Evaluating The Performance of Non-Blocking Synchronisation on Modern Shared-Memory Multiprocessors*

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

Abstract

Parallel programs running on shared memory multiprocessors coordinate via shared data objects/structures. To ensure the consistency of the shared data structures, programs typically rely on some forms of software synchronisations. Unfortunately typical software synchronisation mechanisms usually result in poor performance because they produce large amounts of memory and interconnection network contention and, more significantly, because they produce convoy effects that degrade significantly in multiprogramming environments: 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 are not well understood on modern systems or on real applications. In this paper we study the impact of the non-blocking synchronisation on parallel applications running on top of a modern, 64 processor, cache-coherent, shared memory multiprocessor system: the SGI Origin 2000. In addition to the performance results on a modern system, we investigate the key synchronisation schemes that are used in multiprocessor applications and their efficient transformation to non-blocking ones.

1 Introduction

Cache-coherent non-uniform memory access (ccNUMA) shared memory multiprocessor systems have attracted considerable research and commercial interest in the last years. They are becoming increasingly popular for running parallel programs and are considered to be the foundation of the next generation shared memory multiprocessors. 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 what hardware primitives should be provided by the systems to support software synchronisation primitives that the user can build. Software implementations of synchronisation constructs are usually included in system libraries. Good

*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.

synchronisation library design can be quite challenging and as it is expected many efficient implementations for locks, barriers and semaphores have been proposed in the literature. Researchers in the field first designed different lock and semaphore implementations that lower the contention when the system is in a high congestion situation, and they 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 [8, 14, 15, 17, 27]. The reason for this is that in any shared memory parallel systems concurrent processes that have been created either by a parallel application or by the operating system very often need to share data or become coordinated, and they do so via shared data objects/structures. To ensure consistency of the shared data structures, programs rely on some forms of software synchronisation. Typical synchronisation is based on blocking and introduces performance bottlenecks. There are two main reasons. The first is that busy-waiting tends to produce a large amount of memory and interconnection network contention. The second reason is the convoying effect that blocking synchronisation suffers from: if a process holding a lock is preempted, any other process waiting for the lock is unable to perform any useful work until the process that hold the locks is scheduled. In a typical environment we expect that the machine running our parallel program is used in a multiprogramming environment. Other processes run for periods of time 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 blocking synchronisation 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 structures.

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 structures, one process is not allowed to block another process, non-blocking shared data structures 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.

As it was expected, non-blocking synchronisation has attracted the attention of many researchers that developed efficient non-blocking implementations for several data structures. Some studies have been 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. Some evaluation studies have also been performed for specific data structure implementations [15]. Most of these performance evaluations were using micro-benchmarks and were performed on small scale symmetric multiprocessors, as well as distributed memory machines [3, 9, 10, 7] or simulators [9, 12]. 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 for this programming model. 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 [17] the authors assess the performance and scalability of several software synchronisation algorithms, as well as the interrelationship between synchronisation, multiprogramming and parallel job scheduling. The main body of their evaluation is performed with micro-benchmarks executed in dedicated and multiprogrammed environments on a 64-processor Origin2000. This is the first paper, to the best of our knowledge, that uses 3 applications from the SPLASH-2 benchmark suit, Cholemsky, Radiosity and LU, to assess the performance of synchronisation algorithms under realistic conditions. In their evaluation, i) minor modifications are applied in the synchronisation code of each application and ii) the applications selected because they spend a significant amount of time in synchronisation points. In the work presented here, continuing the work presented in [17] we try to understand how non-blocking synchronisation affects the performance of parallel applications in general, not only to ones that spend o lot of time in communication. Henceforth, i) we try to use a big set of applications with different characteristics, making sure that we include also applications that do not spend a lot of time in synchronisation, ii) we also try to modify all the lock-based synchronisation points of these applications if possible.

The goal of the work presented here is to provide an in depth understanding of how non-blocking can improve the performance of modern parallel applications. More specifically, the main issues addressed in this paper include: i) The architectural implications of the ccNUMA on the design of non-blocking synchronisation. ii) The identification of the basic locking operations that parallel programmers use in their applications. iii) The efficient non-blocking implementation of these synchronisation operations. iv) The experimental comparison of the lock-based and lock-free versions of the respective applications on a cache-coherent non-uniform memory access shared memory multiprocessor system. v) The identification of the structural differences between applications that benefit more from non-blocking synchronisation than others. We examine these issues, using a set of applications on a 64 processor SGI Origin 2000 multiprocessor system. This machine is attractive for the study because it provides an aggressive communication architecture and support for both in cache and at memory synchronisation primitives. It should be clear however that the conclusions and the methods presented in this paper have general applicability in other realizations of cache-coherent non-uniform memory access machines. 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. The contributions of this work are both in the results obtained as well as in the methodologies described and used.

The majority of the applications that we choose are of the SPLASH-2 shared-address-space parallel applications [24]. The SPLASH-2 applications are quite 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. Interestingly, we found that although we did not undo any of the optimisations, the non-blocking synchronisation improved a lot the performance of many of these applications and does not worsen any of them.

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. Finally, Section 5 concludes this paper.

2 Origin 2000

The SGI Origin2000 [10] is a commercial ccNUMA machine with fast, MIPS R10000 processors [26], 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. ccNUMA systems are expected to become the dominant systems on large high performance systems over the next few years. The reasons are: a) they scale up to as many processors as needed. b) they support the cache-coherent globally addressable memory model c) their entry level and incremental costs are relatively low.

2.1 The Platform

The SGI Origin2000 [10] is a scalable shared memory multiprocessing architecture, as shown in Figure 1. 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 [10, 11], and provides remote memory bandwidth equal to local memory bandwidth (780MB/s each) [10, 11, 13]. 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` and `store_conditional` instruction. The `load_linked` and `store_conditional` is comprised by two simpler operations, the `load_linked` and the `store_conditional` one. The `load_linked` loads a word from the memory to a register. The matching `store_conditional` stores back possibly a new value into the memory

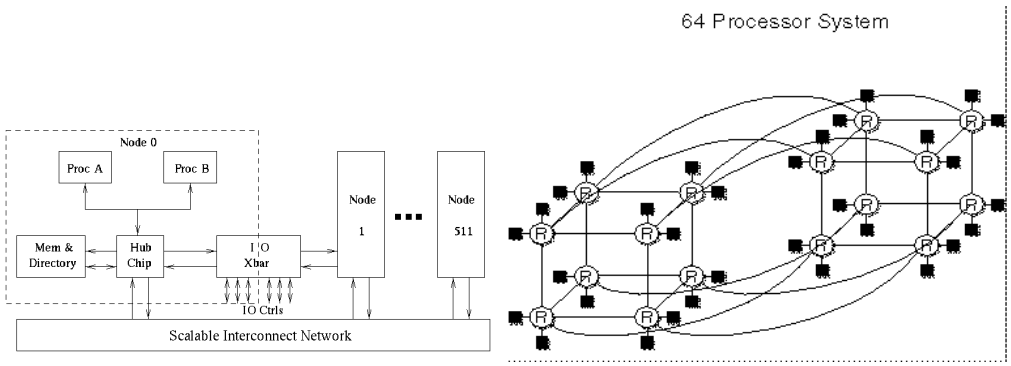


Figure 1: SGI Origin 2000 Architecture

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.

```
LL(p_i, O)                          SC(p_i, v, O)
Pset(O) := Pset(O) ∪ {p_i}            if p_i ∈ Pset(O)
return value(O)                            value(O):= v
                                           Pset(O):= ∅
                                           return true
                                      else
                                           return false
```

Figure 2: The `load_linked_and_store_conditional` primitive

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 and only a few atomic operations are supported on this machine. The specification of this operation 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 a powerful synchronisation power that operations like the `compare_and_swap`, that can atomically check the and exchange the contents of a memory location, has.

For more information on the SGI Origin 2000 the reader is referred to [20, 10].

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

Figure 3: The `fetch_and_op` primitive

3 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 use a large group of applications, some of which are from the SPLASH-2 [24] suite, and some of which were developed more recently and are from the Spark98 kernels suit [18]. Below we briefly describe the applications that we have used. The actual descriptions of the applications can be found in [19, 22, 24, 21].

Ocean simulates eddy currents in an ocean basin [25]. It consists largely of nearest neighbour calculations on regular grids, including a multigrid solver [2]. Both its inherent and induced (at page granularity) data referencing patterns generally involve one producer with one consumer. Read and write accesses are coarse grained internally to a partition and along row-oriented partition boundaries, but fine-grained along column-oriented boundaries; i.e., when a process reads a word from its neighbour along a column-oriented boundary, because of the way memory is laid out, it reads only a single word on each page. Thus, there is significant fragmentation in communicating remote data in pages at column boundaries in shared memory systems.

Volrend renders three dimensional volume data into an image using a ray-casting method [16]. 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]. 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. The structure of the computation and the access patterns to data structures are highly irregular. Parallelism is managed by distributed task queues, one per processor, with task stealing for load balancing. No attempt is made at intelligent data distribution. See [21] for more details.

Water-Nsquared is an improved version of the Water program in SPLASH [22]. This application evaluates forces and potentials that occur over time in a system of water molecules. The forces and potentials are computed using an $0(n^2)$ algorithm , and a predictor-corrector method is used to integrate the motion of the water molecules over time. The main difference from the SPLASH program is that the locking strategy in the updates to the accelerations is improved. 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. For example, the Quake project at Carnegie Mellon University uses a sequence of more than 16,000 sparse matrix-vector product operations to simulate the motion of the ground during the first 40 seconds of an aftershock of the 1994 Northridge earthquake in the San Fernando Valley [1]. The sparse matrix consists of over 13 million rows and 180 million nonzero entries, where each nonzero entry is a dense sub-matrix of double precision floating point numbers. 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. Irregular applications based on sparse matrices are at the core of many important scientific computations. Since the importance of such applications is likely to increase in the future, high-performance parallel systems must provide adequate support for such applications.

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

4 The Non-blocking Transformations

In this section, we show how to transform the lock-based synchronisations in the applications mentioned in the previous section into non-blocking ones. For doing so: i) We first, studied the above applications, in-oder to identify all the lock-based high level synchronisation operations that they use. ii) Second, we propose a set of efficient lock-free implementations for these synchronisations. These implementations are general enough and can be used in other parallel applications.

4.1 Typical Lock-Based Synchronisation Operations And Their Translations to Non-blocking Ones

Before describing the detailed modifications for each application, we will first describe two very common uses of locks in these and other parallel applications. A locks in many parallel applications is used in order to protect a global shared variable which is updated after a simple arithmetic calculation is performed on the value carried before in this shared variable. These shared variables are used in the application programs to either: i) assign consecutive values to the processes, or ii) to sum up values computed by processes of the system, or iii) to act as the index of an array.

We call this kind of Locks as *SimpleLocks*. It was easy to observe that these kind of locks can be replaced with `fetch_and_op` operations to achieve the same functionality without locking.

One problem with the `fetch_and_op` operations is that they do not provide support for floating point numbers. In high performance scientific computing though, people often use floating point numbers. In order to overcome this shortcoming of the hardware we propose an efficient software implementation of a `fetch_and_op` that supports floating point numbers. We call these operations `double_fetch_and_op` operations. We implemented them using 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 SimpleLocks in any parallel application.

4.2 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:

```
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 immplementation of the conditional update of `error_lock`

- `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.

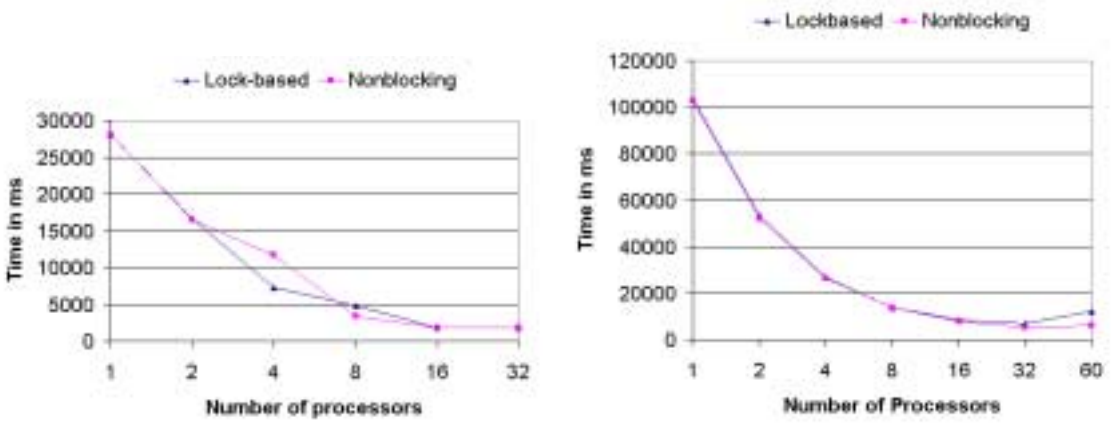
We replaced the first three of these locks with FAD or DFAD operations using the methods described in the previous subsection. The fourth lock (`error_lock`) protects a global variable which is updated conditionally as follows:

```
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 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 the performance. Ocean is a regular application with very regular communication patterns and moreover below 32 processors, the synchronisation time does not contribute much to the total execution time.



(a) OCEAN (b) RADIOSITY

Figure 7: Performance results: OCEAN and RADIOSITY

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 determines when parts of the computation should stop.
- `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 CAS (`compare_and_swap`) atomic operation to implement a non-blocking version of `bsp_tree`. The specification of the CAS 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_and_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 CAS 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

`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

```
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_and_store_conditional`

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 prcesses to determine the task that enters the function `check_task_counter`. We implement this counter in a lock-free manner using the CAS 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`

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 [23], 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.

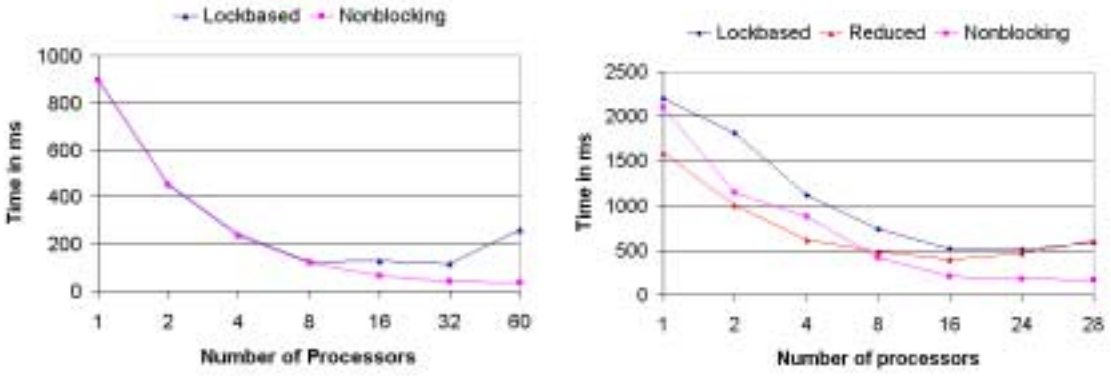
Data Object Name	Functionality
`free_patch`	no enqueue operations run in parallel
`free_element`	no enqueue operations run in parallel
`free_interaction`	enqueue and dequeue operations run in parallel
`free_elemvertex`	no enqueue operations are running in parallel
`free_edge`	no enqueue operations are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

Figure 12: Data objects in **Radiosity**

Volrend in contrast with radiosity does not use many locks. It uses only two SimpleLocks 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 FAD `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. Lmv is a parallel shared memory program based on locks. Rmv is a parallel shared memory program based

on a reduction of the number of locks that are used in lvm. 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 SimpleLocks 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 SimpleLocks and they are used in order to update the force on all moleculars.
- `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 SimpleLocks.

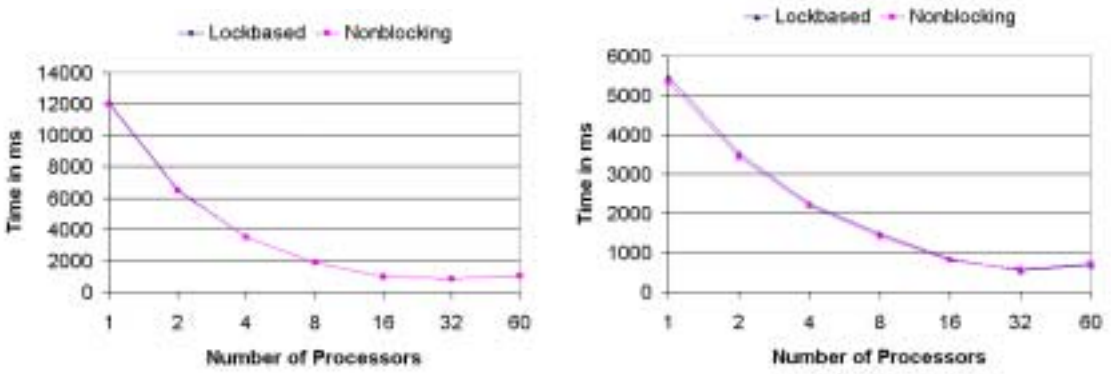
Water-spatial uses 7 different locks. Five of these are SimpleLocks, the first five SimleLocks that are listed in the **Water-nsquared** above (`IndexLock, IntrafVirLock, InterfVirLock, KinetiSumLock, PotengSumLock`). We used the implementations described in the previous subsection in order to replace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

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.

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.

5 Conclusion and Discussions

The main conclusions of our study are the following:



(a) WATER-NSQUARED (b) WATER-SPATIAL

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

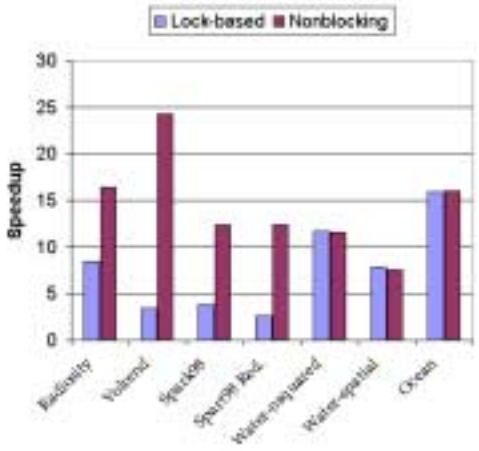


Figure 15: Speedup for the non-blocking and the original versions

- 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 shows 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 phycics department) loaded ORIGIN 2000.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] P. Tsigas and Y. Zhang, A Simple, Fast and Scalable Non-Blocking Concurrent FIFO queue for Shared Memory Multiprocessor Systems, Technical Report 2000-1, Department of Computing Science, Chalmers University of Technology, 2000.

[24] 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.

[25] 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.

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

[27] 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.