Integrating Non-blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies[*]

Philippas Tsigas   Yi Zhang
Department of Computing Science
Chalmers University of Technology
<tsigas, yzhang>@cs.chalmers.se

Abstract

In this paper we investigate how performance and speedup of applications would be affected by using non-blocking rather than blocking synchronisation in parallel systems. The results obtained show that for many applications, non-blocking synchronisation lead to significant speedups for a fairly large number of processors, while it never slows the applications down. As part of this investigation this paper also provides a set of efficient and simple translations that show how typical blocking operations found in parallel applications, such as simple locks, queues and lock trees can be translated into non-blocking equivalents that use hardware primitives common in modern multiprocessor systems. With these translations this paper clearly demonstrates that it is easy for the application designer/programmer to replace the blocking operations commonly found on with non-blocking equivalents ones. For the empirical results a set of representative applications running on a large-scale ccNUMA machine were used.

1 Introduction

Parallel programs running on shared memory multiprocessors coordinate via shared data objects. To ensure the consistency of the shared data objects, programs typically rely on some forms of software synchronisations. Typical software synchronisation mechanisms are based on blocking that unfortunately results in poor performance because it produces high levels of memory and interconnection network contention and, more significantly, because it causes convoy effects: if one process holding a lock is preempted, other processes on different processors waiting for the lock will not be able to proceed. Researchers have introduced non-blocking synchronisation to address the above problems. However, its performance implications on modern systems or on real applications are not well understood. In this paper we study the impact of the non-blocking synchronisation on representative applications running on a large-scale ccNUMA machine: a 64 processor SGI Origin 2000. Our results show that non-blocking synchronisation lead to significant speedups for a fairly large number of processors, while, more interestingly, it never slows the applications down. We used several SPLASH-2 applications [26] with different communication requirements and some Spark98 kernels [19]. The SPLASH-2 suite of parallel applications has been developed to facilitate the study of centralised and distributed and shared-address-space multiprocessor systems and has been used extensively by the parallel architecture community. The set of Spark98 kernels is a collection of sparse matrix kernels for shared memory and message passing systems. Spark98 kernels have been developed to facilitate

[*]This work is partially supported by: i) the national Swedish Real-Time Systems research initiative ARTES (www.artes.uu.se) supported by the Swedish Foundation for Strategic Research and ii) the Swedish Research Council for Engineering Sciences.

system builders with a set of example sparse matrix codes that are simple, realistic, and portable. The identification of the key synchronisation schemes that are used in multiprocessor applications and the efficient transformation to non-blocking ones with the use hardware primitives that are commonly found in multiprocessor systems are integral parts of the study presented here.

Cache-coherent non-uniform memory access (ccNUMA) shared memory multiprocessor systems have attracted considerable research and commercial interest in the last years. Unfortunately, synchronisation is still an intrusive source of bottlenecks in many parallel programs running on shared memory multiprocessors. Synchronisation in these systems is explicit via high-level synchronisation operations like locks, barriers, semaphores, etc. The systems typically provide a set of hardware primitives in order to support the software implementation of these high-level synchronisation operations. There has been a considerable debate about how much hardware support and which hardware primitives should be provided by the systems to support software synchronisation primitives that the user can build.

1.1 Blocking vs. Non-Blocking Synchronisation

Software implementations of synchronisation constructs are usually included in system libraries. Good synchronisation library design can be challenging and as it is expected many efficient implementations for for the basic synchronisation constructs (locks, barriers and semaphores) have been proposed in the literature. Many such implementations have been designed with the aim to lower the contention when the system is in a high congestion situation. These implementations give different execution times under different contention instances. But still the time spend by the processes on the synchronisation can form a substantial part of the program execution time [9, 15, 16, 18, 29]. The reason for this is that typical synchronisation is based on blocking that introduces performance bottlenecks because of busy-waiting and convoying. Busy-waiting tends to produce a large amount of memory and interconnection network contention. The convoying effect that takes place when a process holding a lock is preempted (slows down), all other process waiting for the same lock that become unable to perform any useful work until the process that holds the locks is scheduled back. In a typical multiprocessor environment processes run for periods of time (multiprogramming environment) or, even if the machine is used exclusively, background daemons run from time to time, processes are interrupted by page faults, I/O interrupts. These events can cause the rate at which processes make progress to vary considerably. With synchronisation that is based on blocking the parallel program as a whole slows down when one process is slowed (convoying effect). To address the problems that arise from blocking researchers have proposed non-blocking implementations of shared data objects.

Non-blocking implementation of shared data objects is a new alternative approach to the problem of designing scalable shared data objects for multiprocessor systems. Non-blocking implementations allow multiple tasks to access a shared object at the same time, but without enforcing mutual exclusion to accomplish this. Since, in non-blocking implementations of shared data objects, one process is not allowed to block another process, non-blocking shared data objects have the following significant advantages over lock-based ones:

1. they avoid lock convoys and contention points (locks).
2. they provide high fault tolerance (processor failures will never corrupt shared data objects) and eliminates deadlock scenarios, where two or more tasks are waiting for locks held by the other.
3. they do not give priority inversion scenarios.

The above features of non-blocking synchronisation makes it ideal for parallel and real-time systems.

1.2 Previous and Current Work

As it was expected, non-blocking synchronisation has attracted the attention of many researchers that developed efficient non-blocking implementations for several data objects. Some studies have focused on the developing of better software algorithms, while others have identified the properties of different atomic transaction operations in terms of their synchronisation power [7]. Some evaluation studies have also been performed for specific data structure implementations. Most of these performance evaluations were using micro-benchmarks and were performed on small scale symmetric multiprocessors, as well as distributed memory machines [3, 8, 10, 11, 16] or simulators [10, 13]. Micro-benchmarks are useful since they enable easy isolation of performance issues, but the real goal of better synchronisation methods is to improve performance of real applications, which micro-benchmarks may not represent well. A substantial number of realistic scalable applications now exist. On the systems side, scalable, hardware coherent machines with physically distributed memory have become very popular for moderate to large scale computing. It is important to evaluate the benefits of non-blocking synchronisation in a range of interesting applications running on top of modern realizations of these systems. In [18] the authors assess the performance and scalability of several software synchronisation algorithms, as well as the interrelationship between synchronisation, multiprogramming and parallel job scheduling. In their evaluation, minor modifications are applied in the synchronisation code of small number of applications that spend a significant amount of time in synchronisation.

In the work presented here, we try to provide an in depth understanding of the performance benefits of integrating non-blocking synchronisation in general parallel applications. That is the reason that applications like volrent, from SPLASH-2, that do not spend a lot of time in communication were included. Simple methodologies that could transform all major lock-based synchronisations used in the applications had to be introduced. We tried to select representative applications well known and with characteristics that are well-understood. We selected applications from the SPLASH-2 shared-address-space parallel applications suit [26] and the Spark98 kernels [19].

More specifically, the main issues addressed in this paper include: i) The identification of the basic locking operations that parallel programmers use in their applications. ii) The study of the architectural support found in modern ccNUMA architectures like the SGI ORIGIN 2000 machine that could be used to support non-blocking synchronisation mechanisms. iii) The efficient translation of the lock-based synchronisation operations found in the applications to non-blocking semantically equivalent ones. iv) The experimental comparison of the lock-based and lock-free versions of the applications selected. The work presented here shows that although the applications used are optimised for parallel performance and usually perform synchronisation only when really needed — It is reasonable to expect versions of the same or similar applications to be produced by non-expert programmers with more synchronisation — the integration of non-blocking synchronisation to them lead to significant speedups for a fairly large number of processors, and more surprisingly never slowed the applications down. Another integrated result that is presented in this paper is that it is easy to replace the lock-based synchronisation operations with non-blocking equivalent ones. We think that this is a strong argument for making non-blocking synchronisation a common practice. Our results can benefit the parallel programmers in two ways. First, to understand the benefits of non-blocking synchronisation, and then to transform some typical lock-based synchronisation operations that are probably used in their programs to non-blocking ones by using the general translations that we provide in this paper. Although for our examination we used a set of applications on a 64 processor SGI Origin 2000 multiprocessor system the conclusions and the methods presented in this paper have general applicability in other realizations A preliminary and short version of this paper presented at a poster session at the ACM Conference of Sigmetrics/Performance [25].

The rest of the paper is organised as follows. Section 2 outlines the Origin 2000 architecture and its hardware support for synchronisation. Section 3 discusses the applications that we used for our evaluation. Section 4 presents the transformations that we applied in order to translate the basic blocking synchronisation operations used in these applications to non-blocking ones. In the same section we also present the experimental results. Section 5 discusses the questions addressed in this work together with the results obtained. Finally, Section 6 concludes this paper.

2 Origin 2000

The SGI Origin2000 [11] is a commercial ccNUMA machine with fast, MIPS R10000 processors [28], and an aggressive, scalable distributed shared memory (DSM) architecture. ccNUMA is a relatively new system topology that is the foundation for many next-generation shared memory multiprocessor systems. Based on "commodity" processing modules and a distributed, but unified, coherent memory, ccNUMA extends the power and performance of shared memory multiprocessor systems while preserving the shared memory programming model. ccNUMA systems maintain a unified, global coherent memory and all resources are managed by a single copy of the operating system. A hardware-based cache coherency scheme ensures that data held in memory is consistent on a system-wide basis.

2.1 The Platform

The SGI Origin2000 [11] is a scalable shared memory multiprocessing architecture, Figure 1 describes the architecture of a 64 processor machine. It provides global address spaces not only for memory, but also for the I/O subsystem. The communication architecture is much more tightly integrated than in other recent commercial distributed shared memory (DSM) systems, with the stated goal of treating a local access as simply as an optimisation of a general DSM memory reference. The two processors within a node do not function as a snoopy share memory multiprocessor cluster, but operate separately over the single multiplexed physical bus and are governed by the same, one-level directory protocol. Less snooping keeps both absolute memory latency and the ratio of remote to local latency low [11, 12], and provides remote memory bandwidth equal to local memory bandwidth (780MB/s each) [11, 12, 14]. The two processors within a node share a hardwired coherence controller called the Hub that implements the directory based cache coherence protocol.

Two nodes (4 processors) are connected to each router, and routers are connected by CrayLinks [4]. Within a node, each processor has separate 32KB first level I and D caches, and a unified 4MB second-level cache with 2 way associativity and 128 byte block size. The machine we use has Sixty four 195MHz MIPS R10000 CPUs with 4MB L2 cache and 15.5GB main memory.

2.2 Hardware Support for Synchronisation

The SGI Origin 2000 provides two transactional instructions that can be used to implement any other transactional synchronisation operation. The first instruction is the `load_linked store_conditional` instruction. The `load_linked store_conditional` is comprised by two simpler operations, the `load_linked` and the `store_conditional` one. The `load_linked` (or `LL`) loads a word from the memory to a register. The matching `store_conditional` (or `SC`) stores back possibly a new value into the memory word, unless the value at the memory word has been modified in the meantime by another process. If the word has not been modified, the store succeeds and a 1 is returned. Otherwise the, `store_conditional` fails, the memory is not modified, and a 0 is returned. The specification of this operation is shown in Figure 2.

The second hardware synchronisation mechanism is a group of `fetch_and_op` operations. The `fetch_and_op` operations are implemented at the node memory and supports at-memory atomic read-modify-write operations to special uncached memory locations. These operations are called `fetchops`

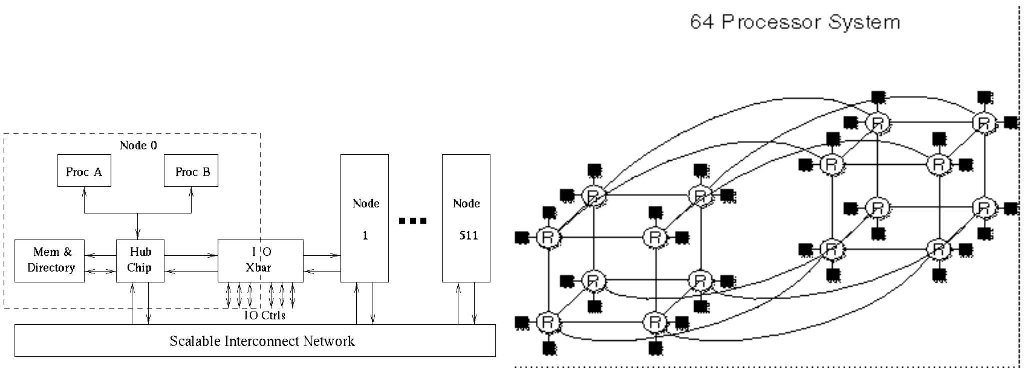


Figure 1: SGI Origin 2000 Architecture

```
LL($p_i$,$O$)                     SC($p_i$,$v$,$O$)
$Pset(O) := Pset(O) \cup \{p_i\}$      if $p_i \in Pset(O)$
return value(O)                          value(O):= v
                                          Pset(O):= $\emptyset$
                                          return true
                                      else
                                          return false
```

Figure 2: The `load_linked store_conditional` primitive

and only a few atomic operations are supported on this machine. The specification of this set of operations is shown in Figure 3. The operations that are supported in Origin 2000 include `fetch_and_and`, `fetch_and_or`, `fetch_and_increment`, `fetch_and_decrement`, `fetch_and_exchange_with_zero`. The `fetch_and_and` was first introduced by the NYU Ultracomputer Project [6]. Reads and updates of `fetchop` memory blocks require a single message in the interconnection network and do not generate coherence traffic. A shortcoming of `fetchops` is the read latency experienced by a processor that spins on an uncacheable variable; spinning on `fetchop` variables may generate significant network traffic. A second drawback of `fetchops` is that they lack the synchronisation power that operations like the `compare_and_swap`, that can atomically check the and exchange the contents of a memory location, have. `LL&SC` or `compare_and_swap` are *universal* atomic primitives, while `fetchops` are not.

```
fetch_and_op(int *address,int value)
{
   int temp;
   temp = *address;
   *address = op(temp,value);
   return temp;
}
```

Figure 3: The `fetch_and_op` primitive

For more information on the SGI Origin 2000 the reader is referred to [11, 21].

3 The Applications

Evaluating the impact of the synchronisation performance on applications is important for several reasons. First, micro-benchmarks can not capture every aspect of primitive performance. It is hard to predict the primitive impact on the application performance. For example, a lock or barrier that generates a lot of additional network traffic might have little impact on applications. Second, even in applications that spend significant time in synchronisation operations, the synchronisation time might be dominated by the waiting time due to load imbalance and serialisation in the application itself, which better implementations of locks and barriers may not be helpful in reducing. Third, micro-benchmarks rarely capture (generate) scenarios that occur in real applications.

We used all the applications from the SPLASH-2 [26] shared memory benchmark suite that use locks for synchronisation, and the kernels for shared memory machines from the Spark98 kernels suit [19]. We included the Spark98 kernels since they cover irregular applications based on sparse matrices. Such applications are at the core of many important scientific computations that simulate physical systems. The importance of such applications is likely to increase in the future.

Later in this section we briefly describe the applications that we have used. The actual descriptions of the applications can be found in [20, 22, 23, 26].

3.1 Problem Size

Problem size is a very important issue. Generally, the larger the problem size the lower the frequency of synchronisation relative to computation. On one hand, using large problem sizes will therefore make synchronisation operations seem less important. On the other hand, small problem sizes might result in very low speedup making them uninteresting on a machine of this scale. Because we wanted to make the evaluation on realistic problem sizes for this machines, we selected significant problem sizes that do not favour synchronisation, but still as we will show later the improvements were big in many applications. Figure 4 shows the inputs that we used for each of the applications.

Application	Input
Ocean	1026
radiosity	largeroom
volrend	256x256x126
spark98	sf5.1.pack
water-spatial	1331 molecules
water-nsquared	1331 molecules

Figure 4: Applications and inputs

3.2 Application Description

Ocean simulates eddy currents in an ocean basin [27]. Both its inherent and induced (at page granularity) data referencing patterns generally involve one producer with one consumer.

Volrend renders three dimensional volume data into an image using a ray-casting method [17]. The volume data are read only. Its inherent data referencing pattern on data that are written (task queues and image data) is migratory, while its induced pattern at page granularity involves multiple producers with multiple consumers. Both the read accesses to the read only volume and the write accesses to task queues and image data are fine grained, so it suffers both fragmentation and false sharing.

Radiosity computes the equilibrium distribution of light in a scene using the iterative hierarchical diffuse radiosity method [5]. The structure of the computation and the access patterns to data objects are highly irregular.

Water-Nsquared is an improved version of the Water program in SPLASH [23]. This application evaluates forces and potentials that occur over time in a system of water molecules. A process updates a local copy of the particle accelerations as it computes them, and accumulates into the shared copy once at the end.

Water-Spatial solves the same problem as Water-Nsquared, but uses a more efficient algorithm. It imposes a uniform 3-D grid of cells on the problem domain, and uses an O(n) algorithm which is more efficient than Water-Nsquared for large numbers of molecules. The advantage of the grid of cells is that processors which own a cell need only look at neighbouring cells to find molecules that might be within the cutoff radius of molecules in the box it owns. The movement of molecules into and out of cells causes cell lists to be updated, resulting in communication.

Spark98 is a collection of sparse matrix kernels for shared memory and message passing systems. Each kernel performs a sequence of sparse matrix vector product operations using matrices that are derived from a family of three dimensional finite element earthquake applications. The multiplication of a sparse matrix by a dense vector is central to many computer applications, including scheduling applications based on linear programming and applications that simulate physical systems. These applications are irregular applications based on sparse matrices. The running time of these applications is dominated by a sparse matrix-vector product (SMVP) operation that is repeated thousands of times, and the SMVP is the only operation besides I/O that requires the transfer of data between processors.

4 Typical Lock-Based Synchronisation Operations and Their Translations to Non-blocking Ones

In all parallel applications that we looked at, the most frequent use of a lock is to protect a single global shared variable while a process first reads the variable, then it performs the operation on the number it has just read and finally it writes the number back to the variable (the Read-Modify-Write problem). These shared variables are used in the application programs to either: i) assign consecutive values to a set of processes, or ii) to sum up values computed by processes of the system, or iii) as simple indexes of arrays.

We call this kind of locks *SimpleLocks*. It is easy to observe that such a lock can be replaced with the respective `fetch_and_op` operation to achieve the same functionality without enforcing locking, if the variable protected by the lock is of integer type. One shortcoming of the `fetch_and_op` operations is that they do not provide support for floating point numbers. In high performance scientific computing though, computations based on floating point numbers are very common. In order to overcome this shortcoming of the hardware, an efficient software implementations for the `fetch_and_op` that could supports floating point numbers was developed. For the rest of the paper we will refer to these `fetch_and_op` operation that can support floating point numbers as `double_fetch_and_op` (denoted DFAD also) operations. As a building block in our implementation we used the `load_link` and `store_conditional` primitives. The specification of the new `double_fetch_and_add` operation is given in Figure 5.
Now, with the help of the FAD (`fetch_and_add`) and DFAD (`double_fetch_and_add`) operations we can remove all *SimpleLock*s in any parallel application. As it is going to become clear in the next subsection the big majority of the locks that we found are *SimpleLock*s.

```
double_fetch_and_add(double *address,double value)
{
   double temp;
   temp = *address;
   *address = temp + value;
   return temp;
}
```

Figure 5: The `double_fetch_and_add` primitive

```
do
{
  temp = LL(multi->err_multi);
  if (local_err > temp)
     rtn = SC(multi->err_multi, local_err);
}
while(rtn == TRUE)
```

Figure 6: Lock-free implementation of the conditional update of `error_lock`

4.1 The Applications and Their Synchronisation

In this subsection, we describe the different lock-based synchronisation operations that are used in the applications that we examine, together with our transformations that transform them to non-blocking ones with the same functionality.

In the **OCEAN** application 4 different locks are used:

* `idlock` is a *SimpleLock* that protects the global variable `index`.

* `psiailock` is also a *SimpleLock* that protects the global variable `psiai` that carries floating point numbers.

* `psibilock` is also a *SimpleLock* that protects the global variable `psibi` that carries floating point numbers.

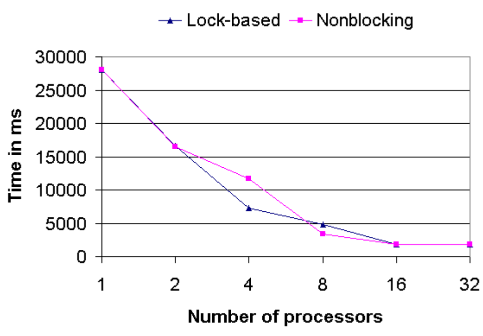
* `error_lock` on the other hand is not a *SimpleLock*, and, it protects the global variable `err_multi`. The use of `err_multi` is describe below.

We replaced the first three of these locks with FAD or DFAD operations using the methods described before in this section. The fourth lock (`error_lock`) protects a global variable which is updated conditionally as follows:

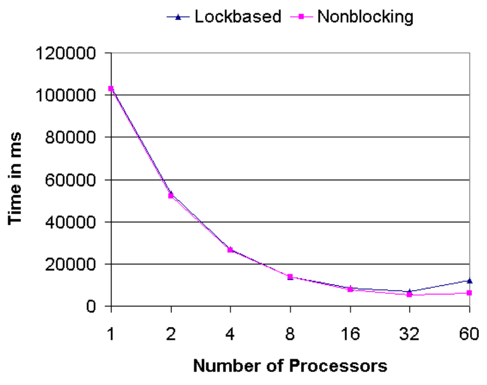
```
LOCK(locks->error_lock)
if (local_err > multi->err_multi) {
  multi->err_multi = local_err;
}
UNLOCK(locks->error_lock)
```

For this lock we had to implement a non-blocking synchronisation with the same functionality to replace it, in our implementation we used the `load_link` and `store_conditional` primitives. Figure 6 describes our simple implementation.

Figure 7(a) shows performance results for the original version and the modified non-blocking version of the OCEAN application. Because the ocean application requires the number of processes to be power of 2, we could only do the experiments for up to 32 processors. For this particular application we do not observe any significant improvement after the modification, but, we also notice that the non-blocking synchronisation do not hamper it's performance. Ocean is a regular application with very regular communication patterns.



(a) OCEAN



(b) RADIOSITY

Figure 7: Performance results: OCEAN and RADIOSITY

In radiosity a scene is initially modelled as a number of large input polygons. Light transport interactions are computed among these polygons. and polygons are hierarchically subdivided into patches as necessary to improve accuracy. In each step, the algorithm iterates over the current interaction lists of patches, subdivides patches recursively, and modifies interaction lists as necessary. At the end of each step, the patch radiosities are combined via an up-ward pass through the quad-trees of patches to determine if the overall radiosity has converged. The main data structures represent patches, interactions, interaction lists, the quad-tree structures, and a BSP tree which facilitates efficient visibility computation between pairs of polygons. Parallelism is managed by distributed task queues, one per processor, with task stealing for load balancing.
Radiosity uses 11 different locks:

* `index_lock` is a *SimpleLock* that protects the variable `index`.
* `bsp_tree_lock` is a lock that protects the `bsp_tree` structure.
* `pbar_lock` is a lock that protects the global variable `pbar_counter`.
* `task_counter_lock` is a lock that protects the global shared variable `task_counter`.
* `free_patch_lock` is the lock that protects the global shared data object `Patch` that is implemented as a queue where free "patches" are queued.
* `free_element_lock` is the lock that protects the global shared data object `Element`. `Element` is implemented as a queue where processes queue free "elements".
* `free_interaction_lock` is the lock that protects the global shared data object `Interaction`. `Interaction` is a queue structure where "interactions" are queued.
* `free_elemvertex_lock` is the lock that protects the global shared data object `Elemvertex`. `Elemvertex` is also implemented as a queue where "free elements" are stored.
* `free_edge_lock` is the lock that protects the global shared data object `Edge`. `Edge` is also a queue structure where "free edges" are queued.
* `avg_radiosity_lock` is a lock that acts as a barrier that processes can use in order to determine.
* `q_lock` protects the task queue.

The `bsp_tree` is protected by the `bsp_tree_lock` that has also a tree structure. We used the `compare_and_swap` (CAS) atomic operation to implement a non-blocking version of the `bsp_tree`. The specification of the `compare_and_swap` primitive is shown in Figure 9. For the SGI Origin 2000 system we had to emulate the `compare_and_swap` atomic primitive with the `load_linked` `store_conditional` instruction; this implementation is shown in Figure 10.

```
do
{
  ...  ...
  traversal the tree to find the leaf to add the node
  ...  ...
}
while(CAS(leaf's address, NULL, node))
```

Figure 8: Non-blocking operations on `bsp_tree`

In the program, nodes are only added to the `bsp_tree` and they are never deleted from it. Moreover, there is no operation that can change the position of a node that is already in the tree. New nodes are added as leaves. Because of these special properties of the `bsp_tree`, we do not face the ABA problem that most non-blocking protocols that use `compare_and_swap` have to phase. The ABA problem arises when a process p reads the value A from a shared memory location, computes a new value based on A, and using `compare_and_swap` updates the same memory location after checking that the value in this memory location is still A and mistakenly concluding that there was no operation that changed the value to this memory location in the meantime. But between the read and the `compare_and_swap` operation, other processes may have changed the context of the memory location from A to B and then back to A again. Our lock-free implementation for the `bsp_tree` is described in Figure 8.

```
compare_and_swap(int *mem, register old, new) {
   temp = *mem;
   if (temp == old) {
       *mem = new;
       new = old;
   } else
       new = *mem;
}
```

Figure 9: The `compare_and_swap` primitive

```
compare_and_swap(int *mem, register old, new) {
   do {
    temp = LL(mem);
    if  (temp != old)  return FALSE;
   }while(!SC(mem,new));
   return TRUE;
}
```

Figure 10: Emulating `compare_and_swap` with `load_linked store_conditional`

The variable `pbar_counter` is a counter that counts the number of working processors. It also emulates the behaviour of a barrier; when there is no processor working, the program will exit the current iteration and will check the radiosity convergence to determine whether to continue the iterations or not. We used the FAD operation to replace the locks, in this way we achieved the same functionality without using locks.

The `task_counter` is used by the processes to determine the task that enters the function `check_task_counter`. We implement this counter in a lock-free manner using the `compare_and_swap` primitive, our implementation is shown in Figure 11.

```
check_task_counter(process_id)
{
    do {
        tempold = global->task_counter;
        tempnew = (tempold + 1) % n_processors;
    } while( CAS(global->task_counter, tempold, tempnew) == 0);
    flag = !tempold;
    return( flag ) ;
}
```

Figure 11: Non-blocking version of the `check_task_counter`

Data Object Name	Functionality
`free_patch`	no enqueue operations run in parallel
`free_element`	no enqueue operations run in parallel
`free_interaction`	enqueue and dequeue operations run in parallel
`free_elemvertex`	no enqueue operations are running in parallel
`free_edge`	no enqueue operations are running in parallel
`task_queue`	enqueue and dequeue operation are running in parallel

Figure 12: Data objects in **Radiosity**

The remaining shared data objects that are protected by locks (`free_patch`, `free_element`, `free_interaction`, `free_elemvertex`, `free_edge`, `task_queue`) are implemented as queues. Figure 12, describes some special properties of these queues.

We used the non-blocking queue implementation presented in [24], to replace the lock-based implementations for the queue based shared objects mentioned before.

Figure 7(b) shows the performance of our non-blocking version comparing with original one. There is no big difference between the two versions until we reach 32 processors where synchronisation becomes a significant part of the total computing time. With 32 processors, the non-blocking version is about 34% faster than the lock-based one and as the number of processors increases the improvement on the performance also increase reaching a 93% better performance when using 60 processors, the maximum number of processors that we could use exclusively for running this application. The access patterns to shared data structures in Radiosity are highly irregular, as we mentioned in the previous section.

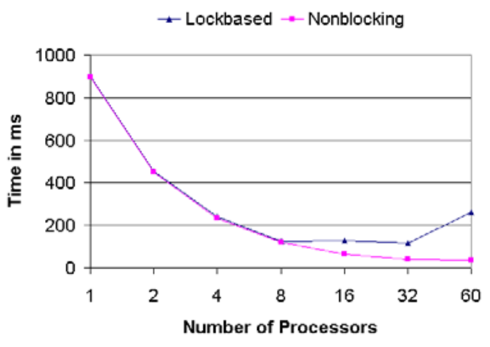
Volrend in contrast with radiosity does not use many locks. It uses only two *SimpleLock*s and an array lock. These locks are described below:

* `IndexLock` is a *SimpleLock* that protects the shared variable `index`.

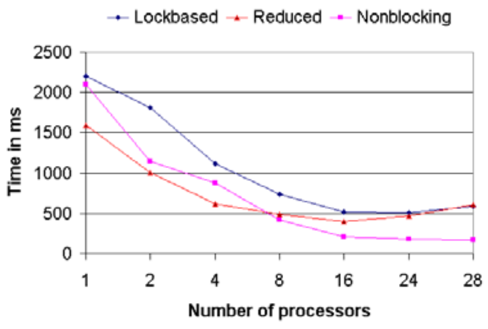
* `CountLock` is a *SimpleLock* that protects the shared variable `Counter`.

* `QLock` is an array lock used to protect a global queue. The global queue is implemented as an array. The protection is on the index of the array. As there is only one arithmetic operation, we used a normal `fetch_and_add` to translate it into a non-blocking one.

Figure 13(a) shows the performance of our non-blocking version comparing with original one. The performance advantage of the non-blocking version starts to show as the number of processors becomes greater than 8. The performance of the non-blocking one is close to optimal since its speed up is very close to the theoretical limit. **Volrend's** inherent data referencing pattern on data that are written (task queues and image data) is migratory, while its induced pattern at page granularity involves multiple producers with multiple consumers.



(a) VOLREND



(b) SPARK98

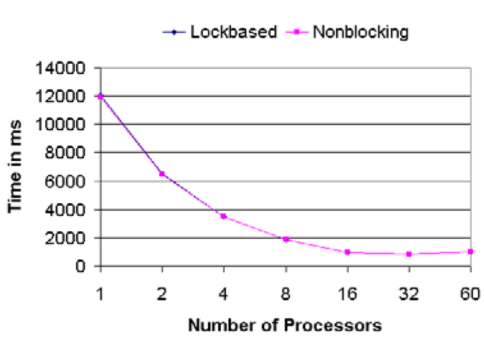
Figure 13: Performance results: VOLREND and SPARK98

From the **Spark98** kernel we used the shared memory applications, the *lmv* and the *rmv*. The *lmv* is a parallel shared memory program based on locks. The *rmv* is a parallel shared memory program based on a reduction of the number of locks that are used in *lmv*. Based on the naming schemes that the developers of **Spark98** have used, we named our version *nmv*. In order to create this non-blocking version we used the *lmv* version from the kernel. All locks in this program are *SimpleLock*s and they handle floating point numbers. Due to the limited time for exclusive use that we had we performed the experiments for up to 28 processors for this application. The results, graphically shown in Figure 13(b), clearly show the power of non-blocking synchronisation for unstructured applications like this one. The speedup of *rmv* and *lmv* stop when we go above 16 processors while *nmv* scales uniformly. This allows us to conjecture that non-blocking will dramatically increase the performance of these applications as the number of processors increases.
In **Water-nsquared** although 10 different locks are defined, only 7 are used. These 7 are described bellow:

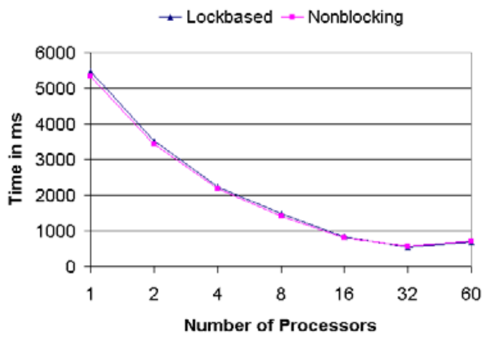
* `IndexLock` is a *SimpleLock* that protects the global variable `Index`
* `IntrafVirLock` is a *SimpleLock* that protects the global variable `VIR` when computing the intra-molecular force/mass acting.
* `InterfVirLock` is a *SimpleLock* that protects the variable `VIR` when computing the inter-molecular force.
* `KinetiSumLock` is a *SimpleLock* that protects the array `SUM`
* `PotengSumLock` is a *SimpleLock* that protects the variables `POTA, POTR, POTRF`.
* `MolLock`, is an array of locks, all of them are *SimpleLock*s and they are used in order to update the force on all molecular.
* `IOLock` is a special lock that is used for I/O control.
We used the implementations described in the previous subsection in order to replace all *SimpleLock*s.

Water-spatial uses 7 different locks. Five of these are *SimpleLock*s, the first five *SimpleLock*s that are listed in the **Water-nsquared** above (`IndexLock, IntrafVirLock, InterfVirLock, KinetiSumLock, PotengSumLock`). We used the implementations described in the previous subsection in order to replace all *SimpleLock*s.

In **Water-nsquared** and **Water-spatial** the communication and the sharing of the data is very simple: A process updates a local copy of the particle accelerations as it computes them, and accumulates into the shared copy once at the end. This simple communication pattern does not give the opportunity to lock-free synchronisation to show it's power. On the other hand, the experiments show that lock-free synchronisation does not harm the performance of the applications. The lock-free versions of both applications perform as well as the respective lock-based ones.



(a) WATER-NSQUARED



(b) WATER-SPATIAL

Figure 14: Performance results: WATER-NSQUARED and WATER-SPATIAL

Figure 15 summarises our experimental results. It graphically shows the maximum speedup of the lock-free and the respective lock-based implementation for each of our implementations.

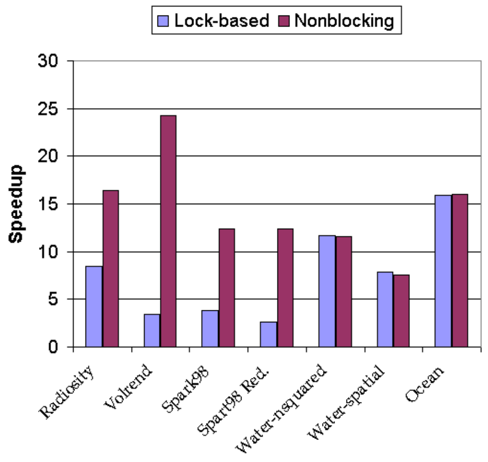


Figure 15: Speedup for the non-blocking and the original versions

5 Discussion

There has been much advocacy arising from the theory community for the use of non-blocking synchronization primitives, rather than blocking ones, in the design of inter-process communication mechanisms for parallel and high performance computing. This advocacy is intuitive, but has not been investigated on top of real and well-understood applications; such an investigation could also reveal the effectiveness of non-blocking synchronization on different applications. There has been a need for an investigation of how performance and speedup in parallel applications would be affected by using non-blocking rather than blocking synchronization primitives. From our interaction with practitioners we could definitely conclude that one of the main reasons why non-blocking synchronization has not become popular among them is the lack of such an investigation. One other significant reason is that many non-blocking synchronization mechanisms are quite complex. In this paper we want to address in an effective way this issue, by performing, a fair evaluation of non-blocking synchronization in the context of well-established parallel benchmark applications.

The results in this paper come to support the general belief of people working in the research area of non-blocking synchronization that advocated that non-blocking synchronization can lead to better performance in the context of parallel applications. They clearly show that applications that have irregular communication patterns and spend significant part of their execution time on communication can benefit a lot from non-blocking synchronization. Something that was not that clear from the beginning, was that the performance of applications with regular communication patterns that do not generate congestion are not going to be affected negatively from the introduction of non-blocking synchronization. Our intuition on this was not that clear, since, lock-based synchronization performs usually better that non-blocking synchronization when contention is very low. Since this work was aiming at clarifying the practical aspects of non-blocking synchronization, we also wanted to demonstrate that it is easy to replace the blocking operations with non-blocking equivalents and get the benefits of non-blocking synchronization, which is a strong argument for making non-blocking synchronization common practice. As part of this investigation, this paper also provides a set of efficient and simple translations that show how typical blocking operations found in parallel applications, such as simple locks, queues and lock trees can be translated into non-blocking equivalents that use hardware primitives common in modern multiprocessor systems.

We believe that our work is a first step and more experiments are needed to reveal the effectiveness of non-blocking synchronization on different applications. But we think that it is a fair evaluation of the proposed non-blocking primitives in the context of well-established parallel benchmark applications.

6 Conclusion

The main conclusions of our study are the following:

- For the fairly wide range of applications examined, non-blocking synchronisation performs as well, and often better than the respective blocking synchronisation.
- For certain applications, the use of non-blocking synchronisation yields great performance improvement. Figure 15 shows graphically the maximum speedup of the lock-free and the respective lock-based implementation for each of our implementations. With 60 processors, the non-blocking version of radiosity is about two times faster than the lock-based one; non-blocking Volrend is about 7 times faster that the lock based one.
- Irregular applications benefit the most from non-blocking synchronisation. Since the importance of such applications is likely to increase in the future, the importance of lock-free synchronisation in high-performance parallel systems is also expected to increase.
- The methods that we introduce to remove lock based synchronisations are quite simple and can be used in any parallel application.

Acknowledgements

We are grateful to Carl Hallen, Andy Polyakov and Paul Waserbrot, they made the impossible possible and at the end we could have exclusive access to our heavily (thanks to our physics department) loaded ORIGIN 2000. Many thanks to Marina Papatriantafilou for her always helpful comments.

References

[1] H. Bao, J. Bielak, O. Ghattas, L. F. Kallivokas, D. R. O'Hallaron, J. R. Shewchuk and J. Xu, Earthquake Ground Motion Modelling on Parallel Computers, in Proceedings of Supercomputing'96, IEEE, November 1996.

[2] A. Brandt, Multi-level adaptive solutions to boundary-value problems, Mathematics of Computation 31(138), pp. 333-390,1977.

[3] A. Eichenberger and S. Abraham, Impact of Load Imbalance on the Design of Software Barriers, in Proceedings of the 1995 International Conference on Parallel Processing, pp. 63-72, August 1995.

[4] M. Galles, Scalable Pipelined Interconnect for Distributed Endpoint Routing: The SGI Spider Chip,in Proceeding Hot Interconnects IV, pp. 141-146, 1996.

[5] P. Hanrahan and D. Salzman, A Rapid Hierarchical Radiosity Algorithm, in Proceeding of SIGGRAPH, pp. 197-206,1991.

[6] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer - Designing a MIMD Shared-Memory Parallel Machine", *IEEE Trans. on Computers*, 32(2), p. 175, February 1983.

[7] M. Herlihy, Wait-Free Synchronization, ACM Transactions on Programming Languages and Systems, 13(1), pp. 124-149, January 1991.

[8] D. Jiang and J. Singh, A Methodology and an Evaluation of the SGI Origin2000, in Proceedings of ACM SIGMETRICS 1998, pp. 171-181.

[9] A. Karlin and K. Li and M. Manasse and S.Owicki, Empirical studies of competitive spinning for a shared-memory multiprocessor, in Proceedings of the 13th ACM Symposium on Operating Systems Principles, pp. 41-55, October 1991.

[10] A. Kägi, D. Burger and J. Goodman, Efficient Synchronization: Let Them Eat QOLB, in Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97), pp. 170-180, ACM Press, June 2-4 1997.

[11] J. Laudon and D. Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server, in Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97), Computer Architecture News, Vol. 25,2, pp. 241-251, ACM Press, June 2-4 1997.

[12] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and John Hennessy, The DASH prototype: Logic overhead and performance, IEEE Transactions on Parallel and Distributed Systems, 4(1), pp. 41-61, January 1993.

[13] B. Lim and A. Agarwal, Reactive Synchronization Algorithms for Multiprocessors, in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), pp. 25-35, October 1994.

[14] T. Lovett and R. Clapp, STiNG : A CC-NUMA Computer System for the Commercial Marketplace,in Proceedings of the 23rd Annual International Symposium on Computer Architecture, pp. 308-317, ACM Press, May 22-24 1996.

[15] J. M. Mellor-Crummey and M. L. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, ACM Trans. on Computer Systems, 9(1), pp. 21-65 February 1991.

[16] M. M. Michael and M. L. Scott, Nonblocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors, Journal of Parallel and Distributed Computing 51(1), pp. 1-26, 1998.

[17] J. Nieh and M. Levoy, Volume Rendering on Scalable Shared Memory MIMD Architectures, in Proceeding of the 1992 Workshop on Volume Visualization, pp 17-24, October 1992.

[18] D. S. Nikolopoulos and T. S. Papatheodorou, A Quantitative Architectural Evaluation of Synchronization Algorithms and Disciplines on ccNUMA Systems: The Case of the SGI Origin2000, in Proceedings of the 1999 Conference on Supercomputing, ACM SIGARCH, pp. 319-328, June 1999.

[19] D. R. O'Hallaron, Spark98: Sparse Matrix Kernels for Shared Memory and Message Passing Systems, Technical Report CMU-CS-97-178, October 1997.

[20] E. Rothberg, J. P. Singh and A. Gupta, Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors, in Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 14-26, IEEE Computer Society Press, May 1993.

[21] SGI, SGI TechPubs Library, http://techpubs.sgi.com/, 2000.

[22] J. P. Singh, A. Gupta and Marc Levoy, Parallel Visualization Algorithms: Performance and Architectural Implications, Computer, 27(7), pp. 45-55, July 1994.

[23] J. P. Singh, W. D. Weber and Anoop Gupta, SPLASH: Stanford Parallel Applications for Shared-Memory, Computer Architecture News, 20(1), pp. 2-12, March 1992.

[24] P. Tsigas and Y. Zhang, A Simple, Fast and Scalable Non-Blocking Concurrent FIFO queue for Shared Memory Multiprocessor Systems, in Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures, pp. 134-143, July 2001.

[25] P. Tsigas and Y. Zhang, Evaluating The Performance of Non-Blocking Synchronisation on Shared-Memory Multiprocessors (Poster Paper), in Proceedings of SIGMETRICS 2001, June 2001.

[26] S. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, in Proceedings of the 22nd International Symposium on Computer Architectures, pp. 24-36, June 1995.

[27] S. C. Woo, J. P. Singh and J. L. Hennessy, The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors, in Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 219-229, October 4-7, 1994.

[28] K. Yeager, The MIPS R10000 superscalar microprocessor, IEEE Micro, 16(2), pp. 28-40, April 1996.

[29] J. Zahorjan and E. D. Lazowska and D. L. Eager, The effect of scheduling discipline on spin overhead in shared memory parallel systems, IEEE Transactions on Parallel and Distributed Systems, 2(2), pp. 180-198, April 1991.