The Non-blocking Programming Paradigm in Large Scale Scientific Computations*

Philippas Tsigas and Yi Zhang

Department of Computing Science,
Chalmers University of Technology,
SE-412 60, Gothenburg, Sweden

Abstract. 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. As shown in [1, 2], non-blocking synchronisation has better performance in certain application than blocking synchronisation. In this paper, we try to provide an in depth understanding of the performance benefits of integrating non-blocking synchronisation in scientific computing applications.

1 Introduction

Shared memory multiprocessors are ideal systems for Large Scale Scientific Computations. Modern multiprocessors offer the shared memory programming paradigm together with low latency and high bandwidth interprocessor communication. Fast interprocessor communication gives to the programmers the possibility of exploring fine grain parallelism in their programs. Moreover, because processors communicate with each other by using conventional memory operations on shared memory, it is relatively easy to transfer sequential programs into parallel ones that run on top of shared memory multiprocessors.

A shared memory multiprocessor system consists of multiple processors, provides a single address space for programming, and supports communication between processors through operations on shared memory. Applications running on such systems may use more than one processor at the same time. Programs can improve their execution speed by exploiting the parallelism available on such systems. Single address space shared memory provides an easy programming model

* This work was partially supported by the Swedish Research Council (VR).

R. Wyrzykowski et al. (Eds.): PPAM 2003, LNCS 3019, pp. 1114–1124, 2004.

to programmers. Shared memory operations can be implemented in hardware or software.

To programmers, programming for shared memory multiprocessors is similar to traditional sequential programming for uniprocessor systems. Communication between processors in shared memory multiprocessors is implicit and transparent via conventional memory access instructions, such as Read/Write, that are also used in sequential programming. Therefore, programmers do not have to consider details of low-level communication between processors and can focus mainly on the applications themselves. When an application is running on shared memory multiprocessors, all processes of the application share the same address space; traditional sequential programming also treats memory as a single address space. Such similarity in programming between shared memory multiprocessors and uniprocessors makes shared memory multiprocessors attractive.

Programming for shared memory multiprocessors introduces synchronisation problems that sequential programming does not need to address. Processes in shared memory multiprocessors communicate and coordinate with each other through reading from and writing to shared memory locations. Such Read/Write operations on memory can be executed simultaneously on several processors. The final results of these operations depend on their interleaving. To maintain consistency, synchronisation is used to guarantee that only desired interleaving of operations can happen. There are two ways to do synchronisation in shared memory: mutual exclusion and non-blocking synchronisation.

Mutual exclusion ensures that certain sections of code will not be executed by more than one process simultaneously. The standard solution to mutual exclusion at kernel level in uniprocessor systems is to momentarily disable interrupts to guarantee that the operation of a shared memory object will not be preempted before it completes. This solution is not feasible for uniprocessor systems at user level, where users do not have the privilege to disable interrupts. In multiprocessor systems, where processes execute on several processors, disabling interrupts at kernel level is too costly. In such cases, locks are used to guarantee that only one process can access a shared memory object: before a process accesses a shared memory object, it must get the lock associated with the object; after accessing the object, it will release the lock. Usually only one lock protects an object. The part of code that the process executes in order to access the object is called code in "critical section". If a process cannot get the lock of an object, then another process owns the lock and is working on the object in the critical section.

Non-blocking synchronisation is an alternative to mutual exclusion for implementing shared data objects. Shared data objects implemented with non-blocking synchronisation do not rely on mutual exclusion and do not require any communication with the kernel. Rather, they rely on hardware atomic primitives such as Compare-and-Swap or the pair Load-Link and Store-Conditional.

An implementation of a shared data object is called non-blocking if first it supports concurrency: several processes can perform operations on the shared data object concurrently; and moreover if it ensures that at any point of time

some/all of the non-fault concurrent processes will complete their operations on the object in a bounded time regardless of the speed or status of other processes. If an implementation guarantees progress of some non-fault processes, it is called lock-free; if it guarantees progress of all non-fault processes, it is called wait-free. This requirement rules out the use of locks for non-blocking synchronisation: if a process crashes while holding a lock, no process waiting for the lock can make any progress.

Compared to mutual exclusion, non-blocking synchronisation has the following significant advantages:

1. it avoids lock convoying effects [3]: if a process holding a lock is preempted or delayed, any other process waiting for the lock is unable to perform any useful work until the process holding the locks has finished its access to the shared object.
2. it provides high fault tolerance. By the definition of non-blocking synchronisation, failures of processes should never corrupt the shared data objects. When using mutual exclusion, a process which dies during modifying a shared object in its critical section might leave the shared object in an invalid state. Some kind of fault recovery technique must be used to recover the object then.
3. it eliminates deadlock scenarios, where two or more tasks are waiting for locks held by the other.
4. it does not give priority inversion scenarios.

Non-blocking programming paradigm is different from the lock-based programming paradigm. In this paper, we investigate how the performance of scientific computing applications is effected by adopting the non-blocking programming paradigm.

2 Previous and Current Work

Previously micro-benchmarks have been widely used to evaluation the performance of synchronisation mechanisms on small scale symmetric multiprocessors, as well as distributed memory machines [4,5,6,7,8] or simulators [6,9]. Although micro-benchmarks are useful since they may isolate performance issues, they do not represent the behaviours of real applications. The goal of designing efficient synchronisation mechanisms is to improve performance of real applications, which micro-benchmarks may not represent well.

For non-blocking synchronisation, many researchers proposed the use of non-blocking synchronisation, rather than blocking one, 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 synchronisation on different applications. To address this need, in [1,2], Tsigas and Zhang showed how performance and speedup in parallel

applications would be affected by using non-blocking rather than blocking synchronisation. They performed a fair evaluation of non-blocking synchronisation and blocking based synchronisation in the context of well-established parallel benchmark applications.

In this paper, we try to provide an in depth understanding of the performance benefits of integrating non-blocking synchronisation in scientific computing applications.

3 Performance Impact of Non-blocking

As shown in [1,2], non-blocking synchronisation gives better performance in certain applications than the respective blocking synchronisation. The fact that non-blocking synchronisation avoids lock convoying effects is believed to be one of the main contributors to the performance improvement that comes with non-blocking synchronisation. Lock convoying effects are caused because of preemption of the processes running the applications. From our experience with non-blocking synchronisation, preemption of processes does contribute to performance degradation on applications with blocking based synchronisation. But it is not the only fact that effect the performance. We performed a set of experiments by running different scientific computing applications with exclusive use and without exclusive use on a cache coherent multiprocessor machine. The results we received with exclusive use are, of course, better than the results without exclusive use due to less frequent preemption. However, the performance gap between blocking and non-blocking synchronisation does not narrow much when changing from non-exclusive use to exclusive use. Avoiding the lock convoy effects only contributes a small part into the performance improvement that comes with non-blocking synchronisation.

On modern cache-coherent shared memory multiprocessors, the cache behaviour of an application also effect the performance of the application. The speed of improvement of processor speed exceeds the speed of improvement of memory accessing speed. This results to a bigger and bigger speed gap between processor speed and memory accessing speed. Cache, a small and fast memory located close to processors, is introduced to reduce the performance gap. However when the data required by a processor is not in the cache, a *cache miss*, takes place and operations on memory still need to be performed. Therefore *cache misses* are expensive for the performance of a program. Researchers of parallel applications are developing cache-conscious algorithms and applications to minimise cache misses during the execution of the application.

The performance difference between blocking and non-blocking synchronisation in applications on cache-coherent shared memory machines makes it interesting to investigate the cache behaviour of different synchronisation mechanisms.

Applications with blocking synchronisation usually use a lock to protect shared variables. When shared variables need to be updated, a lock must be acquired. Then variables can be computed and updated. After the variables

are updated, the lock must be released. A scenario of such operations from the Spark98 [10] is given in figure 1.

```
spark_setlock(lockid);
w[col][0] += A[Anext][0][0]*v[i][0] + ... ...;
w[col][1] += A[Anext][0][1]*v[i][0] + ... ...;
w[col][2] += A[Anext][0][2]*v[i][0] + ... ...;
spark_unsetlock(lockid);
```

Fig. 1. Lock-based operations in Spark98

when they update the shared variables, the operation that acquires the lock and the operations that update shared variables may cause cache misses. The lock usually become a memory bottleneck as all process want to access and modify it. The lock is usually not located at the same cache line with the shared variables which it protects.

Comparing it with blocking synchronisation, non-blocking synchronisation has better cache behaviour. The code for the same operation in the non-blocking programming paradigm is given in figure 2. Here, only the operations on the shared variables themselves may cause cache misses.

```
dfad(&w[col][0], A[Anext][0][0]*v[i][0] + ... ...);
dfad(&w[col][1], A[Anext][0][1]*v[i][0] + ... ...);
dfad(&w[col][2], A[Anext][0][2]*v[i][0] + ... ...);
```

Fig. 2. The non-blocking version of the previous operations in Spark98

Comparing the two programming paradigms, the non-blocking one requires low number of memory accesses and suffer less from cache misses. When the accessing pattern of shared memory is irregular, the cache system cannot predicate its pattern and the application has more chances to suffer from cache misses. To verify the above claim, we designed and performed the experiments described in next section.

4 Experiments

The purpose of these experiments is to compare the performance of applications that use blocking synchronisation and non-blocking synchronisation. We measured the time each application spend in different parts of the application; we also measure the number of cache misses generated by the application. All of our experiments were perform on a SGI Origin 2000 machine with 29 processors. A brief introduction of the system we used is given below.

4.1 SGI Origin 2000 Platform

The SGI Origin2000 [7] is a typical commercial cache coherent non-uniform memory access (ccNUMA) machine. It has an aggressive, scalable distributed shared memory (DSM) architecture. The ccNUMA architecture maintain a unified, global coherent memory and all resources are managed by a single copy of the operating system. The architecture is much more tightly integrated than in other recent commercial distributed shared memory (DSM) systems. A hardware-based directory cache coherency scheme ensures that data held in memory is consistent on a system-wide basis. Comparing with cache snooping, such a scheme keeps both absolute memory latency and the ratio of remote to local latency low, and provides remote memory bandwidth equal to local memory bandwidth (780MB/s each) [7].

In SGI Origin 2000, two processors form a node and share the same secondary cache. Directory based cache coherent protocol maintains coherence between nodes within one machine. The machine we use has twenty-nine 250MHz MIPS R10000 CPUs with 4MB L2 cache and 20GB main memory.

The SGI Origin 2000 provides two groups of transactional instructions that can be used to implement any other transactional synchronisation operations. The one used in this paper contains two simple operations, `load_linked` and `store_conditional`. 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.

For more information on the SGI Origin 2000 the reader is referred to [7,11].

4.2 Experiments and Results

The first application that we used is the Spark98 kernel suite. Spark98 kernels is a collection of sparse matrix kernels for shared memory and message passing systems. Spark98 kernels have been developed to facilitate system builders with a set of example sparse matrix codes that are simple, realistic, and portable. 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. More information about Spark98 can be found in [10].

In [2], we showed that the non-blocking version of Spark98 performs better than the lock-based version and also better than the reduction-based version of Spark98. In this section, we examine the reason that the non-blocking version performs better than both blocking versions. More specifically, we want to identify the part of the application that has been improved. We measured for the lock-based version the execution time spend in critical section. For the

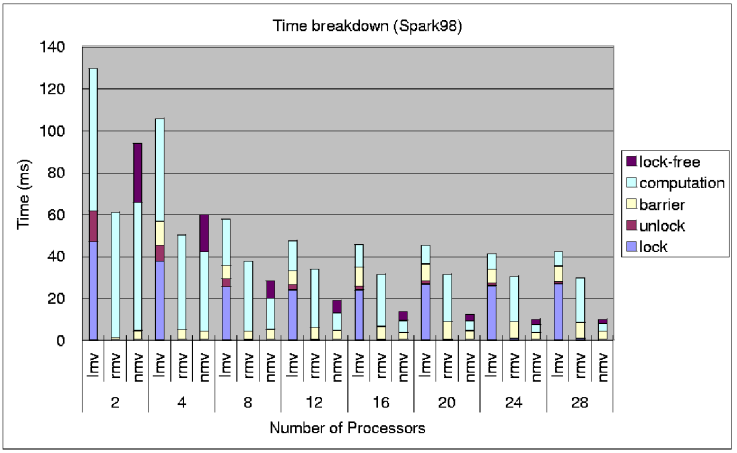

Fig. 3. Time breakdown of different Spark98 version

non-blocking version of the application, which has been improved by adopting the non-blocking synchronisation, we measured the time spend on the lock-free computing part. For all version, the time spend on barrier synchronisation is measured. All versions use barrier operations to synchronise process in different execution phase. Furthermore, the reduction-based version of Spark98 heavily rely on barriers to avoid lock operation. Figure 3 shows the results we have observed. When the number of processors is small, the reduction-based version of Spark98 performs the best: almost all the execution time is dedicated to computation. On the other side, the lock-based and non-blocking versions spend substantial time in synchronisation. On the other hand, when the number of processors becomes larger than 12, the speedup of the reduction-based version of Spark98 stops and the time spend on barrier synchronisation increases as the number of processors increases. The lock-based version can keep its speedup up to 24 processors. Although it is not the best one at the beginning, the non-blocking version performs the best when number of processors become larger than 8. The time spend in lock-free computing and the real computation keep the speedup nicely up to 28 processors. The time spend in barriers is almost constant for the non-blocking version; but for the lock-based and reduction-based ones, the time is almost zero for 2 processors and it is twice as much as non-blocking version when the number of processors reach 28. The larger the time spend in barriers, the more uneven the working load is distributed among processors. The non-blocking version seems to evenly distribute the working load among the processors.

The cache behaviour of these applications are shown in figure 4. As it was described at the beginning of this section, in the SGI Origin 2000 machine, two processors within one node share the same secondary cache. Therefore, only one secondary cache memory caches the main memory in the two-processor case.

When the number of processors is larger than 2, the cache coherent protocol becomes active in order to maintain coherence between several secondary caches. A memory access operation in one node may invalidate a secondary cache line in another node. This is why there is a large difference on the number of secondary cache misses between the 2-processor and the 4-processor experiments shown in figure 4.

From figure 4, the number of cache misses keeps increasing for the reduction-based version; but it keeps decreasing for the lock-based version and the non-blocking version after reaching 4 processors. The number of cache misses for the non-blocking version is always the smallest compared to both the other two versions. When the number of cache misses becomes stable, after 16 processors, it is less than one third of the respective number for the lock-based version which is also stable. The number of cache misses for the reduction-based version keeps increasing as the number of processors increases.

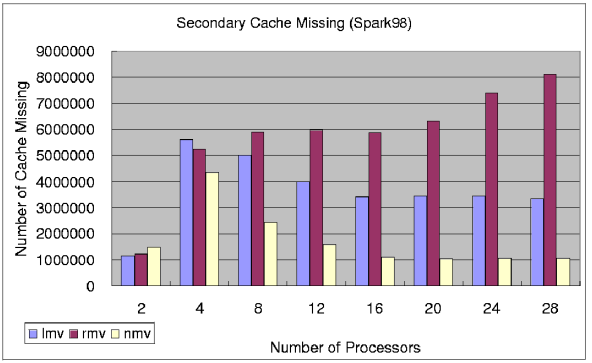

Fig. 4. Cache miss of different Spark98 version

Another application we investigated in this paper is Volrend. Volrend is an application from the SPLASH2 parallel application benchmark [12]. It renders three dimensional volume data into an image using a ray-casting method [13]. 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.

As shown in [2], there is also a large performance difference between the lock-based version and the non-blocking version. We perform the same experiments as we did with Spark98 to investigate the time distribution and cache behaviour of both version.

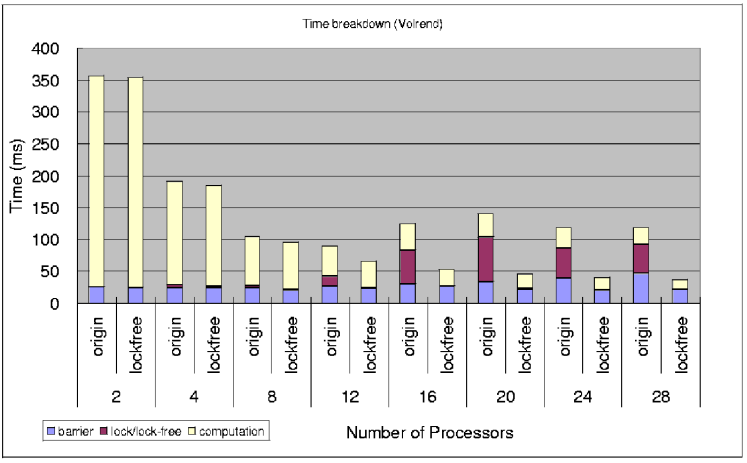

Fig. 5. Time breakdown of different Volrend version

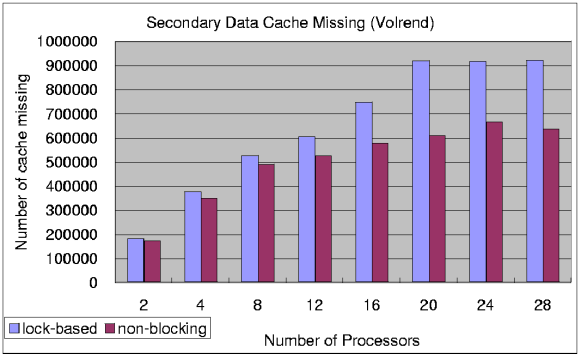

Fig. 6. Cache miss of different Volrend version

The time breakdown for Volrend is shown in figure 5. In the figure, the execution time stops to decrease for the lock-based version when there are more than 12 processors. The time spend in synchronisation increases dramatically when the number of processors reaches 16. But the non-blocking version continues to perform well up to 28 processors and the time spend in lock-free computing is negligible. The time spend in barriers is also almost constant for the non-blocking version of Volrend; but this time doubles for the lock-based version from 2 processors to 28 processors, which means that the non-blocking version offers more fair and balanced working load to processors. The even and balanced working load also contributes to the performance improvements.

The cache behaviour of the two versions of Volrend is shown in figure 6. There is also a large jump on the number of cache misses between 2-processor

and 4-processor as Spark98. In the figure, the non-blocking version also has smaller number of cache misses than the lock-based version in all cases. When all numbers become stable after we reach 20 processors, the number of cache misses for the non-blocking version is about two third of the respective number for the lock-based version.

5 Conclusion

In this paper, we investigate the reason that non-blocking synchronisation performs better than blocking synchronisation in scientific applications. We observed applications using non-blocking synchronisation generate less cache misses than the ones using lock-based synchronisation. Non-blocking synchronisation also balances better the work load among the processors when compared with lock-based synchronisation. Low number of cache misses and balanced work load are the two main reasons that give non-blocking synchronisation better performance. To help parallel programmers who are not experts on non-blocking synchronisation to use non-blocking synchronisation in their applications, a library that supports non-blocking synchronisation called NOBLE [14] has been developed at Chalmers University of Technology, Sweden. The library provides a collection of the most commonly used data types and protocols.

References

1. Tsigas, P., Zhang, Y.: Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In: Proceedings of the ACM SIGMETRICS 2001/Performance 2001, ACM press (2001) 320–321
2. Tsigas, P., Zhang, Y.: Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In: Proceedings of the 3rd ACM Workshop on Software and Performance (WOSP'02), ACM press (2002) 55–67
3. Kopetz, H., Reisinge, J.: The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In: Proceedings of the Real-Time Systems Symposium, Raleigh-Durham, NC, IEEE Computer Society Press (1993) 131–137
4. Eichenberger, A., Abraham, S.: Impact of load imbalance on the design of software barriers. In: Proceedings of the 1995 International Conference on Parallel Processing. (1995) 63–72
5. Kumar, S., Jiang, D., Singh, J.P., Chandra, R.: Evaluating synchronization on shared address space multiprocessors: Methodology and performance. In: Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems (SIGMETRICS-99). Volume 27,1., ACM Press (1999) 23–34
6. Kaumlgi, A., Burger, D., Goodman, J.R.: Efficient synchronization: Let them eat QOLB. In: 24th Annual International Symposium on Computer Architecture (24th ISCA'97), Computer Architecture News, ACM SIGARCH (1997) 170–180
7. Laudon, J., Lenoski, D.: The SGI origin: A ccNUMA highly scalable server. In: Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA-97). Volume 25,2 of Computer Architecture News., New York, ACM Press (1997) 241–251

8. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. Journal of Parallel and Distributed Computing **51** (1998) 1–26
9. Lim, B.H., Agarwal, A.: Reactive synchronization algorithms for multiprocessors. In: Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI), ACM press (1994) 25–35
10. O'Hallaron, D.R.: Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, CMU (1997)
11. Cortesi, D.: Origin 2000 and onyx2 performance tuning and optimization guide. http://techpubs.sgi.com/library/, SGI Inc. (1998)
12. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characteriation and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, ACM Press (1995) 24–37
13. Nieh, J., Levoy, M.: Volume rendering on scalable shared memory mimd architectures. In: Proceeding of the 1992 Workshop on Volume Visualization. (1992) 17–24
14. Sundell, H., Tsigas, P.: Noble: A non-blocking inter-process communication library. In: Proceedings of the Sixth Workshop on Languages, Compilers and Run-time Systems for Scalable Computers. (2002)