DEPARTAMENTO DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Efficient implementation of the ordered read write locks model on multicore architectures

Diciembre 2018

Rodrigo Campos Catelin
`rodrigo@sdfg.com.ar`

Director
Jens Gustedt
`jens.gustedt@inria.fr`

Co-Director
Diego Gabervetsky
`diegog@dc.uba.ar`

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires
República Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

Abstract

Multi-cores are ubiquitous in present computing, from mobile phones and servers to watches and glasses. However, their adoption raises some challenges among which concurrency control is probably one of the toughest to overcome. Current programming languages provide rudimentary support to successfully exploit these systems, and there are numerous ways to tackle this issue. Each one has advantages and disadvantages and there is no silver bullet.

This work presents an efficient implementation of the Ordered Read Write Locks model, an approach to address this problem that eliminates the possibility of deadlock and achieves high CPU parallelization. Such implementation was motivated by insights from a custom instrumentation created to that end, and observations about invariants imposed by the model.

This new implementation improves the performance on multi-core architectures several times, scaling better with increasing number of cores while still being usable for small computations.

Agradecimientos

Acknowledgments

Contents

Chapter 1

Introduction

1.1 The Fort de Mutzig Conundrum

The Fort de Mutzig, also known as Feste Kaiser Wilhelm II, is a fort located near the town of Mutzig, in the Bas-Rhin *departement* of France. It is one of the forts built by Germany at the end of the 19th century to defend Strasbourg.

It has been historically disputed between Germany and France and one of the things that were constantly remodeled when the fort changed hands were the bathrooms. Under French control the bathroom had latrines, but under German control they were changed into toilets. Besides this cosmetic detail, Germans even had a rule to use the bathroom: while a General is using the bathroom, no other soldier can use it.

In other words, a General has exclusive access to the bathroom while the soldiers can all go in at the same time. This brings-up some questions on how to enforce the rule: will a General always wait until the bathroom is empty when they want to use it? If so, an important question is how to guarantee that a General in a hurry will be able to use it anytime soon. And otherwise, if the soldiers are going to wait until no General wants to use the bathroom, how can soldiers in need be guaranteed they will be able to use it soon?

Respecting military hierarchy and guaranteeing that soldiers and Generals can both use the bathroom in a fair way is no different to some problems in computer science. For instance, we can use this as an analogy for read-write locks, where the bathroom is the arbitrated resource, the soldiers want to read and the Generals want to write. The previous questions on how to guarantee a fair access to soldiers and Generals are equivalent to the problem known in computer science as starvation. Starvation was a problem back then at the fort, and it is a problem today in computer science.

Problems at the fort are out of scope for this work. However, the Ordered Read Write Locks model presented here guarantees no starvation.

1.2 Multi-cores

Moore's law is becoming no longer accurate for single-core computing. David Patterson recently said: "We are now a factor of 15 behind where we should be if Moore's Law were still operative. We are in the post–Moore's Law era".

Technology used in modern processors is getting closer to physical limits and it is not expected that advances in this front will be able to keep pace with performance demands. This, inevitably, leads to the use of multiple processing cores in order to meet the new performance requirements.

Nowadays multi-core systems can be found from traditional server and desktop computers to mobile phones and smartwatches. It is possible today to buy systems with 61 cores and 244 threads, like the one provided by the Intel Xeon Phi co-processor. Multi-core systems are a reality now and there is no reason to think that they will go away anytime soon.

However, the adoption of multi-core platforms raises some challenges of which concurrency control is probably one of the toughest to overcome. Although strictly speaking current programming languages do provide rudimentary support to take advantage of these kinds of systems, they don not provide friendly ways to effectively take advantage of them.

A possible avenue for improving upon this is to design and implement completely new programming languages and paradigms. This approach, while valid and worth pursuing, also has some drawbacks; and the tradeoffs when compared to improving the existing languages are not always convenient.

The Ordered Read Write Locks model presented in this work approaches the problem within widely used programming paradigms. Furthermore, this model completely eliminates the possibility of deadlock, one of the main challenges when programming concurrent systems.

1.3 Parallel programming

One of the keys to successfully exploit multicore systems is making an application run in parallel. Probably the most simple way of doing this is running multiple instances of a sequential program, but this is usually not an option. Alternative solutions are available, but the ones that are actually useful in real-life are hard to implement because of their complexity. Sometimes most of the complexity can be hidden inside the compiler, and sometimes it needs to be addressed directly by the programmer. But complexity is always present.

Compilers focused on automatic parallelization have been in the scientific computing arena for many years. They are able to create parallel programs from sequential code without the programmer's intervention. One way they accomplish this is optimizing the *for* loops accessing arrays. The theory behind the optimizations applied in this way is based on a mathematical framework called the polytope model[16, 11, 10]. Examples of such compilers are SUIF[17], Polaris[4], PIPS[8], Pluto[14], VMAD[13] and Apollo[5]. However, there is no established compiler using this technique, as there is no general consensus that the complexity is worth the performance gains.

If complexity is not dealt with inside the compiler, it must be addressed by the programmer. That means programmers might need to make difficult decisions that can vary from marking which parts of the code can run in parallel to modeling the problem in a completely different way, like using message passing techniques. This wide spectrum is supported by different tools. Some are different forms of CPU threads (POSIX[1], OpenMP [18], TBB [19]), accelerator threads for GPUs (CUDA [20], OpenCL [21], OpenACC [22]), and message passing (MPI [23]). Other examples include GPH[2], HiHP[15], Chapel[6] and

Sequoia[9].

When using these languages the programmer has to indicate, in one way or another, which parts of the code can run in parallel. But this is not ideal, as the programmer is required to handle complex issues such as selecting a convenient algorithm for parallelization, analyzing the dependencies between parts of the code, ensuring correct semantics and using a suitable programming model. Nevertheless, this approach is the current industry-standard, and manages to achieve the best results in practice.

1.4 About this work and it's organization

The Ordered Read Write Locks model was created some years ago along with it's reference implementation. The main goal of this work is to improve its performance on multi-core architectures.

Before looking into the proposed improvements, a detailed explanation of the ORWL model is given (chapter 2). This includes all that is needed to follow this work and some key invariants of the model.

To achieve performance improvements, the first step is to understand the current implementation and its run time behavior (chapter 3). To that end, a special instrumentation was incorporated. The combination of runtime analyses and observations about the model and its invariants led to the proposal of a new implementation with different algorithms and data structures (chapter 4).

Finally, performance is compared between the reference and the new implementation (chapter 5).

Chapter 2

The Ordered Read Write Locks model

The Ordered Read Write Locks model is similar to POSIX read-write locks, suitable for data-flow or iterative computations, and fits within existing programming paradigms and languages.

It introduces some restrictions upon the locked code, which are easy to abide in most common cases, and in exchange it can guarantee the absence of deadlocks and starvation. Some typical concurrency tasks, like adding a new node to the cluster where the task executes, using more cores, or using the GPU of a node to compute, are tasks that are expected to be relatively simple when using the ORWL model.

These advantages are achieved by a trade-off on the type of problems this model can represent. Not all algorithms are suitable to implement using the ORWL model. ORWL can handle algorithms where the output of a task is the input to one or more tasks, and the read and write access to the shared data cannot be done atomically. An example of an algorithm suitable for ORWL is block oriented matrix computations[3], where the data chunks that are needed for reading are far too big to allow for an atomic wait-free operation on system level.

2.1 Major concepts

ORWL is based on four major concepts that compose its model of computation: **tasks** as units of program execution, **resources** as an abstraction of data or hardware on which tasks interact, **handles** as means of access of the tasks to these resources, and **critical sections** to organize access to resources and computation. Major theoretical properties and proofs for them can be found in[7]; in particular, this paper shows how to construct iterative applications as a set of autonomous tasks such that any execution has guarantees of liveness and fairness.

In this work, *iterative computations* refers to computations which show data dependencies that allow to run parallel execution by slicing data and feeding a pipeline. An iterative algorithm generally computes until stabilization or a predefined number of iterations, and the computation over a single data block is performed with the following loop: wait to acquire the lock, then perform computation, then release the lock.

Tasks Tasks are units of program execution. A given set of tasks T is to perform some computation, and data dependencies exist between these tasks. Tasks may be recurrent, as part of an iterative application. Data dependencies are distinguished read and write operations that are not necessarily atomic. Therefore a dependency of task v from task w is modeled by v reading data that w has written previously. Hence, v may only be executed while w is not. This model provides a way to control the execution order of tasks algorithmically based on their data dependencies. ORWL tasks run concurrently, they are controlled autonomously through dependencies on resources (there is no centralized scheduler) and they may be heterogeneous, that is each task may run a different compute kernel designed to be executed on a CPU core, on GPU cores or on other accelerators. ORWL tasks and OS processes or threads are only loosely related. In fact, one OS process can perform one or several tasks, while several OS threads will be used by ORWL to perform a task.

There are two phases in an ORWL program: an initialization phase, and a computation phase. During the initialization phase, tasks declare all resources that will be needed during a computation. Then, once all tasks have been initialized, they begin the computations phase. During the computation phase, they perform computations over the resources.

Resources Each task has a fixed number of control data structures coined *resources*. Usually, resources represent a particular data structure in RAM, but they can also represent files or hardware entities such as CPUs or GPUs.

To each such resource ORWL associates a FIFO queue that regulates the access to it.

- Prior to an access, a tasks inserts a request in the FIFO of the resource.
- Access is then granted, once that request becomes the first element in the FIFO.
- In order to grant access to the following requests in the FIFO, the task releases the resource as soon as it may.

Such requests follow semantics of read-write locks: several adjacent read request are served simultaneously, and write requests are exclusive.

Handles In contrast with traditional concurrency control tools such as POSIX mutexes, semaphores or read-write locks, access to an ORWL resource is not granted to a process, thread or task ID but to a *handle*, which is a separate data structure. There is no separation between the lock and the data protected by it (as happens with POSIX read-write locks), as the only way to access the resource is through a handle.

A handle is declared during the initialization phase. It is bound to a particular resource, and is given initial priority, which determines in which order access will be granted (via the resource FIFO queue).

A task can have multiple handles for the same resource, and each task may hold several handles that are linked to the same resource with different priorities in the FIFO. Each handle can be used to either write (exclusive access) or read (shared access) from the resource.

Critical Sections A critical section is a block of code that guarantees the desired access to a resource, either exclusively or shared, via a handle. The resource is locked on entry of the block and data is mapped in the address space of the task; on exit of the block, the lock on the resource is released and the mapping into the address space is invalidated.

The emphasis of the ORWL programming model lies in its expressiveness of a localized view of the computation. For each task it is necessary to identify the data resources that it manages and how the access to these resources relates to the access of other tasks. After that, programming of the effective computation phase is easy: ORWL guarantees that computation only takes place if data is available, routes computed data as needed by tasks, and guarantees that the computation is deadlock free and fair overall.

2.2 Example

The following example illustrates the concepts just presented. The example consists of 10 tasks numerated with an id from 0 to 9. Each task has a single in-memory resource associated with it. Each task and its resource receive the same id number.

During the initialization phase, handles are declared and tasks request write access to a resource with the same id.

During the computation phase, tasks write to the resource for a predefined number of iterations. The content written depends on the task id (if the task id is even, they write a 0 and if it is odd, they write a 1).

```
void *task(int task_id)
{
        // Initialization phase

        struct orwl_handle here;
        orwl_handle2_init(&here);

        // Request access to write location with same id
        orwl_handle2_write_insert(&here, task_id, 1);

        // Initialization phase finishes and
        // computation phase starts.
        orwl_schedule(task_id);

        for (int i = 0; i < 100000; i++) {
                orwl_handle2_acquire(&here);

                double *here_mem = orwl_write_map(&here);
                if (task_id % 2 == 0)
                        *here_mem = 0;
                else
                *here_mem = 1;

            orwl_handle2_release(&here);
        }
        return NULL;
}
```

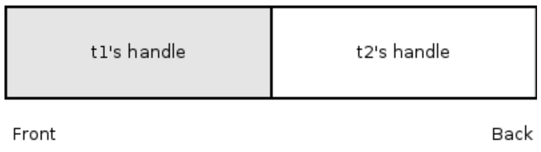
This example intends to illustrate and help to correlate the concepts in previous sections with real code, focusing in the following aspects:

- Tasks may run very similar code that differs on what is executed on run-time, using some variables as the task id. For example, some tasks write a 0 and some others write a 1 in the previous example.
- Access to a resource only happens via the handle, so there is no separation between the data and the lock as it happens with regular POSIX mutex, read-write locks, etc.
- The critical section, delimited via the `orwl_handle2_acquire()` call and it's `_release()` counterpart, guarantees that the requested access (read or write) is provided when the handle is on the top of the FIFO queue.
- The data mapping of the resource, via `orwl_write_map()`, is only valid to use inside critical sections.

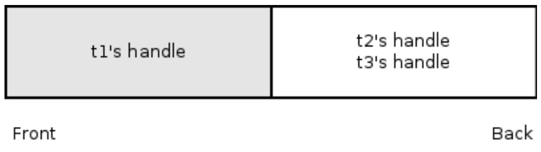
2.3 Handles, resources and their FIFO queue

When a *task* wants to access a *resource* via a *handle*, it needs to request access to read or write the resource, including which priority to use for access.

For example, a task t_1 wants to have priority 1 to write on a resource and a task t_2 wants to have priority 2 to read to the same resource. The FIFO of the resource will look like this:



If, there is also a task t_3 that wants to read with priority 2, the FIFO will look like this:



This is because read access is not exclusive, so both task can read at the same time. Please note the grey background for write and white for read access.

The priorities and types of accesses (read or write) for all handles must be given to the ORWL framework at compile time. There is an algorithm[7] to compute them from a given set of data dependencies, but in many cases they can easily be determined manually. In any case, at compilation time of an ORWL program these priorities and access types have to be known.

In addition, these priorities need to be consistent between all tasks. The only case in which different requests can access the same resource with same priority is if they are read requests. In any other case, it is not valid as the write request would not be exclusive.

There are two types of handles that are designed for different situations:

- `orwl_handle`: this type of handle is used to add a request to the FIFO queue of the resource. Access is granted to the resource via a handle of this type only once. This is okay in some cases, but iterative computations usually access more than once.

- `orwl_handle2`: this type of handle is identical to the previous, except once the access is granted, a new request is automatically appended to the FIFO of the resource.

It is important to understand how the FIFO queues evolve during an ORWL computation. As said at the beginning of the chapter, the computation over a single data block is usually performed with the following loop: wait to acquire the lock, then perform computation, then release the lock. Thus, the following example illustrates how FIFOs associated with resources evolve during such computation.

The example consists of two tasks, with id 1 and 2 respectively. Each task has a resource identified with the same number. The code for each tasks is as follows:

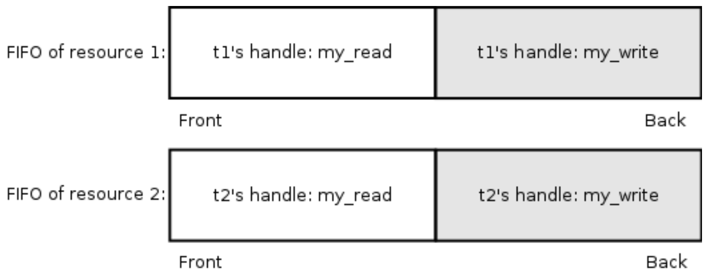
```
1  void *task(int task_id)
2  {
3          // Initialization phase
4
5          struct orwl_handle my_read;
6          orwl_handle2_init(&my_read);
7
8          struct orwl_handle my_write;
9          orwl_handle2_init(&my_write);
10
11         // the signature is: <handle>, <loc_id>, <prio>
12         orwl_handle2_read_insert(&my_read, task_id, 1);
13         orwl_handle2_write_insert(&my_write, task_id, 2);
14
15         // Initialization phase finishes
16         // Computation phase starts
17         orwl_schedule(task_id);
18
19         for (int i = 0; i < 100000; i++) {
20                 orwl_handle2_acquire(&my_read);
21                 orwl_handle2_release(&my_read);
22
23                 orwl_handle2_acquire(&my_write);
24                 orwl_handle2_release(&my_write);
25         }
26 }
```

In the example:

- Lines 2 and 3 add a read/write request to the task's own location

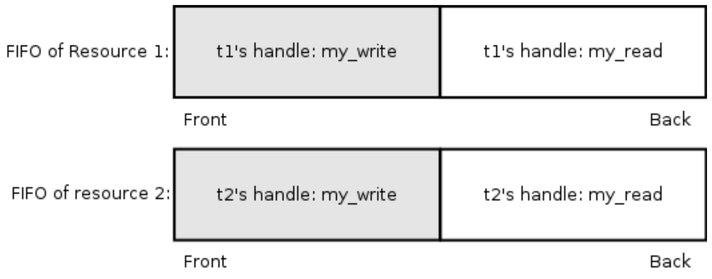
- A read lock is acquired and released
- A write lock is acquired and released
- Iterates several times doing the same procedure

The FIFO queues of the resources look like this in line 18:



This is because the reads have a lower priority, thus they are on the top.

Nevertheless, the queues will look like this when line 22 is executed in both tasks:



The difference is subtle: now the write lock is on the top of the queue, so it is ready to be acquired, and a new entry at the end of the queue is added for the read request. This is because the handle just released is of type `orwl_handle2`.

2.4 The initialization phase: important observations

Its important to note that handles can only be created during the initialization phase, by its definition. Therefore, as resources can only be accessed via handles, after the initialization phase there is a fixed number of handles and resources that can be accessed: the ones created during the initialization phase.

In contrast with general programming environments, where resources and ways to access them can be created at any point during the computation, when developing a program under the ORWL model resources can only be crated during initialization. This is an important limitation to take into account when creating a program to run under the ORWL model.

These limitations are something the work proposed here will take advantage of, as explained in chapter 4.

2.5 Main benefits

The main benefits of using the ORWL model are:

1. The program is deadlock free: as long as the initial data dependency is deadlock free, it will be deadlock free for the whole life of the program.
2. The number of tasks can easily be changed before the start of the computation.
3. Fairness to each task is guaranteed.

Proofs for properties 1 and 3 can be found here[7].

Regarding property 2, the number of tasks used is a parameter to the ORWL framework. So, it is usually trivial to change the number of tasks, as long as the framework is used properly.

Chapter 3

Reference library evaluation

The reference library can run in two different ways: several processes that communicate between them over the network or one process with several threads. This works focuses on the latter because this is how it is run on each node when running a computation on a cluster. So, improvements done when running on one multi-thread process would probably also reflect when running on clusters as well. Therefore, the focus on this work is on the ORWL use on single machine, multi-core architectures.

The library evaluation is divided in two parts: runtime evaluation, and code analysis. The runtime evaluation focuses on performance and describes the instrumentation created for assessing it. The code analysis part studies what operations are performed in the hot path and whether there are algorithms that can be simplified.

3.1 Definitions

User threads refers to threads that are doing the computation the user of the library wants to do.
Library threads are internal threads needed for book keeping.
CPU time refers to the time spent using a single CPU core.
Wall-clock time refers to the time taken to execute the program, as measured by a wall clock, regardless of how many cores are used.

For example, if a system with 140 cores executes a program using all of its cores in 1 second, the wall-clock time of that run will be 1 second, whereas the CPU time will be 140 seconds, as all 140 cores executed for 1 second.

3.2 Runtime evaluation

There are some relevant characteristics to take into account when trying to measure performance of the ORWL library:

- There are several user threads doing work.
- The library has inline functions.

- There are many library internal threads for book keeping (more than one per user thread).

So, there are many more threads than cores, as the number of user threads usually matches the number of cores.

3.2.1 Measurement strategy

The focus is on measuring CPU time as the main driver for performance. This is due to the fact that ORWL is mostly used with CPU intensive applications so the main concern is the overhead CPU time added by the library.

This is usually done via profiling, but most common techniques are sample-based (which sacrifice accuracy) and do not handle inline functions. In addition, inline functions in the library are expected to be short (run fast) but to consume a big percentage of the overhead added, as they are called very often. In order to get accurate measurements it was decided to avoid profiling and instead add a custom instrumentation.

The library run time performance measurements is approached from two complementary angles: the CPU time overhead the ORWL library adds, using the custom instrumentation, and checking that "as much CPU time as possible" is used during the computation. The idea is that if the overhead is small and the CPU time is high, the execution of a CPU bound program is very efficient.

The "uses as much CPU time as possible" part stated previously is also a critical part. It is very easy to have bottlenecks synchronizing hundreds of cores and have the penalty for synchronizing increase when more cores are used. This can even make the program slower to execute when more cores are used. Section 3.2.5 gives a guide to deal with this.

With these two guides a big picture of the current library performance is presented.

3.2.2 Instrumentation

A complementary library, called *deep instrumentation*, was created and extended the current ORWL implementation to use it.

Deep instrumentation can be thought as the UNIX `time` command, but for the CPU time spent inside the library only (not the whole process, because that would include the user computation). The main features of this instrumentation are:

- Accurate measurements of the CPU time spent inside the ORWL library
- Negligible branch miss-predictions and CPU time overhead
- Zero overhead if disabled at compile time

This instrumentation tries to minimize the CPU time overhead when enabled, at the cost of having a bigger impact on the wall-clock time. This is not a problem as the focus is on the CPU time and not wall-clock time.

3.2.2.1 Alternatives considered

The path to create *deep instrumentation* was not an obvious one and several other ways to create it were considered. This section briefly analyze the failed attempts.

The initial alternative that we explored was to instrument low-level locking primitives, measure time waiting for locks, and check if the caller was the library or the user code. The problem with this approach is that, due that the library and the user code use the same ORWL functions to wait on locks, the two were indistinguishable.

The second alternative was a tweak to the first one: differentiate the locks that the user and the library use. This would require to rename functions, or do some macro tricks at compile time. It was discarded because a fundamental flaw was found. It doesn't really matter if there is lot of time waiting as long as all cores are active. It is expected for the library to have idle threads, by design. Therefore, attempts to measure the idle time per thread were discarded as didn't provide useful information.

Learning from the above, a third alternative considered was to measure the CPU-time spent inside the library. This was the first attempt to measure the consumed CPU-time, instead of the idle CPU-time. To measure the time inside the library `ltrace` was considered and rejected, as it was not actively maintained, only worked on a few architectures, and most importantly it throw segmentation fault errors. Another way to measure CPU-time considered was using `gcc` with the option `-finstrument-functions` and custom callbacks created to that end. We attempted this route, as it should even work with inline functions, but ended up discarding it as the functionality did not work and gcc had a bug reported about it for almost a year. This was eventually fixed in gcc, three years later *deep instrumentation* was finished.

The fourth alternative explored was to use custom code to instrument the inline functions. To store the CPU-time an array was used, with an element for each thread. However, at initialization there was no easy way to know the number of threads, hence there was no easy way to know the number of elements the array needed. Work-arounds were found for that limitations, nonetheless it was a complex solution as it had to be cache-aligned, using posix_memalign() and other tricks to reduce the overhead.

By the time the prototype for the fourth alternative was working an easier solution was found: mix the previous attempt with thread-local variables and atomic operations. The fifth attempt was, then, to re-write it using this idea.

At this point, the instrumentation was using `getrusage()` to measure time. It was supposed to work for obtaining the CPU-time and had extra flexibility for returning more information if needed. However, when doing some preliminary tests, it was discovered that it had low precision (in the order of 10ms) which rendered it unusable for our use case, since the windows of time measured were many orders of magnitude smaller

To replace `getrusage()`, we explored using the RDTS instruction, which is an assembler instruction for x86 architecture that measures CPU-cycles. The problem, though, is that it doesn't take into account CPU migrations, hence it cannot be trusted for an application that has many more threads than cores (as it is expected for threads to be migrated in that case). Although using CPU affinity to avoid migrations when running with instrumentation is an option it was discarded after all: the consequences when using cpu affinity with programs that have several times more threads than cores are difficult to estimate, as it can have a big hit on performance, and the instrumentation might end up measuring something very different than the original program performance.

Therefore, we explored how linux-perf dealt with this problem and found that the timing information was retrieved using hardware support (confirmed by looking at the code and, also, by talking to its maintainer, Arnaldo Carvalho de Melo). Hardware support was very new at the time, and imposed a significant constraint into the portability and usability of the instrumentation. Finally, a promising alternative was found: to use clock_gettime(). This alternative become the heart of *deep instrumentation*, which is explained in detail in the following sections.

3.2.3 Meassuring per-call CPU time

The *deep instrumentation* implementation depends heavily on the `clock_gettime()` syscall. This syscall provides per-thread CPU usage. It is similar to RDTS, but it is POSIX compliant, not x86 dependent and, when implemented with OS support, takes into account CPU migrations. The latter is critical as the programs that use ORWL have many more threads than cores and see thread migration frequently.

Measurements are taken as follows:

- When a library function is called, a call to `clock_gettime()` is made to know the current CPU usage by the calling thread
- The current CPU usage is stored in a thread local variable
- When the library function that was originally called ends, another call to `clock_gettime()` is made to get the current CPU usage
- The elapsed CPU usage inside the library is calculated and stored in a thread local variable

The overhead is dominated by the call to `clock_gettime()`, as using thread local variables is usually cheaper than a syscall.

Last but not least, before the ORWL computation is finished, all thread local variables are added up, with the result being the total CPU time spent inside the library. This is a cheap way for all threads to collaborate without using locks or synchronization during the computation, thus it avoids having a big impact on performance. The synchronization is made at the end, after the computation is finished.

Although the current functionality is to expose the time spent by the ORWL library, without any kind on granularity where that time was spent inside the ORWL library, some granularity can be easily added. It is possible to extend its output, for example, to the CPU time spent on different ORWL subsystems. This can be done by saving the elapsed time on different (per subsystem) variables.

3.2.4 Adding instrumentation

The name *deep instrumentation* is due to the major overhead this instrumentation adds. When compiling the ORWL library with instrumentation 2 system calls are added per library call. The wall-clock time overhead is not negligible when using *deep instrumentation.* Nevertheless, the CPU time overhead is smaller.

An important requirement is that the instrumentation should be possible to be disabled and in that case there should be no performance penalty at all. This is very important because ORWL programs are CPU intensive and cannot afford reducing it's performance to incorporate instrumentation. *Deep instrumentation* achieves that by adding zero overhead when compiling without it. All instrumentations calls, when disabled, are expanded to NOPs that are then optimized out by the compiler.

By instrumenting all library function calls, we ensure the resulting measurements are complete, and not based on sampling.

Special attention is needed to handle correctly the case where an instrumented library call ends up calling (directly or indirectly) another library function that is instrumented too. To handle this, a counter of how many "levels" inside the library is used and only the outermost does the accounting.

3.2.4.1 Definitions

As the focus of the instrumentation is CPU time, the use of the term time in the following sections should be interpreted to mean CPU time.

3.2.4.2 Instrumentation of inline functions

To instrument an inline function, calls to functions named _start() and _stop() were added around each function's body. These functions are exported by the deep instrumentation library, and inlined to avoid introducing unexpected function calls. Also, their code is carefully written to minimize overhead.

3.2.4.3 Instrumentation of non inline functions

To instrument non inline functions, a small shared library is used with `LD_PRELOAD` to intercept library functions. The main reason is that it is easier compared to adding code to every function needed to overload (as it is done with inline functions) and is simpler to maintain. Wrappers are automatically generated.

A way to get the pointer to the real function is added to the library initialization, overloading `orwl_init(),` too. After that, the instrumentation stores pointers to these functions and they are used when needed.

3.2.4.4 Instrumentation of library internal threads

The ORWL library has a thread pool implementation that is used for library internal threads and for user tasks. A given thread blocks until there is work to do, does it, and blocks again. As expected, the instrumentation adds `_thread_start()` and `_thread_stop()` calls around each execution, which are analogous to the ones shown above.

The only difference with the `_start()` and `_stop()` functions used before is that the thread variants checks if the thread is a library internal thread and returns immediately if not.

3.2.4.5 Estimate CPU-time spent inside the library

When using *deep instrumentation* an estimation of the time inside the library is made, $EstimatedTL$. This estimation also accounts the overhead of the instrumentation. In other words:

$$EstimatedT_L = T_L + \delta$$

where TL stands for the time inside the library when running without instrumentation.

Consider the CPU time when running with *deep instrumentation* enabled as calculated with the `time` unix command. It returns the CPU time spent in user-space and system. The sum is the total CPU time used by the program:

$$\alpha_1 = U_{ser} + S_{ystem}$$

The estimation of time spent inside the library, hence, can be estimated as:

$$\frac{EstimatedTL}{\alpha_1} \times 100$$

This percentage estimates the CPU-time inside the ORWL library. And as the instrumentation does not add much CPU-time overhead, it seems a reasonable estimation (the absolute CPU time when running with and without instrumentation for the whole computation was measured to change about 1-2%).

3.2.4.6 Requirements from the OS and libc

This technique uses `clock_gettime()` with `CLOCK_THREAD_CPUTIME_ID` as clock_id to get the CPU usage of the current thread. Some platforms used to gives bogus results for this function and, as expected, *deep instrumentation* will give bogus results on those platforms too.

For example, old versions of glibc (< 2.4) implemented this clock on many platforms using timer registers from the CPUs (TSC on i386, AR.ITC on Itanium). These registers may differ between CPUs and as a consequence this clock may return bogus results if a thread is migrated to another CPU.

Since glibc 2.4, the wrapper functions for the system call avoid the problems mentioned above by employing the Linux kernel implementation of `clock_gettime()` when it's available (Linux 2.6.12 and later).

When using Linux and glibc, Linux $\geq$ 2.6.12 and glibc $\geq$ 2.4 is needed. When using something else, it is recommended to check that the `clock_gettime()` syscall works as expected on that platform.

It is worth to mention that the manpage for `clock_gettime()` had an outdated note about it not working on SMP systems. A bug was filled and submitted a patch to fix it after some investigation to know that it was indeed outdated. The fix is already merged upstream as commit 78638aa.

3.2.4.7 Other benefits

Deep instrumentation has some advantages when running on non x86-64 platforms as well: there is no need for profiler support (the valgrind tool suite, for example, is not ported to many architectures) nor hardware requirements, like performance counters.

As mentioned above, the only the requirements are: `clock_gettime()` with OS support and a C compiler.

A small set of requirements can be very useful. For example, on the Xeon Phi co-processor programs must be compiled with *icc Intel C compiler*, with special flags for the Xeon Phi, and the platform it runs is Linux $\geq$ 2.6.12 with glibc $\geq$ 2.4. In other words, there are quite a few restrictions but none of them are a problem when using *deep instrumentation*.

Using *deep instrumentation* turned out to be an easy way to get measurements on any platform. We discovered this later on the project, when trying to use the Intel Xeon Phi, and it was a pleasant surprise.

3.2.4.8 Disadvantages

The instrumentation has some disadvantages, too, that should be taken into account when using it for an analysis. One already mentioned is that it introduces some runtime overhead.

Another disadvantage is that it uses a syscall. This syscall cannot be served in user-space and has a non negligible overhead in the wall-clock time, which can cause several problems. For example, if the thread that performs the syscall is holding a spinlock, this thread will take more time to unlock it, causing other threads to consume more CPU too. Furthermore, in some cases this may affect how the operating system schedules threads to run during the computation.

To validate this was not a significant issue during the analysis, several measures were done. It was validated that the CPU time when running with and without instrumentation was very similar (close to 1% overhead); we also compared most metrics exposed by linux-perf (branch mis-predictions, etc.), and removed the usage of spin-locks.

3.2.4.9 Future improvements

Linux provides a kernel mechanism for exporting a carefully selected set of kernel space routines to user space applications so that applications can call these kernel space routines in-process, without incurring the performance penalty of a context switch that is inherent when calling these same kernel space routines by means of the system call interface. This mechanism is called vDSO (virtual dynamically linked shared object).

Several syscalls use it, and `clock_gettime()` uses it depending the parameters it's called with. At the time of this writing (Linux 4.15) this mechanism is not used for the parameters the *deep instrumentation* uses to `clock_gettime()`. It was considered to write a patch to the Linux kernel to make it use the vDSO functionality, and even talked to Peter Ziljstra (Linux Scheduler maintainer) for advise. But it is left as future work.

Another topic for future improvements is making *deep instrumentation* easy to plug with any other program. The code itself is agnostic of the program it instruments and can be plugged to any program. In fact, *deep instrumentation* was used to instrument the new proposed library in chapter 4 and it was a trivial task. Nevertheless, right now it needs to be injected into the program manually and it is not a standalone project. In the future it can be separated as a standalone project and modified to automate most of the tasks to instrument any program.

3.2.5 Estimation of waiting time

Consider the time it takes for a program to execute. It is easy to see that the CPU time of that execution is not more than the wall-clock time it took times the number of cores/threads. This will be called ideal CPU time. In other words, let *cores* be the number of cores and *elapsed* be the wall-clock time it takes to execute a program. The ideal CPU time, then, is defined as:

$$ideal\, cpu\, time = cores \times elapsed$$

For example, consider a program running on a 4 cores CPU and takes 3.03 seconds to execute. The ideal cpu time for that program is to use the 4 cores during the whole 3.03 seconds it took to run. In other words, the ideal cpu time for that program is 4×3.03 seconds.

Consider also the CPU time a given program execution has. Let it be *cpu time*. And consider the ideal cpu time for that execution: the number of cores the system has and the wall-clock time the execution took. Another interesting number is the percentage of the *ideal cpu time* an execution used:

$$\frac{cpu\, time}{ideal\, cpu\, time} \times 100$$

If the percentage is high, then there is not much time waiting during the execution. Time spent waiting is, by definition, time not using the CPU.

To see how this number changes with the number of cores can help to see if the library scales well. This is very important, as coordinating tasks on a high number of CPUs needs coordination and it is very easy to have bottlenecks on this part. This analysis helps to make sure if this happens or not and how big of a problem it is.

3.3 Results and interpretation

Previous section introduced techniques, using a custom instrumentation, to measure performance of the ORWL library. They are used here to understand the ORWL library performance and evaluate if it seems possible to improve it.

The the well-known block cyclic algorithm for dense matrix multiplication is used for the evaluation. This algorithm uses ATLAS (Automatically Tuned Linear Algebra Software) for optimal performance and has been tuned for other ORWL publications[12]. A real world algorithm is used due to synthetic benchmarks and simple examples being susceptible to present artifacts, like exacerbate scalability problems or the absolute opposite, that may not happen on real world examples.

The test were run on an AMD Opteron Processor 6172 with 24 cores, running at 2.1 GHz under Linux 4.15 x86-64 and with 32 GB of RAM memory.

The following figures show the percentage of the ideal CPU time used, the percentage of CPU time spent inside the ORWL library and the wall-clock time for different matrix sizes.

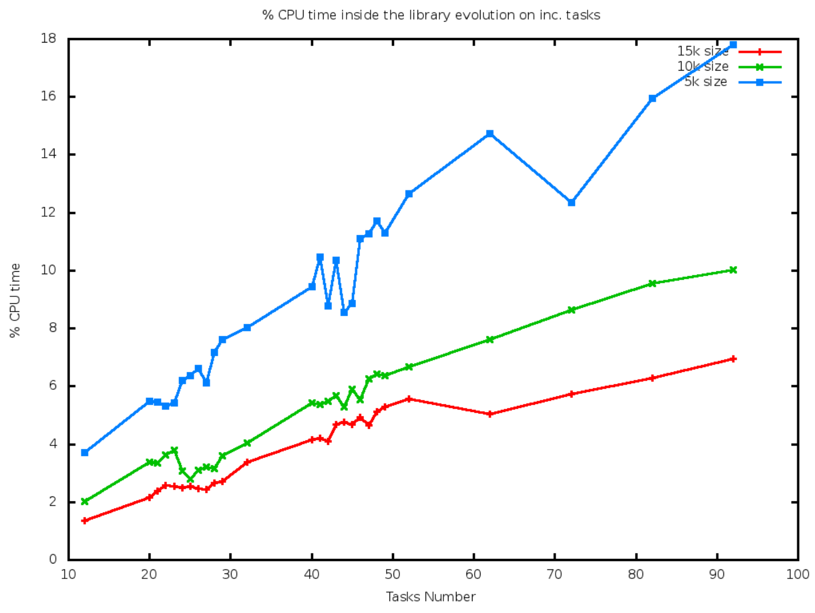


Figure 3.1: Percentage of cpu time spent inside the library when the number of ORWL tasks increases

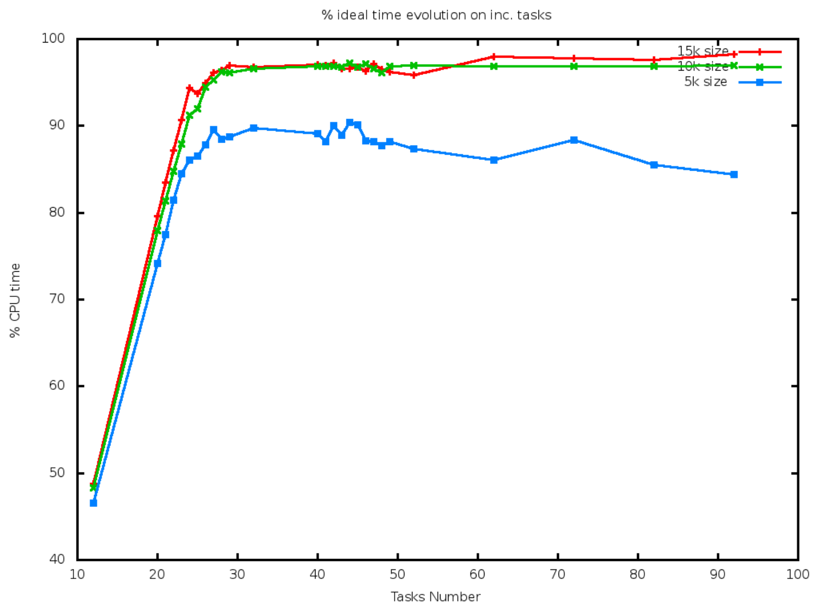


Figure 3.2: Percentage ideal cpu time when the number of ORWL tasks increases

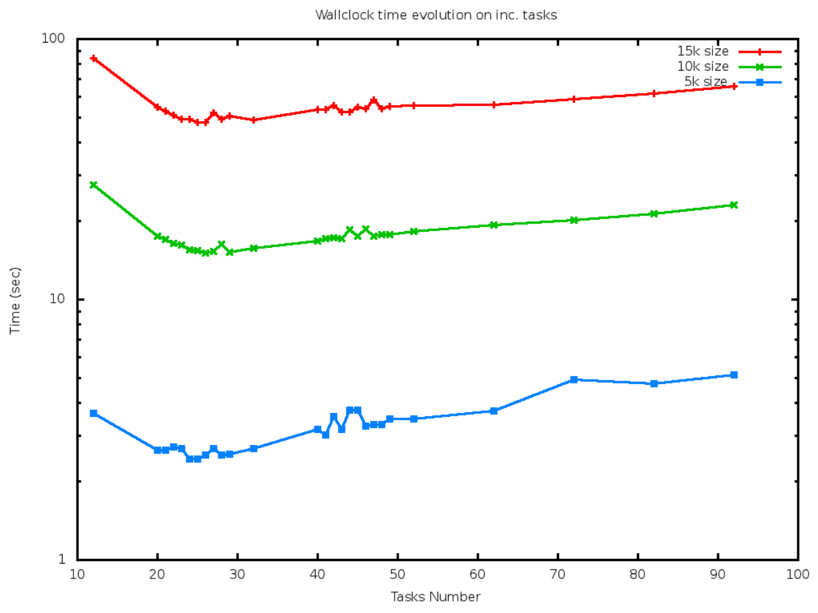


Figure 3.3: Wall-clock time when the number of ORWL tasks increases

Figure 3.1 shows that the percentage is above 5% for the matrix with $n = 15000$, when the number of tasks is bellow 30. For $n = 10000$ it is close to 5%, too. And, although $\sim 5\%$ is not a extremely high overhead, it not a low percentage of CPU time to consume from a CPU intensive computation. Hence, it is enough to hint about possibilities to improve the CPU time spent inside the ORWL library.

In Figure 3.2 the percentage of ideal CPU time is higher when the matrix is bigger. Also, it is higher when the number of tasks is higher. These are desirable outcomes. However, the percentage is quite low when the matrix is small and uses less than 30 tasks (bellow 85%), thus this suggest that ORWL has opportunities for further enhancement on smaller computations.

The wall-clock time hits a minimum value when using 24 tasks (the number of cores the system has), as can be seen in figure 3.3, and shows that adding tasks above that value is slower for the overall computation. For that reason, this indicates that for more than ∼24 tasks the values in previous figures should be ignored. In particular, the high percentages achieved in previous figures with more than ∼24 tasks are, consequently, not relevant.

Therefore, as the ideal CPU time is a quite low percentage for values bellow ∼24 tasks (bellow 85%) and the CPU time inside the library suggest there is room for improvement, these analyses hint that, overall, there might be opportunities for further enhancement in the ORWL library.

3.4 Code Analysis

Tn order to improve the performance on multi-core architectures, besides run-time measurements as described above, we performed a fair amount of code analysis, with an emphasis on the hot paths. These will be used, in conjunction with previous benchmarks, as motivation to try other algorithms and data-structures that might be more efficient.

The ORWL library is a complex piece of code. It is written in C, with heavy macro usage and tricks to work-around limitations in the C programming language, like using default arguments to functions. Other examples to show it is not trivial are that it has a thread pool implementation and a reference counting mechanism.

As usually happens with complex code over the years, it is not a trivial task to understand when first reading it. For the following observations the reader should probably start by the macro for ORWL_SECTION and the lock insertion/release when using `orwl_handle2`. Those are enough for the scope of this section.

The observations about the ORWL implementation that are useful to compare with the new proposed design are:

1. The FIFO queue used is implemented using an arbitrary long queue and needs a memory allocation on insertion
2. When using one of the most common handle types (`orwl_handle2`), releasing a lock implies to:
    (a) Hold a mutex
    (b) Remove the current element on top of the queue
    (c) Do some atomic ops
    (d) Allocate memory to insert in the queue again

The hypothesis around these are that a not contended mutex is usually a cheap operation, the allocations are probably fast and atomic operations are not a big issue performance-wise.

And there is a fair justifications for all these steps. The mutex is needed to coordinate with remote nodes when a lock is added. The atomic operations are needed to avoid other types of locks. And a queue with dynamic length is quite standard, easy to implement and gives some flexibility if needed in the future.

As seen in chapter 2, a typical use of ORWL for iterative computations does a heavy use of `orwl_acquire()` and `orwl_release()` with `orwl_handle2` handle type. And that is why those functions are in the hot-path of an ORWL computation and important to look at.

However, the expectation about those steps being fast can be untrue and definitely worth challenging. If a function in the hot-path of the computation can be improved, it can have a big effect on the overall improvement.

As seen also in chapter 2, there are some important properties about the initialization phase that seem ignored by the current library. Hence, it's not crystal clear that this is the more efficient way to do it. Furthermore, some questions are not easy to answer, like:

- Is a dynamic length queue really needed and worth the performance costs?
- Is there a way to reduce the locking in these hot-path functions?
- Can this be implemented in a more efficient way?

The next chapter will shed some light into this.

Chapter 4

Proposed design for multi-core architectures

To try out the proposed designed, explained in later sections, a new library was written from scratch in plain C. The design and it's data-structures is quite different from the ones used in the current ORWL library. When trying to modify the current ORWL library to try them out, some details were not easy to figure out. Therefore, this new library was written as a small proof of concept that deals, as much as possible, only with the essential complexity needed to try those algorithms.

The simplified data-structures proposed with the reduced scope in it's functionality resulted in some other unexpected benefits, besides the performance gains: really small in LOCs, reduced object size and compile-time, easy to debug with gdb and valgrind and simplicity to run.

This chapter explains in detail the proposed data-structures, algorithms and their correctness, the motivations for these changes, the implementation per-se, and the main differences between both ORWL libraries.

4.1 Definitions

The **new ORWL library** refers to the library created with the proposed design explained in this chapter. It is also called *tiny ORWL* or *torwl*.
The **current ORWL library** refers to the reference implementation and is the library that was analyzed in previous chapters.

4.2 Why write a new library?

The main reason is simplicity. The ORWL API is quite simple and seems easy to fulfill with the new design.

In addition, as the data-structures proposed here and the ones in the current ORWL library are quite different, they do not fit well together. The current ORWL library has complex semantics for internal structures (like a thread pool implementation, several internal locks mechanism, uses reference counting to delete unused objects, etc) and it proved too hard to correctly mix the current library with the new algorithms. The devil really is in the details.

Another reason to write a new library is that the current ORWL library doesn't support running with gdb and valgrind. Those are very valuable tools when writing such a complex piece of code. It is, of course, possible to fix that limitation (it is a C program after all). But it is not trivial and doing it postpones challenging the main hypothesis: see if these new data-structures are useful performance-wise in an ORWL computation.

Therefore, a new library was written from scratch. It is possible to use with valgrind and gdb and easy to run without initialization scripts.

4.3 Proposed changes in algorithms and data structures

This section explains the changes proposed. The critical changes are in the ORWL hot-path of a computation and this is where most of the explanations focus. The rest of the code just accommodate to work with these changes, it does not contain anything complex nor important performance-wise.

The explanations try to make obvious the reasoning and path it was explored. Of course this was not so linear in real life, but hopefully the explanation makes it seem natural to go down this route.

The new orwl implementation treats `orwl_handle` and `orwl_handle2` almost indistinctly. Therefore, examples only for `orwl_handle2` are shown in most sections.

4.3.1 Important observations

The proposed design tries to take advantage of some invariants imposed by the ORWL model to make more efficient the hot-path. In this section some observations are made that will lead to new ideas to explore in the hot-path.

In chapter 2.4 there is one observation that states:

- There is a fixed amount of resources and handles after the initialization: the ones created during the initialization

Each handle is represented by an element in the location's FIFO queue. One immediate consequence of noting that there is a fixed amount of handles is that the FIFO queue size is known after the initialization. And as no handles can be created nor destroyed, the size is constant during the computation.

This implies that the queue size is fixed after the initialization phase and, then, it might be possible to manage the queue without memory allocations. This is the *first observation* that the proposal will leverage.

Another key observation arise from looking at handle type `orwl_handle2` and how FIFO queues evolve during an ORWL computation. The definition for both handle types are:

- `orwl_handle`: this type of handle is used to add a request to the FIFO queue of the resource. Access is granted to the resource via a handle of this type only once. This is okay in some cases, but iterative computations usually access more than once.

- `orwl_handle2`: this type of handle is identical to the previous, except once the access is granted, a new request is automatically appended to the FIFO of the resource. Thereby the system ensures that the access to that same resource is granted iteratively in the same initial FIFO ordering.

Consider a FIFO queue filled with only `orwl_handle2` handles during an ORWL computation. Then, the queue will always have the same handles (instead of new or different handles during the execution), in the same relative order, modulus a rotation. Because the requests are inserted again.

In other words, let n be the number of handles a FIFO queue has and let $x_1, x_2, ..., x_n$ be the handles type `orwl_handle2`. Let the queue at the start of a computation have the requests in the following order:

$$x_1, x_2, x_3, ..., x_n$$

After the first item (x_1) is processed, it is deleted from the queue and added back again, by definition of `orwl_handle2`. Then, as it is a FIFO queue, the new order is:

$$x_2, x_3, ..., x_n, x_1$$

The next item processed will be x_2, with the expected results. Therefore, during a computation the queue will always look like:

$$x_i, x_{i+1}, ..., x_n, x_1, x_2, ..., x_{i-1}$$

In this way, it is easy to see that the elements of the queue will always be the same handles, in the same relative order, modulus a rotation. Thus, an easy way to model this FIFO queue, where items do not change and are added right back again when processed, it is using a circular buffer.

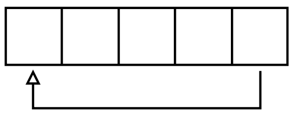


Figure 4.1: Circular buffer using an array

A circular buffer can be implemented with an array as the figure shows, going back to the first element when the array size is met. This can easily handle this types of queues with no need for memory allocations nor complicated operations: just moving a pointer to the next element.

To also handle `orwl_handle`, and not just `orwl_handle2`, a small adaptation can be used over a circular buffer: mark `orwl_handle` items as processed once they were processed. Then, when moving the pointer to the next element of the queue, it needs to skip all handles type `orwl_handle` already marked as processed.

Hence, the *second observation* is that a fixed-size FIFO queue with handles type `orwl_handle` or `orwl_handle2` items can be implemented using a circular buffer and needs cheap operations only: moving a pointer and no memory allocations.

4.3.2 From idea to implementation

The presented observations are nice ideas to implement new algorithms and data-structures for ORWL. But to successfully do it some other problems need to be solved, like choose a mechanism to wake-up threads or how to take advantage of these structures without using locking schemes that slow things down. In other words, this idea seems promising but it is still not clear if it can be used without major drawbacks. Next sections will deepen on these problems.

4.3.2.1 Locations, handles and resources overview

Before discussing in detail how to satisfy the ORWL API, a brief summary of locations, handles and how threads interact with them is presented.

A program might have any numbers of threads, let's say *threads_nr*. Each thread has its own `id`, identified with a number from 0 to $threads\_nr - 1$. Each thread, by default, also has its own location that is identified with the same number.

Each location, during the initialization phase, receives concurrent requests from others threads to read or write it with certain priority. After the initialization phase, the locations are accessed by means of `orwl_acquire()` and `orwl_release()`.

The functions that allow a specific handle to access a given location have the following signature:

```
void orwl_handle2_write_insert(struct orwl_handle *h,
    size_t locid, size_t prio);
void orwl_handle2_read_insert(struct orwl_handle *h,
    size_t locid, size_t prio);
```

4.3.2.2 ORWL hot-path API

The hot-path of an ORWL computation, as was discussed previously, is calling functions that access critical sections. To access a critical section a program uses `orwl_acquire()` and its `orwl_release()` counter-part. Their signature is as follows:

```
void orwl_handle2_acquire(struct orwl_handle *h);
void orwl_handle2_release(struct orwl_handle *h);
```

The acquire function takes the handle as an argument and blocks until its on the top of the corresponding FIFO queue to grant access. When access is not needed anymore, the resource must be released with `orwl_release()`. When all locks are released, the next item of the FIFO queue is moved to the top.

This has two consequences that are relevant when creating an implementation:

- Once an item is on the top of the FIFO queue, all threads waiting for it should be unblocked
- Once all handles are released for the item on the top, the next item in the FIFO queue should be processed

Therefore, a way to wake-up or unblock threads should be used and there should be a way to know when all handles for the item on top of a location have been released.

How to wake-up threads and do it efficiently is a problem on itself and it is not the main goal of this work to have a detailed analysis of all the alternatives. Semaphores, Linux futex and condition variables were tried as mechanisms and semaphores were chose. Section 4.4 explains the reasoning.

To realize when to process the next item of the FIFO queue, it was decided that `orwl_release()` will:

- Count the number of calls to `orwl_release()`
- When the last handle is being released, it process the next item in the FIFO queue

And a way to achieve this in a correct and performant way is as follows.

The number of calls to `orwl_release()` is always an integer number. Hence, an easy way for different threads to collaborate counting calls to `orwl_release()` in an efficient way is using atomic operations (i.e avoids mutex or other more expensive mechanisms). Therefore, when a lock is released an atomic operation to modify the count of released handles is done. Only one thread will account for the last released handle and this thread has the extra task of processing the next item in the FIFO queue.

It is important to note that exactly because only one thread accounts for the last released lock and does the extra process, that thread is the only to process the FIFO queue at a given time. And this has an important consequence: the FIFO queue doesn't need to be thread safe, as only one thread modifies it at any given time. This is another advantage performance-wise the *torwl* implementation will use in it's favor.

4.3.3 New data-structure for ORWL resources queues

Previous sections discussed how to address some issues when creating a new implementation while also propose some ideas to implement the internal FIFO queue with a circular buffer. These concepts are quite coupled between them, so a high level overview of the algorithm is presented here and details are expanded in later sections for better clarity. In particular, this section explains a high level overview of how to articulate the initialization phase with circular-buffer FIFO queues.

On one hand, there is the circular buffer FIFO queue. The proposed implementation, however, has some requirements:

1. The FIFO queue size must be known at creation time
2. It is not thread safe

The reasons for this are:

1. The circular-buffer FIFO queue is implemented using an array, which always is fixed-size. Hence, if the size is not known at creation time the array underneath the circular-buffer FIFO queue will have to be destroyed, created again with the new size, and data moved.

2. As was explained, the circular-buffer FIFO queue does not need to be thread safe during the hot-path. This make it faster during the hot-path (the critical part this algorithm tries to optimize)

On the other hand, the initialization phase needs to articulate with these FIFO queues. During the initialization phase an arbitrary number of handles are created for a resource. Therefore, the final circular-buffer FIFO queue cannot be created before the initialization phase finishes. But as was noted in section 2.4, this size is constant after the initialization phase.

A simple algorithm to mix the initialization phase with circular-buffer FIFO queues with these restrictions can simply:

- During initialization phase: accumulate the handle requests and it's characteristics (orwl handle type, priority, etc.) in some collection, like a linked list
- After the initialization phase finishes, the size for each FIFO queue is known. Therefore, the algorithm can iterate over all the accumulated handle requests and create a circular-buffer FIFO queue for each location

This is a high-level overview of how the circular-buffer FIFO queues idea will be integrated in *torwl* and more details are in the next sections.

4.3.4 Critical ORWL functions implementation

This section explains in detail the implementation of the most important functions in *torwl*. All the previous discussed ideas and scope-reduced observations come together to join all the pieces in new clear algorithms and data structures in *torwl* that provide the complete functionality.

For better clarity, the pseudo-code shown in this section doesn't implement error handling. The *torwl* implementation, though, does check for errors and even has an optional mechanism for easy debugging common errors.

4.3.4.1 The initialization phase

During the initialization phase, request are accumulated as explained. These are accumulated in an static global array:

```
struct orwl_tmp_loc {
        size_t id;
        size_t prio;
        size_t waiters_nr;
        bool read_only;
        orwl_ww_t *handler;
        bool active;
};

#define LOC_SIZE 1000
static struct orwl_tmp_loc tmp_locs[LOC_SIZE];
static pthread_mutex_t tmp_locs_mtx =
    PTHREAD_MUTEX_INITIALIZER;
```

A long fixed-size array is used in conjunction with a bool to know if the position in the array is already used or not.

Calls to add read or write requests, then, iterate this array on the valid positions (`active=true`), matches for the `id` and `prio` and add the request. A pseudo-code for this is:

```
void orwl_handle2_write_insert(struct orwl_handle *h,
    size_t locid, size_t prio)
{
        pthread_mutex_lock(&tmp_locs_mtx);

        orwl_handle2_insert(h, false, locid, prio);

        pthread_mutex_unlock(&tmp_locs_mtx);
        return;
}

void orwl_handle2_read_insert(struct orwl_handle *h,
    size_t locid, size_t prio)
{
        pthread_mutex_lock(&tmp_locs_mtx);

        orwl_handle2_insert(h, true, locid, prio);

        pthread_mutex_unlock(&tmp_locs_mtx);
        return;
}

void orwl_handle2_insert(struct orwl_handle *h, bool
    read_req, size_t locid, size_t prio)
{
        // Check if entry for (locid, prio) exist
        // Returns NULL if it doesn't exist
        struct orwl_tmp_loc *loc = find_loc(locid, prio);

        // Check if it's a read request
        if (read_req && loc != NULL) {
                loc->waiters_nr++;
                return;
        }

        // A new location is allocated
        struct orwl_tmp_loc *new_loc = find_loc_free();

        // Init the tmp location with the id, prio, etc.
        orwl_tmp_loc_init(new_loc, locid, prio, read_req,
             h->handler);
}
```

The functions are protected with a mutex as they can be called concurrently. This way access to the global array is serialized and thread safe.

After all requests are created, each thread calls `orwl_schedule()`. After this function is called on all threads, the initialization phase is finished and the circular-buffer FIFO queues can be created. Each orwl task calls `orwl_schedule()` with its default location id as parameter, and each task has a different id. The pseudo-code for `orwl_schedule()` is:

```
void orwl_schedule(size_t locid)
{
        // Make sure everyone finished adding locks
        pthread_barrier_wait(&schedule_start);

        // Load all requests in orwl_tmp_loc array
        // to a circular-buffer FIFO queue
        create_loc_from_tmp_loc(locid);

        // Make sure all finished creating the locations
        // before starting the computation
        pthread_barrier_wait(&schedule_end);

        // Wakeup the first one
        orwl_loc_wakeup_curr(locid);
        return;
}
```

4.3.4.2 The computation phase

After the initialization phase finishes, the critical sections are accessed by means of `orwl_acquire()` and `orwl_release()`.

The `orwl_acquire()` function waits for the item to be on top, using the abstraction for the synchronization primitive created:

```
void orwl_handle2_acquire(struct orwl_handle *h)
{
        orwl_ww_wait(h->handler, &h->handler_ctl);
}
```

The `orwl_ww_*` is the abstraction created for synchronization primitives. In the case of a semaphore, for example, `orwl_ww_wait()` calls `sem_wait()` underneath.

The `struct orwl_handle` is not expanded here for simplicity. It is a simple struct that has the handle's relevant properties (if it's a read or write handle, the location associated with, etc.).

The release function, though, is more complex as it has to count all threads that have released the lock to process the next item in the queue when appropriate. The pseudo-code is as follows:

```
void orwl_handle2_release(struct orwl_handle *h)
{
        // Get loc associated with handle
        struct orwl_loc *loc = h->loc;

        // Get the request associated, it has the number
        // of threads waiting, if its read/write, etc.
        struct o_rwl_loc_req *req = orwl_loc_curr(loc);

        // Do the release
        orwl_ww_release(h->handler, &h->handler_ctl);

        // It the curr request is a writer, we are the
            last one to release it
        if (!req->read_only)
                goto wakeupnext;

        // If we are the lasts to release the lock, wake
            up the next one
        // XXX: atomic_fetch_add() returns the value
            before the add
        size_t released_nr = atomic_fetch_add(&req->
            released_nr, 1);
        released_nr++;

        if (released_nr == req->waiters_nr) {
                atomic_store(&req->released_nr, 0);
                goto wakeupnext;
        }

        return;
wakeupnext:
        orwl_loc_wakeup_next(loc);
        return;
}
```

This way, the abstraction is used to release the acquired lock calling `orwl_ww_release()` and it uses atomic operations to count the number of times `orwl_release()` has been called with the location's current item.

It is critical to note that as atomic operations are used `atomic_fetch_add()` returns a value that makes the last `if` statement true only in one thread. The rest of the code executed after that is safe to execute concurrently between threads releasing the lock (it is a `return` statement on all threads except on one). This observation is key to make sure some common race-conditions do not happen.

4.3.4.3 Summary

All the critical function for the ORWL implementation, and how they articulate between each other, have been explained in detail. This is the proposed algorithm that, as can be seen, has the following characteristics:

- Uses circular-buffer FIFO queues during the ORWL hot-path
- This implies only waking up threads and moving a pointer to the next item on the queue are the only operations done on the hot-path

- The FIFO queue implementation is not thread safe, as `orwl_release()` uses atomic operations to guarantee that only one thread will modify the structure at a time
- This implies that the FIFO queue implementation doesn't use synchronization mechanisms and can be implemented in very lightweight fashion (an array and some pointers operations)

These are the most important changes this new implementation proposes, as it simplifies a lot the hot-path compared to the current ORWL library (see section 4.6).

4.4 Design decisions

As it was already stated, there are some decisions that are not core to the library but needed to make it work. This section explains some of those decisions and the motivation to chose between the options.

torwl uses semaphores as a synchronization primitive internally, as briefly mentioned. They are used when a user calls to functions like `orwl_acquire()` or `orwl_release()`, that imply waiting or waking up other threads. Some other mechanism were tried (like condition variables, futex, etc.) but in the end semaphores were used. Semaphores are efficient and their semantics correlate naturally with a number of threads to wake up: set the value of the semaphore to the number of threads waiting. More importantly, semaphores are portable (part of POSIX Realtime Extension). Nevertheless, this is a key part of the hot-path and can be revisited in the future.

To easily review this in the future and compare different synchronization primitives, an abstraction to wake up threads is created. This abstraction allows to choose the synchronization primitive used at compile time. Right now only semaphores are supported, but it should be easy to add new primitives. This way, *torwl* should be easy to extend to compile with different flags, like use semaphores or futex as synchronization primitives, and comparisons can easily be done.

During the initialization phase a fixed-size array is used to accumulate the arbitrary number of handles. The reason is that a small number of handles are usually used and a big static array is usually more than enough. If, for some reason there is no room, it fails with a clear error. This was never hit in practice, though, and is easy to change to a a bigger array or even to a linked list, to handle an arbitrary number of handles.

In addition, the initialization phase is not part of the hot-path of an ORWL computation and, therefore, not being optimized by the proposed changes. All concurrent calls to request access to locations are serialized, with an obvious impact in their parallelism and performance. However, this decision can be revisited later, if needed. The authors, nevertheless, suppose that this won't be needed in the near future as usually time the initialization is a very small percentage of the time used by the ORWL computation.

4.5 Usage and dependencies

The library dependencies are:

- C99 compiler with gcc extensions
- make
- the only vendored dependency is P99 as a git-submodule
- POSIX compatibility (POSIX.1, POSIX Real Time Extension)

The P99 dependency is used for atomic operations and to provide the compiler with branch prediction information (if a condition is likely or unlikely to happen). Recent versions of some compilers (like `gcc` and `clang`) implement atomic operations, so the dependency with P99 might be removed. However, other popular compilers like `icc` do not implement them yet.

The library is instrumented with the *deep instrumentation* presented in chapter 3. This instrumentation is key to understand the cpu time inside the *torwl* library.

4.6 Main differences with current ORWL library

Differences can be considered in many aspects and this section deals with the main ones. Some where a side effect of others decisions, but still worth to mention.

- Algorithms and data structures: the computation hot-path is changed a lot to require only a few operations. This is expected to have a big improvement on performance and is achieved using the circular buffer, as explained in the sections above.
- Compilation and object size: Compilation time is reduced compared to the current ORWL library. The new library is small in LOC, does not have many dependencies and the makefiles are really simple yet effective. The dynamic library size is also reduced considerably, also as a consequence of the library being implemented from scratch in a simple fashion.
- Debugging: The new orwl library, *torwl*, has full support to run with valgrind, gdb and throws clear errors as soon as possible on unexpected behavior (like trying to acquire two times the same handle). None of these is true for the current ORWL library. The error checking mechanism that has built-in (and can be enabled/disabled at compile time) is a consequence of creating a library with debug in mind.
- Portability: *torwl* compiles with gcc and with most compilers that support gcc extensions. It worked with no issues using clang and icc on different platforms (x86, x86-64, Intel Xeon Phi). The current ORWL library tries to be extremely portable according to POSIX standards and ends up causing some portability issues. Although with some patches it was possible to fix the compilation issues found.
- Running a program: *torwl* creates a binary that can be executed. This is not the case for the current ORWL library, as it needs a ruby script to initialize. When trying to run on an Intel Xeon Phi, this script was ported to bash as ruby was not possible to use in the platform. Removing the

need for the initialization script in the current ORWL was evaluated but not easy to perform.

- No locks added on runtime: the current ORWL library inserts a lock on the FIFO queue every time an `orwl_handle2` handle is processed, and currently this procedure is fragile when communicating with remote nodes. As *torwl* doesn't do that (it uses a circular-buffer FIFO queue and no inserts during the computation), it could be avoided if the current ORWL library adopted *torwl* ideas.

- API changes: *torwl* implements most of the ORWL API but not all. It does not implement some syntax sugar the current ORWL library has, like `ORWL_SECTION` macros, or some added responsibilities, like manage thread creation for the library user, among others. There are several reasons for this, from reducing responsibilities and scope of the library to keep macros simple. Most of the reasons, though, have trade-offs and none is obvious, therefore the decision in most cases is the author personal choice.

Chapter 5

Results and experimentation

Previous section introduced techniques, using a custom instrumentation, to measure performance of the ORWL library and proposed to use different data-structures during an ORWL computation. This section evaluates the performance of such change.

5.1 About the examples used

The proposed changes try to optimize the hot-path of an ORWL computation. To observe this, several synthetic benchmarks that stress this aspect of the library are used.

Unfortunately, the dense matrix multiplication algorithm was not possible to benchmark, because it is written using advanced ORWL functions not (yet) implemented in *torwl*. Nevertheless, a real world example (named test-matrix) is used. This example executes computations more intense on the CPU and serves as a guide to observe behavior in real world examples.

5.2 Benchmarks

The tests were run on an AMD Opteron Processor 6172 with 24 cores, running at 2.1 GHz under Linux 4.15 x86-64 and with 32 GB of RAM memory.

Due to time constraints to perform this research, it was not possible to run these experiments on an Intel Xeon Phi with 240 CPU threads. Some preparatory test, though, show promising results on synthetic benchmarks.

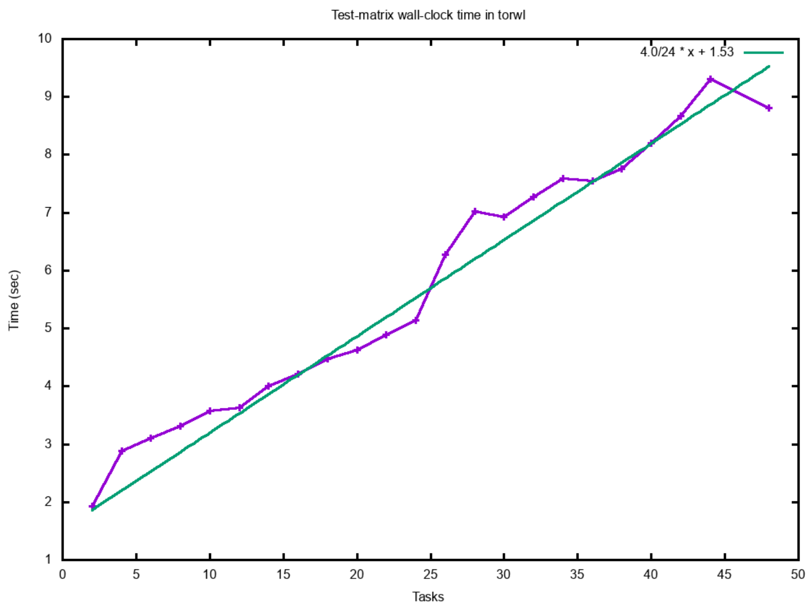


Figure 5.1: Test-matrix wall-clock time while increasing the tasks

The wall-clock time with torwl increase linearly with the number of cores. Hence, it suggest that adding more cores doesn't cause any scaling problem. In addition, preliminary results run on a Xeon Phi with 240 CPU threads show that the wall-clock time, for several synthetic benchmarks stressing the library, presents encouraging speedups (50x). These preliminary analyses also suggest that torwl scales as expected with hundreds of cores.

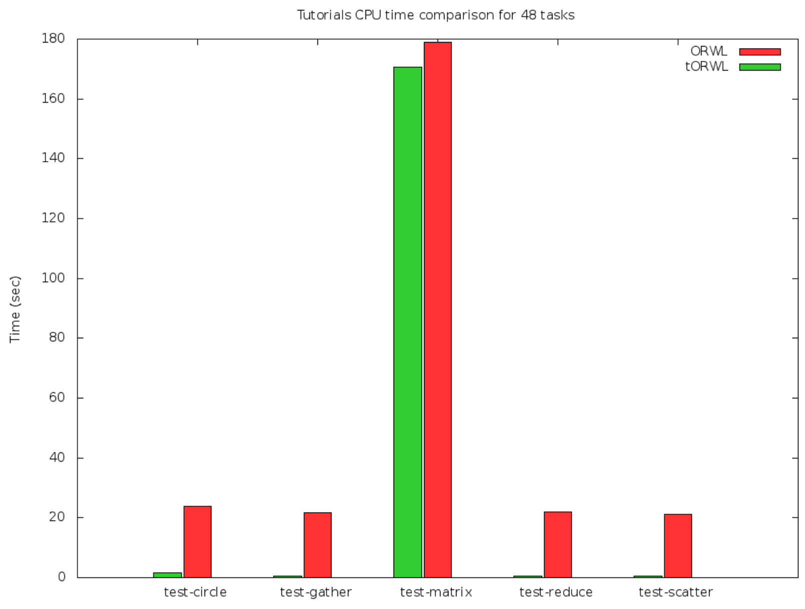


Figure 5.2: CPU time comparison

This figure compares the CPU time used between ORWL and torwl. The CPU time in the overall computation is always less when running with torwl. This means that the CPU time in the library is reduced considerably with torwl.

The percentage of CPU time inside the library shows similar results:

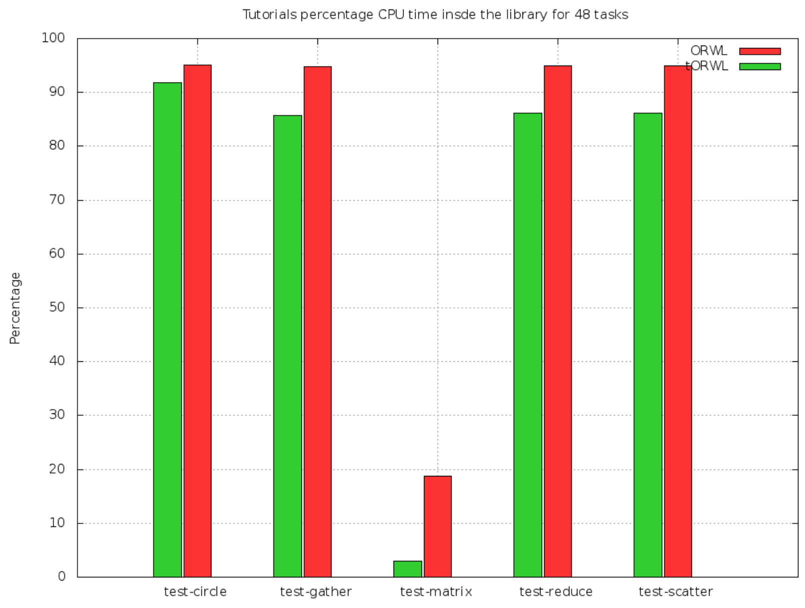


Figure 5.3: Percentage of CPU time inside the library

The percentage of time spent inside the library is reduced in all the cases. It is important to note that for the synthetic benchmarks the percentage inside the library is high because they are designed to stress the library. For the test-matrix benchmark, as it is a real world example, the percentage spent inside the library is less. Nevertheless, the percentage is significantly reduced in that case too.

In fact, the speed-up for the absolute CPU time spent inside the library is:

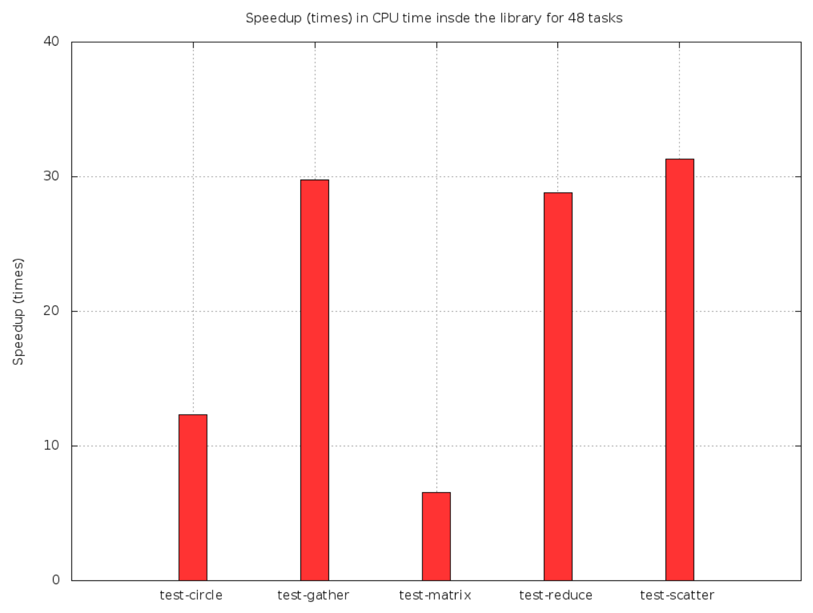


Figure 5.4: Number of times the absolute CPU time is reduced when using torwl

This figure compares the absolute CPU time spent inside the library, for both libraries, and shows how many times it is reduced when using torwl. In other words, it divides de CPU time spent in the current ORWL library by the CPU time spent in torwl.

The CPU time spent inside the library is reduced by a factor of 7 for the text-matrix benchmark and up to a factor of 31. Furthermore, all benchmarks tested are reduced by a factor between 7 and 31.

Therefore, the proposed changes reach their goal of reducing the CPU time overhead the library adds.

The ideal cpu time is ignored for synthetic benchmarks, it is expected to be low as they are meant to stress the library. For the test-matrix example the percentage of ideal CPU time is 73% with ORWL and 75% with torwl. In other words, it is slightly improved for the test-matrix benchmark.

The wall-clock time for these benchmarks is also relevant:

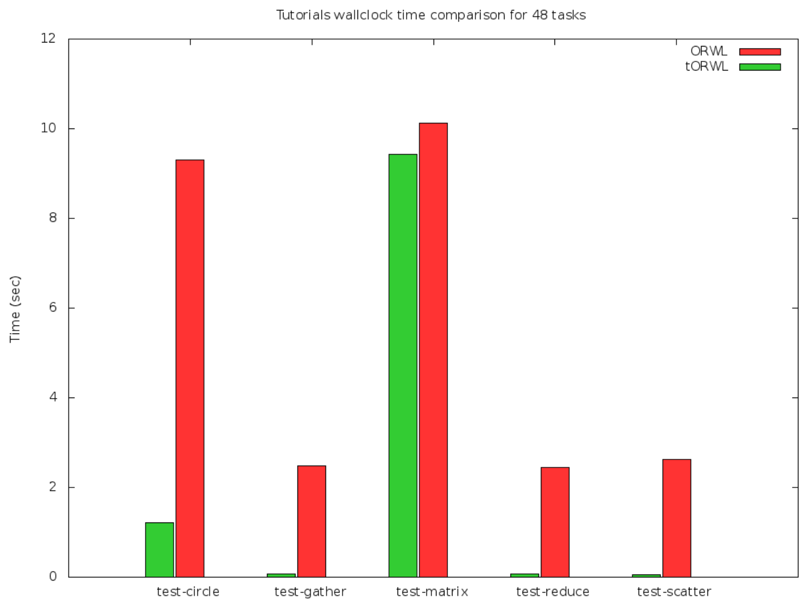


Figure 5.5: Wall-clock time comparison

This shows that the proposed changes always reduce the wall-clock time and even a $\sim 10\%$ reduction is achieved for the test-matrix benchmark.

Furthermore, the wall-clock time is reduced in a very significant way for the synthetic benchmarks. This implies that the overhead of using ORWL for small computations is considerably reduced and expands the scope of ORWL to be used in such computations. It might even help to alleviate the problem reported here for ORWL on smaller computations reported here[12].

Chapter 6

Conclusions

The Ordered Read Write Locks (ORWL) model is a new paradigm for parallel programming and this work studies it and its reference implementation in detail.

A custom instrumentation was created to analyze the runtime behavior of the implementation, that successfully helped to create insights about the components that had room for improvements. As a consequence of the runtime analyses and the study of ORWL, some invariants were noted that motivated the creation of a new implementation for the ORWL model with different algorithms and data structures.

This new implementation is shown to improve several times the performance on multi-core architectures, and expands the scope of ORWL to be used in small computations as well.

Bibliography

[1] Austin Group, “Threads,” in The Open Group Base Specifications, A. Josey et al., Eds. The Open Group, 2013, vol. Issue 7, p. chapter 2.9. [Online]. Available: http://pubs.opengroup.org/onlinepubs/009695399/

[2] M. Aswad, P. W. Trinder, and H.-W. Loidl, “Architecture aware parallel programming in Glasgow Parallel Haskell (GPH),” in ICCS, ser. Procedia Computer Science, H. H. Ali, Y. Shi, D. Khazanchi, M. Lees, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, Eds. Elsevier, pp. 1807-1816

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia, PA, 1994.

[4] Bill Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: The next generation in parallelizing compilers. In *Workshop on Languages and Compilers for Parallel Computing*, pages 10–1. Springer-Verlag, Berlin/Heidelberg, 1994.

[5] Juan Manuel Martinez Caamaño, Aravind Sukumaran-Rajam, Artiom Baloian, Manuel Selva, Philippe Clauss. APOLLO: Automatic speculative POLyhedral Loop Optimizer. IMPACT 2017 - 7th International Workshop on Polyhedral Compilation Techniques, Jan 2017, Stockholm, Sweden. pp.8, 2017. [Online]. Available: https://hal.inria.fr/hal-01533692

[6] B. Chamberlain, D. Callahan, and H. Zima, “Parallel programmability and the Chapel language,” Int. J. High Perform. Comput. Appl., vol. 21, no. 3, pp. 291-312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[7] P.-N. Clauss and J. Gustedt, “Iterative Computations with Ordered Read-Write Locks,” *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 496-504, 2010. [Online]. Available: http://hal.inria.fr/inria-00330024/en

[8] Laurent Daverio, Corinne Ancourt, Fabien Coelho, Stéphanie Even, Serge Guelton, François Irigoin, Pierre Jouvelot, Ronan Keryell, and Frédérique Silber-Chaussumier. PIPS – An Interprocedural, Extensible, Source-to- Source Compiler Infrastructure for Code Transformations and Instrumen- tations. Tutorial at PPoPP, Bengalore, India, January 2010; Tutorial at CGO, Chamonix, France, April 2011.

http://pips4u.org/doc/tutorial/tutorial-no-animations.pdf presented by François Irigoin, Serge Guelton, Ronan Keryell and Frédérique Silber-Chaussumier.

[9] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the memory hierarchy," in Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188543

[10] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 1 : one dimensional time. International Journal of Parallel Programming, 21(5):313–348, 1992.

[11] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part 2 : multidimensional time. International Journal of Parallel Programming, 21(6), 1992.

[12] Jens Gustedt, Stéphane Vialle, Patrick Mercier. Resource Centered Computing delivering high parallel performance. Heterogeneity in Computing Workshop (HCW 2014), May 2014, Phenix, AZ, United States. IEEE, 2014, Heterogeneity in Computing Workshop (HCW 2014), workshop of 28th IEEE International Parallel & Distributed Processing Symposium. [Online]. Available: https://hal.inria.fr/hal-00921128

[13] Alexandra Jimborean. Adapting the polytope model for dynamic and speculative parallelization. Phd thesis, Université de Strasbourg, September 2012.

[14] Uday Kumar Reddy Bondhugula. *Effective automatic parallelization and locality optimization using the polyhedral model.* PhD thesis, Ohio State University, Columbus, OH, USA, 2008. AAI3325799.

[15] F. Mueller and Y. Zhang, "HiDP: A hierarchical data parallel language," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, ser. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1-11. [Online]. Available: http://dx.doi.org/10.1109/CGO.2013.6494994

[16] Alexander Schrijver. Theory of linear and integer programming. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[17] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson, Steve Tjiang, Shih Liao, Chau Tseng, Mary Hall, Monica Lam, and John Hennessy. The suif compiler system: a parallelizing and optimizing research compiler. Technical report, Stanford University, Stanford, CA, USA, 1994.

[18] "OpenMP multi-threaded programming API." [Online]. Available: http://www.openmp.org

[19] "Threading building blocks," Intel Corp. [Online]. Available: http://software.intel.com/ en-us/intel-tbb

[20] NVIDIA CUDA C Programming Guide 9.2, NVIDIA, August 2018. [Online]. Available: https://docs.nvidia.com/cuda/archive/9.2/pdf/CUDA_C_Programming_Guide.pdf

[21] Khronos OpenCL Working Group, *OpenCL Specification*, Khronos Group, 2018, version 2.2. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf

[22] "The OpenACC^{TM} Application Programming Interface," OpenACC-Standard.org, Tech. Rep., Nov. 2018, version 2.7. [Online]. Available: https://www.openacc.org/sites/default/files/inline-files/OpenACC.2.7.pdf

[23] "Message passing interface." [Online]. Available: http://www.mpi-forum.org/docs