Correctly Synchronised POSIX-threads
Benchmark Applications

Christos Sakalis
Abstract

Correctly Synchronised POSIX-threads Benchmark Applications

Christos Sakalis

With the future of high performance computing quickly moving towards a higher and higher count of CPU cores, the need for efficient memory coherence models is becoming more and more prevalent. Strict memory models, while convenient for the programmer, limit the scalability and overall performance of multi- and manycore systems. For this reason, relaxed memory models are looked into, both in academia and in the industry. Applications written for stronger memory models often contain data races, which cause unexpected behaviour in more relaxed models, many of which rely on data race free code to work. At the same time, some of the most widely used programming languages now require data race free code. For these reasons, the need for benchmarks based on properly synchronised code is bigger than ever. In this thesis, we will identify data races in major benchmark suites, remove them, and then quantify and compare the performance differences between the unmodified and the properly synchronised versions.
Contents

1 Introduction ................................................................. 1
2 Background ................................................................. 3
  2.1 Memory Consistency .................................................. 3
    2.1.1 Happened-Before Relationship ................................. 3
    2.1.2 Consistency Models ............................................... 4
    2.1.3 Data Races ..................................................... 5
  2.2 Memory Coherence .................................................... 6
    2.2.1 MESI Protocol .................................................. 6
    2.2.2 VIPS-M Protocol ............................................... 7
    2.2.3 Forward Self-Invalidation/Self-Downgrade .................... 8
  2.3 Synchronisation Mechanisms ....................................... 9
    2.3.1 Barriers and Fences ............................................ 9
    2.3.2 Locks and Semaphores ......................................... 9
    2.3.3 Conditional Variables and Monitors ......................... 10
    2.3.4 Atomic Operations ........................................... 11
    2.3.5 Transactional Memory ......................................... 11
  2.4 Software Tools .................................................... 12
    2.4.1 Helgrind ....................................................... 12
    2.4.2 ThreadSanitizer ............................................... 12
    2.4.3 Fast&Furious .................................................. 13
3 Application Modifications ............................................. 15
  3.1 Synchronisation Methods .......................................... 15
    3.1.1 Mutual Exclusion .............................................. 15
    3.1.2 Atomic Operations ............................................. 16
    3.1.3 Exposing the Synchronisation to the Hardware ............. 16
    3.1.4 FSID Fences .................................................. 17
  3.2 The SPLASH-2 Applications ....................................... 17
    3.2.1 Barnes ....................................................... 18
    3.2.2 Cholesky ..................................................... 20
    3.2.3 FMM ........................................................ 21
    3.2.4 Ocean ....................................................... 22
    3.2.5 Radiosity .................................................... 23
    3.2.6 Raytrace ..................................................... 25
    3.2.7 Volrend ..................................................... 26
  3.3 The PARSEC Applications ......................................... 27
    3.3.1 Streamcluster ................................................ 27
4 Evaluation ...................................................................... 29
  4.1 Synchronisation Primitives ......................................... 30
  4.2 Performance Characteristics ....................................... 31
5 Conclusion ................................................................. 34

Bibliography
A Additional Listings
B Annotation & Traces
C Simulation Parameters
Parallel programming is of increasing importance [Sut05], and multicore architectures now dominate the market. A lot of research, both in academia and in the industry, is focused on improving the performance of parallel architectures, as well as making parallel programming easier to use and less error prone.

A part of that research is dedicated to memory caches, memory coherence, and the memory consistency models used by various systems. In this field, given the unnecessary performance restrictions caused by strict memory models, we see relaxed memory models becoming more and more popular. These systems require more explicit synchronisation than many applications currently use, while data races can cause the programs to not run correctly. At the same time, major programming languages, such as C [ISO11], C++ [ISO15] and Java [Gos+14], have already introduced relaxed memory models, where data races as regarded as logic errors in the code. The memory models defined in these languages are known as “Sequential Consistency for Data Race Free” (SC for DRF), which means that as long as the code does not contain any data races, the program will behave as if run under a sequential consistency memory model (Section 2.1.2). Finally, some times programmers employ data races while naively expecting some specific behaviour, but that behaviour is not necessarily guaranteed. This leads to bugs that are very hard to identify and fix [MA04].

In this thesis, we investigated modifying two parallel benchmark suites (SPLASH-2 [Woo+95] and PARSEC 2.1 [Bie11]) into data race free code. We begin by giving an overview of memory coherence, memory models, and synchronisation methods (Section 2). We then explain where we found data races, what causes them, and how to remove them (Section 3). We also present a version with all the synchronisation exposed to the hardware, using special “magic” functions (Section 3.1.3). This version can be used in conjunction with trace based simulation for more accurate results [Nil+]. Finally, we present the performance differences, if any, introduced by the changes we made (Section 4). By doing so, we enable other users to use the data race free suites as a benchmark tool for any further research.

The benchmark suites we are using are well known and used widely in both academia and the industry. Specifically, the SPLASH-2 suite has been around since the 90’s, with some of the programs included being even older than that. It is used widely in conjunction with simulators to measure various performance aspects of new designs. As far as we know, at the time this thesis is being written, there is no other data race free version of the SPLASH-2 suite. Similarly, the PARSEC suite is also well established in the field, even though it is much newer than SPLASH-2.

For detecting the data races, we depend on the Fast&Furious tool, developed by Ros and Kaxiras [RK15]. The tool is designed specifically for detecting potentially harmful data races under the release relaxed memory model (Section 2.1.2).

Finally, we work with self-invalidation and self-downgrade coherence protocols [RK12; KR12; KR13], as well as the forward self-invalidation/self-downgrade model, which is described later and is part of a not yet published work by Ros et al.

It should be noted that we will often talk about “threads”, “cores” and “proces-
sors”, terms which are used quite interchangeably in much of the bibliography. This is due to the fact that much of the bibliography comes from different but related fields of computer science. At the same time, some of the concepts are old now and the hardware systems then used to be quite different. For example, processors would only have one core and were able to run only one thread in parallel. We will mostly use the term “thread” when referring to parallel execution, the term “processor” for a CPU that might be able to run multiple threads in parallel, and the term “core” for the units that execute those threads. Similarly, while memory coherence protocols and memory models are different concepts, in our case they are directly linked with each other, as each coherence protocol described comes with a matching memory model.
2.1 Memory Consistency

Modern CPUs have reached such high processing speeds that they often spend a lot of time waiting for main memory accesses. However, faster memories are both more expensive and harder to add in large quantities. For that reason, most modern CPU architectures feature a hierarchy of caches in increasing size and decreasing access speed. Since those caches keep copies of the data, and those data might be shared between different threads, cores, and processors, there is a need for keeping those data consistent across both different caches and the main memory.

2.1.1 Happened-Before Relationship

“Happened-before relationships” is a fundamental concept in any computer system where different events can run asynchronously. It was first introduced by Lamport [Lam78] in his seminal paper regarding the ordering of events in distributed systems. Lamport correctly observed that while in real life it can be easily established if an event $a$ occurred before or after some other event $b$ by simply comparing the timestamps of these two events, in a distributed system it is not always possible to do so. Lamport established the “happened-before” relationship between two events, to be symbolised as $\rightarrow$, as follows:

1. If $a$ and $b$ happen in the same process and $a$ is executed before $b$, then $a \rightarrow b$.

   In the case of one processor, it is trivial to verify the event ordering.

2. If $a$ is the sending of a message from a process and $b$ is the receiving of that message from some process, then $a \rightarrow b$.

3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$ (transitive property).

If a happened-before relationship cannot be established between two distinct events $a$ and $b$, then these two events are called concurrent events.

While Lamport was focused in distributed systems, the same concepts apply to the memory coherent multithreaded systems we are using. Lamport’s processes are now threads running on different cores or processors, and the signals being sent and received are synchronisation operations, such as the release and acquire operations (Section 2.1.2). The happened-before relationships are important in these memory coherent systems because if shared data is read and written by different threads without establishing some sort of order between them (i.e. concurrently), then data races can occur (Section 2.1.3) and the application can behave in an unexpected manner.
2.1.2 Consistency Models

A consistency model is a concept mainly found in distributed or parallel systems. It is a set of rules governing the software and the data manipulation done on the system, and a set of expected results when these rules are followed. In the context of parallel programming, we refer to memory (consistency) models, which govern the rules of accessing shared data, how synchronisation works, and what are the expected execution results of a program that follows these rules. Essentially, a memory model can be seen as a contract between the software and the hardware, where the hardware guarantees some observed behaviour as long as the software abides by the rules.

The memory models are often characterised as “strong/strict” or “weak/relaxed”. A naive programmer would expect all operations in an application to happen in the same order as they appear in the code (program order). That is however not true, since the actual order of both the operations and the observed side effects can differ from the program order. This happens because both the compiler and the CPU might decide to alter that order, in order to improve performance. At the same time, in parallel and distributed systems, there is also the inability to absolutely order all the events, since the data needs to be sent over the bus or the network. As such, the stronger a memory model is, the closer it adheres to the expected program order execution of the code. On the contrary, weaker memory models are free to change that order, and often require the programmer to explicitly mark the cases where the order needs to be enforced. Generally speaking, stronger models are easier for the programmer to understand and use, since they follow the order one would logically expect, while weaker memory models trade ease of use for more optimisation opportunities and thus better performance.

Sequential Consistency

Sequential consistency (SC), first defined by Lamport [Lam79], is one of the simplest and strongest models. It states that all the memory operations, both the reads and the writes, should appear as if executed in the same order as they appear in the code [Gha+90]. Since in parallel programs that order is not always well defined, with operations running in parallel (Section 2.1.1), it also states that between all the different threads, the observable order should be the same, as if all the operations were queued using one FIFO queue. Essentially, all the threads should observe the same interleaving of operations. Another way to see it is that the execution of all the threads should be as if they were executed serially by the same thread, regardless of the way they are interleaved.

The sequential consistency model is perhaps one of the easiest models for the programmer to use and understand, since it very closely follows the program order. However, since it is a very strong model, it also means that the compiler and the CPU are not allowed to perform almost any optimisations, and this can be detrimental to performance.

Total Store Order

Total store order (TSO) is a memory model that can be found on the Sparc processors [Wea94; SFC92]. Also, the x86 family of processors defines, albeit badly, a memory models that is very close to TSO [OSS09; Sew+10].

The TSO model is closely related to the sequential consistency model, with the difference that write buffers are introduced. Read operations still happen in program order, but the writes can be put into a FIFO buffer, which hides them from other cores. This means that while a write might be visible in one core, it might not be in another. However, since a FIFO buffer is used, once a write is visible, all the writes preceding it (from the same core) have already become visible. This implies that is possible to use normal write operations as an implicit synchronisation mechanism. Given the popularity of the x86 platform for consumer computers, many programs
take advantage of the TSO model to avoid using explicit synchronisation in their programs, in order to improve performance.

Weak Consistency
In a weak consistency model, the order of both the read and the write operations is not guaranteed, unless properly synchronised using memory barriers, also known as memory fences. The compiler and the processor are free to reorder the memory operations as they see fit, with the exception of synchronisation operations. Also, operations are not allowed to be reordered relative to synchronisation operations, i.e. everything that appears in the code before a synchronisation operation must be executed before the synchronisation, and everything that appears in the code after the synchronisation must be executed after. Other than that, no guarantees are made on the observable order of execution or when writes will be made visible to the other cores. Weak memory models are very popular on distributed systems, where the cost of synchronisation is especially high.

Release Consistency
Release consistency is a special case of weak consistency. It has two variants, “eager” and “lazy”, depending on if the writes are made visible directly after the release or only after an acquire. We will assume the lazy version.

Like weak consistency, the order of the reads and writes is not guaranteed, unless explicit synchronisation is used. However, the model provides two different synchronisation operations. A release operation will make all the writes performed by a thread visible to any other thread that performs an acquire. Essentially, a simple way to think about it is that the release operation can be thought as flushing all the pending write operations, while an acquire operation makes sure all read operations return the most current data than can be found in the main memory. Usually, the systems that provide a release memory models also provide a full synchronisation operation that performs both release and acquire, much like the one found in the weak memory model.

Obviously, the implicit synchronisation that works on models such as TSO will not work on this model, since unless explicitly told not to do so, the compiler and the CPU are free to reorder the memory accesses as they see fit. At the same time, in order to achieve correctness and maximum performance, the programmer needs to know when to use a release or an acquire operation. These make this model a bit hard to use. However, it is a very popular model, with many of the ARM processors supporting it. Also, some of the most popular programming languages, such as C, C++, and Java specify release memory models. Finally, the VIPS-M protocol, discussed in subsection 2.2.2, provides a release model.

2.1.3 Data Races
Data races happen when multiple threads of execution access the same data without establishing a happened-before relationship (Section 2.1.1), under the condition that at least one of those threads writes to these data. Multiple reads from the same data do not need to be ordered and do not constitute a data race.

Data races can appear in applications either by accident or on purpose. In the first case, the developers of the application are not aware that there are shared data between threads, or they failed to synchronise the accesses properly. In the second case, races often occur either as implicit synchronisation mechanisms, of which the software and hardware do not know about, or because the programmers deemed the synchronisation unnecessary and wanted to avoid it.

Even if the data races happen on purpose, they still constitute a problem. First of all, many modern programming languages, such as C, C++, and Java, define memory models that require data race free code. These models...
guarantee sequential consistency (Section 2.1.2) and correctness of the program only if the code does not contain any races (SC for DRF). Specifically, in C and C++, code that contains data races leads to undefined behaviour. Writing code that contains data races can lead to some every unexpected and obscure bugs [MA04].

Another problem is that software that depends on data races following some specific behaviour is often not portable. For example, the Intel x86 architecture defines a somewhat strict memory model [Int13; OSS09; Sew+10] thus allowing the programmers to avoid, in many cases, using explicit synchronisation. However, software that take advantage of the strict memory model on x86 will most likely not work properly on a system that employs a more relaxed model, such as many ARM processors [SSW04].

2.2 Memory Coherence

In order to achieve memory consistency as described above, special protocols for the cache and memory communications are being used. By “memory coherence” (also “cache coherence”) we refer to these protocols and operations. While it is possible for each coherence protocol to support different memory models, some of them match together more closely than others.

2.2.1 MESI Protocol

The MESI protocol is a memory coherence protocol that it was developed at the university of Illinois in the 80’s [PP84], but is still widely used today.

The MESI protocol classifies each cache line with one of four different states. It then operates as a finite-state automaton, switching between the states based on the current state and the input, where the input can be either an operation from the CPU(s) owning the memory or a bus message originating from a different CPU that wants to access the data. The four states are:

Modified: The cache line exists in the cache, it has been modified, and the changes have not been written back (i.e. it is dirty). No other private cache contains the cache line.

1Caches shared among all the cores do not need to be kept consistent in the same way.

Figure 2.1: The MESI state and transition diagram.
Exclusive: The cache line exists in the cache but it has not been modified (i.e. it is clean). No other private cache contains the cache line.

Shared: The cache line exists in the cache and it has not been modified. Other private caches might also contain the same cache line in the same state.

Invalid: The cache line does not exist in the cache.

Figure 2.1 shows how the protocol switches between states based on the CPU and bus (i.e. other CPU(s)) operations. The black lines represent operations by the CPU, while the red lines represent bus messages. The labels on the black lines show what the input was, and what operation the cache takes to respond to that input, while the labels on the red lines are the other way around.

Let us take as an example the waiting for and the setting of a flag. We will assume two processors A and B, each with its own private cache. A will start first and execute a while loop, checking the value of the flag, while B will run later and set the flag.

Both A and B start with the cache line state set to Invalid. A wants to read the flag data, so it will issue a Read operation. The cache, receiving the Read, will ask for the memory over the bus, indicating that it only wants to read the data, and when it receives the data, along with the information that no other cache line has that data, it will set the cache line state to Exclusive. Now it is B’s turn to run. Since it wants to write to the data, the processor with issue a Write operation. The cache, upon receiving that operation, will check to see if it contains the data. Since it does not, it will issue a Read-to-Write message over the bus, signalling that it needs the data and that it also intends to modify them. Cache A, receiving that message, will automatically downgrade the cache line from Exclusive to Invalid. Then, cache B, will receive the data, set the cache line to Modified and proceed with the write. Now, it is processor’s A turn to run again. Once again, it will issue a Read operation, and the cache will ask for the data over the bus. Cache B, will receive the request, downgrade itself to Shared and send the modified data over the bus. Then, cache A will receive the data it requested, set the cache line to Shared and proceed with the other operations. So, at the end, both caches will contain the same data, and both cache lines will be set to the Shared state.

2.2.2 VIPS-M Protocol

The VIPS-M protocol is a new memory coherence protocol aiming to simplify memory coherence, thus allowing for faster and more efficient caches [RK12; KR12; KR13]. It is based on the self-invalidation/self-downgrade (SISD) paradigm and it provides a release memory consistency model.

VIPS-M has two states and two classifications for the cache lines. Each cache line can be either Valid or Invalid, signifying that the cache line does or does not exist in the cache. The cache lines switch between those two states simply by reading the data from the main memory (or a higher level shared cache). At the same time, each page can be either Private or Shared. A page is classified as Private if all of the cache lines in it are being accessed only by a single core, and Shared if by more than one. Normally, all pages start as Private, until they are accessed by more than one CPU at the same time, in which case they are switched to Shared. Depending on the implementation, a page that is classified as Shared might never revert back to Private again.

The VIPS-M protocol is based on self-downgrade and self-invalidation. Self-downgrade means that each cache is responsible for writing back to the main memory (or, again, a higher level shared cache) any changes it has performed. This can be done gradually during normal operation, but it also has to be enforced during a release synchronisation operation. Similarly, self-invalidation means that the cache is

\[\text{VIPS-M performs the classification on the operating system level, so page level granularity is used.}\]
responsible for acquiring any updated data that can be found in the main memory. Often, self-invalidation occurs after an acquire synchronisation operation. By using the \textit{Private} and \textit{Shared} classifications, it is possible to delay or avoid self-downgrade and self-invalidation, since cache lines in a \textit{Private} page are guaranteed to not be read or written in any other cache.

During self-downgrade, it is possible that false-sharing can occur. To avoid overwriting valid data, the VIPS-M protocol specifies that all the changes made into the same cache line by different caches need to be merged together. In order to achieve that, each piece of data, depending on the granularity, needs to be written by at most one core. If more than one cores need to write to the same data, then they need to synchronise, and only one core at the same time is allowed to write. In other words, VIPS-M requires explicitly synchronised data race free code. If the DRF criterion is met, then we are guaranteed to see a sequential consistent execution. That means that the VIPS-M protocol, and other protocols inspired by it, support and require the same DRF for SC memory model as some modern programming languages do.

\subsection{Forward Self-Invalidation/Self-Downgrade}

In the VIPS-M protocol described above, a cache self-invalidates all its shared data \textit{conservatively} when entering a critical section, in case some of them will be needed inside or after the critical section. Since it is invalidating data it has previously touched, this approach is called \textit{backward} self-invalidation. It is however the case that sometimes a critical section is only there to ensure the atomicity of all the accesses made, rather than provide synchronisation. An example is a counter variable, where we simply want to make sure that between reading the current value and updating it with the new one, no other thread modifies the counter. In comparison, a good example of critical sections causing ordering is a task queue, where the critical section contains only dequeuing a task out of the queue, but at the same time all the rest of the relevant task data need to be available after the critical section.

We can see that for the first case, invalidating all of the shared data in our cache is not required, and it can also be quite wasteful. As a matter of fact, only the data that \textit{will be touched} within the critical section need to be invalidated. It is thus possible, instead of conservatively invalidating all our data, to only invalidate data \textit{as needed} within the critical section. Practically, this can be achieved by having an extra self-invalidation bit in each cache line, which is set to true when a piece of data that has this bit set is touched, then the data is invalidated and refetched. The bit is then unset, as to not invalidate the data multiple times in the same critical section. Finally, upon exiting the critical section, all the bits are unset.

This protocol is called \textit{forward} self-invalidation, in contrast with the backward self-invalidation described above. Similar to self-invalidation, forward self-downgrade works by only downgrading data that were touched within the critical section, and not the whole of a thread’s shared data. Combined, forward self-invalidation and forward self-downgrade allow for less invalidations and downgrades, which has the potential of improving both performance and power consumption. However, lock acquire and release operations no longer provide synchronisation, as they did in the traditional self-invalidation/self-downgrade protocol.

Since locks no longer provide synchronisation, a new memory model is required. The model is called scoped release consistency, and is very similar to release consistency with the difference that locks can no longer be used to establish happened-before relationships for the data accesses surrounding them. If a critical section needs to establish a happens-before relationship, as in the task queue example mentioned above, then backward self-invalidation/self-downgrade is required. All the other synchronisation primitives, such as fences or barriers work in exactly the same way as before. Only conditional variables (Section 2.3.3) need to be slightly modified, as to ensure that backward self-invalidation happens even if we decide not to wait on the variable.
In addition to the traditional synchronisation primitives, which are discussed in detail later (Section 2.3), a new scoped memory fence is also introduced. This fence is comprised of two parts, one that starts a scoped region and one that ends it. Within a scoped region, loads and stores are treated as if they were within a critical section, and are thus forward self-invalidated and forward self-downgraded as needed. This scoped fence mechanism is called a forward self-invalidation/self-downgrade (FSID) fence.

## 2.3 Synchronisation Mechanisms

In order to establish happened-before relationships and avoid data races, we need to use the synchronisation mechanisms provided either by the software or the hardware we are writing our applications on. While there are many mechanisms to that end, we will discuss here some of the most basic ones. We will often refer to “critical sections”, which are parts of the execution path that at most one thread may execute at any given time, as described by Dijkstra [Dij01].

### 2.3.1 Barriers and Fences

There are two types of barriers, memory barriers (also known as fences) and thread barriers. We have already talked about memory barriers in section 2.1, so we will focus on thread barriers here.

Thread barriers work by forcing all the threads to stop until they have all reached a certain point in the program. A naive implementation would feature a counter that each thread increments when reaching the barrier, and a loop checking if that counter is equal to the number of threads. After all the threads have reached the barrier, the counter will have the correct value and all the threads will exit the loop and continue executing. Of course, the actual barrier implementations found on production systems are not as simplistic as that. Also, after exiting a barrier, usually a full memory fence is performed.

### 2.3.2 Locks and Semaphores

Locks are designed to solve the critical section problem and are described by Dijkstra [Dij01] in the same paper that he defines the term “critical sections”. Locks are also known as “mutexes”, short for “mutual exclusion”, since they only allow one thread at a time to proceed in the critical section. The way usually locks are used is by locking/acquiring the lock before entering the critical section and then unlocking/releasing it when exiting the critical section. As the naming of the operation implies, acquiring and releasing a lock triggers the equivalent memory operations as well, thus establishing happened-before relationships for all the threads entering and exiting the critical section. Combined with the fact that only one thread can access the data within the critical section at a time, locks prevent data races, as long as all the stores and loads of the shared data happen within them. The disadvantage of locks is that they serialise the execution of the critical sections, which can lead to a decrease in parallel performance.

While we describe the mutual exclusion locks, there are also different types of locks, such as reader-writer locks (shared locks) [CHP71]. These locks protect critical sections much like normal locks do, but allow multiple threads to enter the same critical section as long as none of the threads modify any of the shared data. Since multiple threads reading the same data, without any one of them modifying them, does not constitute a race, the program correctness is maintained. When a thread needs to change the data, then it needs to acquire the lock as a writer, in which case no other threads are allowed in the critical section until the lock is released. The obvious advantage of the reader-writer locks is that the reader threads are no longer serialised, which can lead to an increase in parallel performance. However, since
detecting data dependencies is not always easy to be done automatically, the work of assigning reader or writer privileges is left to the programmer, which increases the code complexity.

Another different type of locks are the Queue Delegation Locks (QDL), as described by Klafthenegger, Sagonas, and Winblad [KSW14]. These locks, as the name implies, delegate the execution of the critical section to the thread that acquires the lock. This has two advantages. First, the thread that failed to acquire the lock does not need to wait until the lock is available. Instead it can continue working on other tasks. Waiting is only necessary at the point where data from within the critical section are needed. The second advantage is that since all the threads entering a critical section operate on the same shared data, by having one thread execute all the critical sections at once, we improve data locality. This is especially advantageous in systems where data invalidation and movement might be very expensive, such as NUMA or Distributed Shared Memory systems [Kax+15].

Finally, we have semaphores [Dij02]. The semaphores’ usage more closely resembles that of signals, but their mechanism is much closer to locks. Semaphores work with an initial value and two operations, \( V \) and \( P \). \( V \) is similar to acquiring a lock and it decrements the value of the semaphore. However, if the value of the semaphore is 0, then \( V \) will wait until it is increased before decrementing it. \( P \) is the opposite operation and is similar to releasing a lock. We can see that the mutual exclusion locks are similar to a semaphore with initial value 1. However, it is more common for semaphores to be utilised as signals, as they are described in the next subsection.

### 2.3.3 Conditional Variables and Monitors

A conditional variable is a construct that allows a thread to wait until a certain condition is met. They are similar to semaphores in that they implement wait and signal semantics, but the concept and usage is different. Conditional variables are combined with locks in what is known as a “monitor”. Monitors were first introduced by Hoare [Hoa74], and nowadays are usually referred to just as “conditional variables”. After all, conditional variables are almost always combined with locks and a standalone implementation would make little sense. So, we will also refer to the monitor constructs as conditional variables.

Conditional variables work in the following way. First, the lock for the critical section is acquired. Then, the condition is checked. If the condition is not satisfied, then the lock is released and the thread is suspended. This frees the shared resources to be used by other threads. The suspended thread will now remain in sleep until some other thread signals the conditional variable. Usually, the programmers can choose to signal either just one thread or all of the threads that are waiting on a conditional variable, depending on the needs of the algorithm. After the thread wakes up, it then reacquires the lock and continues its work in the critical section.

One of the most common examples of conditional variables is the producer-consumer queue. A consumer that wants to remove an item from the queue, first acquires the lock and then checks if the queue is empty. If the queue is empty, then the consumer will sleep until a producer signals that the queue now contains some items. The alternative to the conditional variable would have been a spinloop, where the consumer constantly polls a memory location to check if the queue is empty. If that check was done without any synchronisation, it would lead to a data race and constitute undefined behaviour. At the same time, in the case where the spinloop was properly synchronised, the cost of constantly synchronising in a tight loop could impact the performance of the program significantly.

---

3 Also known as wait and signal

4 Conditional variables are usually susceptible to spurious wakeups, so the condition is usually rechecked after the thread wakes
2.3.4 Atomic Operations

An operation is “atomic” if it appears as if it was executed as one indivisible operation. Technically, a critical section within locks is an atomic operation, however, when talking about atomic operations, or “atomics”, we refer to hardware enforced atomicity. Specifically, the hardware guarantees us that the operation will be executed as if it was one instruction, without any other instructions interleaved during its execution. If that is not possible, then the atomic instruction should fail to execute and not cause any visible side-effects to the system. Practically, the definition of an atomic operation might differ between different hardware systems and programming languages and libraries. For example, in C++, there is an atomic type template that guarantees the atomicity of the operations, but it does not guarantee that the implementation will not just use locks to achieve that.

The advantage of atomic operations is their speed, when compared to locked critical sections. That, of course, assuming that special atomic instructions are available, as otherwise the implementation might use locks and lose any speed improvement. In modern x86 systems, there are atomic operations available at least for the integer (and in extension the pointer) types.

Regardless of the implementation details, there is a set a well known and commonly available atomics.

**Atomic Read and Atomic Write** operations make sure that there are no partial reads and writes, which, for example, is what happens if during a write another thread reads an incomplete variable. At the same time, an atomic read triggers a memory acquire operation, while an atomic load triggers a memory release.

**Test and Set** operations perform an atomic write and return the old value of the variable. This is done atomically, meaning that we are guaranteed to read the old value and set the new one before any other thread is able to modify it. Since these atomics both read and write to a memory location, they trigger both an acquire and a release operation.

**Compare and Swap** operations check if the variable has an expected value and if it does, then they replace it with a new one. They are similar to the test and set operation, but they incorporate a conditional check beforehand.

**Fetch and Add** operations are the atomic equivalent of the postfix increment operator (\texttt{var++}) in C-like languages. They issue both a release and an acquire operation.

2.3.5 Transactional Memory

In all the methods described above, the weight for protecting the shared data fell on the programmer. They had to determine which parts of the code needed locks or atomics, as well as try to minimise the performance cost associated with those mechanisms. Also, the focus was on the data rather than the program flow.

Transactional memory differs from that approach in two ways. First of all, the underlying system, be it the software or the hardware, has to make sure that the data are accessed in a data race free way. All the programmer has to do is designate which parts of the program should be run atomically, without having to worry about how the system is going to ensure that. There are various ways of achieving that \cite{Yen+07}, but they are beyond the scope of this thesis. Secondly, the focus is on the program flow rather than the data. The programmer does not have to explicitly state which data are shared or not, just which parts of the code should be executed transactionally.

We did not use transactional memory in our code, but it is still worth mentioning, since it is a re-emerging technology.
2.4 Software Tools

Detecting data races is not always an easy job. The races can appear in the program both on purpose and as the result of some programmer or even compiler induced bug. For that reason, specialised applications are used. We will discuss two of the most well known FOSS ones, as well the custom tool we used for detecting the races.

All of the tools we will present perform a dynamic analysis of the program at runtime. This has two disadvantages. First, the tools cannot detect races that might happen in code paths that were not executed. Also, if a race is detected or not depends on the different interleaving of instructions on each run. For that reason, such tools usually need to be run more than once. The second problem is that the execution time of the target program is affected, sometimes to the point of making running the tool for large workloads impractical. However, they are still faster (and easier to use) than offline model checking tools that try to check all possible interleavings.

2.4.1 Helgrind

Helgrind [Val14] is a tool in the Valgrind [NS07] suite. It detects errors dynamically by performing Just In Time (JIT) instrumentation of the target executable. All stores and loads are intercepted, as well as all synchronisation operations. Helgrind tries to establish happened-before relationships for all the loads and stores, based on the synchronisation it has observed, and if it fails to do so for some of them, they are reported as a potential race. Since it does not actually check the memory data, all the data races reported are only potential and they might have not actually happened.

Since Helgrind does not require the program to be compiled with any special options, it is very easy to use it in any project. Except from data races, it also detects other threading problems, such as deadlocks. All of these come at a cost of up to 100x running time overhead, at least according to the online documentation. Also, Helgrind is designed to work specifically with POSIX threads, and thus can only be productively used in programs that only use them. For example, in our case, Helgrind did not recognise the C11 atomic operations as synchronisation and reported a large number of false positives. However, due to the fact that it ran much faster than Fast&Furious, we used it as a preliminary checking tool for some of the versions we produced.

2.4.2 ThreadSanitizer

ThreadSanitizer (TSan) is a tool developed by Google. It started as part of the Valgrind suite [SI09; Thr13] but now it is available through the Clang and GCC compilers [Thr14]. It is similar to Helgrind in the sense that it tries to establish happened-before relationships between data accesses, based on the observed synchronisation calls. It does so by using the hybrid data race detection method described by O’Callahana and Choi [OC03]. Since instrumentation is done at compile time, access to the source code and recompilation is necessary. However, the runtime overhead is less than that of Helgrind, in the range of 2-20x, at least according to the online documentation. This relatively low overhead makes TSan practical to use even for larger workloads.

TSan works by using what is called “shadow memory”. For each piece of data accessed in the program, TSan stores which thread performed the access, a timestamp and some other metadata. How many of those metadata are stored depends on user specified parameters and affects the memory usage but also the precision of the program. Also, information about locks or other synchronisation is stored. On each memory access, TSan tries to see if happened-before relationships can be established for all the threads that have accessed that memory. If no such relationship is found, then the access is reported as a data race. The authors claim that the algorithm used has less false positives while detecting more races than Helgrind.

---

5Free and Open Source Software
In this project, we did not use TSan because our workloads were small enough for Helgrind to be practical. Also, TSan requires specific compilers and recompilation of the whole program, which requires extra effort. However, regardless of our specific use-case, it is a tool that has gathered a lot of attention and is actively used by large projects, such as the Chrome web browser.

2.4.3 Fast&Furious

Fast&Furious (F&F) \cite{RK15} is a tool built on top of the Pin Dynamic Binary Instrumentation Tool \cite{Luk+05}. It is similar to Helgrind in that it performs dynamic (JIT) instrumentation but the method for detecting data races is completely different. F&F is specifically designed to work with applications designed to run on systems that guarantee Sequential Consistency for code that is data race free (SC for DRF). It models the weak memory model of Release Consistency (Section 2.1.2).

F&F works by maintaining private software caches of unlimited size. Theoretically, all the data a thread loads should be fetched there on the first time they are accessed and all following loads should be subsequently performed from the cache. All stores should also be performed to the cache. Since the cache is of unlimited size, data invalidation is performed only during explicit synchronisation operations. This creates an effect similar to moving all reads of a thread above all the writes of the other threads. This is permitted because of the SC for DRF memory model used. In practice, while the software cache is fully maintained, its data are not used by the target application, only by the tool itself. If the data were used, the runtime behaviour of the application would be altered and, in many cases, the application would not run properly.

With the private software caches maintained, the tool compares the value of every read operation between the cached value and the actual read value. If there is a discrepancy between those two values, then a data race must have happened. Since the caches are flushed only during explicit synchronisation, implicit synchronisation as well as all the data it protects are seen simply as data races.

In addition to the release model that the Fast&Furious tool was written for, we also produced a modified version for the forward protocol (Section 2.2.3). The tool operates in the same way as before, only it uses forward self-invalidation/self-downgrade when locks are encountered. This was done quite simply by adding two boolean variables that mark which data have already been self-invalidated (for forward self-invalidation) within the current critical section and which data have been written to (for forward self-downgrade) in the current critical section. Other than that, we also added FSID fences to the tool. No other modifications where needed to make the tool work with the forward protocol.

The Fast&Furious tool has three disadvantages. First of all, with the current version, all synchronisation needs to be annotated as so; even well known primitives such as those belonging to POSIX pthreads. This means that the tool can not be used without editing and recompiling the code. The second disadvantage is that it can only detect races if a write actually happens before a read. If the write happens after a read, then the tool will not detect the discrepancy and thus it will not detect the data race. Finally, in our case, we observed a runtime overhead in the order of hundreds and thousands of times running time increase. This makes the tool impractical for any but the smallest of workloads. There is however room for optimisation, since, at the time that this project was undertaken, we are using an early first version of the tool.

However, the tool has some significant advantages over the other tools presented here. All the races detected, assuming that the code is annotated properly, are guaranteed to not be false positives. The tool checks the data values read from memory, so the only way for it to report a data race is if the values have been actually altered. This is very important, because data races are hard to detect and reason about, so confidence in the tool results is necessary. Also, due to the software architecture of
the tool, it was easy for us to modify it to fit our specific memory models. Finally, it is possible to integrate the tool with the tracer application we used for the final benchmarks, which ensures that the traces are produced from a correct (i.e. properly synchronised) program execution.
Using the F&F tool described in the introduction, we found data races in a number of applications. These include both implicit synchronisation and simple data races where the authors considered synchronisation unnecessary. Also, we got a number of races that can be considered as a form of false positives, since they occur because F&F does not recognise the implicit synchronisation used as synchronisation. Furthermore, some applications contain some potential bugs\footnote{Special thanks to Carl Leonardsson (Uppsala University - Information Technology Department) for identifying those bugs.} that are caused either by data races or by other race conditions; those were fixed as well.

Our initial plan was to remove all data races, but after some consideration, we decided to only remove races that are used for implicit synchronisation and any other races that might cause issues. We left in (for the most part) data races that we considered “benign”, that is to say data races where consistency was not strictly required. Such races are still an issue according to the memory models used, but practically they work. We decided to allow those races because our focus is on memory coherence and we did not want to affect the applications’ performance too much.

3.1 Synchronisation Methods

In order to eliminate the races we found, we used a number of different methods. This serves two purposes. First of all, not all systems support all of the methods we used. For example, not all hardware has support for atomic instructions. Secondly, we wanted to measure how different methods affect the performance characteristics of the applications. This should allow the benchmarking of different hardware designs that depend on those methods.

In both the mutual exclusion and the atomic operations versions, we removed any volatile qualifiers from the variables used for synchronisation. Not only making the synchronisation explicit makes the volatile qualifiers obsolete, but they are problematic to begin with\footnote{Special thanks to Carl Leonardsson (Uppsala University - Information Technology Department) for identifying those bugs.}. We only left them in the exposed version, for reasons explained below.

3.1.1 Mutual Exclusion

The first method we used (referred to as the “locks” version) is one of the simplest and most widely used ones. Different threads are prevented from accessing the same shared data at the same time, regardless of the type of operation they want to perform, be it a read or a write.

In order to implement mutual exclusion, we used locks and, when applicable, conditional variables. SPLASH-2 already uses locks for a lot of the synchronisation, and being one of the most basic primitives, locks are available on almost every system. Removing the races with locks is fairly simple, as all we had to do is identify which is
the appropriate lock to acquire for the given piece of data, and then simple add code
to lock and unlock it. In many cases, the lock was already acquired for writing to the
data, which meant that we only needed to lock the read operations.

However, there are some cases where a variable is polled constantly, while the
thread is waiting for it to take some specific value. One example is the Cholesky code
in listing 3.3 line 171. We can imagine that constantly acquiring and releasing a lock
there would cause both contention on the lock and unnecessary synchronisation. By
analysing the code, we determined that what the programmer wanted to accomplish
for the consumer thread (since we have a queue) is to wait for a producer thread.
This is a classic usage case for conditional variables, so we replaced the loop with
one conditional variable. Now, if one thread enqueues something in the task queue,
it signals one of the threads waiting for data to wake up and check the queue again.
Practically, since conditional variables can cause a thread to wake up spuriously, the
loop is still in place, but now the threads waiting spend most of the time in sleep,
instead of constantly polling the shared variable. For the scoped release model, we
made sure that on every dequeue operation, either a conditional wait or an acquire
fence is issued, as otherwise all the data that has been modified outside the critical
section would not be made visible to the dequeuing thread. This was done for all the
applications we introduced conditional variables to and not just for Cholesky.

Finally, there are some cases where removing the race makes more sense than
adding extra locks. Specifically, the double check locks. Since the sole reason of using
a DCL is to avoid acquiring a lock unnecessarily, locking the first check completely
defeat the purpose. So, instead of synchronising those racy accesses, we simply re-
moved them completely, or when complete removal was not appropriate, we moved
the first check into the same critical section as the second one.

3.1.2 Atomic Operations
The second method (the “atomics” version) involves turning all the racy accesses
into atomic accesses. Many modern hardware support atomic operations for integer
and pointer types, which often means that they are faster than locks. While some
compilers, such as GCC, support atomic intrinsics, we decided to use the atomic types
found in C11. These are better documented, and work on any standard compliant
compiler.

In many cases, the races present used locks for writing to the shared data, but
not for reading. In those cases, we only converted the reads to atomic operations,
since the writes were already protected by locks. If that was not the case, then we
converted both accesses. We also substituted some locks for atomic operations, for
example when the lock was acquired just to increase an integer value and nothing else
was done in the critical section.

Other than those changes, we did not change the synchronisation much. Unlike
when we used locks, we can not substitute spinloops for conditional variables, since
those require locks. Also, presuming that atomic operations are faster than acquiring
and releasing a lock, we also left the DCL in place, making the first check atomic.

3.1.3 Exposing the Synchronisation to the Hardware
In this version (the “exposed” one), we did not actually remove the races. Instead,
we marked them, using the same special “magic” functions that we use in the F&F
tool, and let the hardware deal with them. Of course, at this point, since modern
consumer hardware does not actually handle this kind of synchronisation, only the
simulator is able to recognise the races and handle them appropriately. However, the
reason why we chose to do this exposed version, is exactly so that is can be used with
the simulator. By abstracting the synchronisation, and letting the simulator know
what kind of synchronisation it is, we can simulate it in many different ways. For
example, the **SPIN WHILE** that is described below, can be simulated either as a **while** spinloop, or as a conditional variable that puts the threads to sleep while waiting.

What we did is introduce “magic” functions (and their corresponding m4 macros — Appendix B) for the implicit synchronisation concepts that we found in the applications. Unfortunately, some of them are very specialised to the application, so there is a limit to how much we abstracted them.

**SPIN WHILE** marks a spinloop on some variables(s), waiting for them to take a specific value. Essentially, this is what was replaced with conditional variables in the mutual exclusion version. The “magic” functions include all the information we need to simulate the wait, such as the address(es) accessed and the conditions for exiting the loop.

**DCL IF** marks a double checked loop. It includes information such as the address(es) and the condition, but also the lock that is going to be acquired if the check succeeds. For simplicity, the programmer is still responsible for performing the actual locking and unlocking.

**DO WHILE** is not really a synchronisation concept, but it hard to categorise each of them separately. It marks loops that do not just spin on a variable but also perform some useful work. It might be something similar to a DCL, or a barrier that does work stealing while waiting.

All of the above macros should be accompanied by matching **DO STORE**, which mark the stores on the shared data. Unlike the atomics version, we need to mark all the stores, even the ones that are already in the critical section. The simulator needs all the stores and their values to know how the threads should progress.

### 3.1.4 FSID Fences

For the special case of the forward self-invalidation/self-downgrade model (the “fsid” version), we introduced additional forward self-invalidation/self-downgrade (FSID) fences (Section 2.2.3) on top of the locks version. Specifically, we added FSID fences around the data races where we could not identify some primitive that performs backward self-invalidation/self-downgrade close to the accesses in question. If such a primitive was identified, then we considered the additional fences to be an unnecessary overhead and we did not add them. This was done on top of the additional synchronisation described in the previous sections, and specifically, on top of additional locks. Since the difference between the backward and the forward self-invalidation/self-downgrade protocols lies in how locks are handled, we did not produce an atomics version for the forward protocol.

From all the applications we found to contain data races, only three of them required any additional FSID fences: Barnes, FMM, and Radiosity.

### 3.2 The SPLASH-2 Applications

The SPLASH-2 benchmark suite is a collection of parallel applications used to measure various performance characteristics of shared memory multiprocessors [Woo+95]. The suite includes more applications than the ones we are going to discuss, but not all of them were found to contain data races.

Note that for some data races we give precise source location in the form of line numbers. Since our version of the source is modified, the line numbers might not match the ones in the original code. However, it should be possible to identify which line we are referring too, as the line numbers should be very close to the ones in the original version.
if (qptr == NULL) {
    /* lock the parent cell */
    ALOCK(CellLock->CL, (cellptr) mynode)->seqnum % MAXLOCK);
    if (qptr == NULL) {
        // Create a new leaf at qptr and
        // add it to the parent node (mynode)
    }
}

if (qptr == NULL) {
    // Create a new leaf at qptr and
    // add it to the parent node (mynode)
    AULOCK(CellLock->CL, (cellptr) mynode)->seqnum % MAXLOCK);
}

if (flag && qptr && (Type(qptr) == LEAF)) {
    /* reached a "leaf"? */
    ALOCK(CellLock->CL, (cellptr) mynode)->seqnum % MAXLOCK);
    if (Type(qptr) == LEAF) {
        /* still a "leaf"? */
        // Add bodies to the leaf
    }
}

if (r->done)
    /* unlock the parent cell */
}

Listing 3.1: The double check locks in the Barnes application (load.C).

### 3.2.1 Barnes

The Barnes application calculates the gravitational interactions between particles in a three-dimensional space. As the name implies, it uses the Barnes-Hut simulation algorithm, as described by Barnes and Hut [BH86]. In order to parallelise the workload, both building the octotree and calculating the forces is split among multiple threads. This is done by making each thread responsible for a number of particles. Which particles each thread is responsible for is decided depending on the cost, in number of calculations, associated with calculating the forces applied to each particle. The algorithm tries to assign equal amounts of total cost to all the threads, thus improving the work balance.

The Barnes application contains a large number of races, not all of which are used for synchronisation. Some of them appear to be there in order to avoid the extra cost of synchronisation for operations that do not need to be synchronised. This however causes some problems, including a bug, which we will discuss later.

First of all, we begin with the races that are used as implicit synchronisation mechanisms. These happen in the load.C file, which handles the octotree. The first two of these races are double check locks and can be seen in Figure 3.1. Specifically, the code checks the value of the *qptr variable before locking to avoid acquiring the lock if no operations on it needs to be done. This happens in both lines 248 and 258.

The second implicit synchronisation data race happens in the same file, at line 421 (Listing A.1). There, the value of the r->done variable is checked in a while loop, spinning until it is set to true. Different places in the code can change that value to either true or false, and multiple threads might do so as well. The done variable itself signifies, for the current step the program is on, if the center of mass of the particle/node has been calculated. This signals to the other threads that they can use the value in any calculations they need. It can be easily seen that what the author(s) of the code intended was a conditional variable, so in the locked implementation we replaced the spinloop with a conditional variable. Removing this data race also removes some other races. Before, since Fast&Furious was not aware of the implicit synchronisation, it detected data races when reading the center of mass of a particle or node that another thread had set. Now, the happened-before
relationship has been established by making the synchronisation mechanism explicit.

As we have already mentioned, not all of the data races in the code are used as implicit synchronisation mechanisms. Some of them are there as optimisations, in order to avoid unnecessary synchronisation when not needed.

The first of those races happens in the return statement of the function loadtree in the load.c file (Listing A.2). There the \*qptr node is accessed, in order to ascertain its parent, without acquiring a lock first. The obvious solution would be of course to acquire the appropriate lock before accessing the variable, but that is not actually necessary. We notice that the qptr variable is going to be modified in the while loop just above the return statement. We also notice that the program is only going to exit the loop if the local variable flag is set to false. This is done only within critical sections which have already been acquired for modifying \*qptr. By combining this knowledge, we come to the solution that we just need to access the data while in the critical section, store them locally, and then just return them at the end of the function. After exiting the critical sections, we were not guaranteed to see any changes that another thread might have made to these data anyway. Now, since outside the critical section we only access the locally stored data, the data race is eliminated.

The second data race happens in the files grav.c and code.c (Listing A.3). It happens while reading the position of a particle (Pos(p), lines 38 and 71 in grav.c) in order to calculate the force that it applies to the particle the thread is currently working with. This position is set by the thread responsible for the particle (line 601 in code.c), after having calculated the forces applied to it. Since there is no synchronisation between those two steps, a race can occur. To remove this race, we would need to lock the particle, but that would require too fined grained locking. Instead, we could lock the whole leaf. Within the critical section, we access multiple particles, so the coarser locking also saves us from releasing and acquiring the same lock multiple times consecutively. However, since this race is not in the implicit synchronisation category, we did not remove it.

The third of the data races happens in file code.c, at lines 661, 729, and 743, as well grav.c line 50 (Listing A.4). The cost (Cost(p)) for calculating the forces applied to the particle or node is set and read by different threads without any synchronisation. The solution would be to either lock the nodes or use atomic operations while accessing the cost. As with the other non-synchronisation races, we chose to not remove it.

Finally, we discovered a race that was not detected by Fast&Furious. This race happens in the file code.c, in the SlaveStart function. There, after having initialised the values for process 0, we process to read them and set the values for the other processes as well. However, process 0 also reads and resets those values at the same time. The reason Fast&Furious did not detect the race is because the values written back are the same as the ones read, so no data discrepancy is detected. As a matter of fact, it is possible to argue that no data race is actually happening here.

We mentioned that some of those data races can cause bugs, and some of them are very subtle. We detected one such bug in this application. In load.c, function loadtree, line 273 (Listing A.2), the SubdivideLeaf function is called and the result is stored to \*qptr. SubdivideLeaf creates a new node, sets the current leaf as its first child and then returns that new node. Returning the new node, and setting \*qptr to it is what publishes the change made to the other threads. If however, setting the children and publishing the changes were done in the reverse order, it is possible that some other node might go and set the first child before the current thread. Then the current thread would override that value, causing a leaf to be lost. Such a reordering is not possible to happen in x86, since stores are ordered in program order. However, a compiler could perform this reordering as an optimisation, since we do not indicate that the order is important. As we explain in the introduction, this constitutes undefined behaviour in the latest C and C++ standards. In order to solve this bug, we just need to make sure that all the operations that SubdivideLeaf performs are completed and visible to everyone before publishing the changes by setting \*qptr. This is done by placing a release barrier between those two operations, also known as
an `sfence` instruction on x86. However, the C11 standard includes a memory fence function, so, in the atomics implementation, we preferred to use that one.

After finishing the conversion for the release model, we investigated the inconsistencies found for the scoped release model. We found that the data accesses reported for the scoped release model are the ones we decided not to fix for the release model. Since we could not detect any backward synchronisation close to those data accesses, we decided to enclose them in FSID fences. We tried to make the FSID blocks as coarse as possible, since if they were too fine grained, they could cause more invalidations of the same data that is accessed multiple times.

### 3.2.2 Cholesky

The Cholesky application calculates the Cholesky Decomposition; that is the decomposition of a matrix into the product of a lower triangular matrix and its transpose. The SPLASH-2 implementation works with sparse matrices and utilises a blocked approach for performing the calculations.

The way parallelism is achieved is by distributing the blocks that need to be operated on to different threads, by means of a task queue. Each thread has its own task queue and the work that the thread needs to do is placed there. Since different threads might try to access a thread’s task queue simultaneously, the `push` operation is protected by a lock. The same goes for the `pop` operation, of course. However, the application implements a technique known as “double checked locking”, where a condition is checked outside the critical section, and if the condition is met, then the lock is acquired and the condition is checked again. This technique is employed in order to avoid acquiring a lock unnecessarily, since if the condition is not met in the first place, the data in the critical section are not going to be manipulated. In the Cholesky application, this condition is if the task queue contains any tasks. Of course, if the queue is empty, no `pop` operation is going to be performed to it. Unfortunately, this double checked locking causes a data race in the application. A thread that calls the `push` operation on the queue will have acquired the lock, but another thread might try to read the same variable without acquiring the lock first.

Specifically, there are two functions that manipulate the queue directly, `Send` and `GetBlock` (file `mf.C`). `Send` performs the `push` operation (Listing 3.2) while `GetBlock` performs the `pop` operation (Listing 3.3). While `Send` acquires a block before doing any operations on the queue, `GetBlock` first checks if there are any tasks, by checking if the point to the head of the queue is `NULL` or not (lines 151 and 171 in listing 3.3).
for (; ;)
{
    if (tasks[MyNum].taskQ || tasks[MyNum].probeQ) {
        LOCK(tasks[MyNum].taskLock);
        t = NULL;
        if (tasks[MyNum].probeQ) {
            t = (struct Task *) tasks[MyNum].probeQ;
            tasks[MyNum].probeQ = t->next;
            if (!t->next)
                tasks[MyNum].probeQlast = NULL;
        }
        else if (tasks[MyNum].taskQ) {
            t = (struct Task *) tasks[MyNum].taskQ;
            tasks[MyNum].taskQ = t->next;
            if (!t->next)
                tasks[MyNum].taskQlast = NULL;
        }
        UNLOCK(tasks[MyNum].taskLock);
        if (t)
            break;
    }
    else {
        while (!(tasks[MyNum].taskQ && !tasks[MyNum].probeQ));
    }
}

Listing 3.3: The code getting the tasks from the queue in the Cholesky application (mf.C).

If it is not NULL, then the lock is acquired, the head is checked again and then it is popped. If it is NULL, then the function will spin in a loop, constantly checking the head, until a task if found.

The solution to this race is rather simple. All we have to do is remove the double check and check only inside the critical region. This however is not efficient, since the same lock might be acquired and released multiple times before there is an actual task in the queue. A much better solution would be to use some signalling mechanism, such as conditional variables. GetBlock acquires the lock and checks if the queue is empty. If it is, then the lock is released and GetBlocks sleeps until it is waken. When it is waken, the lock is acquired again automatically and the queue is checked again. If this time there is a task, then the pop operation is performed, otherwise GetBlocks is set to sleep again. Send, on the other hand, works exactly the same as before, with one minor change. Before releasing the lock it holds, it also signals the conditional variable, causing GetBlock to wake up and check for new tasks.

There is of course another solution. It is possible to switch the head of the queue with an atomic variable and do the checks using atomic operations. Since atomic operations act as synchronisation mechanisms, the race is avoided. However, the lock does need to be acquired for performing the pop operation, so atomics might not be the best solution in this application.

3.2.3 FMM

FMM stands for Fast Multipole Method, which, much like the Barnes-Hut algorithm, is used to solve the n-body problem. FMM partitions the space using trees like the Barnes-Hut algorithm, but then the forces are calculated using the Fast Multipole Method, yielding $O(N)$ runtime complexity, as shown by Carrier, Greengard, and Rokhlin [CGR88].

While running the FMM application in the F&F tool, a significant amount of data races appeared. However, some of them only happened because implicit synchronisation was used. Thus, by making the synchronisation explicit, those races disappeared.
as well. In total, we identified four real races in the code. Two of them are used as implicit synchronisation mechanisms, while the other two are used to avoid unnecessary synchronisation, as thought by the original developers of the application.

The first two races occur when accessing either the `box::interaction_synch` or the `box::construct_synch` variables. Even from their names, it is fairly easy to deduce that the first one is used to signal when the interactions between particles in a node have been calculated, while the second one signals that a node is constructed and ready to use. The races are easy to find and solve, since they either reset or read the two variables without acquiring any locks. Of course, incrementing (i.e. signalling) the variables is done within locked critical sections, as the increment operation is not atomic in itself.

The third race occurs during accesses of the `box::children` variable. It would appear that the algorithm maintains correctness even if it does not always have the latest children of a node, presumably because it is correct to access the children from the previous iteration. Also, the children are removed and inserted in a way that maintains the atomicity of the operations, which means that it is not possible to read an invalid state. Finally, the fourth race, occurs when accessing the `box::*_expansion` variables. It appears to be a similar case to the one just described.

The solution for the races mentioned above is fairly simple. All we have to do is make sure the accesses happen either within locked critical section or using atomic variables. However, for the first two races, it is also possible to substitute the spin-loops with conditional variables, which usually are implemented in a more optimal way. Since we decided to not remove any races that are not used for implicit synchronisation, those two were the only races we actually removed.

As for the scoped release model, we found that the data races mentioned above remain, similar to Barnes. Since once again we could not find any backward synchronisation primitives close to those races, we decided to enclose them in FSID fences. However, our pintool also detected some data races unique to the scoped release model. These revolve around the creation of the octotree boxes and the particles they contain. Some, but not all, of those accesses are synchronised through conditional variables, so the FSID fences are not required all of the time. However, determining if and when that is going to be the case, without excellent knowledge of how exactly the application works and handles synchronisation, is practically impossible. The critical path of the execution is fairly convoluted, and most importantly, it relies quite a bit on transient synchronisation over multiple threads and function calls. For those reasons, we decided to choose the “safe road” and add FSID fences around all those accesses. This of course can have a negative effect on performance, but we prefer to err on the side of excessive synchronisation rather than program correctness.

This showcases one of the disadvantages of using the scoped release model. It is hard to convert applications that rely a lot on stricter memory models, especially when the programmer is not completely familiar with the application itself. At the same time, we took a more difficult approach to solving the problem. Normally, one would start by simply having all the locks provide backward self-invalidation and self-downgrade and then optimise the locks that are not used for synchronisation with the forward protocol. But again, that requires excellent knowledge of how the application works and how synchronisation is performed. Also, this would be best done over a number of rounds of modifying and then testing the application, which we can not do since there is not any hardware (that we know of) that supports the forward protocol, and the simulator is not suitable for such a task.

### 3.2.4 Ocean

The Ocean application simulates ocean movement based on currents. There are two different versions, the contiguous and then non-contiguous one. The first one tries to partition and store the data locally to each processor, at the expense of increased code complexity. The latter does not.
if (procid == xprocs-1) {
    sintemp = pi *((double)jmm1) * res / yscal;
    sintemp = sin(sintemp);
    frcng->tauz [0][jm-1] = factor * sintemp;
  }
if (procid == nprocs-1) {
    sintemp = pi *((double)jmm1) * res / yscal;
    sintemp = sin(sintemp);
    frcng->tauz [im-1][jm-1] = frcng->tauz [0][jm-1];
  }

Listing 3.4: The code that causes the data race in the Ocean application (slave1.C).

How parallelism is implemented in this application is not important, since the data race that we discovered (in the slave1.C file) is not used for synchronisation. While calculating various values needed for the simulation, there is a chance of a processor reading a value (line 449 in listing 3.4) that another processor has already calculated (line 444 in listing 3.4) instead of calculating the value locally. This data race only appears in the non-contiguous version the program, as the contiguous one chooses to calculate the value locally instead of reading it from another processor’s data. We do not know the reason for this choice, since all it is required for calculating the value is a simple multiplication operation.

The solution, thus, to this data race is to simply remove the remote access and do the multiplication mentioned. Correctness was verified by making sure the output remains the same. Also, correctness should not be affected, since the contiguous implementation does exactly the same thing. Even though we normally leave the non-synchronisation races in the code, this one was too simple to fix, it does not introduce any extra explicit synchronisation, and it is inconsistent with the contiguous version of the application, so we chose to remove it.

3.2.5 Radiosity

The Radiosity applications calculates the global illumination of a scene, using the radiosity algorithm. The scene is split into patches, which are subdivided into smaller patches, and it is worked on until the required accuracy is reached. Parallelism is achieved through task queues. Each node has one queue, and when finished, job stealing is implemented for load balancing.

Let’s first start with the data races that are used for synchronisation. The application has four different implicit synchronisations. First of all, there is a custom barrier implementation (file taskman.C). The barrier is implemented with a counter variable that indicates how many threads have reached it. If the counter has a value equal to the total number of threads, then the thread is allowed to exit it. While manipulating the counter is done safely within critical sections, checking its value is done without acquiring any locks, as seen in line 119 of listing 3.5. This of course causes a data race. The solution is simple, we either have to lock before checking the value or make the accesses atomic. We can not replace the custom barrier with a system implementation, such as pthread_barrier because while waiting at the barrier, the thread might exit the barrier if it finds any new tasks to work on.

Finally, there also are two double check locks, testing the amount of tasks and free space in a task queue before acquiring the queue’s lock (line 446 – if(tq->ntasks>0), and 523 – if(tq->n_free>0) respectively, Listing A.5). Much like all the other DCLs, we can eliminate the races by either removing the check or making the accesses atomic. We need to pay extra attention to the second one, as some more operations are executed in the if block.

Except from the races used as implicit synchronisation mechanisms, there is also a data race that happens because the author(s) of the code decided to try and avoid unnecessary synchronisation. When calculating the error for each pass, the RGB
/* Barrier. While waiting for other processors to finish, poll the
task queues and resume processing if there is any task */

LOCK(global->pbar_lock);
/* Reset the barrier counter if not initialised */
if (global->pbar_count >= n_processors )
    global->pbar_count = 0 ;
/* Increment the counter */
global->pbar_count++ ;
UNLOCK(global->pbar_lock);
/* barrier spin-wait loop */
while (global->pbar_count < n_processors )
{
    /* Wait for a while and then retry dequeue */
    if ( _process_task_wait_loop() )
        break ;
    /* Waited for a while but other processors are still running. 
Poll the task queue again */
    t = DEQUEUE_TASK( taskqueue_id[process_id] , QUEUES_VISITED,
                      process_id ) ;
    if ( t )
    {
        /* Task found. Exit the barrier and work on it */
        LOCK(global->pbar_lock);
        global->pbar_count-- ;
        UNLOCK(global->pbar_lock);
        goto retry_entry ;
    }
}
BARRIER(global->barrier , n_processors);

Listing 3.5: The custom barrier in the Radiosity application (taskman.C)

values of the patch are needed and are read without acquiring any locks. However,
some other thread might be updating those RGB values at the same time.
Since the error is monotonically decreasing with each pass, the author(s) of the code
decided that calculating a higher error value was cheaper than synchronizing the
threads. To be specific, the loads happen at lines 462-4 and 916-8 in file
elemman.C (inter->destination->rad.\{r,g,b\}), while the stores happen at lines 790-2 and
844-6 in the same file (e->rad.\{r,g,b\}, Listing A.6). We could choose to eliminate
them by either introducing proper locking or making the accesses atomic, but we did
not.

The second race happens in the same file, when the taskq_too_long macro is
called. The macro accesses the number of tasks the queue contains without acquiring
the queue lock first. This is done to avoid extra synchronisation operations that are
not needed, but it causes a data race. Much like all other similar data races, we would
just have to acquire the proper lock or make the accesses atomic. However, since the
race is not used for synchronisation, we left it in the code.

We also found a bug in the application, which can happen in the barrier code
described above. At line 127 (Listing 3.5), the application tries to dequeue a task
for the queue and if that was successful, it exits the barrier to work with it (line
132). However, the dequeue operation and exiting the barrier is not done atomically,
which means that it is possible to get a task from the queue and before exiting the
barrier, all the other threads reach it and go through. Now all the other threads are
finished with their work and are just waiting at the final barrier, while the current
thread is left to process all the new tasks that might be generated from that last task.
Attempting to move exiting of the barrier within the critical region of the dequeue
First, try to get job from pid’s own pool (or pool 0). */

if (gm->wpstat[i][0] == WPS_VALID)
{
    if (GetJob(job, i) == WPS_VALID)
    {
        return (WPS_VALID);
    }
}

/*
 * If that failed, try to get job from another pid’s work pool.
 */
i = (pid + 1) % gm->nprocs;
while (i != pid)
{
    if (gm->wpstat[i][0] == WPS_VALID)
    {
        if (GetJob(job, i) == WPS_VALID)
        {
            return (WPS_VALID);
        }
    }
    i = (i + 1) % gm->nprocs;
}

Listing 3.6: The code that implements the double check locking in the Raytrace application (workpool.C).

operation causes a deadlock, so the solution we came up with is to exit the barrier, check the queue and then reenter the barrier if no tasks were found. This of course requires acquiring the barrier lock one extra time, but it does solve our issue.

Finally, after all the changes mentioned above, we had to modify the application for the scoped release model as well. Unfortunately, Radiosity turned out to be implemented in a way that is very counter-productive for the scoped model.

To begin with, Radiosity uses a task queue. In order for the program to make sense, we added backward fences in the queue, since except from the task itself no other data would have been made visible. However, even if now the queue enforces correct synchronisation, we discovered that it is possible for it to be bypassed. Specifically, for some kinds of tasks, when a thread creates a task, it is possible to execute it itself without going through the queue. This is done if the queue is found to be too loaded or if the task is considered too small. This means that any synchronisation guaranteed by the queue is now gone. Because we did not want to alter the applications drastically, by disallowing bypassing the queue, we had to add explicit FSID fences around a large number of data, on top of all the other added synchronisation. And of course, we also needed to add FSID fences around the races found for the release model, since we can not guarantee that the data will be invalidated and downgraded in a timely manner.

Due to all the additional synchronisation needed, we expect to see a performance hit during program execution. At the same time, modifying the application was not an easy task. It is for these reasons that we think modifying Radiosity for the scoped release protocol is counter-productive.

3.2.6 Raytrace
The Raytrace application performs ray tracing on a scene.

The scene is split into blocks, which are then distributed to all the processors by using workpools/queues. Each processor has its own queue, but job stealing is implemented, so it is possible for a processor to access another processor’s queue. For this reason the pop operations need to be protected by locks. At the same time, all
the push operations are done before all the processors start working on them, so each processor only inserts jobs in its own queue. So the push operations do not need to be protected or be synchronised in any way. The application contains two races, which happen because double checked locking is implemented before getting a job from a queue. In order to avoid extraneous locks, a queue is first checked outside the critical section and the pop operation is executed only if the queue is found not empty. Those checks however constitute a data race, since while checking, some other processor might be popping from the same queue.

Essentially, in the workpool.C file, there is the GetJobs function, which tries to find a job for the processor to execute. First, the processors local queue is checked, and if it contains a job, then it is locked and the job is acquired. Otherwise, the function tries to steal a job from another processor, by iterating over all the other queues and performing the same operation. The race happens when the variable gm->wpstat[i][0] is read (lines 211 and 227 in listing 3.6). This variable stores the state of the queue, as either WPS_VALID or WPS_EMPTY, with the latter being set by the processor that owns the queue when the queue is depleted. Since all the push operations have already been performed beforehand, once the WPS_EMPTY state is set, it will not revert back, as there are no more jobs to insert into the queue.

One solution would be to simply check the status of queue after acquiring a lock, but that would defeat the purpose of double check locking. So, the best solution, would be to either remove the double check locking completely, or make the state variable atomic. In the first case, we also simplify the code, since there is no more need for the state variable. On the other hand, atomic accesses are usually much faster than acquiring a lock, thus allowing the double check lock to remain efficient, while avoiding data races.

### 3.2.7 Volrend

The Volrend application performs volume rendering using ray casting. The application comes with a command line switch which chooses between adaptive and non-adaptive sampling, however the adaptive method does not work, so we only tested the non-adaptive one.

Parallelism is achieved through job queues; job stealing is also employed for load balancing. Each processor is assigned a number of pixels to ray trace. If the processor finishes all of its local jobs, it then proceeds to check the job queues of the other processors and steal any jobs it may find. There is also a “global queue” which indicates if there is any processor at all that has any jobs left. It is used to prevent processors from trying to find new jobs when there are none, and also signals the end of the processing loop. The races happen both when checking this “global queue” and when trying to find other jobs to steal, since both checks happen outside a critical region.

Specifically, the races happen in the file adaptive.C, in the function named RayTrace\_Non\_Adaptively. The local\_node variable represents which processor’s queue (Global->Queue[local\_node]) the thread is currently getting jobs from, while
the `Global->Queue[num_nodes]` is the “global queue” variable. The function will read jobs from the current `local_node` until there are no more jobs left. This is accomplished in a thread safe way by doing all operations inside critical sections. After the current queue has been exhausted, the thread will try to find a new queue to work on. This is done by going over all the queues and checking if they contain any jobs, until one that does is found. This check is done in line 370 (Listing 3.7) and since it is done without any synchronisation, it constitutes a data race. Similarly, checking if all the queues are empty is done in two places, both in while loops, in lines 338 and 370, and are both done without any form of synchronisation, causing two more data races. All other accesses that work as synchronisation mechanisms, i.e. the `pop` operations for both queues, are performed only after a lock has been acquired.

This application also contains a bug. Each processor’s queue is initially at the beginning of the function, before processing begins (line 337 in listing 3.7). It is possible, however unlikely, for a processor to finish its queue and then start working on another processor’s queue, which has not yet been initialised. This is not just a data race used for synchronisation but an error in the program. Thankfully, the solution is very easy because before calling the `Raytrace_Non_Adaptively` function, all the processors are synchronised with a barrier. So, fixing the bug involves only moving the initialisation before the barrier call.

The solution for the other races mentioned above is to enclose all the checks into locks as well. Since they are used as the predicates for while loops, the code can become a bit awkward. Also, given that both the checks and the `pop` operations are simple increment and decrement operations, using locks is perhaps a too heavyweight solution. As a matter of fact, all of the locks in the job queue can be replaced by atomic operations, eliminating the need for locking in general.

### 3.3 The PARSEC Applications

The PARSEC suite [Bie11] is another benchmark suite designed for multiprocessor systems. It is much more modern than SPLASH-2 and it focuses not just on scientific but also on business and other applications. It supports pthreads, as well as Intel’s TBB and OpenMP parallelism. Except for the Intel x86 architecture, it also supports Intel’s Itanium, and the Solaris Sparc architectures. This means that PARSEC is written with release memory models in mind.

From all the PARSEC applications, we only chose to investigate some of them. Specifically, we ignored any applications that used anything other than pthread parallelism. We also ignore the Raytrace application because we have Raytrace in SPLASH-2, the Facesim application because it requires too much memory and the execution traces are too big for simulation, and also the Vips, and x264 applications. In the end, we decided to use the Blackscholes, Ferret, Fluidanimate, Swaptions, Dedup, Bodytrack, Canneal, and Streamcluster applications.

Since, as we mentioned earlier, PARSEC is written for release consistency architectures, it does not contain any data races used as implicit synchronisation mechanisms. As a matter of fact, we found most of the applications to be completely data race free, with the exception of Streamcluster.

#### 3.3.1 Streamcluster

The Streamcluster application solves the online k-means clustering problem. Our pintool detected a number of races around two different cases.

The first case is some of the points being accessed without any mutual exclusion. We presume that this is happening because this is a “benign” data race that does not affect the correctness of the program. Very close to all those accesses, barriers can be found, which means that soon after they are touched, the data are then synchronised. For that reason, and since the accesses are not used for synchronisation, we did not add any locks or atomic instructions.
Listing 3.8: The code where the data race around $\text{gl\_cost\_of\_opening\_x}$ happens in Streamcluster (streamcluster.cpp).

The second case happens when the static variable $\text{gl\_cost\_of\_opening\_x}$ is accessed (in the $\text{pgain}$ function), also without any synchronisation. The race happens when a thread tries to read this variable as part of a condition check (line 1150 listing 3.8) after thread 0 has already written to it during the same iteration (line 1178). Between iterations, these accesses are synchronised between two barriers (lines 1145, 1182), but not for the duration of each individual iteration. The reason why this race does not matter is because thread 0 will only try to set this variable if the condition check fails, and the value that it will set it to also does not satisfy the condition. So, regardless of the write being made visible before the barriers or not, the program behaviour is going to remain exactly the same. For those reasons, we decided to avoid adding any extra synchronisation for this accesses as well.

Since we ended up not modifying Streamcluster, or any other of the PARSEC applications that we considered, we will not include them in the evaluation section.
Except from modifying the applications in order to be able to use them with more relaxed memory models, we are also interested in how our modifications affect the performance characteristics of those applications. There are of course too much data that can be collected on the subject, so we decided to focus on basic coherence related data. We measure the impact our modifications have on the misses in the L1 data cache, the execution time, and the network traffic. We also measure the number of locks, barriers, atomic instructions, conditional variables, and FSID fences each benchmark uses.

Since we are not interested in the different coherence protocols but in the modification we made to the applications, we only benchmarked the applications we modified. This means that we will only include results for Cholesky, Barnes, FMM, Ocean (non-contiguous), Radiosity, Raytrace, and Volrend. We will not include any of the PARSEC applications, since we did not modify any of them. For the SPLASH-2 applications included, we used the recommended input parameters [Woo+95] (more information can be found in Appendix C). We ran each application through both the MESI (Section 2.2.1) and the VIPS-M (SISD) (Section 2.2.2) protocol. Of course, this included all the different versions (Section 3.1). We did not however ran the unmodified version though the VIPS-M protocol, since the results would not be correct. Also, we ran the three FSID modified applications only through the FSISD protocol, since they are a special case and FSID fences are not recognised by any of the other protocols.

In order to collect accurate measurements of the runtime characteristics of our applications, we decided to forego actual runtime measurements and do simulation instead. Especially for SPLASH-2, the inputs are too small to be able to accurately measure without simulation. We use the Wisconsin GEMS [Mar+05] simulator, combined with GARNET network simulator [Aga+09]. We used the network traffic information provided by GARNET to estimate the changes in power consumption, since a) network traffic is also correlated with cache accesses and misses and b) both of those metrics are what affects performance the most when it comes to changes in the memory coherence behaviour. We fed the simulator with traces collected from

<table>
<thead>
<tr>
<th>Application</th>
<th>Unmodified</th>
<th>Locks (Normalised)</th>
<th>Atomics (Normalised)</th>
</tr>
</thead>
<tbody>
<tr>
<td>barnes</td>
<td>34502</td>
<td>387308 (11.23)</td>
<td>34486 (1.00)</td>
</tr>
<tr>
<td>cholesky</td>
<td>26124</td>
<td>26126 (1.00)</td>
<td>26124 (1.00)</td>
</tr>
<tr>
<td>fmm</td>
<td>27794</td>
<td>124997 (4.50)</td>
<td>27840 (1.00)</td>
</tr>
<tr>
<td>oceannc</td>
<td>1296</td>
<td>1296 (1.00)</td>
<td>1296 (1.00)</td>
</tr>
<tr>
<td>radiosity</td>
<td>95112</td>
<td>314070 (3.30)</td>
<td>94374 (0.99)</td>
</tr>
<tr>
<td>raytrace</td>
<td>2064</td>
<td>3750 (1.82)</td>
<td>2064 (1.00)</td>
</tr>
<tr>
<td>volrend</td>
<td>4904</td>
<td>8458 (1.72)</td>
<td>112 (0.02)</td>
</tr>
</tbody>
</table>

Table 4.1: The number of lock/unlock operations issued. Each pair is counted as one operation.
a custom pintool, similar to the way described by Monchiero et al. [Mon+09] and Nilakantan et al. [Nil+]. For each application and version, we collected traces once, and then ran them multiple times in the simulator, with different seeds, in order to account for the deviations in the execution of parallel applications. Taking into consideration simulation time constraints, we ran each simulation 3 times. Ideally, we would have liked to have used the same trace for all the versions, in order to avoid the different execution paths affecting the results between the versions, but this is not possible in our case since there are significant code differences that can not be be easily abstracted in the simulator.

The simulations were performed on resources provided by SNIC through Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX).

### 4.1 Synchronisation Primitives

Before we go into the details of how our modifications affect the performance of the applications, we need to present how they affected the synchronisation of the applications, namely how many more (or less) locks, barriers, atomic instructions, and conditional variables are being used. Since, we did not introduce any new barriers into the application, we will omit those from the results.

Table 4.1 shows the number of lock acquire and release operations each program issued, in the unmodified, locks, and atomics versions. In parentheses there are the same numbers normalised to the unmodified version. Each pair is counted only once, i.e. we do not count each acquire and release separately.

We can see that the programs where we introduced the most locks are Barnes, FMM, and Radiosity. This partly coincides with the applications where we found data races in many places. In both Barnes and FMM, we introduced extra locks during the octotree traversal, which explains the results. In Radiosity on the other hand, we introduced locks in the task queue, and specifically in the fuzzy barrier, which is probably where the bulk of the extra acquire and release operations happen.

Another interesting observation is that in the atomics version, we hardly see any difference in the lock operations issued, with the exception of Volrend. In Volrend, locks were used for simple integer increment/decrement operations. Since the same can be achieved with atomic operations, we removed those locks and replaced them with atomics. Since those locks appeared in the task queue code, we managed to almost eliminate locking from the application. Compared to the locks version, we have just 2% of the locks left.

Table 4.2 presents the number of atomic operations issued by each application. Since only the modified atomics version contains any atomic operations, we can not compare them with anything. We see, however, that Cholesky, followed by Radiosity, are the two applications that issue the most atomic operations. On the other hand, Ocean and Raytrace issue none to very few.

Similarly, table 4.2b presents the number of signal/wait (i.e. conditional variables)
operations issued. Obviously, only Barnes, Cholesky, and FMM have any of these operations, since these are the only applications we introduced them to. There are not really any conclusions we can draw from these numbers, but we can see that Cholesky is much more likely to have to wait for some data, when compared to the other two. As a matter of fact, Barnes only ends up waiting a few times, which means that there is not much contention in that part.

Finally, table 4.2c presents the number of FSID fences encountered during execution. This includes just the explicit FSID fences and not all the FSID regions caused by locked critical sections.

### 4.2 Performance Characteristics

Figure 4.1 presents the number of L1d (“d” for “data”) misses for the different versions and coherence protocols. For the MESI protocol, we see that on average the additional locks increase the number of misses slightly, but most applications are not affected too much. Similarly, the introduction of atomic operations affects the applications little, with the average being almost the same as the unmodified version.

However, for the VIPS-M protocol, the results are quite different. Cholesky and Volrend have much less misses for the locked version, compared to the atomics one. In Cholesky, this can be explained by the fact the instead of the efficient conditional variables, in the atomics version we utilise spinloops, which constantly cause the cache to self-invalidate all its data. Volrend on the other hand, does not utilise any conditional variables. Looking at the simulator results, the atomics version of Volrend performs self-invalidations roughly 5 times more than the locks version. This can be explained by the fact that Volrend uses multiple locks in a tight loop, which increases the contention. Atomics on the other hand suffer from much less contention, and thus can be executed much more often (before the variable gets the correct value and the loop is terminated). We can verify this theory by looking at the number of locks (8K) vs the number of atomic operations in the two versions (51K) (Tables 4.1, 4.2a). The rest of the applications exhibit the expected behaviour, with the atomics versions
performing similarly or better than the locks one. We expected this because atomic operations, unlike locks, do not need to use both acquire and release at the same time. This means that, if for example we just want to store a variable, there is no need to self-invalidate all our data, like a locked critical section would do.

Finally, we have the FSISD protocol. Generally, it performs similarly or better than the locked version for the applications that did not require any additional FSID fences. For the three applications that did require fences, Barnes, FMM, and Radiosity, it performs worse than the SISD protocol.

We also measured the amount of misses in the L2 cache, but we found that it is dominated by the protocol used, rather than the application version. Since we are not interested in comparing the different protocols, we omitted these results from here. It is, however, worth mentioning that the number of L2 misses is very small compared to the L1. On average, the L2 misses are less than 470 times the L1 misses, but this number varies greatly with each application.

Figure 4.2 presents the simulated execution times for all the applications. We see here, that even for applications such as Radiosity, where the number of L1d misses sky-rocketed, the execution time is much less affected. On the contrary, Radiosity exhibits a lower execution time for the locks version. So does Cholesky. This was expected of Cholesky, since in the locks version we have replaced an expensive spinloop with a much more efficient conditional variable. From table 4.2b we can see that Cholesky utilises the wait function quite often, which means that we have avoided a lot of spinloop iterations. This presumption is further reinforced by the fact that the atomic version (which utilises just a spinloop) performs much worse, both for MESI and for VIPS-M. At the same time, the other two applications that we introduced conditional variables to, Barnes and FMM, do not utilise them as much and thus no speed improvement is visible. Overall, we see that there is a correlation between L1d misses and execution time.

Interestingly enough, Radiosity performs much better with the locks even though we have not introduced any conditional variables and the VIPS-M version actually runs with more L1d misses. The explanation we came up with is that the fuzzy barrier (that we fixed — this is where the bug was found in Radiosity) works much
better with the locks, rather than without. We see from the simulation results that the locked version spends much less time waiting on barriers than the other two. This could mean that the work is balanced much better and the all tasks are finished faster. In order to verify this theory, we ran the unmodified version again, but this time with the barrier bug fixed (with locks). The results matched very closely to the locks version, at least as far as the execution time is concerned. The atomics version also spends more time at the barriers, but it also has less misses, which might explain why it runs better than the unmodified one but worse than the locks one. The VIPS-M version also runs better (still worse than the modified MESI versions though), in spite of the increased number of L1d misses, which fits with our theory that the performance improvement is not something coherence related.

Finally, we have the network traffic, pictured in figure 4.3. Except from the network congestion itself, network traffic is also closely correlated with the power consumption caused by changes in the coherence protocol and synchronisation used. We can see that the network traffic follows closely the L1d cache misses, which was expected, since the misses cause a lot of network traffic. The cases where the network traffic is noticeably higher correlate with the cases where the L1d misses are also high. However, the differences between the different versions are not equally pronounced. On average, we can see that, for MESI, the locks version increases the network traffic very slightly while the atomics version maintains it to almost the same level as the baseline. On the contrary, for VIPS-M, the network traffic is lower than the baseline, and at the same time, the difference between the locks and the atomics versions is almost non-existent. This is due to different applications exhibiting vastly different behaviours, and in the end averaging each other out.

As for the network traffic for the FSISD protocol, we see that it is similar or lower than the VIPS-M protocol, but only for the applications we did not need to add explicit fences. On average, because of the bad performance of the applications that did require explicit FSID fences, it is slightly worse than VIPS-M, and almost equal to MESI.

Figure 4.3: Normalised network traffic for all the applications.
We see that it is possible to convert code that contains data races into correctly synchronised code that can be run on relaxed memory models, such as release consistency. While not 100% standard and memory model compliant, it is also possible to reason about and avoid synchronising some data races, in order to avoid a significant performance degradation.

There are different ways to achieve that, from which we chose to use either locking primitives, such as locks and conditional variables, or atomic operations. We found that, in the case of the SPLASH-2 benchmark suite, in most cases the atomic operations cause less cache invalidations, and consecutively also require less power consumption. However, the locks provide slightly better execution time overall. This is due to a couple of applications that benefit quite a lot from the locks. On a case by case basis, the results vary and they depend on the characteristics of the application. Usually atomic operations provide more lightweight synchronisation, but at a higher frequency. However, that is not true for the amount of L1d misses and network traffic. The locks version increases the number of misses and the network traffic for the MESI protocol, while the atomics version does not. Finally, for the VIPM protocol, both version increase the number of misses, but the network traffic is decreased, equally, by both.

We see that explaining the changes in the runtime characteristics of the applications is very hard to do, and predicting the changes is even harder. For this reason, in any future use case of our modified code, the unmodified version should also be used (when possible) as a base case for comparison.

At the same time, converting unfamiliar applications for relaxed memory models proved to be a challenging task, mostly because we were not familiar with the original author(s) intentions and mental model of the application. Furthermore, the applications are written with certain synchronisation costs in mind, which differ from model to model. For example, many of the SPLASH-2 applications rely on the fact that store operations happen in order on the x86 architecture, and thus get synchronisation between stores essentially for free. Converting those applications to a release model, where ordering stores might required some expensive synchronisation methods (since it is not the default behaviour, for which the system is optimised for), can lead to decreased performance.

For future work, one could try to synchronise the applications in a more optimal way. Given our lack of knowledge of the applications and the synchronisation methods, we can not be sure that we have the most optimal solutions. Of course, at the same time, it is not possible to have one optimal solution, since it depends on the input data, the input parameters, and on the system the applications are run on.

Furthermore, it would be interesting to see how our synchronisation affects the scalability of the applications. Granted, many of the SPLASH-2 applications already exhibit fairly bad scalability, but some of them are supposed to scale fairly well [Woo+95].

At the same time, one could also try to remove all the data races from the appli-
cations, in an attempt to make them fully C11 standard compliant. With the current modifications, some applications still retain some of the data races, which means that the behaviour of the applications is not clearly defined. Switching to a different compiler, or even to a different version of the same compiler, can produce quite different results. This is something that we would like to avoid in benchmark applications.

Finally, SPLASH-2 and PARSEC are not the only benchmark applications being used. One could potentially use similar tools and methods as ours to synchronise even more applications, or even adapt them for different and more relaxed memory models.


Appendices
Listing A.1: Barnes: The racy accesses to r->done (load.C).
flag = TRUE;
while (flag) { /* loop descending tree */
    if (l == 0) {
        error("not enough levels in tree\n");
    }
    if (*qptr == NULL) {
        /* lock the parent cell */
        ALOCK(CellLock->CL, ((cellptr)mynode)->seqnum % MAXLOCK);
        if (*qptr == NULL) {
            le = InitLeaf((cellptr)mynode, ProcessId);
            Parent(p) = (nodeptr)le;
            Level(p) = 1;
            ChildNum(p) = le->num_bodies;
            ChildNum(le) = kidIndex;
            Bodyp(le)[le->num_bodies++] = p;
            *qptr = (nodeptr)le;
            RMS_MarkAsRacy(qptr);
            RMS_MarkAsRacy(*qptr);
            flag = FALSE;
        } else {
            le = (leafptr)*qptr;
            if (le->num_bodies == MAX_BODIES_PER_LEAF) {
                *qptr = (nodeptr)SubdivideLeaf(le, (cellptr)mynode, l, ProcessId);
            } else {
                Parent(p) = (nodeptr)le;
                Level(p) = 1;
                ChildNum(p) = le->num_bodies;
                Bodyp(le)[le->num_bodies++] = p;
                flag = FALSE;
            }
        }
        AULOCK(CellLock->CL, ((cellptr)mynode)->seqnum % MAXLOCK);
        /* unlock the parent cell */
    } else {
        /* reached a "leaf"? */
        ALOCK(CellLock->CL, ((cellptr)mynode)->seqnum % MAXLOCK);
        /* lock the parent cell */
        if (Type(*qptr) == LEAF) { /* still a "leaf"? */
            le = (leafptr)*qptr;
            if (le->num_bodies == MAX_BODIES_PER_LEAF) {
                *qptr = (nodeptr)SubdivideLeaf(le, (cellptr)mynode, l, processId);
            } else {
                Parent(p) = (nodeptr)le;
                Level(p) = 1;
                ChildNum(p) = le->num_bodies;
                Bodyp(le)[le->num_bodies++] = p;
                flag = FALSE;
            }
        }
        AULOCK(CellLock->CL, ((cellptr)mynode)->seqnum % MAXLOCK);
        /* unlock the node */
    }
    if (flag) {
        mynode = *qptr;
        RMS_MarkAsRacy(mynode);
        kidIndex = subindex(xp, l);
        qptr = &Subp(*qptr)[kidIndex]; /* move down one level */
        RMS_MarkAsRacy(qptr);
        RMS_MarkAsRacy(*qptr);
        l = l >> 1; /* and test next bit */
    }
}
SETV(Local[ProcessId].Root_Coords, xp);
return Parent((leafptr)*qptr);

Listing A.2: Barnes: The racy accesses to qptr (load.C).
34 { extern gravsub();
35  
36  Local[ProcessId].pskip = p;
37  SETV(Local[ProcessId].pos0, Pos(p));
38  Local[ProcessId].phi0 = 0.0;
39  CLRV(Local[ProcessId].acc0);
40  Local[ProcessId].myn2bterm = 0;
41  Local[ProcessId].mynbcterm = 0;
42  Local[ProcessId].skipself = FALSE;
43  hackwalk(gravsub, ProcessId);
44  Phi(p) = Local[ProcessId].phi0;
45  SETV(Acc(p), Local[ProcessId].acc0);
46  
47  #ifdef QUADPOLE
48  Cost(p) = Local[ProcessId].myn2bterm + NDM * 
49  Local[ProcessId].mynbcterm;
50  #else
51  Cost(p) = Local[ProcessId].myn2bterm + 
52  Local[ProcessId].mynbcterm;
53  #endif
54 }

for (pp = Local[ProcessId].mybodytab;
pp <
Local[ProcessId].mybodytab+Local[ProcessId].mynbody;pp++)
{

p = *pp;
SETV(acc1, Acc(p));
Cost(p)=0;
hackgrav(p, ProcessId);
Local[ProcessId].myn2bcalc += Local[ProcessId].myn2bterm;
Local[ProcessId].mynbcalc += Local[ProcessId].mynbterm;
if (!Local[ProcessId].skipself) { /* did we miss
self-int? */
    Local[ProcessId].myselfint++; /* count another
          goofup */
}
if (Local[ProcessId].nstep > 0) { /* use change in accel to make 2nd order correction to vel */
    SUBV(dacc, Acc(p), acc1);
    MULVS(dvel, dacc, dthf);
    ADDV(Vel(p), Vel(p), dvel);
}

if (Type(mycell) == LEAF) {
    l = (leafptr) mycell;
    for (i = 0; i < l->num_bodies; i++) {
        if (work >= Local[ProcessId].workMin - .1) {
            if ((Local[ProcessId].mynbody+2) > maxmybody) {
                error("find_my_bodies: Processor %d_needs_more_than_%d_bodies: increase_fleaves\n", ProcessId, maxmybody);
            }
            Local[ProcessId].mybodytab[Local[ProcessId].mynbody++] =
                Bodyp(l)[i];
        }
    }
    work += Cost(Bodyp(l)[i]);
    if (work >= Local[ProcessId].workMax - .1) {
        break;
    }
}
else {
    for (i = 0; i < NSUB) && (work < (Local[ProcessId].workMax - .1)); i++){
        qptr = Subp(mycell)[Child_Sequency[direction][i]];
        if (qptr!=NULL) {
            if ((work+Cost(qptr)) >= (Local[ProcessId].workMin
- .1)) {
                find_my_bodies(qptr, work,
                Direction_Sequency[direction][i],
                ProcessId);
            }
        }
        work += Cost(qptr);
    }
}

#endif QUADPOLE
Cost(p) = Local[ProcessId].myn2bterm + NDIM * 
Local[ProcessId].mynbterm;
else
Cost(p) = Local[ProcessId].myn2bterm + Local[ProcessId].mynbterm;
#endif

446  if( tq->n_free > 0 )
447  {
448    LOCK(tq->f_lock);
449    if( tq->free )
450      {
451        /∗ Scan the free list ∗/
452        for( i = 1, p = tq->free ;
453             (i < NALLOCATELOCAL_TASK) && p->next ;
454             i++, p = p->next ) ;
455        task_struct[process_id].local_free_task = tq->free ;
456        task_struct[process_id].n_local_free_task = i ;
457        tq->free = p->next ;
458        tq->n_free -- ;
459        p->next = 0 ;
460        UNLOCK(tq->f_lock);
461        break ;
462      }
463    }
464    UNLOCK(tq->f_lock);
465  }
...
719  if( tq->n_tasks > 0 )
720  {
721    /∗ Lock the task queue ∗/
722    LOCK(tq->q_lock);
723    if( tq->top )
724      {
725        if( qid == taskqueue_id[process_id] )
726          {
727            t = tq->top ;
728            tq->top = t->next ;
729            if( tq->top == 0 )
730              tq->tail = 0 ;
731            tq->n_tasks -- ;
732          }
733        else
734          {
735            /∗ Get tail ∗/
736            for( prev = 0, t = tq->top ; t->next ;
737                prev = t, t = t->next ) ;
738            if( prev == 0 )
739              tq->top = 0 ;
740            else
741              prev->next = 0 ;
742              tq->tail = prev ;
743              tq->n_tasks -- ;
744          }
745        }
746    /∗ Unlock the task queue ∗/
747    UNLOCK(tq->q_lock);
748    break ;
749  }
750
Listing A.5: Radiosity: The DCLs (taskman.c).
rad_avg = (inter->destination->rad_r
+ inter->destination->rad_g
+ inter->destination->rad_b) * (float)(1.0 / 3.0);
...
e->rad.r = e->rad_in.r + e->rad_subtree.r + e->patch->emittance.r;
e->rad.g = e->rad_in.g + e->rad_subtree.g + e->patch->emittance.g;
e->rad.b = e->rad_in.b + e->rad_subtree.b + e->patch->emittance.b;
...
while(e!=0)
{
    /\* Get radiosity of the child and add to my radiosity */
    LOCK(e->elem_lock->lock);
    e->rad_subtree.r += ec->rad_subtree.r * (float)0.25;
    e->rad_subtree.g += ec->rad_subtree.g * (float)0.25;
    e->rad_subtree.b += ec->rad_subtree.b * (float)0.25;
    e->join_counter--;  
    join_flag = (e->join_counter == 0);
    UNLOCK(e->elem_lock->lock);
    if(join_flag == 0)
        /\* Other children are not finished. Return. */
        return;
    /\* This is the continuation called by the last (4th) subprocess. 
    Perform JOIN at this level */
    /\* Update element radiosity */
    e->rad.r = e->rad_in.r + e->rad_subtree.r +
                e->patch->emittance.r;
    e->rad.g = e->rad_in.g + e->rad_subtree.g +
                e->patch->emittance.g;
    e->rad.b = e->rad_in.b + e->rad_subtree.b +
                e->patch->emittance.b;
    /\* Traverse the tree one level up and repeat */
    ec = e;
    e = e->parent;
}
...
while(inter)
{
    /\* Be careful! 
    Use FP(out) to compute incoming energy */
    ff_v = inter->formfactor_out * inter->visibility;
    bf_r = ff_v * inter->destination->rad.r;
    bf_g = ff_v * inter->destination->rad.g;
    bf_b = ff_v * inter->destination->rad.b;
    rad_elem.r += bf_r;
    rad_elem.g += bf_g;
    rad_elem.b += bf_b;
    /\* Update pointers */
    inter = inter->next;
}

Listing A.6: Radiosity: The racy accesses to the radiosity values (elemman.C).
In order to mark the synchronisation for our different version, we use a combination of m4 macros and empty functions. Also, all of these have equivalent outputs in the tracer.

<table>
<thead>
<tr>
<th>M4 Macro(s)</th>
<th>RMS_Initial/Final “magic” Function(s)</th>
<th>Tracer Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATOMIC_LOAD</td>
<td>Atomic_Load(void *addr, size_t size)</td>
<td>&quot;AT_AC&quot; addr size</td>
</tr>
<tr>
<td>ATOMIC_STORE</td>
<td>Atomic_Store(void *addr, size_t size)</td>
<td>&quot;AT_AR&quot; addr size</td>
</tr>
<tr>
<td>ATOMIC_FETCH_ADD/SUB.Fetch</td>
<td>Atomic_AcqRel(void *addr, size_t size)</td>
<td>&quot;AT_ARE&quot; addr size</td>
</tr>
<tr>
<td>RELEASE_FENCE</td>
<td>Atomic_Load(FENCE_TYPE_SS)</td>
<td>&quot;FENCE 3&quot;</td>
</tr>
<tr>
<td>ACQUIRE_FENCE</td>
<td>Atomic_Load(FENCE_TYPE_LL)</td>
<td>&quot;FENCE 2&quot;</td>
</tr>
<tr>
<td>FULL_FENCE</td>
<td>Atomic_Load(FENCE_TYPE_F)</td>
<td>&quot;FENCE 1&quot;</td>
</tr>
<tr>
<td>BEGIN_FSID</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 2&quot;</td>
</tr>
<tr>
<td>END_FSID</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 1&quot;</td>
</tr>
<tr>
<td>LOCK/A_UNLOCK</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>SemaphoreWait</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>CONDVAR_WAIT</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>CONDVAR_CAST</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>SPIN_WHILE</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>DO_WHILE</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>DO_WHILE2</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>DO_STORE</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>DCL_IF</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>DCL_WHILE</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>DCL_WHILE2</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>DCL_STORE</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
<tr>
<td>DCL_Load</td>
<td>Atomic_Load(FENCE_TYPE_FS)</td>
<td>&quot;FENCE 0&quot;</td>
</tr>
</tbody>
</table>

Table B.1: The macros and their matching annotation functions and traces.

Most of the operations are self-explaining; DCL_IF, SPIN_WHILE, DO_WHILE, and DO_STORE are explained in section 3.1.3.
Simulation Parameters

Table C.1 shows some of the most important system values we run the simulator with.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor frequency</td>
<td>3.0GHz</td>
</tr>
<tr>
<td>Block and page size</td>
<td>64 bytes and 4kB</td>
</tr>
<tr>
<td>Private L1 cache</td>
<td>32KB, 4-way</td>
</tr>
<tr>
<td>L1 cache access time</td>
<td>1 cycle</td>
</tr>
<tr>
<td>Shared L2 cache</td>
<td>512KB per bank, 16-way</td>
</tr>
<tr>
<td>L2 cache access time</td>
<td>Tag: 6 cycles; tag+data: 12 cycles</td>
</tr>
<tr>
<td>Memory access time</td>
<td>160 cycles</td>
</tr>
<tr>
<td>Network topology</td>
<td>2D mesh</td>
</tr>
<tr>
<td>Routing technique</td>
<td>Deterministic X-Y</td>
</tr>
<tr>
<td>Flit size</td>
<td>16 bytes</td>
</tr>
<tr>
<td>Switch-to-switch time</td>
<td>6 cycles</td>
</tr>
</tbody>
</table>

Table C.1: Simulator parameters

Table C.2 shows the input values used for the SPLASH-2 applications. For PARSEC, we used the simmedium input for all the applications we run, with the exception of Fluidanimate and Streamcluster, where we used the simsmall input due to simulation constraints.

<table>
<thead>
<tr>
<th>Application</th>
<th>Input Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>barnes</td>
<td>inputs/n16384-p#</td>
</tr>
<tr>
<td>cholesky</td>
<td>-p# inputs/tk15.O</td>
</tr>
<tr>
<td>fft</td>
<td>-p# -m16</td>
</tr>
<tr>
<td>fmm</td>
<td>inputs/input.#.16384</td>
</tr>
<tr>
<td>lu</td>
<td>-p# -n512</td>
</tr>
<tr>
<td>ocean</td>
<td>-p# -n258</td>
</tr>
<tr>
<td>radiosity</td>
<td>-p # -ae 5000 -bf 0.1 -en 0.05 -room -batch</td>
</tr>
<tr>
<td>radix</td>
<td>-p# -n1048576</td>
</tr>
<tr>
<td>raytrace</td>
<td>-p# -m64 inputs/car.env</td>
</tr>
<tr>
<td>volrend</td>
<td># inputs/head</td>
</tr>
<tr>
<td>watern-sq</td>
<td>inputs/n512-p#</td>
</tr>
</tbody>
</table>

Table C.2: Application Parameters. The # symbol is replaced with the number of threads.