Technical Report no. 2002-02

NOBLE: A Non-Blocking Inter-Process Communication Library[1]

Håkan Sundell **Philippas Tsigas**

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, 2002

[1]Performed within the network for Real-Time research and graduate Education in Sweden (ARTES), supported by the Swedish Foundation for Strategic Research (SSF)

Technical Report in Computing Science at
Chalmers University of Technology and Göteborg University

Technical Report no. 2002-02
ISSN: 1650-3023

Department of Computing Science
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, Sweden, 2002

Abstract

Many applications on shared memory multi-processor machines can benefit from the exploitation of parallelism that non-blocking synchronization offers. In this paper, we introduce a library support for multi-process non-blocking synchronization called NOBLE. NOBLE provides an inter-process communication interface that allows the user to select synchronization methods transparently to the one that suits best for the current application. The library provides a collection of the most commonly used data types and protocols in a form, which allows them to be used by non-experts. We describe the functionality and the implementation of the library functions and illustrate the library programming style with example programs. The examples show that using the library can considerably reduce the runtime on distributed shared memory machines.

1 Introduction

Software implementations of synchronization constructs are usually included in system libraries. The design of a good synchronization library can be challenging and many efficient implementations for the basic synchronization 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 spent by the processes on the synchronization can form a substantial part of the program execution time [KarLMS91, MelS91, MicS98, NikP99, ZahLE91]. The reason for this is that typical synchronization is based on blocking that unfortunately results in poor performance. This is because blocking produces high levels of contention on the memory and the interconnection network, 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 synchronization to address the above problems.

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 and many researchers have for more than two decades developed efficient non-blocking implementations for several shared data objects. Despite this, the scientific discoveries on non-blocking synchronization have not have migrated much into practice, even though synchronization is still the bottleneck in many applications. The lack of a standard library of non-blocking synchronization of shared data objects commonly used in parallel applications has played a significant role for this slow migration.

NOBLE offers a library support for multi-process non-blocking synchronization for shared memory systems. NOBLE has been designed in order: I) to provide a collection of shared data objects in a form which allows them to be used by non-experts, II) to offer an orthogonal support for synchronization where the application can change synchronization implementations with minimal changes, III) to be easy to port to different multi-processor systems, IV) to be adaptable for different programming languages and V) to contain efficient known implementations of its shared data objects.

The rest of the paper is organized as follows. Section 2 outlines the basic features of NOBLE and the implementation and design steps that have been taken to support these features. Section 3 illustrates by a way of an example the use of the features of NOBLE in a practical setting. Section 4 presents run-time experiments using NOBLE that show the performance benefits of non-blocking synchronization. Finally, Section 5 concludes this paper.

2 Design and Features of NOBLE

When designing NOBLE we identified a number of characteristics that had to be covered from our design in order to make NOBLE easy to use and usable by a wide range of practitioners. We designed NOBLE to have the following features:

- Usability-Scope - NOBLE provides a collection of fundamental shared data objects that are widely used in parallel and real-time applications.
- Easy to use - NOBLE provides a simple interface that allows the user to use the non-blocking implementations of the shared data objects the same way the user would have used lock-based ones.
- Easy to Adapt - No need for changes at the application level are required by NOBLE and different implementations of the same shared data objects are supported via a uniform interface.
- Efficient - Users will have to experience major improvements in order to decide to replace the existing trusted code with new synchronization methods. NOBLE has been designed to be as efficient as possible.
- Portable - NOBLE has been designed to support general shared memory multi-processor architectures and offers the same user interface on different platforms, although the actual implementations might differ.

- Adaptable for different programming languages.

2.1 Usability-Scope

NOBLE provides a collection of non-blocking implementation of fundamental shared data objects in a form that allows them to be used by non-experts. This collection includes most of the shared data objects that can be found in parallel applications, i.e. stacks, queues, linked lists, snapshots and buffers. NOBLE contains the most efficient realizations known for its shared data objects, [AllEHPST99, Har01, SunT00, SunTZ00, TsiZ99, Val95].

2.2 Easy to use

NOBLE provides a precise and readable specification for each shared data object implemented. The user interface of the NOBLE library is the defined library functions that are described in the specification. At the very beginning the user has only to include the NOBLE header at the top of the code (`#include <Noble.h>`).

In order to make sure that none of the functions or structures that we define causes any name-conflict during the compilation or the linking stage, only the functions and the structures that have to be visible to the user of NOBLE are exported. All other names are invisible outside of the library. The names that are exported start with the three-letter combination NBL. For example the implementation of the shared data object Queue is `struct NBLQueue` in NOBLE.

2.3 Easy to Adapt

NOBLE offers a set of different implementations for many shared data objects realized in NOBLE. The user can select the implementation that better suits the application needs. The selection is done at the moment of creation of the shared data object. For example for the $Queue$ shared data object there are several different creation functions for the different implementations offered by NOBLE, their usage is described below. NOBLE also offers a simple interface to standard lock-based implementations of all shared data objects provided.

```
  NBLQueue *NBLQueueCreateLF(int nrOf-
Blocks); /* Create a Queue using the
implementation LF */
  NBLQueue *NBLQueueCreateLF2(); /*
Create a Queue using the implementation
LF2 */
```

In all other steps of use of the shared data object, the programmer does not have to remember or to supply any information about the implementation (synchronization method) used. This means that all other functions regarding operations on the shared data objects have only to be informed about the actual instance of the shared data object. The latter gives a unified interface to the user: all operations take the same number of arguments and have the same return value(s) independently of the implementation:

```
  NBLQueueFree(handle);
  NBLQueueEnqueue(handle,item);
  NBLQueueDequeue(handle);
```

All names for the operations are the same regardless of the actual implementation type of the shared data object, and more significantly the semantics of the operations are also the same.

2.4 Efficiency

The only information that is passed to the library during the invocation of an operation is just a $handle$ to a private data structure. Henceforth, any information concerning the implementation method, that is used for this particular shared data object instance, had to be inside this private data structure. For fast redirection of the program flow to the correct implementation, function pointers are used. Each instance of the data structure itself contains a set of function pointers, one for each operation that can be applied on it:

```
typedef struct NBLQueue {
     void *data;
     void (*free)(void *data);
     void (*enqueue)(void *data
           ,void *item);
     void *(*dequeue)(void *data);
} NBLQueue;
```

The use of function pointers inside each instance, allows us to produce in-line redirection of the program flow to the correct implementation. Instead of having one central function that redirects, we define several macros that redirect directly from the user-level. From the user's perspective this usually makes no difference, these macros can be used in the same way as pure functions:

```
  #define NBLQueueFree(handle) (handle-
>free(handle->data))
  #define NBLQueueEnqueue(handle,item)
(handle->enqueue(handle->data,item))
  #define NBLQueueDequeue(handle)
(handle->dequeue(handle->data))
```

2.5 Portability

The interface to NOBLE has been designed to be the same independently of platform. Internal library dependencies are not visible outside of the library. Application calls to the library look exactly the same with respect to the NOBLE library when moving to another platform. The implementations that are contained in NOBLE, use hardware synchronization primitives (Compare-And-Swap, Test-And-Set, Fetch-And-Add, Load-Link/Store-Conditional), that are widely available in many commonly used architectures. Still, in order to achieve the same platform independent interface, NOBLE has been designed to provide an internal level abstraction of the hardware synchronization primitives. All hardware dependent operations that are used in the implementation of NOBLE, are reached using the same (abstraction) method:

```
#include "Platform/Primitives.h"
```

This file depends on the actual platform. However, a set of implementations (with the same syntax and semantics) of the different synchronization primitives needed by the implementations have to be provided for the different platforms. These implementations are only visible inside NOBLE and there are no restrictions on the way they are implemented.

NOBLE at this point has been successfully implemented on the Sun Sparc platform as well on the Intel Win32 platform. An implementation for the Silicon Graphics Origin 2000 platform is under construction.

2.6 Adaptable for different programming languages

NOBLE is realized in C and therefore easily adaptable to other popular programming languages that support importing functions from C libraries.

C++ is directly usable with NOBLE. The basic structures and operation calls of the NOBLE shared data objects have been defined in such a way that real C++ class functionality can also be easily achieved by using wrap-around classes, with no loss of performance.

```
class NOBLEQueue {
private:
     NBLQueue* queue;
public:
     NOBLEQueue(int type) {if(type==
           NBL_LOCKFREE) queue=
           NBLQueueCreateLF();
           else ...  }
     ~ NOBLEQueue() {NBLQueueFree(queue);}
     inline void Enqueue(void *item)
           {NBLQueueEnqueue(queue,item);}
     inline void *Dequeue() {return
           NBLQueueDequeue(queue);}
};
```

Because of the inline statements and the fact that the function calls in NOBLE are defined as macros, the function calls of the class members will be resolved in the nearly exactly same way as in C, with almost no performance slowdown.

3 Examples

In this section we give an overview of NOBLE by way of an example: *a shared stack* in a multi-threaded program. First we have to create the stack using the appropriate create functions for the implementation that we want to use for this data object. We decided to use the implementation *LF* that requires to supply the maximum size of the stack as an input to the create function. We selected 10000 stack-elements for the maximum size of the stack:

```
stack=NBLStackCreateLF(10000);
```
where $stack$ is a globally defined pointer variable:
```
NBLStack *stack;
```

Whenever we have a thread that wants to invoke a stack operation the appropriate function has to be called:

```
NBLStackPush(stack, item);
```
or
```
item=NBLStackPop(stack);
```

When our program does not need the stack any more we can do some cleaning and give back the memory allocated for the stack:

```
NBLStackFree(stack);
```

In case that we decide later on to change the implementation of the $stack$ that our program uses, we only have to change one single line in our program. For example if we want to change from the LF implementation to the LB implementation we only have to change the line:

```
stack=NBLStackCreateLF(10000);
```
to
```
stack=NBLStackCreateLB();
```

Experiment	Operation 1	Operation 2	Operation 3	Operation 4	Operation 5
Queue	Enqueue 50%	Dequeue 50%			
Stack	Push 50%	Pop 50%			
Snapshot	Update/Scan 100%				
Singly Linked List	First 10%	Next 20%	Insert 60%	Delete 10%	
Queue - Low	Enqueue 25%	Dequeue 25%	Sleep 50%		
Stack - Low	Push 25%	Pop 25%	Sleep 50%		
Snapshot - Low	Update/Scan 50%	Sleep 50%			
Singly Linked List - Low	First 5%	Next 10%	Insert 30%	Delete 5%	Sleep 50%

Table 1. The distribution characteristics of the random operations

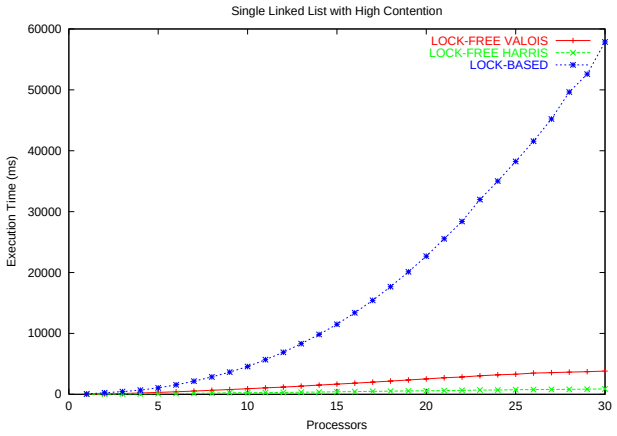

Figure 1. Experiment with singly linked lists and high contention on SUN Enterprise 10000

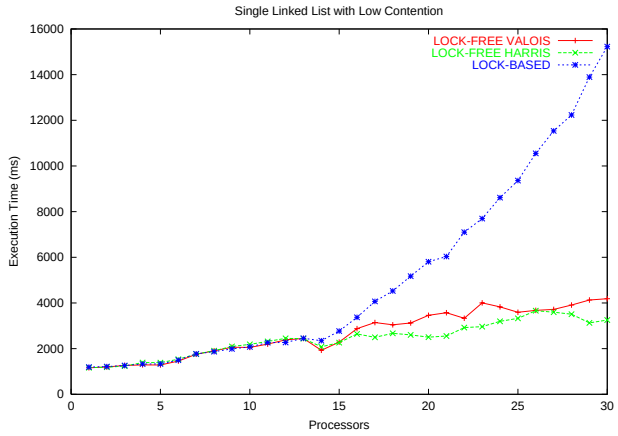

Figure 2. Experiment with singly linked lists and low contention on SUN Enterprise 10000

4 Experiments

We have performed a significant number of experiments in order to measure the performance benefits that can be achieved from the use of NOBLE. Since lock-based synchronization is known to usually perform better than non-blocking synchronization when contention is very low, we used the following micro-benchmarks in order to be as fair as possible:

- High contention - The concurrent threads are continuously invoking operations, one after the other to the shared data object, thus maximizing the contention.
- Low contention - Each concurrent threads performs other tasks between two consecutive operations to the shared data object. The contention in this case is lower, and quite often only one thread is using the shared data object at one time.

In our experiments each concurrent thread performs 50000 randomly chosen sequential operations. Each experiment is repeated 50 times, and an average execution time for each experiment is estimated. Exactly the same sequential operations are performed for all different implementations compared. All lock-based implementations are based on simple spin-locks using the TAS atomic primitive. For the low contention experiments each thread randomly selects to perform a set of 1000 to 2000 sequential writes to a shared memory register with a new computed value. A clean-cache operation was performed just before each sub-experiment. The distributions characteristics of the random operations for each experiment are shown in table 1.

The experiments were performed using different number of threads, varying from 1 to 30. We performed our experiments on a Sun Enterprise 10000 StarFire [Cha98] system. At that point we could have access to 30 processors and each thread could run on its own processor, utilizing full concurrency. A set of experiments was also performed on a

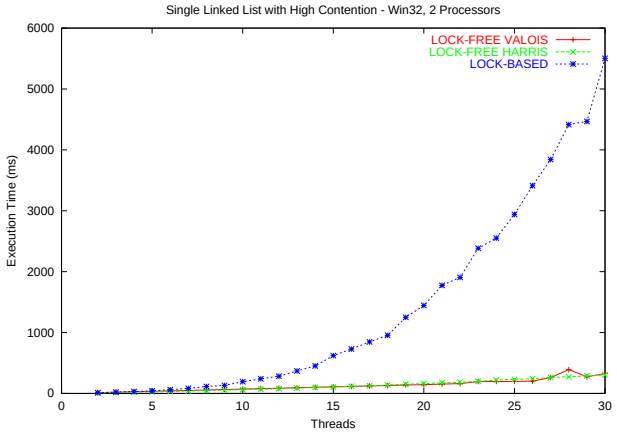

Figure 3. Experiment with singly linked lists and high contention on Dual Pentium II

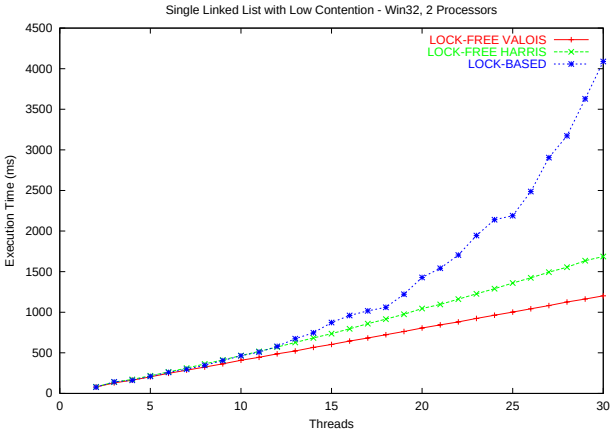

Figure 4. Experiment with singly linked lists and low contention on Dual Pentium II

Compaq dual-processor Pentium II PC.

Because of space constraints only some of the results from these experiments are shown here in figures 1 to 6, the average execution time is drawn as a function of the number of processes. From all the results that we collected we could definitely conclude that NOBLE, largely because of its non-blocking characteristics and partly because of its efficient implementation, outperforms the respective lock-based implementations significantly. For the singly linked list, NOBLE was up to 64 times faster than the lock-based implementation. The performance benefits from using NOBLE and lock-free synchronization methods in general increase with the number of processors. For the experiments with low contention, NOBLE still performs better than the respective lock-based implementations, for high number of processors NOBLE performs up to 3 times faster than the respective lock-based implementations.

5 Conclusions

NOBLE is a library for non-blocking synchronization, that includes implementations of several fundamental and commonly used shared data objects. The library is easy to use and existing programs can be easily adapted to use it. The programs using the library, and the library itself, can be easily tuned to include different synchronization mechanisms for each of the supported shared data objects. Experiments show that the non-blocking implementations in NOBLE offer significant improvements in performance, especially on multi-processor platforms. NOBLE currently supports two platforms, the SUN Sparc and the Intel Win32 architectures. There is work under construction for supporting the Silicon Graphics Origin 2000 platforms.

The first version of NOBLE has just been made available for outside use and can be used freely for the purpose of research and teaching, it is available on the web at http://www.cs.chalmers.se/~noble. We hope that NOBLE will narrow the gap between theoretical research and practical application.

Future work in the NOBLE project includes the extension of NOBLE with more implementations and new shared data objects, as well, as porting NOBLE to platforms more suitable for real-time systems.

References

[AllEHPST99] B. ALLVIN, A. ERMEDAHL, H. HANSSON, M. PAPATRIANTAFILOU, H. SUNDELL, PH. TSIGAS. Evaluating the Performance of Wait-Free Snapshots in Real-Time Systems. *SNART'99 Real Time Systems Conference*, pages 196–207, Aug 1999.

[Cha98] A. CHARLESWORTH. StarFire: Extending the SMP Envelope. *IEEE Micro*, Jan. 1998.

[Har01] T.L. HARRIS A Pragmatic Implementation of Non-Blocking Linked Lists. *Proceedings of the 15th International Symposium of Distributed Computing*, October 2001.

[KarLMS91] A. KARLIN, K. LI, M. MANASSE AND S.OWICKI. Empirical studies of competitive spinning for a shared-memory multiprocessor. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 41-55, Oct. 1991.

[MelS91] J. M. MELLOR-CRUMMEY AND M. L. SCOTT. Algorithms for Scalable Synchronization on Shared-

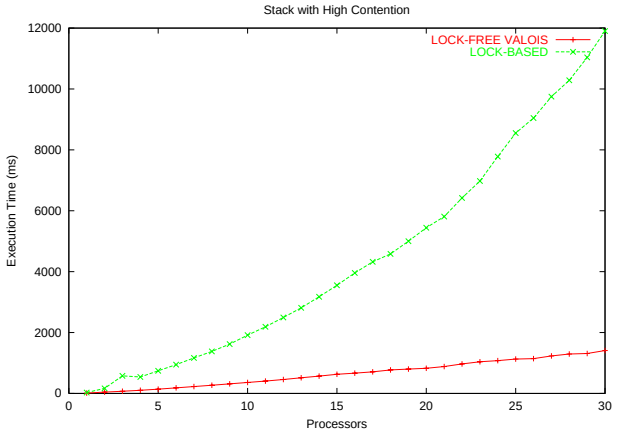

Figure 5. Experiment with stacks and high contention on SUN Enterprise 10000

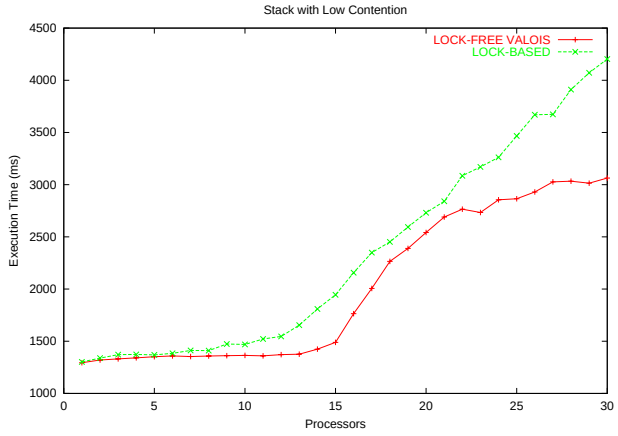

Figure 6. Experiment with stacks and low contention on SUN Enterprise 10000

Memory Multiprocessors. *ACM Trans. on Computer Systems, 9(1)*, pp. 21-65, Feb. 1991.

[MicS98] 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.

[NikP99] 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 *Proceedings of the 1999 Conference on Supercomputing, ACM SIGARCH*, pp. 319-328, 1999.

[SunT00] H. SUNDELL, P. TSIGAS. Space Efficient Wait-Free Buffer Sharing in Multiprocessor Real-Time Systems Based on Timing Information *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applicatons (RTCSA 2000)*, pp. 433-440, IEEE press, 2000.

[SunTZ00] H. SUNDELL, P. TSIGAS, Y. ZHANG. Simple and Fast Wait-Free Snapshots for Real-Time Systems *Proceedings of the 4th International Conference On Principles Of Distributed Systems (OPODIS 2000)*, pp. 91-106, Studia Informatica Universalis, 2000.

[TsiZ99] P. TSIGAS, Y. ZHANG. Non-blocking Data Sharing in Multiprocessor Real-Time Systems. *Proceedings of the 6th International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, IEEE press, pp. 247-254.

[Val95] J.D. VALOIS. Lock-Free Data Structures. *PhD. Thesis, Rensselaer Polytechnic Institute, Troy, New York*, 1995.

[ZahLE91] J. ZAHORJAN, 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, Apr. 1991.