state plus the reward assigned to each transition between consecutive states in π. The reachability reward is defined as

$$\mathrm{ReaRew}(\pi, \Phi) = Rew(\pi(s_0 : s_j))$$

where Φ is a propositional formula describing a set of states, $j = min\{i \mid s_i \models \Phi\}$ and $\pi(s_0 : s_j)$ stands for the sub-path of π starting in s_0 and ending in s_j. The extension of these notions from paths to states and CTMCs is done the usual way and can be found in [4].

Analysing the protocols, we will for example deal with the state reward function that assigns value 1 to each state. In this case ReaRew(Φ) can be interpreted as the average number of processor clock cycles to reach a state where the formula Φ holds. We also wish to count remote memory access operations. For this we assign value 1 to each transition that represents such an operation. ReaRew(Φ) is then the average number of remote memory access operations issued until reaching a Φ-state.

4.2.2 Modelling

To analyse functional and quantitative aspects of the chosen barrier protocols, we model them using two different formalisms and two different software tools. We use ordinary non-deterministic transition systems and the SPIN [23, 47] model checker to verify functional properties. To analyse quantitative properties we employ CTMCs and PRISM [28, 43].

To verify the functional correctness of a barrier, we need to include a reset mechanism into the model, which is not needed for quantitative analysis. By stripping this unnecessary information off the probabilistic model we shrink its state space. This enables us to verify properties more quickly and/or of larger models.

Depending on circumstances we describe part of a system as control flow diagram and part as CTMC. Control flow diagrams contain local variables, shared variables, control flow information and need to be unwound to form a proper CTMC. The state space of the resulting CTMC is then a combination of control flow location and variable assignments.

The CTMCs of the protocols are composed of multiple modules. Composed modules are executed in an interleaved fashion except if transitions require synchronising actions. A synchronising action is an action that can be triggered by at least two modules. A transition with such an action can only fire if all modules, that are able to fire this action eventually, are presently in a state where a transition with this action label is enabled. The rate of such a simultaneous transition is determined by any one module. For each simultaneous transition, in our case, only one module specifies a rate. The presented way of composing modules corresponds to the following SOS rules.

If no other module uses action label α:

$$\frac{s \xrightarrow{\lambda:\alpha} s'}{\langle s, \overline{x} \rangle \xrightarrow{\lambda:\alpha} \langle s', \overline{x} \rangle}$$

where $\overline{x}$ is the tuple of local state of all other modules. The module's execution is interleaved with that of all other modules. If α is used by two modules:

$$\frac{s \xrightarrow{\lambda:\alpha} s', t \xrightarrow{\alpha} t'}{\langle s, t, \overline{x} \rangle \xrightarrow{\lambda:\alpha} \langle s', t', \overline{x} \rangle}$$

Only if α is enabled in both modules, these two transitions fire simultaneously. Analogous rules apply for more than two modules sharing an action label.

Shared Variable In order to exchange information threads write to and read from the same memory location. Memory access latency is very large in comparison to the duration of arithmetic operations. Because synchronisation is mainly about exchanging information between threads, the execution time of barrier protocols is dominated by memory access latency.

Usually memory access is cached. That means repeatedly reading a value is quick, because upon first reading it the processor places a copy of it near its execution units. When a cached value gets changed by another thread that shares it, the cached value needs to be thrown away because otherwise it would soon be incoherent. In order to model execution time and energy consumption of a barrier protocol, we therefore model the caching behaviour of threads.

We identify a shared variable with the cache line it resides on. A cache line is usually a 64 bytes large unit of contiguous memory to which all cache manipulation operations are applied as a whole. It increases the granularity of address space, which is usually one byte, to decrease management overhead for caching. Each thread that has access to this shared variable has one copy of such a cache line. Synchronisation between cache line copies and with other CTMC modules, like the actual algorithm to be modelled, is done via synchronising actions. The operation triggering thread's cache line copy determines the rate of the transition. For convenience we use combinations of action labels like $\alpha \vee \beta$ meaning that there are two transitions, one with action α and one with β between two states. Either action may trigger the transition.

We use the MSI cache coherency protocol, extended by support for atomic operations, to model cache consistency. The MSI protocol is a basic cache model. Further details about it can be found online at for example [37].

In the MSI model a cache line copy can be in one of three states. *Modified* means the thread has the only up-to-date copy of the cache line and all other copies are *invalid*. If a cache line copy is invalid it is outdated and therefore not usable. Being in the *shared* state means the thread has an up-to-date copy, but other threads may have a correct copy, too. We extend these three states by a fourth, called *atomic*. Its meaning is the same as modified, except that as long as a cache line copy is in this state, no other thread may access it.

Figure 11 illustrates how a cache line copy changes its state. Solid transitions are fired by events occurring on the thread that owns the cache line copy, whereas dashed transitions are due to events triggered by other cache line copies. For example if a thread reads a variable and its own cache line copy is invalid, it first needs to make sure that all other threads take notice and change its cache line copy state to shared in case it was modified before. After this the thread fetches the cache line from main memory, marks its own copy shared, reads the variable and continues program execution.

The durations of the cache line copy state transitions are as follows. Reading a modified or shared copy is considered instantaneous, because the copy is up-to-date and no other copies need to be notified. If a thread whose copy is invalid reads a variable, it has to advertise its intent to all other copies. If another copy is modified, it needs to switch to the shared state. Meanwhile the reading thread fetches an up-to-date copy from main memory, sets the state of its local copy to shared, and finally continues program execution. This usually takes around 50 processor clock cycles. A write operation on a modified variable is considered instantaneous, since no other thread has an up-to-date copy and therefore no one needs to be notified. If the cache line copy in question is shared

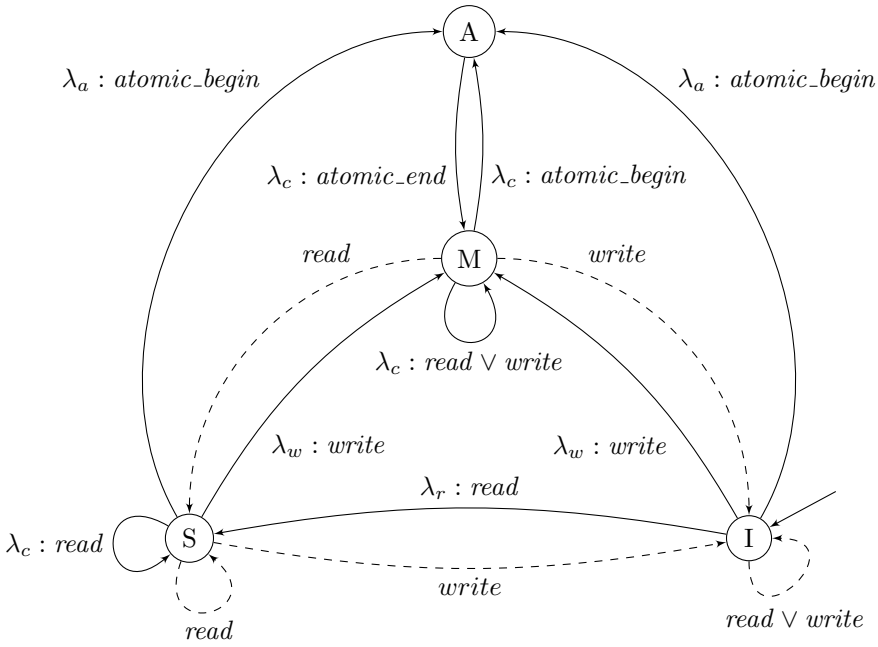

Figure 11: CTMC for one cache line copy

or invalid, it broadcasts an invalidation request to all other copies, and waits until every copy fulfills the request. After this the thread can safely mark its copy modified and finally write to it. This operation commonly takes about 100 cycles. Entering the atomic state works exactly like entering the modified state. During the atomic state the cache line copy postpones all requests to this copy until it transitions to a different state. After leaving the atomic state, it instantly begins to answer these requests. Therefore, leaving the atomic state is considered instantaneous.

The exact number of processor clock cycles these operations take is strongly hardware-dependent. For the CTMC model we assume a cache read to take 50 cycles, whereas writing and entering an atomic operation requires 100 cycles. The rates of the transitions corresponding to the described events is then the reciprocal of the cycle count. The rate of an instantaneous operation, which is assumed to be one cycle, is $\lambda_c = 1$. The rate of a cache read is $\lambda_r = \frac{1}{50}$. And the rate of initiating a write or an atomic operation is $\lambda_w = \lambda_a = \frac{1}{100}$

For technical reasons we switch the rate of entering and leaving an atomic operation. The intuitive result is the same.

Central Counter Barrier The Central Counter Barrier uses one shared variable. Therefore, the final CTMC is composed of one cache line copy module per thread and one algorithm module per thread. The algorithm's control flow diagram (Figure 12) directly results from the pseudocode in Figure 1 (Section 2.2.1). Initially variable `barrier` is set to `threadCount`. The synchronisation with the cache line copy module is hinted at in grey. The rates for the shared operations are determined by the thread's cache line copy module,

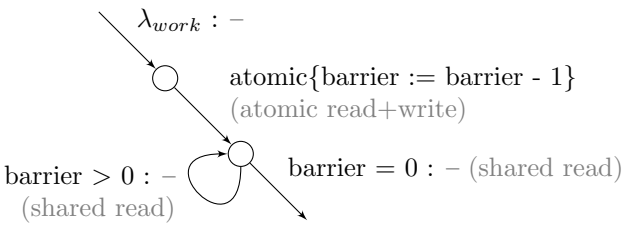

Figure 12: Control flow diagram for one Central Counter Barrier thread

depending on its internal state.

We emulate the interval in which threads arrive at the barrier via an exponential distribution with rate λ_{work}. One can imagine this as a period where the thread is performing computational work.

The atomic operation has been shortened for readability and really contains multiple transitions. The thread initiates the atomic operation on `barrier`, reads it, subtracts one from it, writes it back, and finally ends the atomic operation.

Unlike the quantitative model, the functional model repeatedly executes the protocol. Resetting is achieved via the triple barrier method, which is explained in Section 2.3.

B1 Barrier The B1 Barrier (Figure 7 in Section 3.2) allocates one cache line per array element. Therefore, the final model is composed of n algorithm modules plus n^2 cache line copy modules where n is the number of threads.

The algorithm's control flow is illustrated in Figure 13. As for the Central Counter Barrier, the B1 Barrier starts with an initial work period, its rates are determined by thread's cache line copy module, and the grey transition labelling indicates the interaction with the cache. Initially each array element

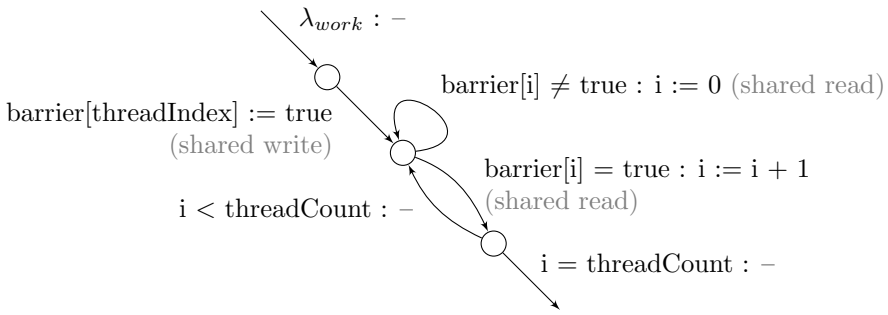

Figure 13: Control flow diagram for one B1 Barrier thread

is set to false and the loop index to zero. In comparison to the pseudocode, the assignments are slightly different, but ultimately result in the same behaviour.

Deviating from the quantitative model, the functional model implements repetition and resetting by replacing the boolean array elements with counters.

The principle is described in detail in Section 2.3. Additionally, the functional model splits some of the transition into multiple transitions to allow for more interleaving and therefore reveal more potential errors in the protocol.

Splitting these transitions in the quantitative model would increase model size majorly. It is important not to merge multiple transitions that depend on shared memory. Merging or splitting other transitions influences timing only in a minor way.

Dissemination Barrier The Dissemination Barrier (Figure 3, Section 2.2.2), as a representative for distributed barrier algorithms, does not have shared memory. Therefore, the CTMC model consists of one algorithm module per process and nothing else.

Figure 14 depicts the control flow diagram of the protocol. The distance is initialised to one and each array element to false. As in the two previous paragraphs, a work period precedes the protocol. The expressions `to` and `from` are shorthands for `processIndex` $\pm$ `dist (mod processCount)`. Local oper-

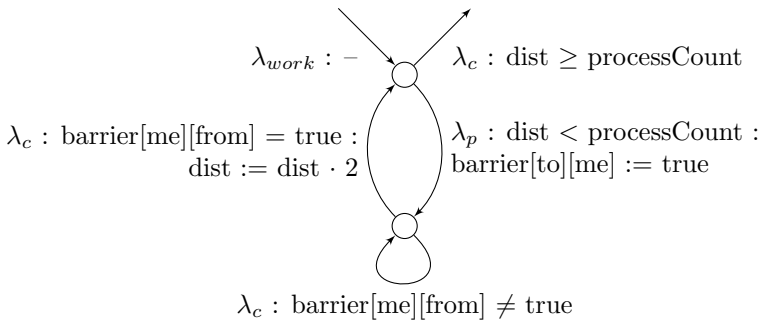

Figure 14: Control flow diagram for one Dissemination Barrier process

ations take far less time than remote operations – between ten to a thousand times usually. Therefore, we assign the rate λ_c (one processor clock cycle) to local operations. The duration of a remote write (also called put) operation, which is strongly hardware-dependent, is assumed to take 100 cycles. Therefore, we assign $\lambda_p = \frac{1}{100}$ to each corresponding transition.

In the CTMC model, resulting from unwinding this control flow diagram, a remote write operation is represented as a synchronised action. When a process executes a put, the target process then changes its local state accordingly.

The functional model includes repetition and resetting with counters instead of boolean variables as described in Section 2.3.

B2 Barrier The B2 Barrier, introduced in Section 3.3, is modelled as a composition of one algorithm module per process.

Figure 15 illustrates the control flow of the protocol. Initially each bitset is empty and the loop index is set to `processIndex`. The rates of operations are the same as for the Dissemination Barrier except that, instead of remote writing, we assign a rate of $\lambda_g = \frac{1}{100}$ to remote read (also called get) transitions.

The functional model additionally includes repetition and resetting using the triple barrier method explained in Section 2.3. Similar to previously presented

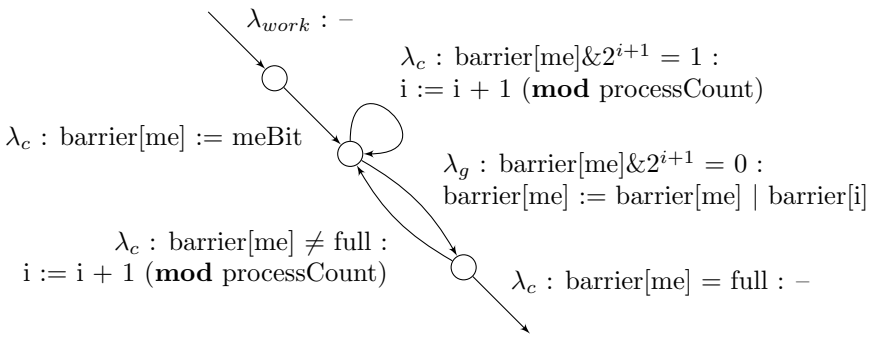

Figure 15: Control flow diagram for one B2 Barrier process

models the functional model splits some of the transitions into multiple ones to allow for more interleaving. This way more potential errors can be revealed. The quantitative model refrains from splitting them, to reduce the model's size.

4.2.3 Functional Properties

In this section we use the words *thread* and *process* interchangeably, because conditions of functional correctness apply to both shared and distributed memory protocols.

There are two basic conditions that every barrier protocol has to fulfill. It has to properly synchronise threads, i.e. no thread leaves before all threads arrived at the barrier, and it may not deadlock. We express these two properties using the linear temporal logic.

(A) *"A thread may only exit the barrier if all threads have entered it."*

$$\Box(\mathit{one_left} \implies \mathit{all_entered})$$

(B) *"If all threads entered the barrier, each one leaves it in a finite amount of time."*

$$\Box(\mathit{all_entered} \implies \Diamond \mathit{all_left})$$

We use state predicates to describe sets of states with the intuitive meaning. For example *all_left* represents the set of states where each thread has passed the barrier.

In order for B to hold we have to assume that each thread is scheduled infinitely often. This property is also called *fairness*. In fact assuming weak fairness already suffices here. Without this mild side assumption a barrier protocol can deadlock since one thread might never leave the barrier.

We implemented each of the four chosen protocols in the modelling language Promela and verified properties A and B using the SPIN model checker. All barrier protocols fulfill both conditions.

4.2.4 Quantitative Properties

As in the previous section we use the words *thread* and *process* interchangeably, since some properties apply to both shared and distributed memory barrier protocols.

The section is divided into two parts. We first identify and formalise interesting quantitative properties of barrier protocols. Second, we present the results of the automated analysis.

Identification and Formalisation There are a number of interesting questions to ask about barrier protocols. First, let us enumerate what we can quantify. Measuring time is obviously of great interest to us. We want to determine how many processor clock cycles certain operations take. In order to measure cycles, we assign a state reward of 1 to each state. The accumulated reward is then the desired value.

Another interesting quantity to measure is energy consumption (unit joule) and its rate, measured in watt (joule per second). To measure these, we use a mix of transition and state rewards. The following, very limited assumptions are based on a sample measurement conducted on a personal laptop. We use a processor clock rate of 2.5 GHz to convert between seconds and cycles. We assume the processor to consume 11 watts as a baseline. Local operations are assigned no reward. For shared memory protocols we assign a cost of 2 nanojoules (nJ) to each instantaneous memory operation and 200 nanojoules to slower memory operations. For distributed memory we assume a cost 200 nanojoules per remote memory operation and 2 nanojoules for each local memory operation.

For distributed memory protocols we are also interested in the number of remote operations issued. To obtain this measure, we assign a reward of 1 to each corresponding transition and take the accumulated reward.

Having listed the three measures we want to quantify, we now consider the points in time when we want to measure. There are four of these: the moment when the first (A), and the last (B), thread enters the barrier, and the instant when the first (C), and last (D), thread leaves the barrier. For the shared memory barriers we additionally consider the moment when the writing phase ends (W), i.e. when the last thread finishes writing. Since the Dissemination Barrier consists of multiple rounds, we are also interested in measuring at the end of each round (Ri). That is the point in time when the last thread leaves round i.

Formalising this enumeration we present the list of properties used for model checking.

(A) *"Average reward until any one thread entered the barrier"*

$$\mathrm{ReaRew}(\mathit{one_entered})$$

(B) *"Average reward until all threads entered the barrier"*

$$\mathrm{ReaRew}(\mathit{all_entered})$$

(C) *"Average reward until any one thread left the barrier"*

$$\mathrm{ReaRew}(\mathit{one_left})$$

(D) *"Average reward until all threads left the barrier"*

$$\text{ReaRew}(\mathit{all_left})$$

(W) *"Average reward until all threads finished writing"*

$$\text{ReaRew}(\mathit{all_done_writing})$$

(Ri) *"Average reward until all threads left round i"*

$$\text{ReaRew}(\mathit{all_left_round}_i)$$

The state predicates have the intuitive meaning. For example $\mathit{all_left_round}_i$ describes the set of states where all threads passed round i. We additionally consider combinations of the above properties, for example D minus B, which measures the average reward accumulated between the last thread entering the barrier and the last thread leaving the barrier.

There are two main parameters to variate for the measurements. The first one is the number of threads participating in the barrier synchronisation. Because the model size grows exponentially in the number of threads we are mostly restricted to thread counts of up to four. The second value to variate is the initial work period's rate. By changing the work rate, we can consider scenarios where threads arrive in smaller and larger intervals.

Evaluation We formalised the four chosen barrier protocols using CTMCs and the suggested properties using the logic CSRL. We then supplied the PRISM model checker with both, in order to retrieve the desired quantitative model checking results. In the following section we present and comment on these results. The evaluation is split into shared and distributed memory protocols.

Shared Memory Barriers First, we assume the initial work period to take 100 cycles on average, which is extremely low considering that barrier synchronisation takes more time than the actual computation task.

Figure 16 presents the basic timing of our two shared memory barriers for different thread counts. We exclude the time before the last thread completes its work period, because the barrier cannot finish earlier. The execution time until the last thread leaves the barrier (dark red) is similar for both the Central Counter Barrier and the B1 Barrier. Considering the first thread to leave, the B1 Barrier performs better.

Figure 17 shows the total time spent and energy consumed for both barriers and thread counts between two and four. The diagram presents the four basic points in time we chose, and the moment the last thread finishes writing, in a stacked fashion. Phase A covers the accumulated time or energy from the starting point until the first thread enters the barrier. During B there is at least one thread that has not entered the barrier. When the last thread arrives, the writing phase begins. The writing phase covers the area where at least one thread has not committed itself to the barrier. Phase C describes the interval in which every thread has written its date, but no thread has left the barrier, yet. Phase D depicts the accumulated time or energy from the first to the last thread leaving the barrier.

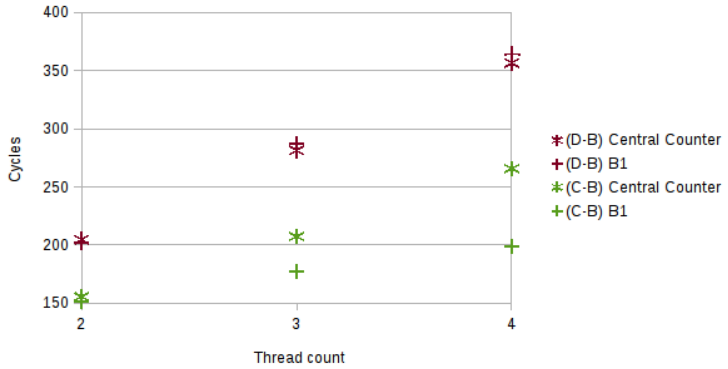

Figure 16: Average execution time for the Central Counter Barrier and B1 Barrier, and a short work period

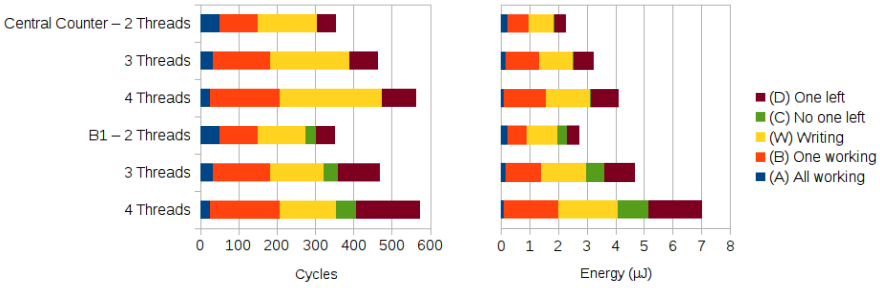

Figure 17: Average execution time and energy consumption for the Central Counter Barrier and B1 Barrier, and a short work period

The length of phase A and B depends only on the thread count. They are equally long for both protocols. In the Central Counter Barrier the last thread to commit itself, very likely immediately exits the barrier. Therefore, there is few time, on average, between the last thread writing and the first one leaving the barrier (green).

As conjectured in Section 4.1 writing is quicker for the B1 Barrier and reading, i.e. phase C and D combined, for the Central Counter Barrier.

All phases take longer with increasing thread count, because of the CTMC semantic for simultaneously enabled transitions, described in Section 4.2.1. The exit rate of a transition is the sum of the rates of all currently enabled transitions. In particular suppose you have n concurrent threads of execution, modelled as interleaved CTMC modules, and in each a transition with rate λ is enabled. The average delay of the first transition to fire is then $\frac{1}{n \cdot \lambda}$. The average time of a parallel execution of all n transitions is $\sum_{i=1}^{n} \frac{1}{i \cdot \lambda}$ rather than the intuitive $\frac{1}{\lambda}$. For example the expected number of cycles until the first thread enters the barrier is $\frac{100}{n}$. The last thread is expected to enter the barrier after roughly 150, 183 and 208 cycles, for two, three and four processes. Because of this influence, phase D for the Central Counter Barrier takes 50, 75 and 92 cycles for two, three and four threads. It is expected to be about the same across all thread counts, since each thread issues exactly one shared read operation. For the same reason the duration of the writing phase for the B1 Barrier should be around 100 cycles instead of 125, 138 and 146 for two, three and four threads.

The energy consumption is similar to the timing. The B1 Barrier requires more energy than the Central Counter Barrier, since it issues more memory operations in a similar amount of time. The writing phase of the Central Counter Barrier uses few energy since it spends considerable time waiting for atomic operations to finish, during which no operations on the same variable are performed. Where the Central Counter Barrier's energy consumption scales similarly to the execution time, the B1 Barrier's energy consumption grows faster than the time needed. Because a thread in the waiting loop of the B1 Barrier iterates between 1 and n times, where n is the number of threads, before leaving the barrier, the two last phases consume considerably more energy than in the Central Counter Barrier.

Aside from the accumulated energy consumption its rate during the different phases, depicted in Figure 18, is also interesting. Phase A uses the baseline power consumption of 11 watts. The overall power consumption of the Central Counter Barrier lies between 16 and 18.3 watts. The B1 Barrier's is considerably higher with 19.4 to 30.6 watts. Because the Central Counter Barrier uses atomic operations and waits busily on only one variable, its power consumption is less in almost every phase. The only part where power consumption is similar, is phase D. The Central Counter barrier only issues shared read operations in this phase, whereas the B1 Barrier executes a mix of shared and local read operations. The execution rate to energy reward ratio is similar, though. Both Barriers have very high power consumption between the moment the last thread commits itself and the first one leaves (green). For the Central Counter Barrier this is due to the shortness of this phase. The B1 Barrier issues almost exclusively shared read operations, which are modelled to consume more energy per cycle than other operations.

Figure 19 shows the ratio of time spent writing versus reading for both protocols. Writing begins when the first thread enters the barrier and ends

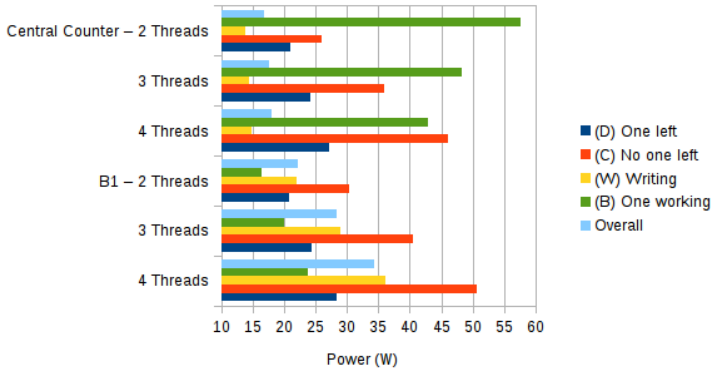

Figure 18: Average energy consumption rate for the Central Counter Barrier and B1 Barrier, and a short work period

when the last thread announces its arrival. Reading begins when the last thread committed itself and ends when the barrier completes. Deviating from our expectation, that the writing percentage would increase, the Central Counter Barrier's ratio is the same across all thread counts. This is due to the CTMC interleaving behaviour, explained earlier in this section. As expected, from our informal analysis in Section 4.1, with increasing thread count the ration of writing to reading time for the B1 Barrier decreases.

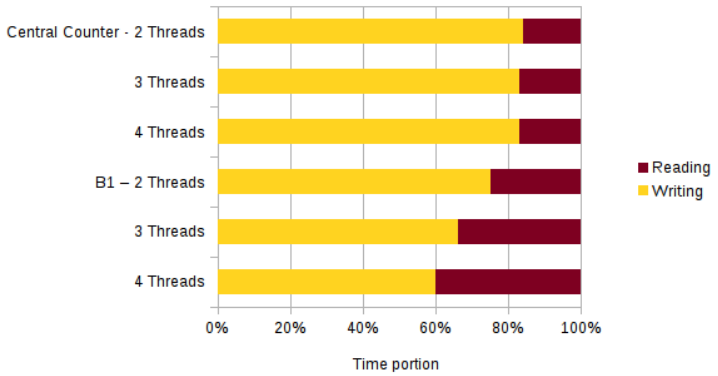

Figure 19: Average time spent reading versus writing for the Central Counter Barrier and B1 Barrier, and a short work period

Having reviewed different metrics for a rather short initial work period, we now have a look at similarly organised figures with an increased work period duration of 1000 cycles. Note that in comparison to real life scenarios this is still small. According to [44] each process at a barrier spends 8194 cycles idly[1]. But an average of 1000 cycles is already enough to emulate threads arriving in

[1] 1813 microseconds accumulated idle time at 450 MHz on 99.6 processes

intervals large enough so that each can conduct its desired work before the next one arrives. Increasing the work period further does not reveal new effects.

As one can observe in Figure 20 the basic execution time of the barriers is very different from before. Being naturally at a disadvantage in the reading phase, and because write operations are serialised due to the large arrival intervals, the B1 Barrier performs worse in every respect. The duration between the last thread to enter and the first thread to leave for the Central Counter Barrier converges to 100 cycles – one atomic operation plus a few additional cycles.

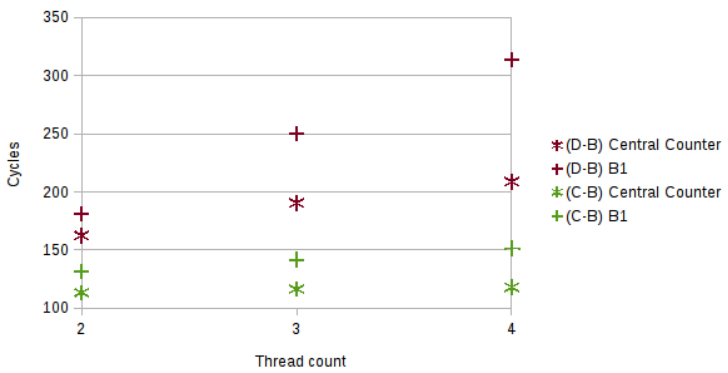

Figure 20: Average execution time for the Central Counter Barrier and B1 Barrier, and a long work period

Figure 21, again, shows the total time spent and energy consumed during the different phases. Phases A and B are excluded because they are large and behave similarly for both protocols.

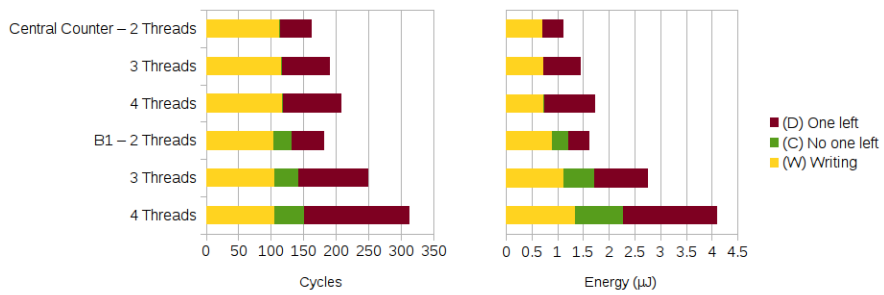

Figure 21: Average execution time and energy consumption for the Central Counter Barrier and B1 Barrier, and a long work period

The timing behaves as expected. The Central Counter Barrier needs one atomic operation and a few more short instructions from the last thread to enter until the first leaves. The B1 Barrier uses a very similar amount of time – a write operation and a few read operations. Since changing the work period only influences the phases before each thread has committed its date, the interval between C and D is the same as in Figure 17.

Energy consumption during the writing phase of the Central Counter Barrier is constant across all thread counts, because atomic operations block all other threads from busily reading the barrier variable. The B1 Barrier in turn polls constantly, and therefore energy consumption increases with increasing thread count.

The rate of energy consumption behaves similarly to the previous measurement, where we used an average work period duration of 100 cycles. The overall power consumption is now dominated by phases A and B, which are similar across protocols. The B1 Barrier requires between 15.3 and 23.5 watts and the Central Counter Barrier uses 14.6 to 20.9 watts. The B1 Barrier can catch up considerably.

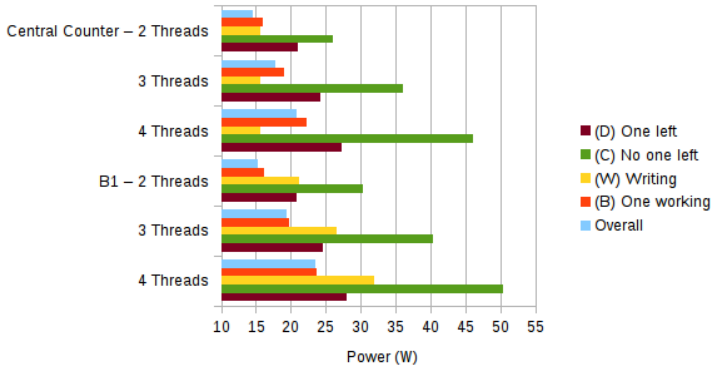

Figure 22: Average energy consumption rate for the Central Counter Barrier and B1 Barrier, and a long work period

If one increases the length of the work period beyond 1000 cycles, both algorithms issue few serial write operations followed by busy-waiting. The timing is then similar to our 1000 cycle case. The power consumption converges to the same value for both protocols.

Distributed Memory Barriers This section presents a comparison of the Dissemination Barrier and the B2 Barrier, similarly organised as the previous section. Additionally, we analyse the Dissemination Barrier with up to eight concurrent processes. If we attempt the same for the B2 Barrier, the model size exceeds the manageable. Furthermore, we quantify the number of remote operations the B2 Barrier issues.

We begin with a work period of 100 cycles, simulating that processes arrive in extremely small intervals. The stacked diagrams in Figure 23 show the total time spent and energy consumed for both protocols and varying thread counts through the different phases. Both protocols complete similarly fast. The B2 Barrier has an advantage at over two processes. The Dissemination Barrier finishes 23 cycles earlier for two processes, because remote writes can be issued during another process' work period, whereas the B2 Barrier needs to execute two more remote read operations after the last process enters the barrier. For the B2 Barrier after the last process arrives, the first one leaves in 50 cycles for two

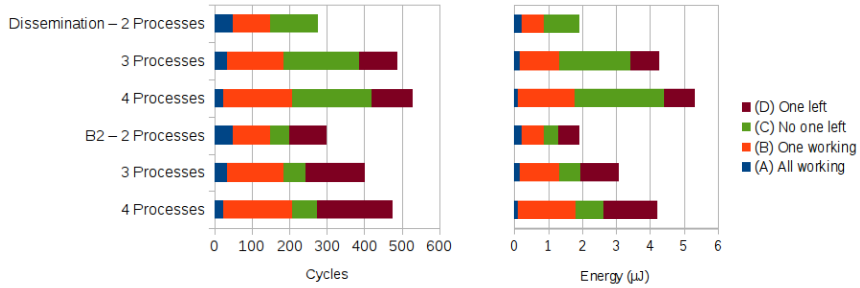

Figure 23: Average execution time and energy consumption for the Dissemination Barrier and B2 Barrier, and a short work period

processes. Since a remote read takes 100 cycles on average, this is unrealistically early. The CTMC interleaving behaviour, explained in the previous section, causes this effect. Furthermore, one can observe that there is a wider spread between the first and the last process to leave for the B2 Barrier. For the Dissemination Barrier at two processes the first and the last process leave at nearly the same time, because when the first realises its peer has completed its remote write, it already completed its own. Both remote writes finish before either process can leave and therefore phase D is one cycle short.

Energy consumption behaves similarly to execution time through the different phases.

Figure 24 presents execution time and energy consumption of the Dissemination Barrier through its rounds for up to eight processes. As suspected in

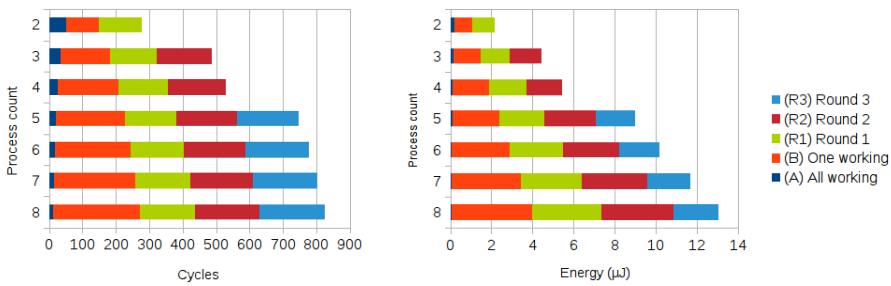

Figure 24: Average execution time and energy consumption for the Dissemination Barrier, and a short work period

Section 4.1 it shows a stair-shaped execution time distribution. With each new round the duration increases by a large step, whereas adding processes and maintaining the number of rounds increases runtime very little. Normally, because remote writes can be issued concurrently, one expects the runtime to not increase at all. But due to the CTMC interleaving semantic an increase in concurrently issued operations increases the average time it takes to complete all of them. Here one can interpret this effect as resource contention, i.e. issuing multiple messages concurrently increases the time to transfer all of them. We expect this behaviour to be visible in runtime benchmarks as well.

Energy consumption with increasing process count and constant number of rounds increases faster than the execution time. There are two reasons for this. First, more remote operations are issued if processes are added, as described in Section 4.1. Second, having more processes means more processes are waiting busily and therefore increase energy consumption.

The B2 Barrier's rate of energy consumption is similar to the Dissemination Barrier's as can be observed in Figure 25. That is, because the execution time and the number of issued remote operations is similar. The Dissemination Barrier uses between 17.4 and 25.2 watts, whereas the B2 Barrier requires between 16.1 and 22.2 watts. Only phase D differs, but not significantly. This is due

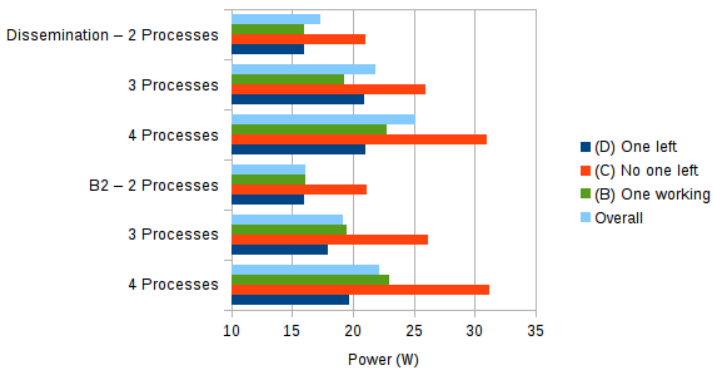

Figure 25: Average energy consumption rate for the Dissemination Barrier and B2 Barrier, and a short work period

to the fact that all processes in the Dissemination Barrier poll actively on local memory, where in the B2 Barrier they poll remotely and therefore more slowly.

Switching to a 1000 cycle long work period, thus emulating that processes arrive in larger intervals, Figure 26 illustrates the distribution of execution time and energy consumption for both barrier protocols. As in the previous section we omit phase A and B, because they are long and behave equally in both protocols. The measurement is similar to the 100 cycle case. Here, too, the first process to leave the B2 Barrier does so unrealistically early.

Figure 27 shows the same measures for the Dissemination Barrier with up to eight processes. Comparing to Figure 24 there is almost no additional execution time per process involved, if the number of rounds does not increase. Remote writes inside one round are approximately serial. Therefore, a write is oftentimes enabled at the same time as other processes repeatedly execute local reads. The expected time such a write takes, according to CTMC semantics explained in Section 4.2.1, is then 100 processor cycles. The duration of each round is related to the explanation on progress in Section 4.1. When the last process arrives at the barrier, exactly one remote write needs to be executed to finish round one, two more are needed to finish round two, and four writes are required to complete the third round. Therefore, and because of CTMC interleaving semantics, approximately 100, 150 and 183 cycles are needed to finish rounds one, two and three.

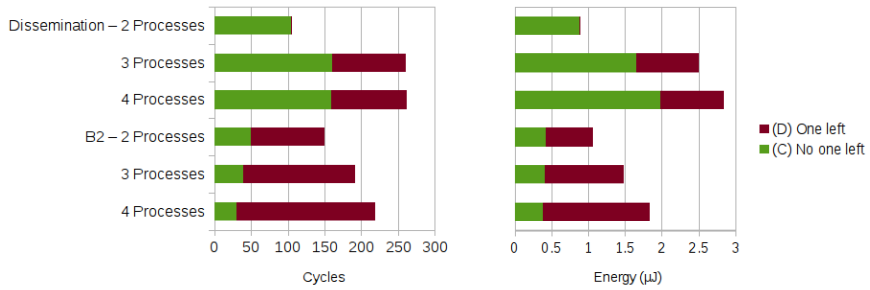

Figure 26: Average execution time and energy consumption for the Dissemination Barrier and B2 Barrier, and a long work period

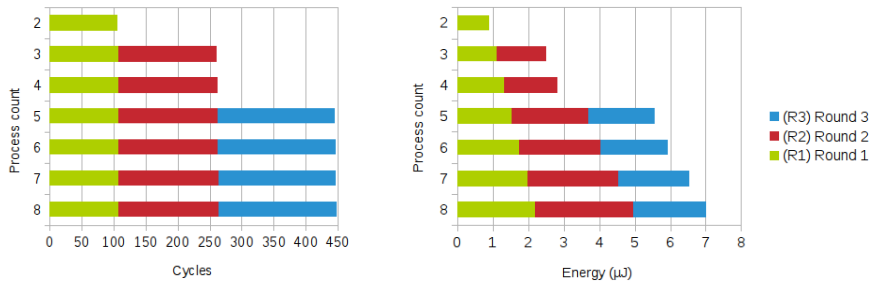

Figure 27: Average execution time and energy consumption for the Dissemination Barrier, and a long work period

Energy consumption behaves similarly to that illustrated in Figure 24 where we assumed a work period of 100 cycles.

As we observed, there is a certain cost for each additional round in the Dissemination Barrier. Due to model checking limitations we are unable to compare the Dissemination Barrier to the B2 Barrier for more than four processes, i.e. two rounds. In Figure 26 one can see that the B2 Barrier's execution time is spread more evenly than that of the Dissemination Barrier. We expect this advantage to become greater with increasing process count. Comparing to the 100 cycle work period results, the B2 Barrier does not increase its advantage. The B2 Barrier in phase C and D uses 84.6 and 53.5 fewer cycles for three and four processes than the Dissemination Barrier, whereas for the larger work period it uses 69.5 and 43.4 fewer cycles. This is surprising, and there are a number of possible explanations. First, the B2 Barrier might handle the case where processes arrive in large intervals less efficient than expected. Second, the fact that the Dissemination Barrier exchanges relatively few messages might be more important than assumed. This means there is no clear indication that the progress problem, described in Section 4.1 exists. It might surface, if one is able to analyse beyond four processes, though.

Figure 28 displays the rate of energy consumption for both protocols. It is similar to the 100 cycle work period case, except that it uses overall less power, because a longer work period means that more time is spent idly.

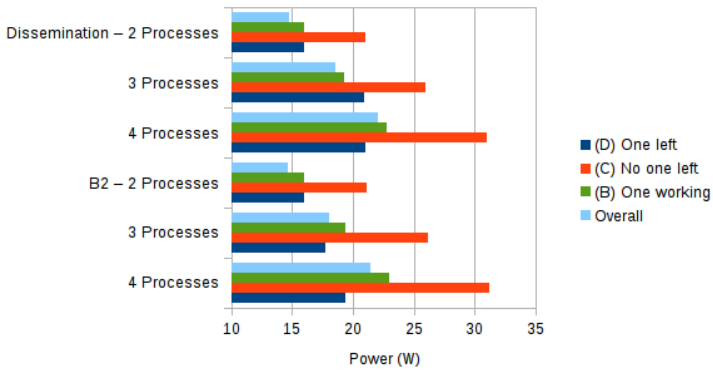

Figure 28: Average energy consumption rate for the Dissemination Barrier and B2 Barrier, and a long work period

Aside from execution time and energy consumption, the number of remote transfer operations is another interesting measure to evaluate. As explained in Section 4.1, this number is fixed for the Dissemination Barrier – $n \cdot \lceil \log_2 n \rceil$ where n is the process count. The number of successful, i.e. non-zero, remote read operations for the B2 Barrier lies between $2 \cdot (n-1)$ and $n \cdot (n-1)$. We are unable to give a better estimate. This is where model checking comes into play.

Table 2 presents the number of remote operations, assuming a work period duration of 100 cycles. The B2 Barrier issues more remote transactions than the Dissemination Barrier, but comparatively few. For the Dissemination Barrier the number of remote operations depends heavily on the number of rounds,

Table 2: Average number of remote transfers for the Dissemination Barrier and B2 Barrier, and a short work period

	Threads	Successful transfers	Failed transfers
Dissemination	2	2	
	3	6	
	4	8	
B2 Barrier	2	2	0.99
	3	4.74	1.73
	4	7.98	2.44

therefore the situation might get better for the B2 Barrier with process counts higher than four.

Table 3 shows the same measurement for an assumed work period length of 1000 cycles. The number of transfers for the Dissemination Barrier is the same

Table 3: Average number of remote transfers for the B2 Barrier, and a long work period

Threads	Successful transfers	Failed transfers
2	2	9.9
3	4.95	22.89
4	8.78	38.05

as before. If processes arrive in large intervals, each time a new one arrives it gets concurrently polled by all others. The resulting communication pattern is more inefficient than if processes would share their presence in a chain- or tree-like fashion. Therefore, the B2 Barrier issues slightly more successful remote reads in this case. The number of failed transfers is, as expected, considerably higher. Through implementing a back-off strategy this number can be lowered in exchange for a higher execution time.

4.3 Discussion

Each analysed protocol is functionally correct in the sense that they indeed perform a barrier synchronisation and do not deadlock. Each barrier is reasonable fast and consumes an expected amount of energy.

Despite relying on atomic operations the Central Counter Barrier performs better than our proposed protocol, the B1 Barrier. The approach to trade less time writing for more time reading does not pay off for two reasons. First, in a balanced scenario the time lost due to queuing atomic operations is small. Second, atomic operations blocking access to a variable results in less busy waiting, which in turn lowers energy consumption.

In the distributed memory case, the B2 Barrier outperforms the Dissemination Barrier by most measures. As conjectured in Section 4.1, the Dissemination Barrier wastes substantial time due to its fixed, round-based communication pattern. For non-power of two process counts it even gets worse. The main drawback of the B2 Barrier is, due to busy waiting on remote memory, it issues

more remote operations overall and therefore allocates more bandwidth than the Dissemination Barrier.

The CTMC model checking semantic for simultaneously enabled transitions is different from our intuitive notion of parallel execution. Because of this, and since the assumptions we make do not closely reflect reality, one has to exercise caution when interpreting the model checking results.

5 Conclusion and Future Work

In this work we present three innovative barrier protocols. We compare two of them to state-of-the-art protocols using model checking techniques. The B2 Barrier, having a shorter execution time and lower energy consumption than the Dissemination Barrier, turns out to be a promising algorithm. We, therefore, conclude that today's barriers can be improved upon. To achieve the displayed performance, the B2 Barrier does not use a fixed communication pattern but relies on randomness. Weakening determinism and using randomness are core principles of the pW/CS lock. Thus, we consider the B2 Barrier an example for the aptitude of these principles to increase synchronisation performance. Model checking enables exhaustive and fine-grained analysis beyond what measurement-based methods deliver. Although model size and accuracy is admittedly limited, we expect future work in this direction to alleviate these deficiencies. We belief that model checking is a powerful tool to aid the development and analysis of algorithms, and synchronisation protocols in particular.

The possible future directions of this work can be divided into five categories. First, this thesis lacks benchmarks. One should repeat the analysis of Section 4.2.3 using measurement. One then sees how the protocols perform in reality, and can also use the obtained results to refine the existing probabilistic models.

Second, new barriers and variations of our presented ones should be evaluated. For example one can try different increment strategies for the iteration variable of the B2 Barrier. Both the B1 and the B2 Barrier exclusively read remotely and write locally. In particular, they repeatedly poll remote memory. By switching to remote writing in conjunction with local reading, one might be able to get rid of busy waiting on remote memory and leverage the better performance of remote write operations on some architectures, like the Epiphany [9] multi-core processor.

Third, model checking should be extended. The unfortunate limit of four processes for our analysis ought to be improved. To achieve this, one can employ techniques like symmetry reduction, partial order reduction and manual fine-tuning. Another interesting pursuit is to model the protocols in greater detail. Today's processors, for example, have a multi-layered cache. In order to get preciser performance predictions one can incorporate such intricacies into the model. Resource contention is another detail we do not model in this work. Our theoretical computer can concurrently transfer data between processes with arbitrarily high bandwidth.

Fourth, we encourage to search for alternative modelling formalisms or to modify/limit the existing CTMC/CSRL semantic. The way interleaving is treated does make sense, if a formula queries information about one module, for example if we ask for the arrival time of a specific thread. But, when modelling multi-threading through multiple modules, if one checks quantitative properties about multiple modules, the behaviour is unintuitive.

The fifth suggestion is to improve tool support. Model checking complements testing and benchmarking algorithms. But, tools like SPIN and PRISM are both very extensive and hard to use. They have esoteric input languages, cumbersome user interfaces, and require the user to have background knowledge about model checking and temporal logics. A specialised tool set to aid the design of parallel algorithms is much desired.

Bibliography

[1] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time markov chains. In *Computer Aided Verification*, pages 269–276. Springer, 1996.

[2] C. Baier, B. Engel, S. Klüppelholz, S. Märcker, H. Tews, and M. Völp. A probabilistic quantitative analysis of probabilistic-write/copy-select. In *NASA Formal Methods*, pages 307–321. Springer, 2013.

[3] C. Baier, B. Haverkort, and H. Hermanns. On the logical characterisation of performability properties. In *Automata, Languages and Programming*, pages 780–792. Springer, 2000.

[4] C. Baier, B. Haverkort, H. Hermanns, and J.-P. Katoen. Model-checking algorithms for continuous-time markov chains. *Software Engineering, IEEE Transactions on*, 29(6):524–541, 2003.

[5] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *4th Symposium on Experimental Distributed and Multiprocessor Systems*, pages 57–71, 1993.

[6] R. Brendel. A lock-less shared memory barrier without atomic operations, Apr. 2013. Available upon request.

[7] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–10. IEEE, 2011.

[8] Programming environments release announcement for Cray XE and Cray XK systems. `http://docs.cray.com/books/S-9407-1306/S-9407-1306.pdf`, june 2013.

[9] Epiphany multicore IP. `http://www.adapteva.com/products/epiphany-ip/`.

[10] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*, page 85. AUUG, Inc., 2002.

[11] A. Gara, M. A. Blumrich, D. Chen, G.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, et al. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2.3):195–212, 2005.

[12] S. R. Garea and T. Hoefler. Modeling communication in cache-coherent smp systems - a case-study with xeon phi. In *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, Jun. 2013.

[13] K. Gharachorloo. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *In Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

[14] The GNU C Library (glibc). `http://www.gnu.org/software/libc`.

[15] GOMP – an OpenMP implementation for GCC. `http://gcc.gnu.org/projects/gomp`.

[16] W. D. Gropp, E. L. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the Message-Passing Interface*, volume 1. the MIT Press, 1999.

[17] R. Gupta, V. Tipparaju, J. Nieplocha, and D. Panda. Efficient barrier using remote memory operations on VIA-based clusters. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 83–90. IEEE, 2002.

[18] S. Habata, M. Yokokawa, and S. Kitawaki. The earth simulator system. *NEC Research and Development*, 44(1):21–26, 2003.

[19] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1988.

[20] T. Hoefler. Evaluation of publicly available barrier-algorithms and improvement of the barrier-operation for large-scale cluster-systems with special attention on InfiniBand™networks, Apr. 2005.

[21] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. Adding low-cost hardware barrier support to small commodity clusters. In *ARCS Workshops*, pages 343–350, 2006.

[22] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. Fast barrier synchronization for InfiniBand™. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 7–pp. IEEE, 2006.

[23] G. J. Holzmann. The model checker SPIN. *Software Engineering, IEEE Transactions on*, 23(5):279–295, 1997.

[24] InfiniBand Trade Association. InfiniBand™architecture specification volume 1, release 1.2.1, november 2007.

[25] J. G. Kemeny and J. L. Snell. *Finite markov chains*, volume 210. Springer-Verlag New York, 1976.

[26] V. G. Kulkarni. *Modeling and analysis of stochastic systems*, volume 36. CRC Press, 1995.

[27] M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *Formal methods for performance evaluation*, pages 220–270. Springer, 2007.

[28] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic model checking for performance and reliability analysis. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009.

[29] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[30] J. Liu, J. Wu, and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[31] N. Mc Guire. Probabilistic write copy select. In *13th Real-Time Linux Workshop*, pages 195–206, Oct. 2011.

[32] N. Mc Guire, P. Okech, and G. Schiesser. Analysis of inherent randomness of the linux kernel. In *Eleventh Real Time Linux Workshop, Dresden, Germany*, 2009.

[33] A Message-Passing Interface standard version 2.2. `http://mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`, september 2009.

[34] A Message-Passing Interface standard version 3.0. `http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf`, september 2012.

[35] MPICH | High-Performance Portable MPI. `http://www.mpich.org/`.

[36] Message Passing Interface forum. `http://mpi-forum.org`.

[37] MSI protocol – Wikipedia. `http://en.wikipedia.org/wiki/MSI_protocol`.

[38] MVAPICH2 (MPI-3 over OpenFabrics-IB, OpenFabrics-iWARP, PSM, uDAPL and TCP/IP). `http://mvapich.cse.ohio-state.edu/overview/mvapich2/`.

[39] OpenMP. `http://openmp.org`.

[40] Open MPI: Open Source High Performance Computing. `http://www.open-mpi.org/`.

[41] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann, 1997.

[42] J. Postel. User Datagram Drotocol, 1980.

[43] PRISM – probabilistic symbolic model checker. `http://www.prismmodelchecker.org`.

[44] R. Rabenseifner. Automatic MPI counter profiling. In *42nd CUG Conference*, 2000.

[45] A hardware-accelerated MPI implementation on SGI® UV™ 2000 systems. `http://www.sgi.com/pdfs/4378.pdf`.

[46] S. F. Siegel and G. S. Avrunin. Modeling MPI programs for verification. *Department of Computer Science, University of Massachusetts, Tech. Rep. UM-CS-2004-75*, 2004.

[47] Spin – formal verification. `http://spinroot.com`.