Relaxed Synchronization with Ordered Read-Write Locks

Jens Gustedt and Emmanuel Jeanvoine

INRIA Nancy, Grand Est, France
LORIA, AlGorille
{jens.gustedt,emmanuel.jeanvoine}@inria.fr

Abstract. This paper promotes the first stand-alone implementation of our adaptive tool for synchronization *ordered read-write locks*, ORWL. It provides new synchronization methods for resource oriented parallel or distributed algorithms for which it allows an implicit deadlock-free and equitable control of a protected resource and provides means to couple lock objects and data tightly. A typical application that uses this framework will run a number of loosely coupled tasks that are exclusively regulated by the data flow. We conducted experiments to prove the validity, efficiency and scalability of our implementation.

Keywords: synchronization, iterative algorithms, read-write locks, experiments.

1 Introduction

Lock or token based mechanisms to protect shared resources have a long tradition in parallel distributed computing. They are closely integrated into nowadays operating systems (POSIX mutex, semaphores and read-write locks), run times (OpenMP), and higher level languages (Java). They act on shared objects (POSIX `rwlock`), file ranges, or distributed entities (Corba, Chord, read-write locks [8]).

In contrast to implicit methods such as atomic snapshots or transactional approaches, see [7,1,6], they require an *explicit* action by the programmer or algorithm designer to mark the parts of her/his code that is judged *critical*. This paper is based on the assumption that such a labeling of critical parts will be provided. On a longer time scale the tool presented here might be a good basis to do such annotations automatically, but such an automatic annotation is not the subject of this paper.

Many parallel or distributed computations follow data dependency patterns between their different computation tasks [2]. Usually the output of one task (producer) is taken as input of other tasks (consumers), but write access (of the producer) and read access (of the consumer) to that data cannot be done atomically. This can occur in a shared memory setting where the data is too

M. Alexander et al. (Eds.): Euro-Par 2011 Workshops, Part I, LNCS 7155, pp. 387–397, 2012.

large to be accessible in one atomic read or in a distributed setting where data is sent and received in slices.

Algorithmically, the commonly implemented tools that we mention above are unsatisfactory with respect to at least one of the following three properties:

Liveness: Guarantees for liveness can in general not be given easily. Usually it needs supplementary tools such as barriers that come with an important cost whence they inhibit dynamic optimizations by the run time. Sporadic deadlocks are common software bugs that are quite costly to debug.

Equity: In case of contention, tools such as POSIX' reader-writer lock or semaphores voluntarily leave the order of lock acquisition either to the system implementation (in the simplest cases) or to the scheduling policy. If the order of treatment by different subtasks is fixed by the algorithm designer and may even be cyclic, guaranteeing equity and a precise flow of control can be challenging.

Efficiency: Using a lock structure (a mutex in the simplest case) introduces fixed points in the program between which a resource needs to be accessible. It usually gives no explicit indication to the run-time which resource is targeted and also what could be done proactively to represent the resource in the address space of the program. Possibilities of overlapping computation and communication (in a broad sense) are easily lost by that, exploiting such possibilities can become tedious to implement. Again in the simplest case of a mutex, a resource is only fetched and pushed when it is accessed where usually the transfer from one task P_1 to another P_2 could be done as soon as P_1 unlocks the mutex.

To target the three criteria from above, in [3,4] we introduced the framework of *ordered read write locks*, ORWL, that are designed to favor algorithmic control and data consistency. This framework for inter-task synchronization is conceptually independent of the execution context and can be implemented in both shared memory or distributed environments.

A first adhoc implementation of this tool was integrated as part of the parXXL library and is only fully available for shared memory. This paper here presents a new implementation that is only based on standard languages and interfaces (C and POSIX) and that can be used in shared, distributed or mixed contexts.

The basics of our model and the designs of the underlying tool for iterative parallel algorithms are briefly reviewed in Section 2, in particular we remind the features that guarantee liveness and equity for iterative settings. Then, in Section 3, we present the three different features that distinguish the use of ORWL from other tools: the possibility to announce the future use of a resource, a comfortable interface for iterative computations and a tight binding between control structures and data. In Section 4, we present benchmarks that address the potential overhead that our implementation introduces. Finally, we conclude and discuss our next steps in Section 5.

2 Ordered Read-Write Locks, an Adaptive Tool for Synchronization

We call the building block of our model *Ordered Read-Write Locks* (ORWL), a special kind of read-write locks. All proofs on properties of the model have been presented in [4]. ORWL have the following features:

1. A waiting queue with FIFO-policy.
2. An explicit association of a lock with application data.
3. A distinction between *request* and *acquire* operations that replace a classical one-step *lock* operation. So the typical sequence for an access is *request*, *acquire* and then *release*.
4. A distinction between locks (as opaque objects) and lock-handles (as user interfaces acting on locks).
5. A distinction into exclusive or write locks and inclusive or read locks.

All of these features have been used previously for lock data structures, see *e.g* [5]. But to the best of our knowledge their intentional combination in a single framework is original.

Property 1 and 2 together ensure a controlled access order of the application to its data. For an important class of applications that will iterate over their data, we must be able to control when and what data is accessed. In addition, Property 2 restricts the access to the data to the time that a lock is held, pointers become invalid outside that time window. We thus enforce data consistency: no thread may write to data that it has not locked and if a read is granted to data it is guaranteed to be invariant while the lock is held.

Property 3 allows us to reserve resources pro-actively. At first, this gives the application programmer the possibility to issue some sort of hint (a request operation) that a resource will be used in the future with a require operation. Such a hint is non-blocking and incurs only negligible cost by itself. This is a big advantage for the *programming logic* of iterative algorithms which access data in a cyclical pattern. They may insert their request for the next iteration in the FIFO while holding a lock for the current one. The other advantage lies in *performance* issues. The run time system then may use this information to anticipate the access, e.g by doing a data prefetch.

When doing such a pro-active locking the Property 4 comes into play: a thread or process may define several handles (usually two in our case) on the same lock and thereby newly request a lock by means of one handle while still actively holding a lock via another handle. The type of request in view of Property 5, namely if the access will be just for reading (and thus potentially shared) or for writing must be specified when the lock is requested and that type is kept track via the handle.

Property 5 ensures that we may easily handle the case that the output of a task is read by several others. It allows for important optimizations: buffer space with read-only data can be shared among threads and processes; data that is only presented for reading may be thrown away once the lock is released and thus costly updates (or just checks for consistency) may be avoided.

Recurring Tasks. To model a recurring task of an iterative process we proceed as follows. Whenever all the lock requests that such a task has requested have been acquired, the task is said to be *active* and can perform its job.

After finishing the computation of the job itself, before releasing any of the locks, a second set of lock handles is used to posts copies of its requests for the next iteration, first. These guarantee the reservation of the resources for the next iteration. The task then releases the acquired locks to pass the control over to other tasks that operate on the same data. This procedure guarantees that access to the resources is given in a cyclic pattern and thus that all tasks iteratively get access to the data in an equitable way, see [4]. Ensuring liveness of such system needs an additional effort. It has been shown that for this property the initialization order of the lock handles in the FIFO is crucial. Any initialization that doesn't contain certain types of cyclic dependencies never will run into a deadlock and that such an initialization is always possible.

3 User Interfaces

This section will introduce a handful of data types and functions that compose the user interface of ORWL. There are three data types:

`orwl_mirror` a representation of a local or remote resource.
`orwl_handle` a lock handle to queue up for that resource, and
`orwl_handle2` a pair of lock handles used for recurrent locking requests.

The function interfaces can be classified in three different sets:

- `orwl_read_request` (or `orwl_write_request`), `orwl_acquire` (or `orwl_test`) and `orwl_release` that form a lock sequence on the resource.
- `orwl_truncate`, `orwl_write_map` and `orwl_read_map` that allow to control and access the data that is eventually associated to a resource.
- A set of analog functions with a "2" appended to the name that operate on pairs `orwl_handle2`, such as `orwl_read_request2` or `orwl_write_map2`. They suit particularly the needs for iterative tasks.

3.1 Resource Protection

Simple resource protection that is analogous to a protection of a critical section through a mutex can be implemented in a straight forward manner.

Listing 1.1. Simple exclusive protection of a resource `loc` through a handle `handle`

```
orwl_write_request(&loc, &handle);          /* announce future access */
/* some operation without the resource                                 */
orwl_acquire(&handle);                      /* Block until granted    */
/* some critical operation with locked resource                        */
orwl_release(&handle);                      /* Free the resource      */
```

Here the first call to `orwl_write_request` binds `handle` to resource `loc` and announces the intent to access it. Until the time we call `orwl_acquire` the system may

- satisfy other demands to the resource that have higher priority,
- route the corresponding data to our host
- allocate space in our address space or
- perform other operations that are needed to satisfy the demand.

During that time the application can perform any type of operation that doesn't need access to the resource.

Then, once the application comes to a point it can't proceed further without the resource, `orwl_acquire` ensures that it is blocked until the request can be fulfilled. The critical section then ends by calling `orwl_release`.

If the application has a variety of tasks to perform before access to the resource can't be avoided further, `orwl_test` can be used to query for the lock acquisition and allows to adapt the application at run time, see Listing 1.2.

Listing 1.2. Adapted protection of a resource `loc` through a handle `handle`

```
orwl_read_request(&loc, &handle);                    /*  announce future read */
while (!orwl_test(&handle)) {                        /*     check if produced */
  /* Do some operation while the resource is produced                          */
}
orwl_acquire(&handle);                               /*   block until granted */
/* Do some operation while the resource is stable                             */
orwl_release(&handle);                               /*     free the resource */
```

3.2 Associating Data

Up to now we have introduced a use of ORWL that only uses its controlling aspect, analogous to POSIX' `pthread_mutex_t` or `pthread_rwlock_t`. In addition to that ORWL allows one to associate data to the resource directly, Listing 1.3.

Listing 1.3. Associate data to `loc` and initialize it properly

```
orwl_write_request(&loc, &handle);    /* reserve the resource for maintenance */
orwl_acquire(&handle);
orwl_resize(&handle, 168 * sizeof(uint64_t));    /* write access is needed to */
size_t size;                                     /* get a pointer to the data */
uint64_t* data = orwl_write_map(&handle, &size); /*      in our address space */
assert(size == 168 * sizeof(uint64_t));          /*             check the size */
my_special_initialization(data);                 /*      initialized the data */
orwl_release(&handle);                           /*         free the resource */
data = 0;                                        /*    invalidate the pointer */
```

To associate data to a resource we just have to assign a non-zero size to the data (here 168 units for the type `uint64_t`). Per default the data is initialized to all zero values, in the example it is initialized by a special function. Data is viewed as untyped bytes (`void*`) and the size returned by `orwl_write_map` accounts the number of bytes in the data. To ease the underlying communication routines, data sizes are always multiples of `sizeof(uint64_t)`, usually 8 bytes.

Another task or process may then modify the data without changing its size, Listing 1.4, and any number of readers may inspect the results of that modification simultaneously, see Listing 1.5.

Listing 1.4. Modify the data

```
orwl_write_request(&loc, &handle);    /* reserve the resource for modification */
orwl_acquire(&handle);
size_t size;                                                  /* get a pointer to the data */
uint64_t* data = orwl_write_map(&handle, &size); /*       in our address space */
size /= sizeof(*data);
for (size_t i = 0; i < size; ++i) {                           /*     do some operation with */
  data[i] *= i;                                               /*           exclusive access */
}
orwl_release(&handle);                                        /*          free the resource */
data = 0;                                                     /*     invalidate the pointer */
```

Listing 1.5. Access the associated data

```
orwl_read_request(&loc, &handle);          /* reserve the resource for reading */
orwl_acquire(&handle);
size_t size;                                                  /* get a pointer to the data */
uint64_t const* data = orwl_read_map(&handle, &size); /* in our address space */
size /= sizeof(*data);
for (size_t i = 0; i < size; ++i) {                           /*       some operation while */
  printf(stderr, "item %zu is %" PRIu64 "\n",                 /*                 the resource */
         i, data[i]);                                         /*                    is stable */
}
orwl_release(&handle);                                        /*          free the resource */
data = 0;                                                     /*     invalidate the pointer */
```

3.3 Recurrent Access to Resources

Iterative computations need to be implemented with a lot of care if we want to guarantee liveness for all processes and equity among them. As introduced above ORWL, as an abstract tool can guarantee these properties if we issue a lock request on a resource for the next iteration before we abandon a current lock that we hold. This is facilitated by our library with the type `orwl_handle2`. It represents a pair of `orwl_handle` that are used in alternation for even and odd numbered iterations.

Listing 1.6. A simple iterative procedure with one resource and one `orwl_handle2`

```
orwl_write_request2(&loc, &handle2);          /*                    bind the pair */
while (!done) {                               /*  do until some external event */
  orwl_acquire2(&handle2);                    /*     block until our turn comes */
  /* work exclusively with the resource */
                                              /* insert ourselves in the queue */
  orwl_release2(&handle2);                    /*          for the next iteration */
                                              /*                free the resource */
                                              /* pass control to somebody else */
}
orwl_cancel2(&handle2);                       /*        withdraw from the queue */
```

Here, before entering into the iteration, we bind the first handle of the pair to the resource. Then, at the start of each iteration the handle in the pair with the request pending is acquired and we gain exclusive access to the resource. At the end of the iteration `orwl_release2` first issues a new request on the handle that is currently inactive and then releases the lock on the resource. When going out of the iteration, `orwl_cancel2` has to be called since otherwise one of the handles in the pair would be left with a pending lock request.

Listing 1.7. An iterative procedure with two resources and two `orwl_handle2`

```
orwl_write_request2(&hereResource, &here);      /*            bind the two pairs */
orwl_read_request2(&thereResource, &there);
while (!done) {                                 /*     until some external event */
  orwl_acquire2(&here); orwl_acquire2(&there); /*                  block twice */
  size_t size;                                  /*            the data access is */
  uint64_t* hereData = orwl_write_map2(&here, &size);        /* exclusive here */
  uint64_t const* thereData = orwl_read_map2(&there, &size);/* inclusive there */

  size /= sizeof(*thereData);                   /*                application code: */
  for (size_t i=0; i<size; ++i) {               /*          average componentwise */
    hereData[i]=(hereData[i]+thereData[i])/2;/*          store in hereResource */
  }
                                                /* insert ourselves in the queues */
  orwl_release2(&here);                         /*          for the next iteration */
  orwl_release2(&there);                        /*              free the resources */
                                                /*      pass control to the others */
}
orwl_cancel2(&here); orwl_cancel2(&there);      /*        withdraw from the queues */
```

Finally, with Listing 1.7, let us look into a more complicated pattern, namely with two resources and two pairs of handles that act on them. This can be seen as each process "owning" one resource (called `hereResource`) that he will update and "inspecting" one resource of a "neighboring" process (`thereResource`). The access pattern between different processes could then be any collection of directed circles or trees. The basic scheme is similar, only that always two calls to `orwl_acquire2` and `orwl_release2` are issued in the iteration, one for each resource. Using the mapping feature of ORWL we see how we easily can implement an iterative vector averaging on the associated data.

4 Experiments

4.1 Benchmark Application

Livermoore Loops Kernel 23. To benchmark the library, we use the Livermoore Loops Kernel 23 (LLK23) benchmark (see `http://www.netlib.org/benchmark/livermorec`).

Listing 1.8. Core computation of the Livermoore Loops Kernel 23 benchmark

```
for (i = 2; i < (N - 1); ++i) {
  for (j = 2; j < (M - 1); ++j) {
    q = data[i-1][j] * zb[i][j] + data[i][j-1] * zv[i][j]
      + data[i][j+1] * zu[i][j] + data[i+1][j] * zr[i][j]
      + zz[i][j] - data[i][j];
    data[i][j] += 0.175 * q;
  }
}
```

The core computation of the benchmark is given in Listing 1.8. To simplify, each element of a matrix called `data` is computed using four neighbors elements (N, S, E and W) and five coefficient matrices (`zb`, `zr`, `zu`, `zv`, `zz`).

This application has significant characteristics to test our approach. It is an iterative computation of which the different parts can be executed asynchronously and where each part shows constant progress. Furthermore, for a parallelization, some data exchange is required between the frontiers. This can be done transparently with ORWL; the nested for loops of Listing 1.8 would go in the place marked "**application code**" in Listing 1.7.

Parallelization with ORWL. An intuitive method to parallelize the problem is to decompose `data` into several blocks. For each block, the inner computation is independent from the other blocks whereas the computation of the edges and the corners depends on some neighboring blocks.

Thus, for each block, we define a main task (MT) that performs the computation and eight sub-tasks (ST) that are used to export the frontier data (edges and corners) to the neighboring MT. Figure 1 shows an example of a simple decomposition into four blocks. The four MT are numbered from 0 to 3 and the associated ST are prefixed with their direction.

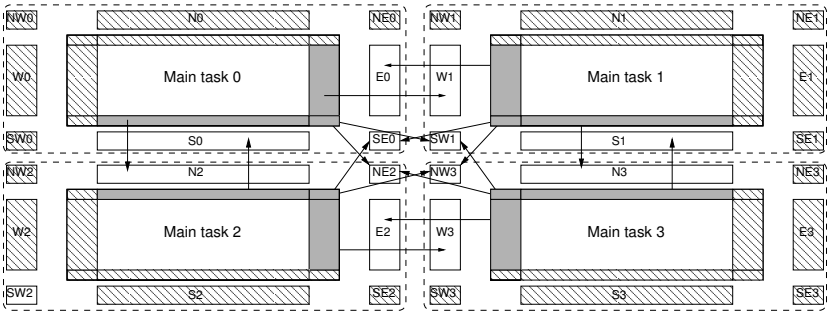


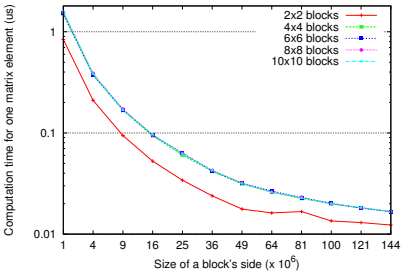
Fig. 1. A four block decomposition example. The gray parts represent the MT parts that require some frontier data to be computed. Hatched parts are constant boundary conditions or unused.

4.2 Experimental Results

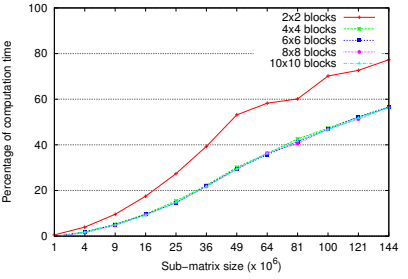
The experiments have been conducted on the *graphene* cluster of the Grid'5000 experimental testbed. Each node is composed of 4 cores at 2.53 GHz and 16 GiB of memory, and a Gigabit Ethernet interconnection network. All the following results are obtained after running 100 iterations of the LLK23 computation.

Average Execution Time per Matrix Element. In this experiment, the goal is to evaluate the average execution time per `data` element. First, we divide `data` in 4, 16, 36, 64 and 100 blocks. Then, for each division, we increase the global problem size by varying the size (in number of elements) of a block. One node is reserved for each block, such that we can reach the limit of a maximum block size that fits into RAM.

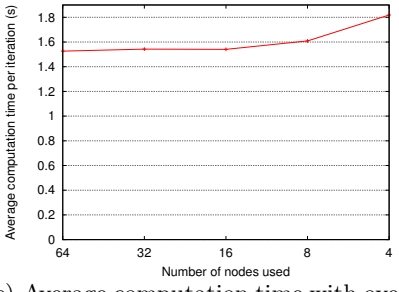
In Figure 2(a) we see that the computation cost per element decreases when the problem size increases, and that it tends to a lower limit. We note that the times corresponding to the 4 block division are below the others. This is due to the simplified connection pattern (see Figure 1) where half of the block boundaries don't participate in communication. The other running times are not distinguishable, which proves that for subdivisions into more parts this effect is already negligible.



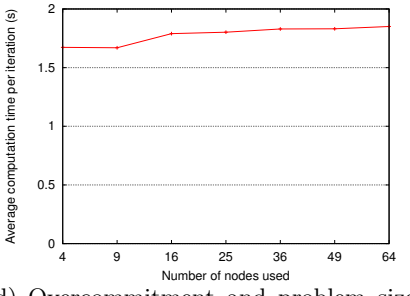
(a) Average computation time for an element



(b) Percentage of wait time



(c) Average computation time with overcommitment



(d) Overcommitment and problem size up

Fig. 2. Experimental results

Computation Efficiency. Because our parallelization of LLK23 requires frontiers of neighboring regions, not all parallel tasks can compute simultaneously. In this experiment, see Figure 2(b), the goal is to relate the time that MT spends either to compute or to wait for some frontier data. The setup is the same as in the previous experiment. We see that for small problems, almost all the time is spent to wait. For larger problems, the computation time increases and finally reaches about 55 % of the total time. Actually, the *absolute* wait time itself (not shown) is basically spent in `orwl_acquire2` and does not vary much, but the computation partially overlaps with these waiting periods.

Overcommitting. In this experiment the global problem size is constant, 4000×4000 elements per blocks with a division into 64 blocks. We launch the computation successively on 64, 32, 16, 8 and 4 nodes; thus commit 1, 2, 4, 8 and 16 MT per quad-core node. The average time to compute an iteration is shown in Figure 2(c). We can see that the overcommitment of several tasks per cores allows to take full advantage of the four cores of each node. The time per iteration only increases slightly when we have much more MT per core.

So, even if the parallelization induces long waiting times for the tasks, ORWL is able to hide this cost: tasks that have received their data autonomously start their execution while other tasks are waiting for theirs.

Scalability. In this last experiment we study the impact of increasing the global problem size. We place 16 MT per quad-core node on 4 to 64 nodes. Each MT computes a 3000x3000 element block. The average time to compute an iteration is shown in Figure 2(d). We see that increasing the problem size does not increase the average computation time much. Thus, ORWL is suitable for the construction of scalable applications.

5 Conclusion and Future Work

In this paper we introduced a new library that implements the ordered read-write lock (ORWL) paradigm to control access to shared or distributed resources. We presented the basic use patterns for that library that ranges from simple implementations of critical sections that allow a pro-active announcement to more involved patterns of alternating resource allocation in iterative computations. Macro-benchmarks show that the library behaves well on multi-core machines and clusters; it realizes almost perfect computation/communication overlap and weak scaling properties.

A forthcoming article will describe the implementation of the library in more detail and present micro-benchmarks of the individual functions and components. Future plans with ORWL include application to other types of applications and architectures. Namely we are currently implementing an application that uses ORWL to control computations on compute cluster equipped with GPU co-processors. Other future work includes improvements to use ORWL as simple and efficient locking features in a shared memory context.

Acknowledgment. Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

1. Afek, Y., Attiya, H., Dolev, D., Gafni, E., Merritt, M., Shavit, N.: Atomic snapshots of shared memory. J. ACM 40(4), 873–890 (1993)
2. Barrett, R., Berry, M., Chan, T.F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., Van der Vorst, H.: Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods. SIAM, Philadelphia (1994)
3. Clauss, P.N., Gustedt, J.: Experimenting Iterative Computations with Ordered Read-Write Locks. In: Danelutto, M., Gross, T., Bourgeois, J. (eds.) 18th Euromicro International Conference on Parallel, Distributed and Network-based Processing, pp. 155–162. IEEE, Pisa (2010), `http://hal.inria.fr/inria-00436417/en`
4. Clauss, P.N., Gustedt, J.: Iterative Computations with Ordered Read-Write Locks. Journal of Parallel and Distributed Computing 70(5), 496–504 (2010), `http://hal.inria.fr/inria-00330024/en`

5. Danek, R., Golab, W.M.: Closing the Complexity Gap between FCFS Mutual Exclusion and Mutual Exclusion. In: Taubenfeld, G. (ed.) DISC 2008. LNCS, vol. 5218, pp. 93–108. Springer, Heidelberg (2008)
6. Herlihy, M., Eliot, J., Moss, B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 289–300 (1993)
7. Vitányi, P.M.B., Awerbuch, B.: Atomic shared register access by asynchronous hardware (detailed abstract). In: FOCS, pp. 233–243. IEEE (1986)
8. Wagner, C., Müller, F.: Token-Based Read/Write-Locks for Distributed Mutual Exclusion. In: Bode, A., Ludwig, T., Karl, W.C., Wismüller, R. (eds.) Euro-Par 2000. LNCS, vol. 1900, pp. 1185–1195. Springer, Heidelberg (2000)