Barrier Synchronisation in Java

Carwyn Ball and Mark Bull

Abstract

Since barrier routines are widely used in the paradigm of shared memory parallel programming, it is important that they are as efficient and reliable as possible. Unfortunately, due to the weak memory model of the Java Virtual Machine, special care must be taken writing such routines, to allow for the case of the JVM executing on a multiprocessor with a weak memory model. *Programmers' safety nets*, which contrain the reordering of code, are used to enforce correctness. However as they can also prevent a number of common code optimisations they can damage performance.

Java and C versions of a central barrier, a software-combining-tree barrier, a butterfly barrier, a dissemination barrier and a static F-way tournament barrier were timed on a Sun Fire 6800. The Java versions included a *naive* case (which works on the Sun Fire 6800 as it employs the Total Store Ordering memory consistency model), an *undelayed synchronized* case, and three *delayed synchronized* cases.

The naive barriers outperform the correctly synchronised barriers by a considerable margin.

1 Introduction

Race conditions can cause a program to execute in a non-deterministic fashion, producing inconsistent results. Synchronisation routines are used to remove race conditions from a code. An *episode* of a synchronisation routine forces a "fast" thread to wait for other threads to reach the same episode. Thus operations on shared data can be separated into different *epochs*, insuring the correct execution of a code.

Barrier synchronisation is a common technique. A thread executing an episode of a barrier waits for all other threads before proceeding to the next epoch. Therefore, when a barrier is reached, all threads are forced to wait for the last thread to arrive.

Use of barriers is common in shared memory parallel programming. The OpenMP library [1], [2], [3], which facilitates the transformation of sequential codes for shared memory multiprocessors, not only contains explicitly called barriers, but a number of the routines also contain hidden or *implicit* barriers.

For parallel programming in Java, it is natural to implement barrier synchronisation routines (no such routine is available in the standard Java libraries). Unfortunately, the Java memory model has raised controversy as it has been suggested [15] that there are anomalies in its design, which in some circumstances may allow code to execute in an inappropriate order and render erroneous results, while also preventing many optimisation techniques.

A selection of barrier algorithms have been written in both C and Java. The C implementations use the OpenMP library and the Java implementation uses the `Threads` class. As the multi-threaded Java memory model has been shown to be unreliable, two different versions of the Java codes were made: a naive version and a completely safe version.

The completely safe version uses the `synchronized() {}` construct to separate read and write operations to individual memory locations, while allowing memory access to different memory locations freely and concurrently. This creates a problem, as there is no way of giving write operations priority over read operations. Hence, there is a very expensive ownership problem with barrier algorithms which have threads busy-waiting for a central flag change to notify completion. Each thread claims the relevant variable for as long as it takes to read its value, and the updating thread must wait in line for its turn. The Java memory model issues are discussed further in Section 2.1.2.

The remainder of this report is organised as follows: Section 2 contains discussions of various issues involved with implementing synchronisation routines on different shared memory multiprocessors and memory consistency models. Section 3 contains descriptions of a number of barrier algorithms. Section 4 contains descriptions of experiments which were performed, and the results of these. Section 5 contains conclusions based on these experiments.

2 Synchronisation and Memory Consistency in C and Java

Non-deterministic and unexpected behaviour can occur when multi-threaded codes are not correctly synchronised. Though this is true of many programming languages, problems are more common with object orientated languages and languages which make safety guarantees. There is much hidden functionality behind the scenes, involving many layers of data structures which are not available to the programmer. The Java Virtual Machine complicates matters further.

Although multithreaded programming is supported by core Java with the `Threads` class, care must be taken to ensure that surprising results are avoided. It is fairly straightforward to write *legal* Java code which will produce unpredictable or obscure results. There are problems with the Java memory model, or more precisely, there are problems with the memory consistency model of the Java Virtual Machine. Memory consistency models are described in Section 2.1. The memory model of the multiprocessor used for the experiments (see Section 4) is described in Section 2.1.1. The Java memory model and its problems are described in Section 2.1.2. There are descriptions of preventative measures for programmers in Section 2.1.3.

2.1 Introduction to Memory Consistency Models

A *memory consistency model* is designed to formalise the appearance of the memory to the programmer. It is an interface between the programmer and the memory hardware structure.

On a single processor machine, a read returns the value of the last write. The last write is easily defined by the program order. This is not the case for shared memory machines. Different operations occur on different processors, and are not related by program order.

The first attempt to extend the single processor model in an intuitive way to shared memory machines is called the *sequential consistency model*. A multiprocessor system is sequentially consistent if:

> the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. [5]

This definition is based on two basic ideas: (a) conserving the order of operations performed by a single processor, and (b) performing operations in a well defined order; one at a time (*atomically*).

On single processor machines, it is sufficient to preserve the order of just data dependencies (recognised by reads from and writes to the same memory address) and control dependencies (such as `if()` control sequences), to maintain sequential consistency. This allows hardware and compilers a great deal of freedom to transform the code to benefit performance. (This process is referred to as *optimisation*.) On multiprocessors, preserving the order of reads and writes per-location like this is insufficient for correctness, as more complicated data dependencies are common. The requirement for sequential consistency curtails the optimisation procedures which may be performed. There are many standard optimisation procedures which cannot be used on a strictly sequentially consistent system without the possibility of changing the outcome (breaking the code).

For systems with a cache based memory hierarchy, enforcing sequential consistency requires that special care must be taken regarding: (a) cache coherence, (b) making threads aware of completion of write-operations and (c) making writes appear atomic. Again, for an implementation to adhere strictly to sequential consistency, a number of cache optimisations must be neglected.

As sequential consistency can be too restrictive for a reasonable optimisation process, a number of more "relaxed" memory consistency models have been devised. These can be characterised by how they relax the sequential consistency model. Sequential consistency is defined by two ideas: program order and atomicity, and can therefore be relaxed in these two ways. The order of reads and writes to different memory addresses can be relaxed. Impositions placed on cache updates and invalidates can also be relaxed.

In all cases, writes are eventually made visible to all threads/processors and all writes to the same memory location are serialised (forced to be atomic) in order preserve data dependencies.

In general, relaxed models require that memory be consistent only at certain synchronisation events. They thus allow buffering, merging, and pipelining writes within certain constraints specified by the model.

Release consistency is an example of a commonly used relaxed model. Memory accesses are classified as ordinary, an *acquire* or a *release*. An acquire indicates that a processor is beginning an operation on which another processor may depend—writes by other processes must be made visible to this processor. A release

indicates that the processor has completed such a job—all this processor's writes must be made visible to any acquiring processors. Thus there is a certain amount of freedom as to when writes are made visible, and memory latency can be hidden behind computation.

As most optimisations break a programmer's intuitive model of what memory should do, there are *safety-nets* available, which a programmer uses to force correctness where it is necessary. An example of a safety-net is a *fence*, which is similar to a synchronisation routine, guaranteeing all outstanding memory operations are completed before proceeding.

Unlike in C, where the code is compiled directly for the machine it executes on, Java is translated into *bytecode* to execute on a (software) *virtual machine*, which is itself executed on the (hardware) real machine. The memory models of these machines are described in subsections 2.1.1 and 2.1.2.

2.1.1 SPARC Memory Consistency Model

The memory consistency model of the SPARC machines is called *total store ordering*(TSO).

TSO works as follows. Writes are buffered and stored in order. Reads are performed in order, but can bypass writes. The store buffer is flushed when a fence is executed. In other words only the write followed by a read condition is relaxed. All other operation orderings are preserved.

This means that there are no important inconsistencies between what we expect to happen and what actually happens. The only safety-net required to ensure correctness when using C is that certain variables are declared as `volatile`, and sufficient synchronisation enforced. In C, `volatile` means that the variable is not stored in a processor register for improved serial performance on a processor. This would delay the change of the variable becoming visible to the other processors. Instead the variable is flushed straight into cache as soon as its value is set.

2.1.2 Java Virtual Machine Memory Consistency Model

The Java memory model [4] is a special case, as it describes the memory structure of a generic virtual machine—which itself is layered upon an SMM. It is described in in terms of variables.

A variable is any memory location which may be stored into: class variables, instance variables and array members. A *master* copy of each variable is stored in *main memory* which is shared by all threads. Every thread has a *working copy* of variables it is using, which it stores in its *working memory*. There is a set of rules defining when a thread should transfer its working copy to the main memory. A thread acquires a lock (also stored in main memory) before updating the main memory.

When data is copied from main memory to a working memory, two actions must occur: a read performed by the main memory, followed by a load action performed by working memory. Correspondingly, transferring data to main memory requires a store to working memory and a write to main memory. There may be a delay between these actions, so actions on one thread may be appear to another thread as having occurred in a different order. For individual variables, the actions in main memory are order-preserved.

There are a set of rules concerning ordering: actions performed by any one thread are totally ordered; actions performed by main memory for any individual variable or lock are totally ordered. This does not mean that their orders are preserved, only that they have an order.

There are a set of rules which constrain actions performed by a thread with respect to a variable, and a set of rules for when a variable becomes visible to threads which use it. Effectively, the orders of operations are completely relaxed. The Java memory model is weaker than release consistency.

For correctness the Java memory model has been shown to require memory *coherence* [6]:

> for each variable in isolation, the uses and assigns to that variable must appear as if they acted directly on global memory in some order that respects the order within each thread (i.e., each variable in isolation is sequentially consistent.) [15]

Implementing a Java Virtual Machine on an SMM with a weak memory model therefore raises issues—no strength is gained using the virtual machine.

There follow some examples of situations which are allowed according to the memory model. The first is shown in Figure 1 (taken from [15]—Figure 1). The code is not executed in a sequentially consistent way. This is allowed because of *prescient stores*. Prescient stores allow store actions to be executed first on each thread.

Initially: `x = y = 0`

Thread 1	Thread 2
`a = x`	`b = y`
`y = 1`	`x = 1`

Result: a = 1, b = 1

Figure 1: Out of order execution is valid because of prescient stores.

Default: `p = new point(1,2)`

Thread 1	Thread 2	Thread 3
`p = new point(3,4)`		
	`a = p.x`	`b = p.x`

Result: a, b = 0(!?), 1 or 3.

Figure 2: Reordering of field initialisation and ref update

Another example is shown in Figure 2 (based on Figure 9 from [16]).

Writes initialising the point (on thread 1) are not required to be sent to main memory before the write of the reference to the newly created point. Different threads accessing the object could see different values—actions may be made visible to different threads at different times. This would account for the values of 1 or 3. `a,b = 0` is possible if the default initialisation is not complete.

Synchronising the writes will not help here. The only way to fix this problem currently is to also synchronise the reader. This may be a problem for applications which involve threads busy-waiting for a central flag to change, as the "changer" thread will have to queue behind the waiting threads to gain a lock to change the flag.

The problem is that two writes to global memory are allowed to be reordered in a weak memory consistency model, by the hardware or by the compiler. As there is no data dependence the order will not be preserved. If these writes are reordered, then the results are unexpected.

It has been said that the Java memory model is "both too weak *and* too strong" [16]. Too strong in that it prevents certain optimisations, and too weak in that much of the code which has been written at present is not guaranteed to be correct according to the model. As current memory consistency models are tending to be weaker and compiler optimisations more aggressive, there may be a large collection of code which currently appears to be safe which will later be revealed to be unsafe.

2.1.3 Java Safety-Nets

synchronized The `synchronized` keyword can be used to prevent threads from running sections of code at the same time. Also, when two threads synchronise on the same object, they become aware of each other's updates. Thus using `synchronized` can prevent unusual behaviour such as that illustrated in Figure 2. It can be used in two ways:

1. `synchronized` method. The syntax is shown in Figure 3 Declaring a method as `synchronized`

```
synchronized boolean doSomething() {
   // lines of code to run exclusively
   System.out.println(‘‘Hello
world.’’);
}
```

Figure 3: Syntax for the declaration of a `synchronized` method.

excludes it from running at the same time as all other `synchronized` methods executed on the same

instance of the object. Two threads can concurrently execute a `synchronized` method on different instances of the same object. They cannot execute a `synchronized` method on the same instance of the object.

2. `synchronized` code block. The syntax is shown in Figure 4 The block is said to be `synchronized`

```
synchronized (object) {
   // lines of code to run exclusively.
   System.out.println(‘‘Hello
world.’’);
}
```

Figure 4: Syntax of a `synchronized` block.

on the object. Blocks which are `synchronized` on the same object are executed exclusively. A section of exclusive code may be executed concurrently with a different exclusive section by using a different `object`.

Declaring a method `synchronized` is the same as placing the code of the method inside a `synchronized` block, using the object which the method is called by as the `object` on which the block is `synchronized`.

There is an overhead associated with using `synchronized`. Calling a `synchronized` method or block is significantly more expensive than calling a non-`synchronized` method or block, because of the extra book-keeping which is required in the virtual machine. The performance of `synchronized` methods has been measured as between six and 100 times slower than that of non-`synchronized` methods.

volatile Similarly to in C, the Java `volatile` prevents the variable from being stored in a processor register and not being made visible to other threads. `volatile` variables are flushed straight into cache after being changed. Furthermore, actions on the master copy of a `volatile` variable are executed in program order.

wait-notify This is useful for multithreaded code which is run on a single processor, but is worth mentioning here. A thread which has finished its current job can call `wait`, which makes it relinquish its time on the CPU, allowing threads with unfinished jobs to take priority. A single `wait`-ing thread can proceed when a `notify` call is made, or all can proceed at once if a thread calls `notifyall`. A barrier can be made which uses this pair of calls for its wake-up phase but the performance is not competitive. For comparison, a `wait-notify` barrier version of the Central Barrier was tested. See Section 3.10 for details.

3 Barrier Routines

An episode of a barrier prevents any thread from entering the next epoch before each thread has left the previous one. Thus, an episode prevents two threads from executing adjacent epochs at the same time. If reads and writes to global data are contained in separate epochs, then their order is guaranteed.

Each episode is typically composed of phases *arrival*, *wake-up notification* and *re-initialisation*. In the arrival phase, threads communicate that they have arrived. Often only one thread “knows” that all have arrived, and it is this thread which performs the wake-up notification. Sometimes the notification phase is not required (for example with the dissemination barrier, synchronisation *disseminates* through all the threads and no single global wake-up is required). Re-initialisation returns the central data structure to a suitable state for the thread to enter another episode successfully, without passing straight through or getting stuck.

3.1 Wake-up Notification

Notification can be achieved in a number of ways, the more efficient depending on the architecture used. A simple method uses a shared central flag, which is changed to indicate that the arrival has been completed. Expectant threads busy-wait for it to change like so:

```
while(flag != local_target);
```

Here the local target is a thread-private boolean variable. For each new episode of the barrier, the sense of this local target is changed. This procedure is known as *sense reversal*. Examples of variations on this theme use linear and exponential delay functions to reduce contention between threads.

An alternative is to use a *wake-up tree*. Each thread except the "aware" thread is designated a parent. The aware thread notifies its children. Each notified parent thread notifies its child threads. Mellor-Crummey and Scott [13] used a fan-out of two because it resulted in the shortest critical path for a uniform tree, but larger fan-outs may be better, as they result in fewer wake-up stages.

The rest of this section contains descriptions of several barrier algorithms. Implementations of a central barrier, software combining tree barrier, butterfly barrier, dissemination barrier and a static f-way barrier have been made using C with OpenMP, and Java. A basic test code executes 1,000,000 iterations of each. The master thread executes the difference in time between the start and the finish. The C barriers have been compared with the OpenMP standard barrier, and various other comparisons have been made. More details about these experiments can be found in Section 4.

3.2 Central Barrier

Each of the N threads entering the barrier atomically decrements a shared integer, the initial value of which is N. If the value of the integer after decrementation is 0 then the thread resets the counter and changes a shared central flag. Otherwise, the thread waits for notification.

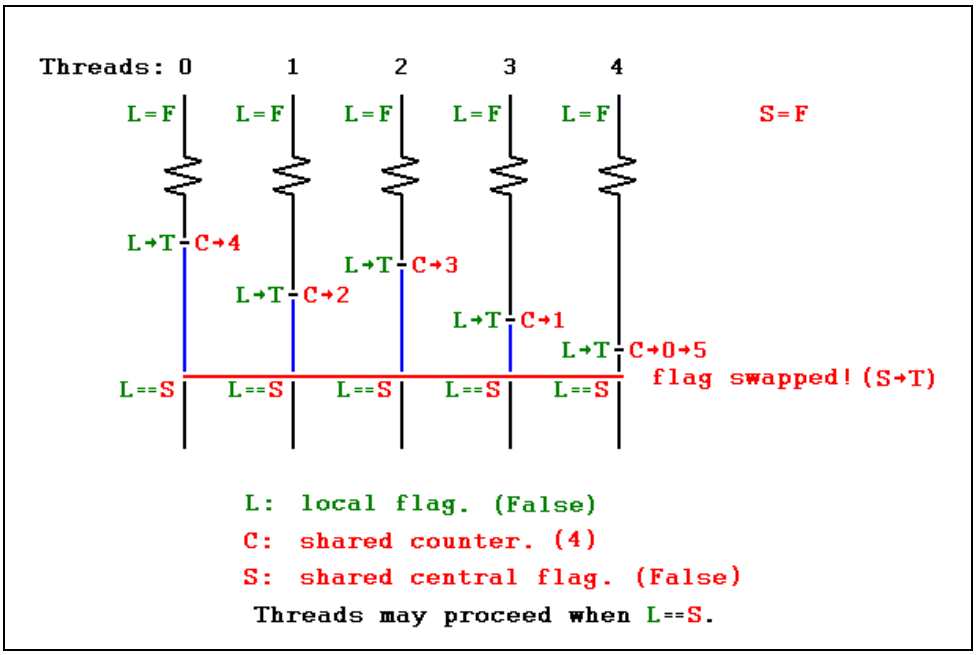

Figure 5: Control flow diagram for Central Barrier Algorithm, with a central flag wake-up.

Figure 5 illustrates the procedure. As each thread enters the barrier, L is swapped, and C is decremented. When C reaches 0, then it is reset and the central flag is swapped. The waiting threads can proceed when L and C have the same value.

The algorithm relies on every thread reading and writing to a single memory location: the counter. This memory location is known as a *hot-spot*, as the threads all need read and write access.

As each increment needs to be performed atomically, the lower bound of the algorithm overhead scales with O(N).

3.3 Software Combining Tree Barrier

The software combining tree barrier, proposed by Yew, Tzeng and Lawrie[12], is composed of groups of threads, which each have a central counter, as in the central barrier. The last thread in the group goes through to the next level, while the others wait for wake-up notification. This procedure is illustrated in Figure 6.

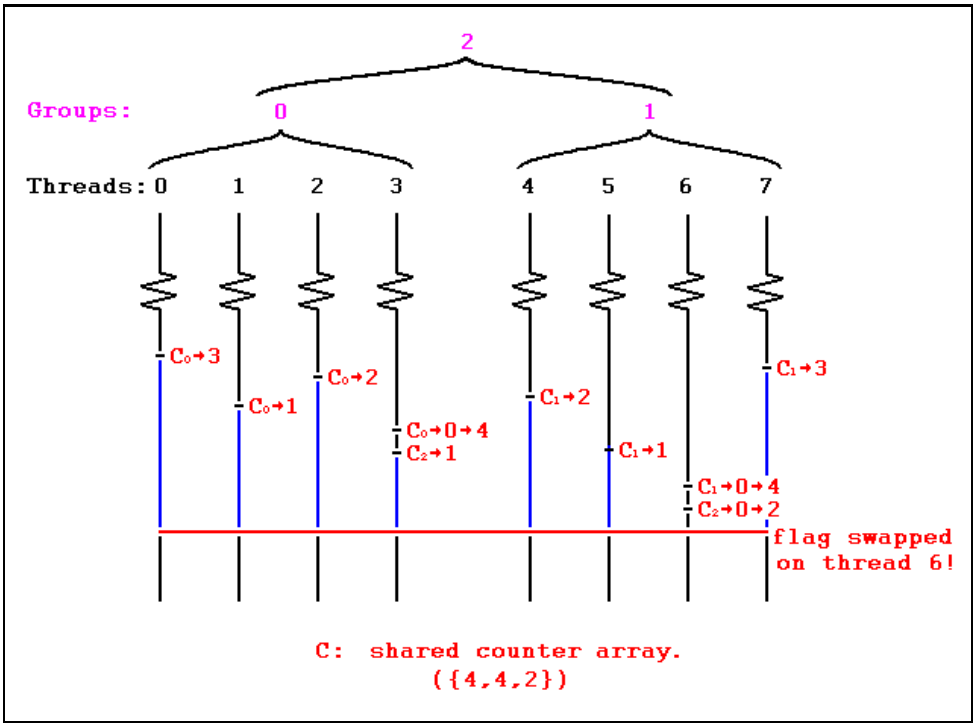

Figure 6: Control flow diagram for Software Tree Barrier Algorithm, with a central flag wake-up.

The first four threads find themselves in group 0, and so decrement counter C_0. The final four threads find themselves in group 1, and so they decrement the counter C_1. The last thread in each of these groups is the thread which sets the counter to 0 (threads 3 and 6). This thread resets the counter to its original value (4) and proceeds to the next level, decrementing the counter C_2. Thread 6 decrements C_2 to 0, resets it to 2 and swaps the shared central flag.

When the barrier is initialised, the number of levels and number groups are calculated, followed by arrays containing the numbers of groups in lower levels and numbers of threads in each group. The counter for each group is initialised with the number of threads in that group.

The number of levels $L = \lceil log_A N \rceil$, where A is a predetermined constant—the maximum number of threads in a group. Yew *et al.* [12] reported that a $A = 4$ produced the best performance with software-combining trees.

At each stage, each thread firstly works out which group it is in, then atomically decrements the relevant counter. If the counter becomes zero, then this thread resets the counter to the value of the number of threads in the group and continues to the next level. Otherwise it busy-waits for the flag to change. The last thread in the last group changes the flag. As in the central barrier, the sense of the flag target is changed at each barrier.

The overhead of this barrier relates to how the threads percolate into groups. The effect of hot-spots is reduced as thread contention is spread across more memory locations—different counters can be decremented at the same time. The length of time taken for a thread to pass through a level is (at least) proportional to the number of threads in a group. The number of levels is (at least) proportional to the logarithm of the number of threads. The overhead therefore scales as O($A \times \lceil log_A N \rceil$).

3.4 Brooks Butterfly Barrier

This multi-stage barrier was proposed by Brooks [10]. It is built out of pair synchronisations and uses a shared array of flags. The pair synchronisation algorithm shown in Figure 7.

At stage 1 each thread waits until any previous instance of the pair-synchronisation has finished. At stage 2 each thread sets its own flag to true. At stage 3, each thread waits until its partner's flag has been set to true. Stage 4 is resetting the partner's flag to false.

When the number of threads is a power of 2, then at each of S ($= log_2[N]$) stages (s), each thread(t) synchronises with thread t XOR 2^s. In doing so, it is also synchronising with each of the threads which the partner thread has synchronised with at previous stages. The array of flags is a $N \times log_2[N]$ shared array of booleans. This procedure is illustrated in Figure 8.

When the number of threads is not a power of two, some of the threads have to synchronise with two threads. The array of flags is $M \times log_2 M$, where M is the next power of two higher than N, the number of threads. If, for example, there are 5 threads, the array of flags is 8×3, and threads 0, 1 and 2 also represent the virtual threads 7, 6 and 5, as illustrated in Figure 9.

The algorithm is therefore most efficient when the number of threads is a power of two—none of the threads are ever required to synchronise with more than one other. When the number of threads is slightly higher than a power of two is when the algorithm is the least efficient, as nearly half of the threads will be synchronising with two. The lower bound of the overhead of the algorithm varies with O($log_2 N$), and there are expected to be are steps at powers of two.

3.5 Dissemination Barrier

The Dissemination Barrier addresses the problem with the Butterfly Barrier—being inefficient for non-powers of two numbers of threads. It is not built out of pair synchronisations, but uses a similar procedure of flag setting and waiting. This is shown in Figure 10.

The procedure is exactly the same as for the previous pair-synchronisation except that the thread whose flag is altered is not the same as the thread which checks that this thread's flag is altered. Thread t checks the flag of the (cyclically) next thread ($t + 2^s mod N$), and waits for its own flag to be set by the cyclically previous thread ($t - 2^s mod N$). The correctness of the algorithm is not obvious, and is detailed in [11]. Figure 11 illustrates the procedure.

Each thread only ever needs to check the flag of one other, so the target threads can be stored in a global $N \times S$ array (where S is $\lceil log_2 N \rceil$), which means that no arithmetic needs to be performed to find the thread pair at any stage after initialisation.

The overhead of this algorithm scales as O($\lceil log_2 N \rceil$).

3.6 Tournament Barrier

The Tournament Barrier was also proposed by Hengsen *et al.* [11]. The barrier is built out of two-thread *games*, which are arranged in a tournament structure—the winners play other winners until there is an overall champion. The losers await wake-up notification which is performed by the champion. The two-thread games are as follows:

```
     Partner 0                       Partner 1

1    while(flag[0]!=false);          while(flag[1]!=false);
2    flag[0]=true;                   flag[1]=true;
3    while(flag[1]!=true);           while(flag[0]!=true);
4    flag[1]=false;                  flag[0]=false;
```

Figure 7: A pair synchronisation algorithm. The `flag` variable is a shared array of booleans, initialised to {false, false}.

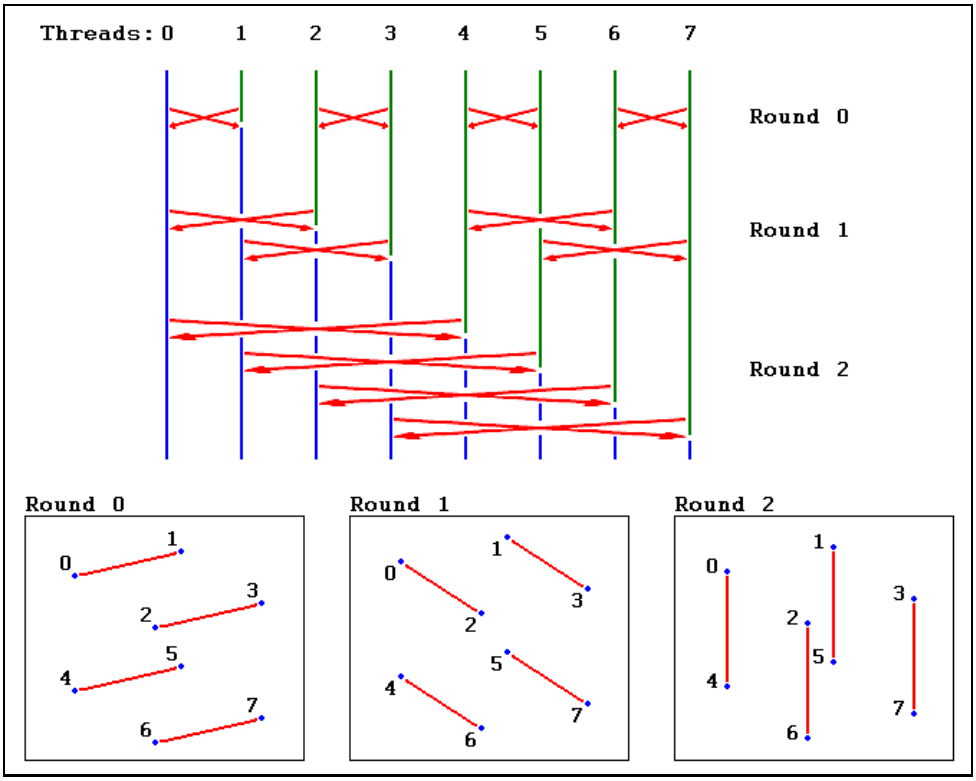

Figure 8: Control flow diagram for Butterfly Barrier Algorithm with a power of 2 number of threads.

In the first step, the winner is just waiting for the loser to arrive. In the second step, the winner resets the loser's flag, while the loser waits for one thread to win overall. Notice, these games are not competitive. The order at which the threads arrive at the game is unimportant. They represent a one-way synchronisation. If the loser arrives first, then its flag is already set when the winner arrives. This means the "winner" can be chosen (arbitrarily) in advance.

As in the previous two algorithms, each episode is made up of $\lceil log_2 N \rceil$ rounds. At each round (s), thread t enters a game with thread t XOR 2^s. The opponents at each round are calculated at initialisation and stored. The communication structure of the algorithm is illustrated in Figure 13.

In the first round, the games are between thread pairs: 0 and 1, 2 and 3, 4 and 5, 6 and 7; in each case, the former is the winner. In round two, the games are between: 0 and 2 and 4 and 6. and in round three, the winners: 0 and 4 play, with 0 becoming the overall "champion". 0 then swaps the central flag and all the threads proceed into the next epoch.

The overhead of this algorithm is again proportional to $\lceil log_2 N \rceil$. The synchronisation-game code is shorter than that of the butterfly and dissemination barriers, so the tournament barrier is expected to be more efficient than these.

3.7 MCS-Tree Barrier

This barrier was proposed by Mellor-Crummey and Scott [13]. Each thread is assigned a unique tree-node, and a *parent*. The barrier is initialised so that the thread's `parent_pointer` variable is directed to the `child_arrived` flag of its parent.

When a thread arrives at an episode of the barrier, it checks or waits for its `child_arrived` flag to be set by each child. Threads without children have these flags set permanently. After the children arrive, the thread resets its `child_arrived` flag, sets its parent's flag, and waits for notification. The top thread notifies. Notification can be achieved by a central flag or a wake-up tree. The communication structure of this algorithm used with 16

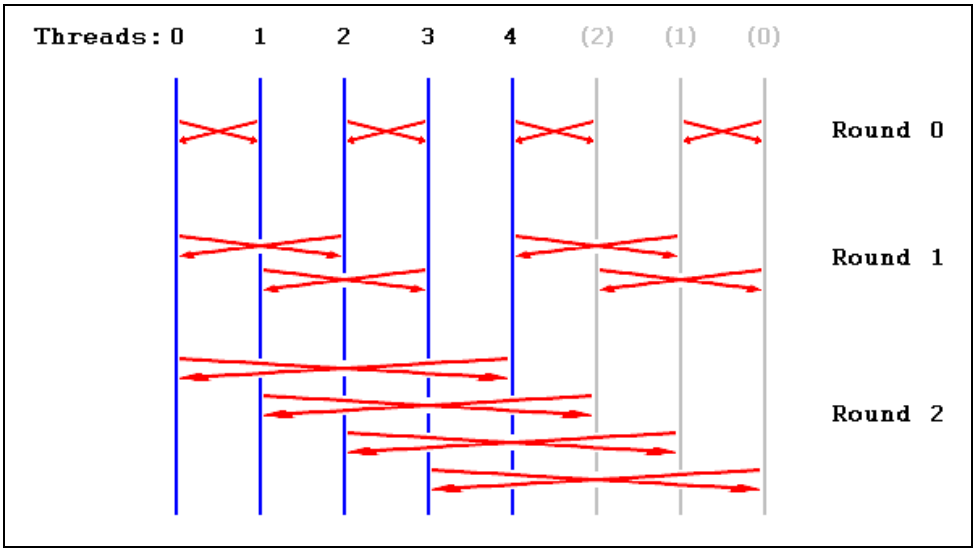

Figure 9: Control flow diagram for Butterfly Barrier Algorithm with a non-power of two number of threads.

```
Partner me

1.  while(flag[me]!=false);
2.  flag[me]=true;
3.  while(flag[other]!=true);
4.  flag[other]=false;
```

Figure 10: Building block of the Dissemination Barrier. Note that his procedure is not the same as Figure 7.

threads is shown in Figure 14.

Mellor-Crummey and Scott used a fan-in of four for the arrival-tree. This value is chosen because Yew *et al.* reported that this value produced the best performance with software-combining trees and it allows the `child_arrived` flags to be packed into a data-structure of length one word. This means that a parent can check all the children have arrived in the same time that it would take to check one, if the memory was not so arranged.

3.8 f-way Tournament Barrier

This algorithm was proposed by Grunwald *et al.* [14]. The idea is the same as the tournament barrier, but the number of threads "competing" at each round is more than two. The number is called the fan-in and the choice of this affects the efficiency if the algorithm. The advantage is that there are fewer rounds.

There are two versions of the f-way tournament barrier: static and dynamic. The static version is more similar to the 2-way tournament barrier described previously. The fan-in f depends on the number and arrangement of threads at any stage, and may differ between levels. The aim is to balance each level as much as possible. The critical path is of length $log_f N$, and has a value between two and the number of bytes in a memory word (usually four). In the same way as with the MCS tree, the flags are packed into a memory word.

The communication structure is illustrated in Figure 15.

With the static tournament barriers and the MCS tree, the readers (parents/winners) and writers (children/losers) are predetermined. At runtime, the order of arrival cannot be predicted, so there can be problems on systems with cache. In the case where a reader thread arrives before a writer thread:

1. The writer will cache-miss as it brings the shared line into cache.

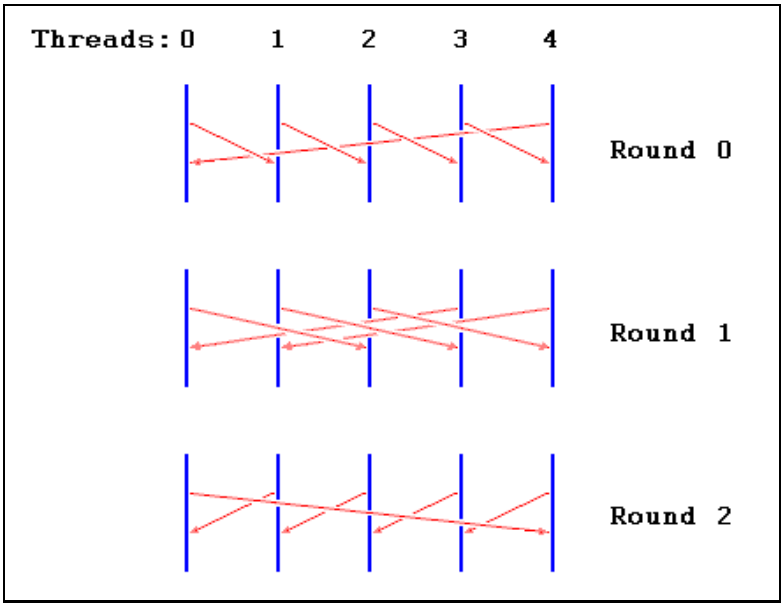

Figure 11: Control flow diagram for Dissemination Barrier Algorithm.

```
     Winner                                  Loser

1    while(SetByLoser==false);               SetByLoser==true;
2    SetByLoser=false;                       while(SetByChampion==false);
```

Figure 12: The building block of the Tournament Barrier.

2. The reader will cache-miss as it reads the shared line which has been invalidated.

The expected number of cache misses for the MCS tree and the static f-way is approximately $\frac{3}{2}f$.

The dynamic f-way tournament differs from the static in that the last thread to arrive at a node goes on to the next level. There are fewer cache misses, as in each case the writers are the first to arrive. There are no cache-line invalidation problems. The only other differences between this and the static version is that a sense-reversal mechanism is used to detect the arrival of the child threads and consecutive epochs of the barrier run in two separate flag-trees.

The overheads of both the static and dynamic f-way tournament barriers are approximately proportional to $log_f N$.

3.9 C implementation

For each barrier, there is an initialisation function, which is executed on each thread. This sets up the threads' private data with the appropriate values, for example the `local_sense` variable. The initialisation function also contains a section to be executed on a single thread—using the `single` OpenMP directive. This sets up global shared data structures with appropriate values. These global data are then acted upon by the barrier routines called by each thread.

As the C version is implemented using OpenMP which requires optimisation level 3 on Sun machines, care must be taken to prevent the compiler breaking the code. This can simply be done by removing the relevant functions to a separate file to be compiled separately.

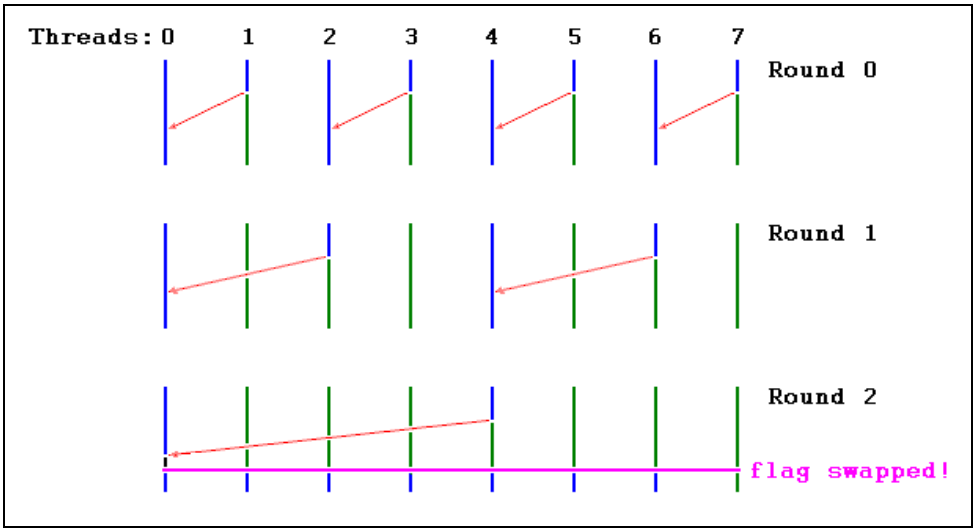

Figure 13: Control flow diagram for Tournament Barrier Algorithm, with a central flag wake-up.

3.10 Java Implementation

This section contains a breakdown of the barrier objects and their methods. Objects and methods are named according to the convention whereby objects begin with capital letters and methods do not.

3.10.1 Timing Code

Each barrier is tested and timed using a similar benchmark class. The main method takes in a command line prompt of the number of threads to use. If nothing is typed, the default is one. It then initialises a *benchmark* object and calls its `run` method, which initialises the *barrier object* and spawns the appropriate number of threads (i.e. initialises that number of thread objects). Each thread then executes the `doBarrier` of the barrier.

3.10.2 Naive Barriers

The Central Barrier uses a simple `FlagObject` and `CounterObject`. The `CounterObject` contains a `counter` volatile integer, `top`, a volatile integer ontaining the maximum value of the counter variable, and an empty `LockObject`. The `LockObject` is used for the `synchronized` blocks, contained in the methods `decrementCounter` (which returns `counter`) and resetCounter. This to make the decrementation of the counter atomic—to prevent overlapping which may cause two decrements to appear as one, stalling the barrier. The `FlagObject` contains a volatile boolean `flag`, and `swapFlag()` and `isDifferentFrom(boolean)` method. The `local_sense` is a thread variable that is passed into the `isDifferentFrom(boolean)` method.

The Software Tree Barrier is composed of `FlagObject`s, an array of `CounterObjects` which are `synchronized` on a corresponding array of `LockObjects`, one for each group. All these objects are the same as those contained in the Central Barrier. Again `local_sense` is a thread variable which is passed in.

The Butterfly Barrier and Dissemination Barrier contain a boolean `flag[]` array. The Static f-Way Barrier contains a boolean `isDone[]` array.

A Wait-Notify barrier from the Java Grande Benchmark Suite [7] [8] [9] is also tested. The algorithm is similar to that of the Central Barrier, but the `wait` replaces busy-waiting, and `notify` replaces swapping the central flag.

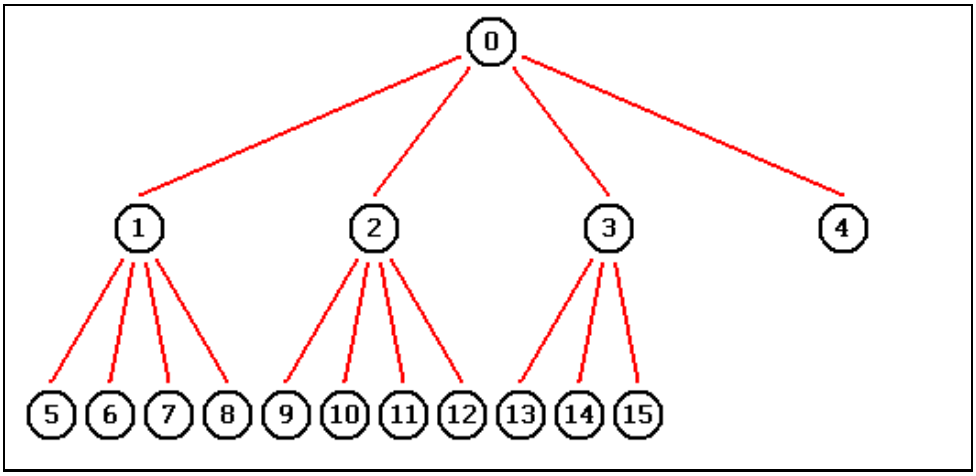

Figure 14: Control flow diagram for an MCS Tree Barrier Algorithm with 16 threads.

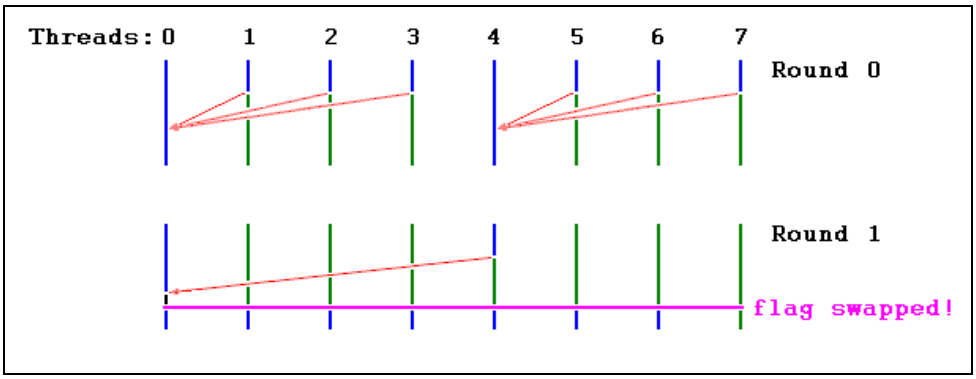

Figure 15: Control flow diagram for f-Way Tournament Barrier Algorithm with a fan-in of 4.

3.10.3 Safe Java

These barriers are largely the same as the naive barriers, except the `synchronized` keyword is used to make reads and writes mutually exclusive. In order to make the barriers safe, all reads of and writes to shared counters and flag variables and arrays must be `synchronized`. Different flag variables must be `synchronized` on different objects. The empty `LockObject` class is used again. Where there are arrays of flags, arrays of `LockObject`s are also defined.

These changes require threads to lock other threads out while they busy-wait for a flag to be set. In effect, the threads must queue up for access to the variable. This leads to problems where all the threads are spinning on a central flag, waiting for a thread to change it. This can allow all the waiting threads to get a head-start in the queue before the changer thread arrives which can result in a huge overhead.

In an attempt to reduce this problem, the busy-wait command can be altered to introduce a delay, as follows:

```
while (flag !=local_sense);
                ⇓
while (flag != local_sense) delay(D);
```

The delay method is defined as in Figure 16.

`D` is referred to as the *delay factor*. The Java compiler is not capable of dependence-analysis sophisticated enough to recognize that the increasing non-negative function `total` will never be negative. (Unfortunately, this could happen in theory if `total` overflowed, but this will not happen for the order of magnitude required here.)

```
private void delay(int D){
    int i,total;

    for(i=0; i<D; i++) {
        total += i;
    }

    if (total < 0)
        System.out.println(''This will
                                never be
printed.'');
}
```

Figure 16: The delay function used to reduce thread contention around flag variables.

The if() statement therefore prevents the compiler from recognizing that the code result `total` is not used, and getting rid of it as a performance optimisation. The value of the delay factor will change the length of the delay. If it is too small, then it will not reduce the thread contention, but if it is too big, then it will introduce a new overhead—the delay overriding the queuing effect. The code was tested with values D of 50, 100 and 150.

4 Experiments

Code was developed and compiled on a Sun HPC 3500 with eight 400MHz UltraSPARC II processors, each with 16Kb Level One cache and 4Mb Level Two cache.

Experiments were performed on a Sun Fire 6800 RISC based SMM with 24 750MHz UltraSPARC III processors. The caches are arranged as follows: Level One—64Kb, is four way associative with write through consistency protocol; Level Two—8Mb, is direct mapped, combined data and instruction cache. The nominal peak performance of the machine is approximately 36Gflops. The (shared) memory is 48Gbytes.

The C compiler used was Sun WorkShop 6 update 2 C Compiler. The Java translator used was Sun Java 2 SDK, Standard Edition Version 1.4.0. All C and naive-Java barrier programs executed 1,000,000 iterations of the relevant barrier routine. Safe Java codes' run-times were significantly longer, so these were only executed 1000 times.

Times were taken, before and after the barrier executions, using `System.currentTimeMillis()` for Java and `gettimeofday()` for C. The difference in the times measured by the master thread (`id=0`) were recorded.

The "safe" Java codes were timed with values of 50, 100 and 150 for the delay factor. All the results obtained from the barrier codes are shown below.

4.1 C

Figure 17 shows the timings obtained from the C implementations of our barriers and the barrier provided by the OpenMP library.

The Central Barrier has the worst performance. The time approaches 90 microseconds for 24 threads. The time scales worse than O(N); it is closer to O(N^2). This is caused by thread-contention for memory hot-spots.

The Butterfly Barrier takes approximately 50 microseconds on 24 threads. It appears to scale as O(N). There are very pronounced steps at powers of two. At these stages, the algorithm requires an extra stage of execution, and most of the threads perform extra work.

The Software Combining Tree Barrier takes approximately 38 microseconds on 24 threads. The curve has a logarithmic shape, as predicted. There is drop in the time on 20 threads, which cannot be explained.

The Dissemination Barrier takes approximately 19 microseconds on 24 threads. It scales with O($logN$). There are steps at powers of two which are not as pronounced as those from the Butterfly Barrier. This is because the

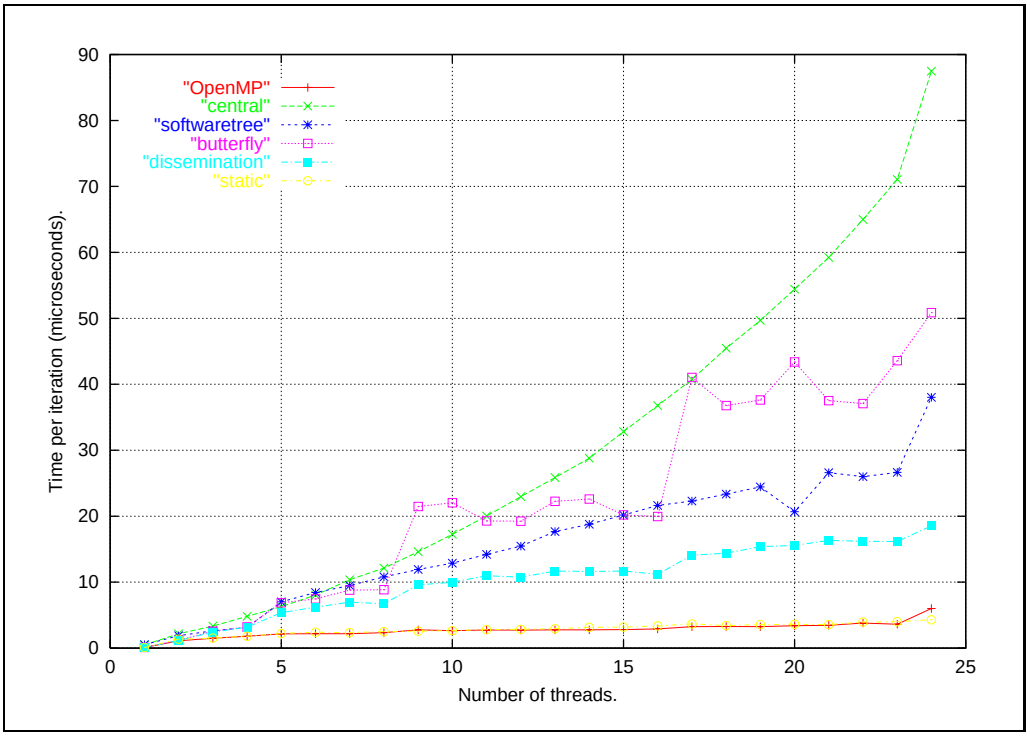

Figure 17: Times of all C barrier implementations.

algorithm requires an extra stage of execution, but extra work is more evenly distributed between the threads. This barrier is more than twice as fast as the Butterfly Barrier.

The Static f-Way Barrier produces the best performance. Its time for one execution is approximately 6 microseconds. It scales with O($logN$). The OpenMP Barrier curve and the Static f-Way Barrier curve are very close, which leads one to speculate that they are the same.

All barriers appear to have a sudden increase when they are executed on 24 threads. This may be caused by contention with operating system processes. These execute on unused processors when there are fewer threads than processors. When all processors are used, there is contention.

4.2 Java

All timings from the naive codes are shown in Figures 18 and 19.

Again the Central Barrier is the worst. It closely resembles O(N) on up to 20 threads, and for more threads the times diverge. The time for 24 threads was so long it was not recorded.

The Software Combining Tree Barrier obtained the next best performance. It appears to scale with O($logN$). Again the time diverges approaching 24 threads.

The Butterfly Barrier again appears to scale with O(N). Again, the graph is stepped at powers of two. The time for 24 threads is approximately 70 microseconds.

The Dissemination Barrier appears to scale with O($logN$). The time for 24 threads steps up to about 20 microseconds. The performance of this is significantly better than that of the Butterfly Barrier above eight threads.

The Static f-Way Barrier is again the best performer. The time for 24 threads is approximately 10 microseconds. It again scales with O($logN$).

The results obtained from the safe code are shown in Figures 20 to 31, without delay (which is referred to as "original safe") and with the delay factor set to 50, 100 and 150. These also contain the times of the naive and C implementations.

Figure 20 shows the results obtained from all the codes executing the Central Barrier. Figure 21 shows the naive Java and C results more clearly. The times from the safe codes are closely grouped. Using no delay is

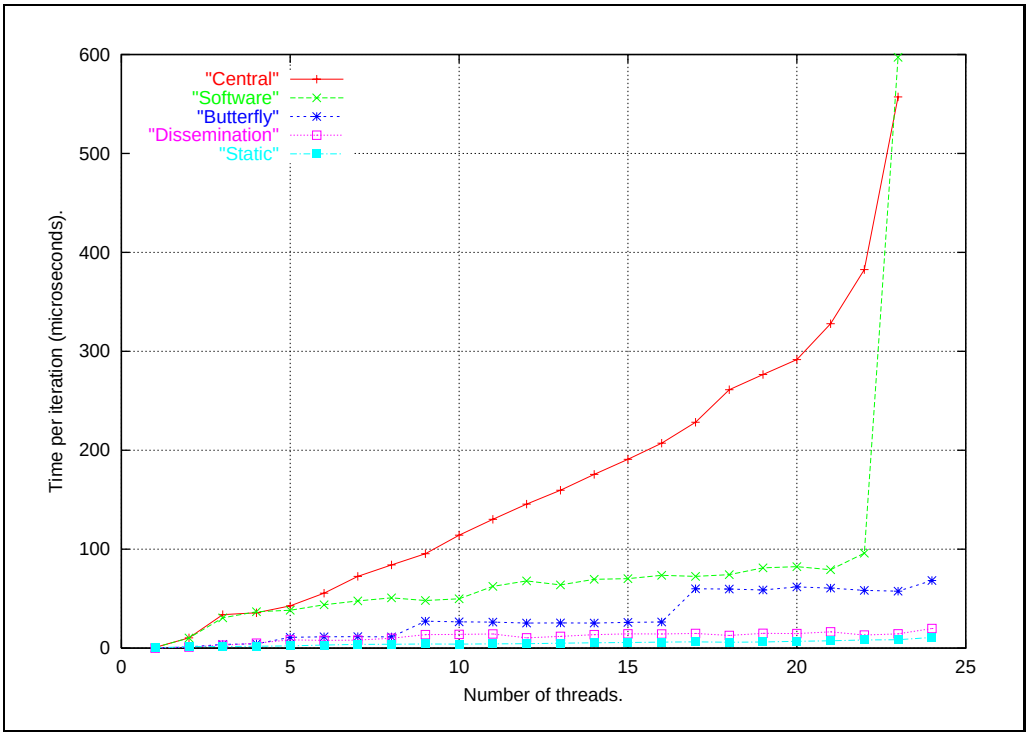

Figure 18: Times of all naive Java barrier implementations.

clearly the most efficient. The safe routines take approximately 4 milliseconds using 24 threads. They scale with O(e^N). The wait-notify code scales with O(N), taking approximately 800 microseconds on 24 threads. There are spikes at N=18 and N=22. The wait-notify code is approximately twice as fast as our safe codes.

The runtime of the naive code is approximately six times faster than the safe code (three times faster than the wait-notify code).

The undelayed safe routine has a runtime spike of approximately 1500 microseconds on two threads. The delayed safe codes do not display similar values. The contention for the shared data is extremely large for this value.

The naive Java routine takes approximately six times longer than the C routine.

Figure 22 shows all the results obtained from the Butterfly routines. Figure 23 shows the same except with the dominating undelayed safe Java timings removed. Figure 24 shows the C and naive Java codes. The undelayed safe Java gives the worst performance. The next is that with delay factor set to 150, which appears to scale with O(e^N). The routines with delay factors of 50 and 100 have a similar runtime overall. A delay factor of 50 produces the best performance on 24 threads, taking approximately 650 microseconds. The safe Java is between 5 and 150 times slower than the naive Java (approximately). The C routine's runtime takes approximately $\frac{2}{3}$ that of the naive Java routine.

Figure 25 shows all the results obtained from the Dissemination codes. Figure 26 shows the same except with the dominating delay=150 results removed. Figure 27 shows the results from the C and naive Java codes. The delay=150 routine gives the worst performance, scaling with O(e^N). The other safe codes perform better overall, staying below 4 milliseconds for all numbers of threads below 23. Again, there is divergence at 24 threads. The delay=50 code give the best performance of the safe codes overall, but reaches approximately 100 milliseconds on 24 threads. The naive code is approximately 30 times faster than the D=50 routine. The naive Java and C codes give similar performance, both taking about 20 microseconds on 24 threads.

Figure 28 shows all the results obtained from Software Combining Tree codes. The safe codes are closely grouped with none clearly giving a best performance. They scale with O(N^2), taking about 3 milliseconds on 24 threads. The naive Java implementation performs approximately five times better than the safe implementations. The C code performs approximately 3 times better than the naive code.

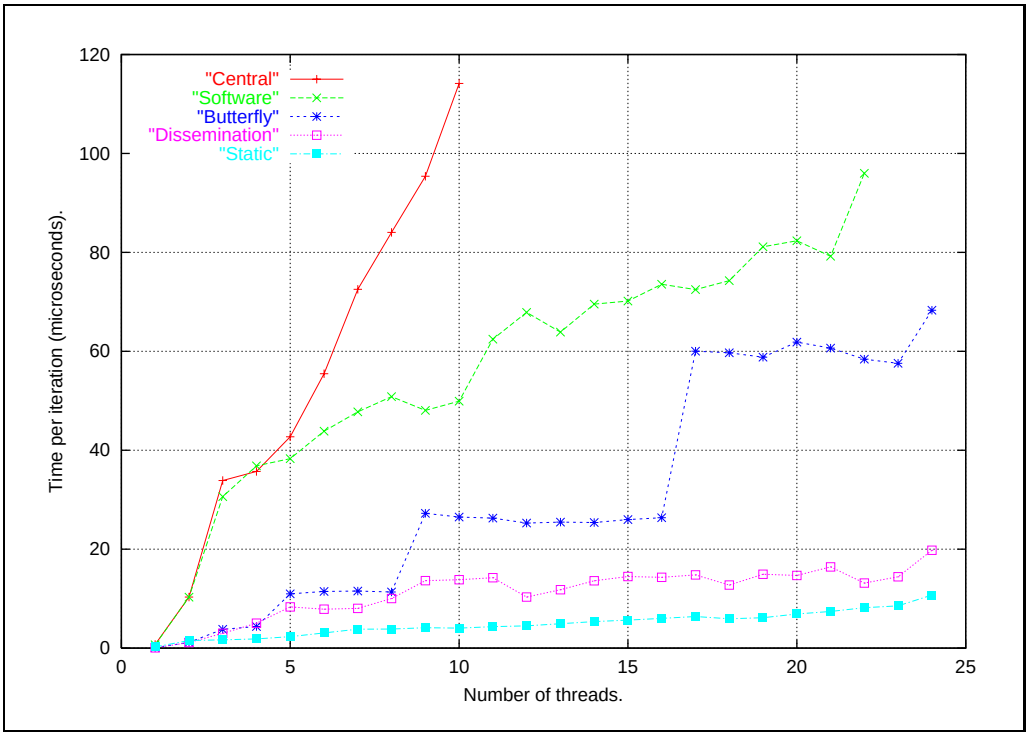

Figure 19: Times of all naive Java barrier implementations—without extreme values.

Figure 30 shows all the results obtained from the Static f-Way codes. Figure 31 shows the results obtained from the C and naive Java codes. All the safe codes scale with O(N^2), and slightly increase on 24 threads. There is unusual spiky behaviour on small numbers of threads for all except the delay=150 code, but on more threads none is clearly the best. They take about 3.5 milliseconds on 24 threads. The naive code is approximately 200 times faster than the safe codes. The C code is approximately twice as fast as the naive Java.

5 Conclusions

Many shared memory codes require a large amount of thread synchronisation in order to correctly operate. It is therefore useful to minimise the overhead of synchronisation routines. The use of barriers is the simplest way to synchronise threads.

A number of barrier routines were implemented using C with OpenMP and Java: a Central Barrier, a Software Tree Barrier, a Butterfly Barrier, a Dissemination Barrier and a Static f-Way Tournament Barrier. 1,000,000 iterations of these were executed on a 24 processor Sun Fire 6800, and the times recorded.

The most efficient barrier for both C and Java was the Static F-Way Tournament Barrier. The C implementation takes approximately 6 microseconds on 24 processors, and the Java implementation takes approximately 18 microseconds. In general the C implementations are faster than the naive Java implementations, typically by a factor of two or three. Bearing in mind that Java is an object-oriented language which runs on a virtual machine, this performance is acceptable.

However, there are problems with the memory consistency model of the Java Virtual Machine, which permit certain unpredictable behaviour on some architectures, if special care is not taken. The Java implementations were extended using `synchronized` blocks, to ensure that the current version of any shared variable is visible to the thread reading it. This means that the threads all “own” the variable while they read or write to it, so must queue for ownership when it is required. As a result the performance of the *safe* versions is much worse that that of the *naive* versions. To reduce this effect, a delay function was included in the busy-wait code.

The safe Java implementations produced times which were between five and 200 times worse than those

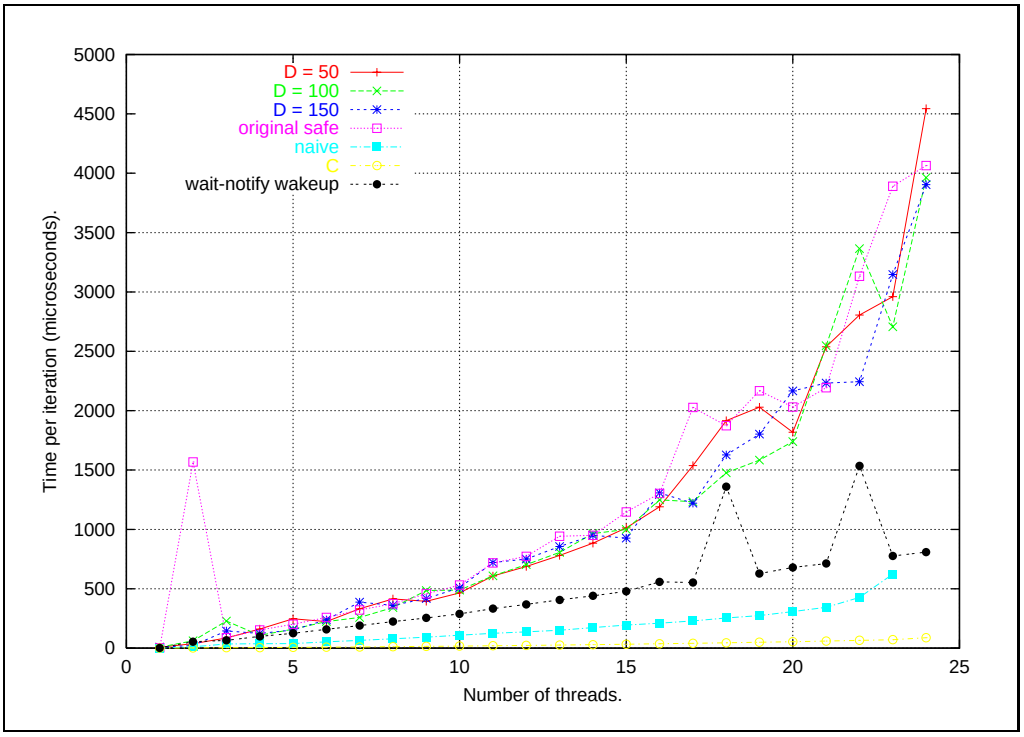

Figure 20: Times of all safe and naive Java (including wait-notify) and C implementations of the Central Barrier.

produced by the naive implementations. The best safe barrier was the Butterfly Barrier, which with the appropriate delay would produce an extrapolated time of 500 microseconds on 24 threads. The time on 24 processors must be extrapolated because timings are slowed down when the number of threads running matches the number of processors. The effect of full machine contention is more of a problem for the Java codes, as the Java Virtual Machine runs threads such as the garbage collector.

The cases for which the delay function was beneficial were the Butterfly and Dissemination Barriers. For the Butterfly Barrier, a delay of 50 produced the best results, improving the performance of the code by as much as 30 times. For the Dissemination Barrier, a delay of 50 was also most productive, improving the performance by a factor of five, approximately. In this case, a delay of 150 significantly damaged performance, making the code scale with O(e^N). For the other barriers, the delay did not have an obvious effect.

The implementation which was most efficient compared to its naive equivalent was the Software Combining Tree. The times of each of its safe routines were grouped around five times slower than those of the naive version. The worst implementations were the safe Java Static f-Way implementations, which were grouped around 200 times slower than the naive code.

The `wait-notify` algorithm is a contention-free version of the central barrier, and scales with O(N), as predicted. A number of implementations of the Central Barrier scale with O(N^2). It appears that thread contention for memory hot-spots scales with O(N) on this architecture. The wait-notify barrier is approximately three times slower than the naive version.

Using `synchronized` is not the most suitable way to overcome the current problems with the Java Memory Model. The contention caused slows down code significantly, particularly for a large number of threads, and for situations where threads are required to busy-wait. `wait-notify` is a safe contention-free wake-up method, but it is still slow.

To gain more of an understanding of synchronisation, it would be advantageous to test out our implementations of the barriers on a number of different machines—particularly those with different architectures and different memory consistency models—and also on a number of different Java Virtual Machines.

Different barrier wake-up stages should be tested for those barriers which require it: the central barrier, software tree barrier and tournament barriers. For example, the wakeup-tree algorithm. The `wait-notify` wake-up

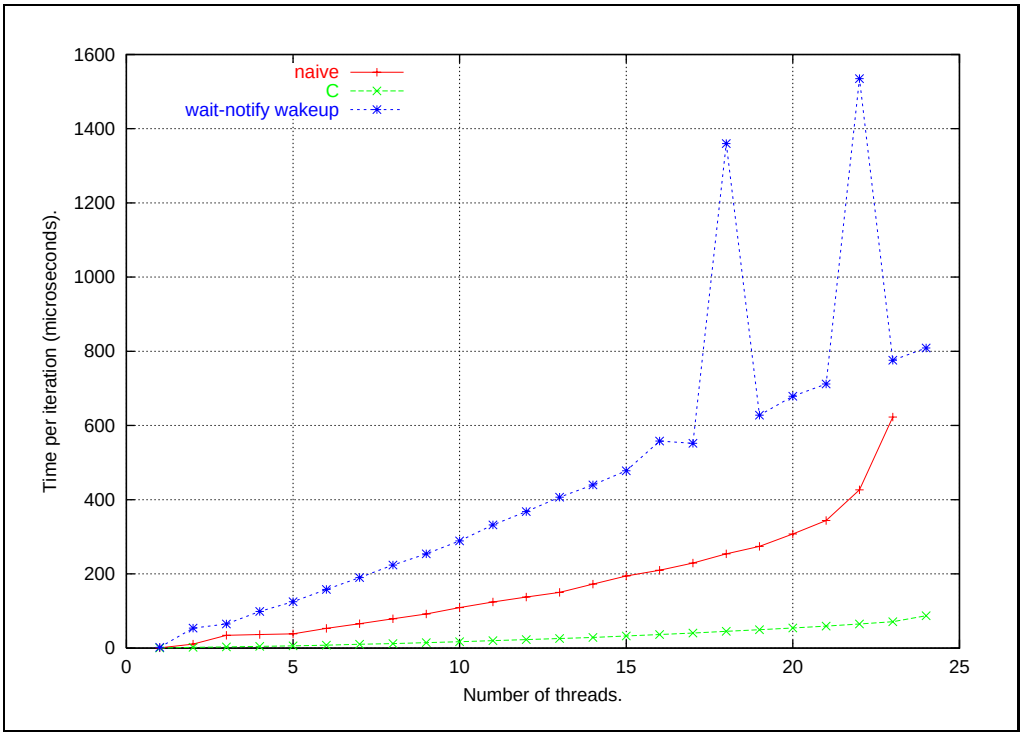

Figure 21: Times of naive and wait-notify Java and C implementations of the Central Barrier.

should also be tested for the rest of the barriers.

References

[1] "The OpenMP Homepage", http://www.openmp.org/

[2] "OpenMP C and C++ Application Program Interface", Version 1.0, October 1998.

[3] "OpenMP Fortran Application Program Interface", Version 2.0, November 2000.

[4] "Java Language Specification", Chapter 17. http://java.sun.com/

[5] "How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor", Leslie Lamport, IEEE Transactions on Computers, 1993.

[6] "The Power of Processor Consistency", M. Ahamad, R. A. Bazzi, R. John, P. Kohli and G. Neiger. In Proceedings of the Fifth ACM Symp. on Parallel Algorithms and Architectures, 1993.

[7] "The Java Grande Forum Homepage."
http://www.javagrande.org/

[8] "Java Grande at EPCC."
http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/

[9] "A Multithreaded Java Grande Benchmark Suite", L. A. Smith and J. M. Bull, Edinburgh Parallel Computing Centre, Edinburgh University, June 2001.

[10] "The Butterfly Barrier", Eugene D. Brooks III. International Journal of Parallel Programming, Vol. 15, No. 4, 1986.

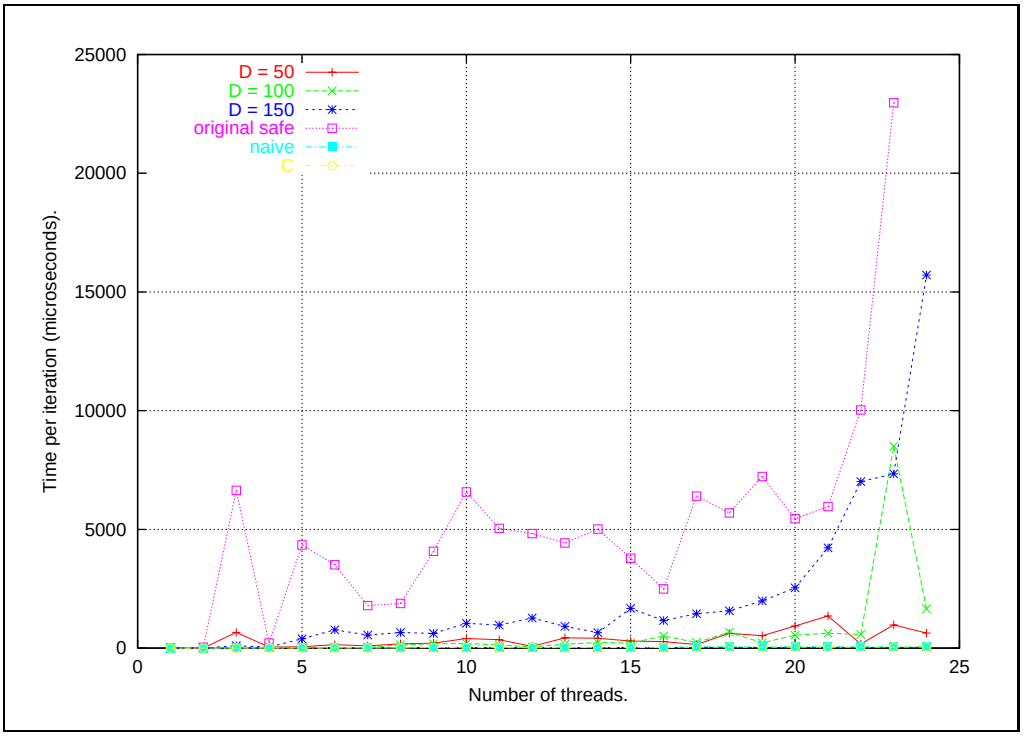

Figure 22: Times of all safe and naive Java and C implementations of the Butterfly Barrier.

[11] “Two Algorithms for Barrier Synchronization”, Debra Hengsen, Raphael Finkel, Udi Manber. International Journal of Parallel Programming, Vol. 17, No. 1, 1988.

[12] “Distributing Hot Spot Addressing in Large Scale Multiprocessors”, P.C. Yew, N.F. Tzeng and D.H. Lawrie. IEEE Trans. on Computers, Vol. C-36, No. 4, pp. 388-395, April 1987.

[13] “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”, John Mellor-Crummey and Michael Scott. ACM Transactions on Computer Systems, Vol. 9, No. 1, February, 1991, Pages 21-65.

[14] “Efficient Barriers for Distributed Shared Memory Computers”, Dirk Grunwald and Suvas Vajracharya. Technical Report CU-CS-703-94-93, Department of Computer Science, University of Colorado at Boulder, 1993.

[15] “The Java Memory Model is Fatally Flawed”, William Pugh. Dept. of Computer Science. Univ. of Maryland, College Park. http://www.cs.umd.edu/ pugh/java/memoryModel/

[16] “Fixing the Java Memory Model”, William Pugh. Dept. of Computer Science. Univ. of Maryland, College Park. http://www.cs.umd.edu/ pugh/java/memoryModel/

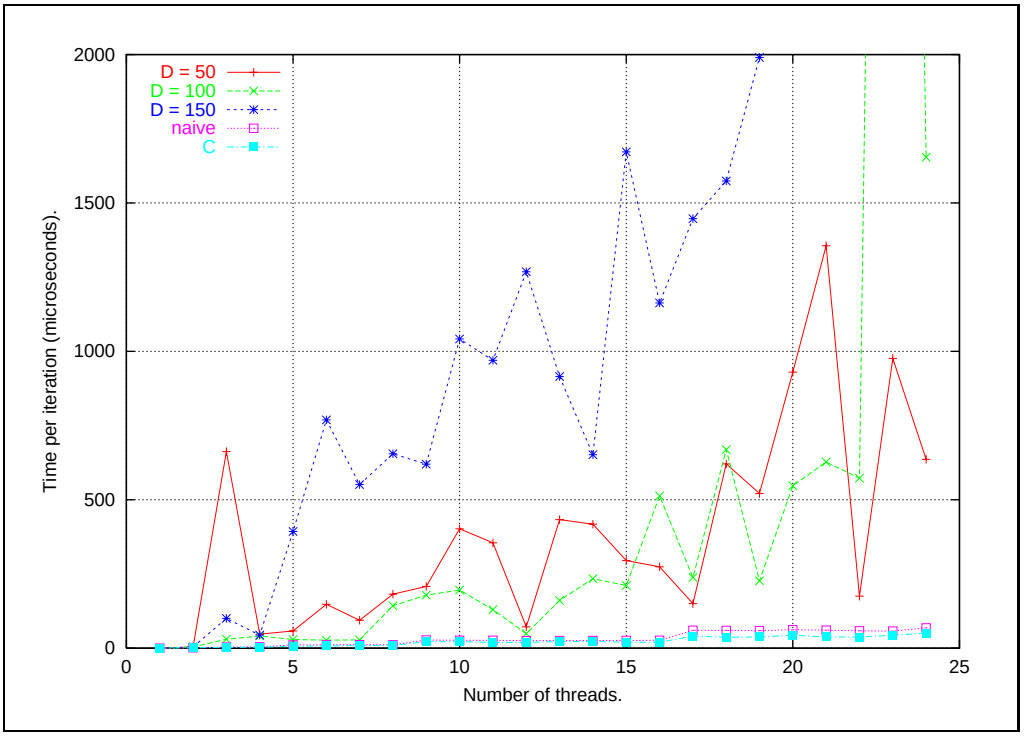

Figure 23: Times of all delayed safe and naive Java and C implementations of the Butterfly Barrier.

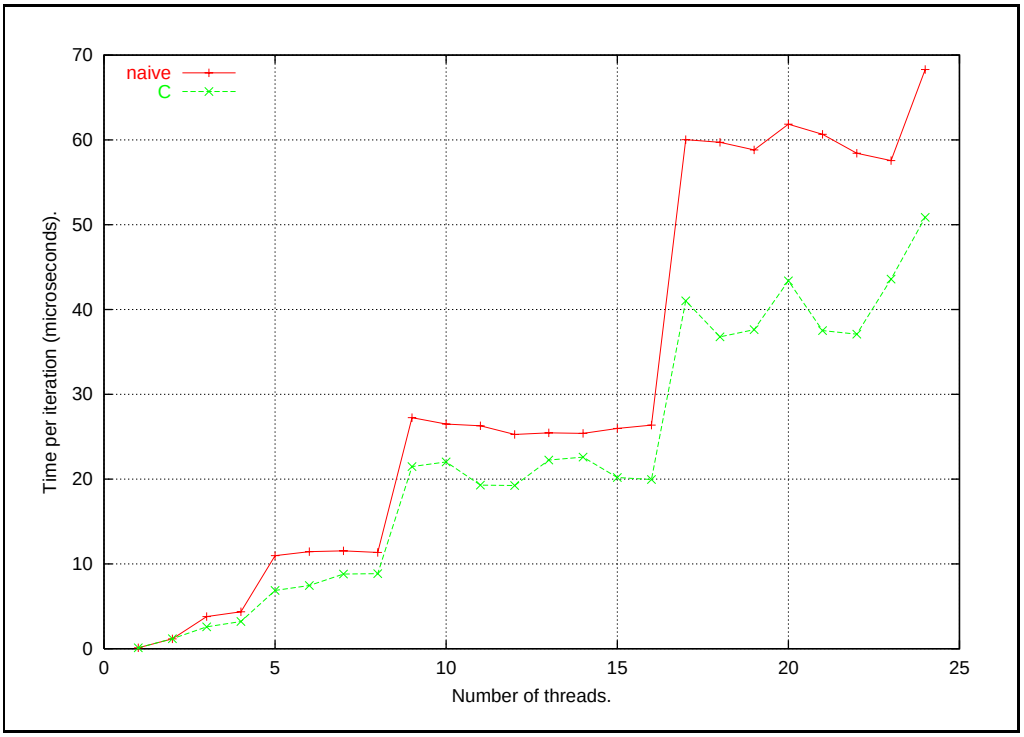

Figure 24: Times of naive Java and C implementations of the Butterfly Barrier.

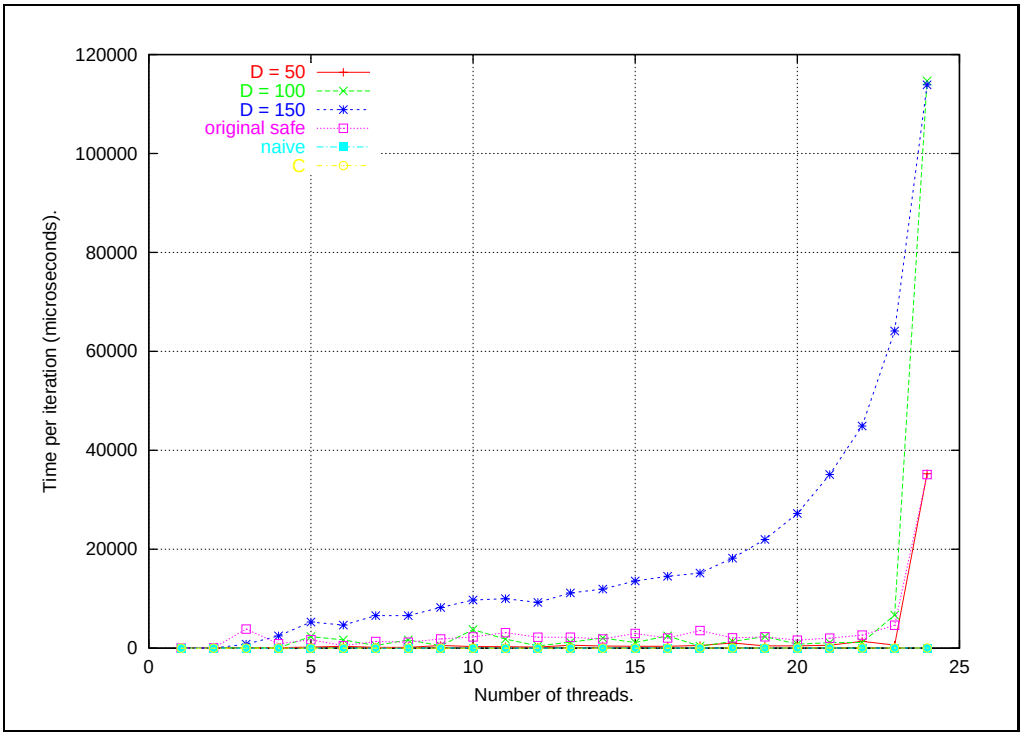

Figure 25: Times of all safe and naive Java and C implementations of the Dissemination Barrier.

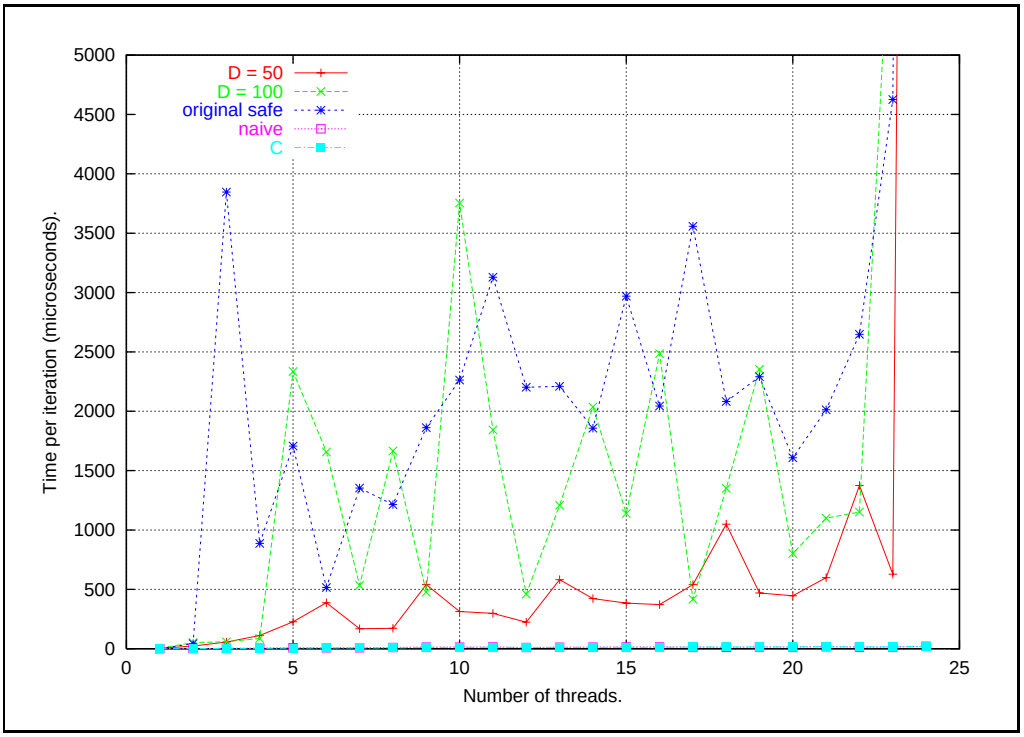

Figure 26: Times of naive, C and a number of safe Java implementations of the Dissemination Barrier.

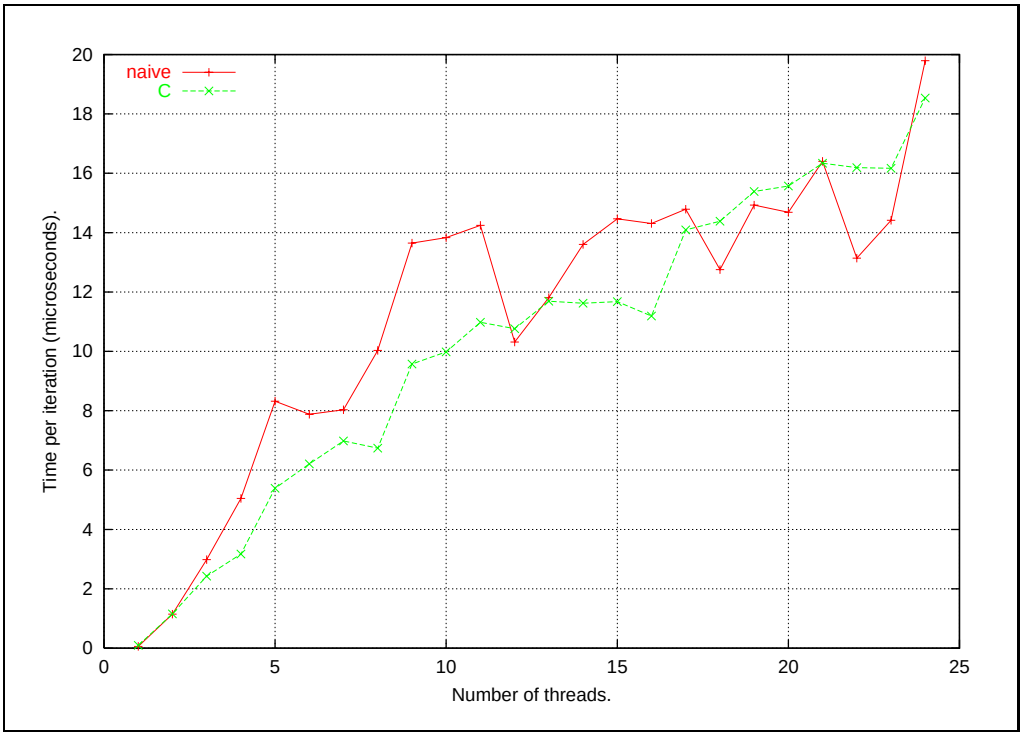

Figure 27: Times of the naive Java and C implementations of the Dissemination Barrier.

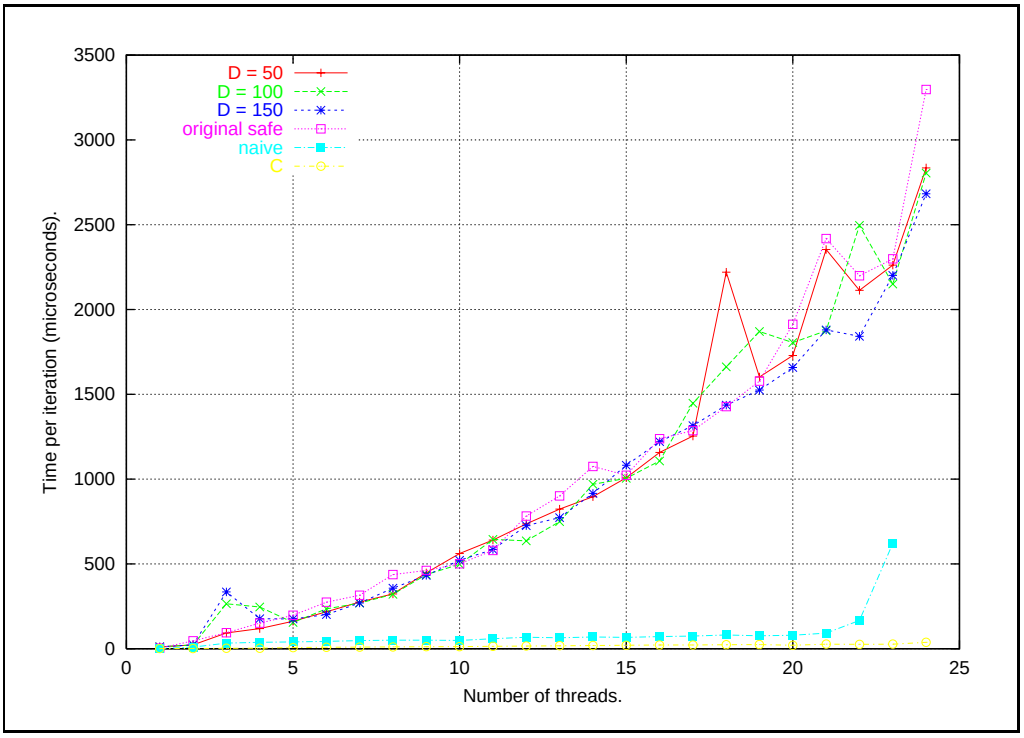

Figure 28: Times of all safe and naive Java and C implementations of the Software Tree Barrier.

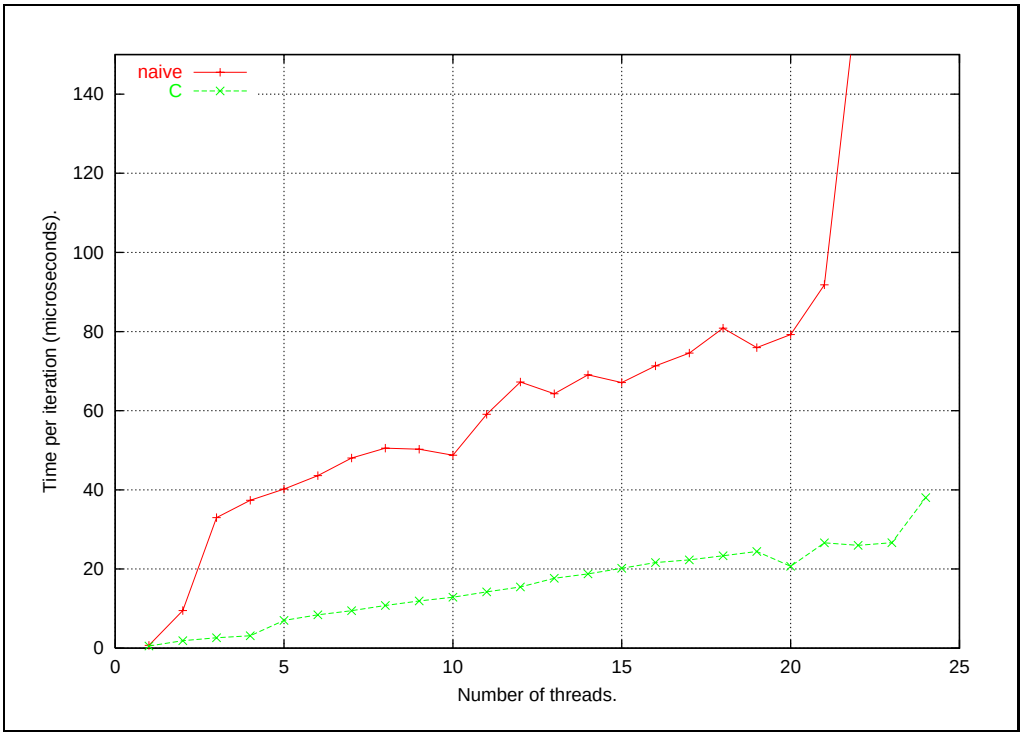

Figure 29: Times of the naive Java and C implementations of the Software Tree Barrier.

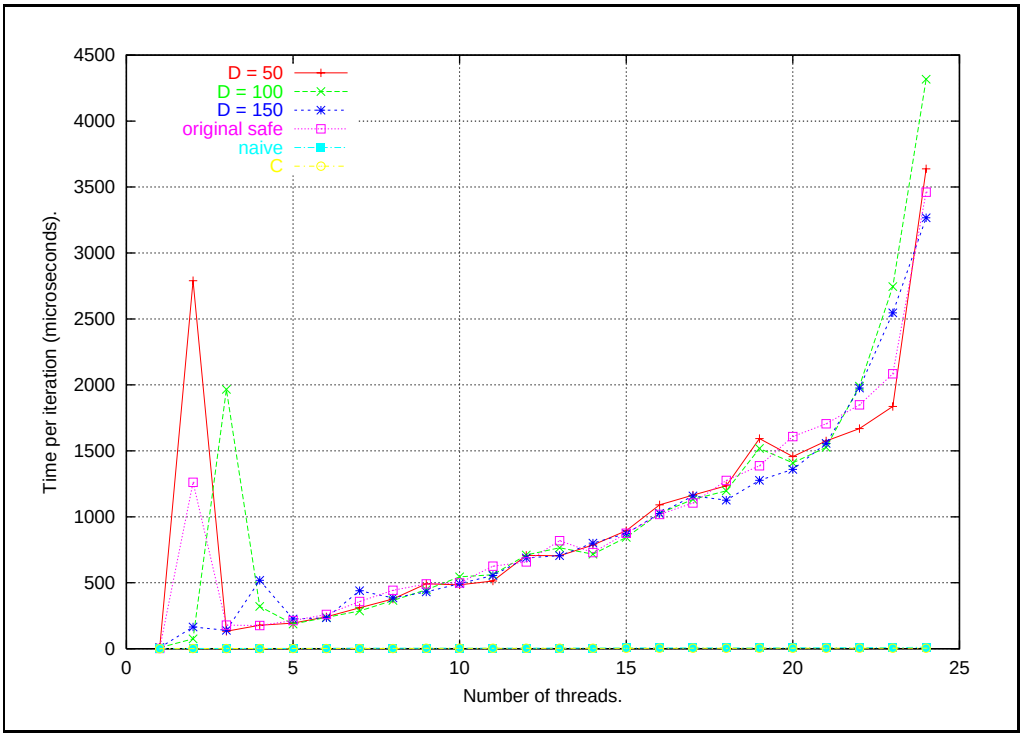

Figure 30: Times of all safe and naive Java and C implementations of the Static f-Way Barrier.

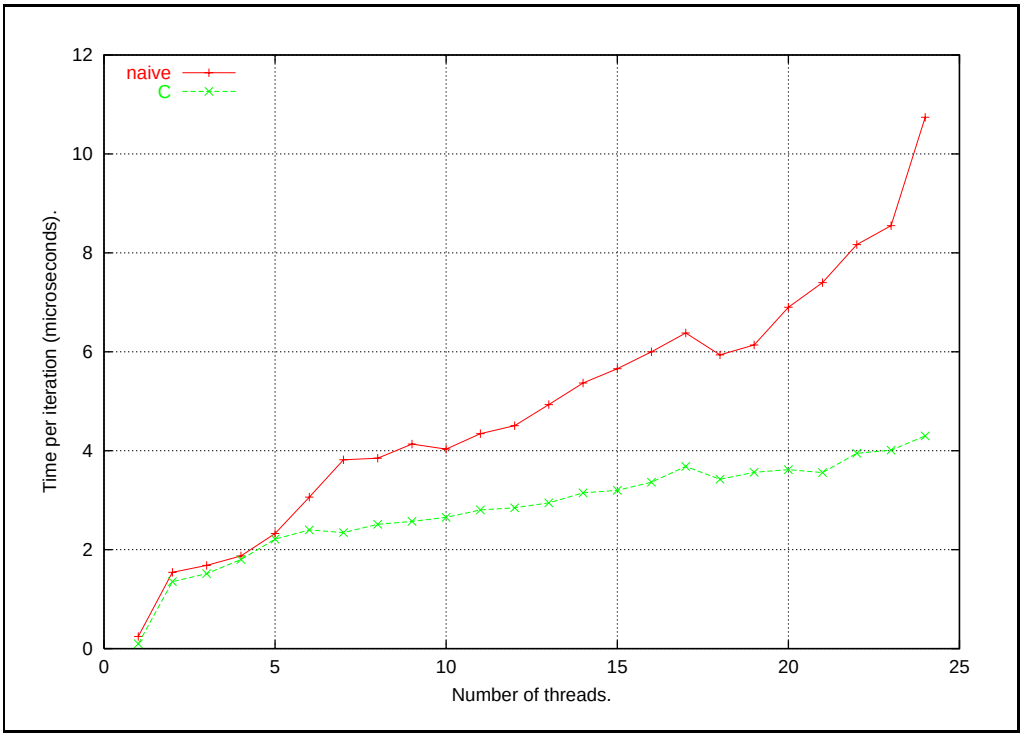

Figure 31: Times of the naive Java and C implementations of the Static f-Way Barrier.