Iterative Computations with Ordered Read-Write Locks

Pierre-Nicolas Clauss, Jens Gustedt

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Iterative Computations with Ordered Read-Write Locks

Pierre-Nicolas Clauss
Nancy University

Jens Gustedt
INRIA Nancy – Grand Est

N° 6685

Octobre 2008

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--6685--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
NANCY - GRAND EST

Iterative Computations with Ordered Read-Write Locks

Pierre-Nicolas Clauss
Nancy University

Jens Gustedt
INRIA Nancy – Grand Est

Thème NUM — Systèmes numériques
Équipe-Projet AlGorille

Rapport de recherche n° 6685 — Octobre 2008 — 20 pages

Abstract: We introduce the framework of *ordered read-write locks*, ORWL, that are characterized by two main features: a strict FIFO policy for access and the attribution of access to *lock-handles* instead of processes or threads. These two properties allow applications to have a controlled pro-active access to resources and thereby to achieve a high degree of asynchronicity between different tasks of the same application. For the case of iterative computations with many parallel tasks which access their resources in a cyclic pattern we provide a generic technique to implement them by means of ORWL. We show that the possible execution patterns for such a system correspond to a combinatorial lattice structure and that this lattice is finite iff the configuration contains a potential deadlock. In addition, we provide efficient algorithms: one that allows for a deadlock-free initialization of such a system and another one for the detection of deadlocks in an already initialized system.

Key-words: synchronization, iterative algorithms, read-write locks

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Calculs itératifs avec des verrous ordonnés de lecteur/écriture

Résumé : Nous présentons la structure des *verrous ordonnés de lecture/écriture*, ORWL, qui sont caractérisés par deux fonctionnalités principales : une politique d'accès FIFO stricte et une attribution des accès à des *gestionnaires de verrou* plutôt qu'à des processus ou des threads. Ces deux propriétés permettent aux applications d'avoir un contrôle dynamique de l'accès aux ressources et donc de parvenir à un haut niveau d'asynchronisme entre les différentes tâches d'une même application. Dans le cas de calculs itératifs avec beaucoup de tâches parallèles accédant à leurs ressources selon un motif cyclique, nous proposons une technique générique d'implémentation avec des ORWL. Nous montrons que les schémas d'exécution possibles pour un tel système forment un treillis combinatoire et que ce treillis est fini si et seulement si la configuration contient un interblocage potentiel. De plus, nous proposons deux algorithmes efficaces : l'un pour une initialisation sans interblocage d'un tel système et l'autre pour la détection d'interblocage dans un système déjà initialisé.

Mots-clés : synchronisation, algorithmes itératifs, verrous lecture/écriture

1 Introduction and Overview

Many iterative computations are be found to observe data dependency patterns between their different computation tasks. In this paper we handle the common case that the output of a task may be input to one or many other tasks *and* that the read and write access to that data cannot be done atomically. Such dependencies hold *e.g* for block oriented matrix computations, see [2], where the data chunks that are needed for reading are far to big to allow for an atomic wait-free operation on system level.

A well-known technique to handle such situations uses *read-write locks* (RWL). RWL are designed to allow efficient concurrent access of several readers without involving more than one atomic operation to check readability, see *e.g* [13, 10] for parallel environments and [14] for an algorithm for distributed environments. Before a task reads data that was the output of some other tasks it acquires a *read-lock* (or shared lock) for that data, meaning that the data can't be modified in the mean time. But eventually other tasks may read the same data without creating a conflict. Before writing, the corresponding task acquires a *write-lock* (or exclusive lock) that hinders any other task to read or write the data until the lock is released.

Implementations of such RWL data structures usually do not prescribe which task is granted a lock in case that several of them simultaneously request access. This is *e.g* the case for both RWL that are specified for POSIX [9], namely `pthread_rwlock_t` and range locks on files (with `fcntl`). Here, the norm explicitly states that no guarantee is given on the order in which locks are granted and that applications should not rely on the ordering in which locks are required. In particular, it explicitly allows to grant access to writers (to avoid writer starvation) even if the write-lock request had been clearly issued after the read-lock request.

While such an underspecification of the lock order may be convenient on the system side it clearly is very annoying for an application designer: because of possible lock inversion, cyclic data dependencies easily lead to deadlocks and which to avoid becomes very tedious. This is probably the primal reason why RWL although present in all modern OS do not have a wide spread use.

This underspecification also has the disadvantage that it becomes difficult to guarantee an equitable progression of all tasks. In case of a contention on a lock, two writers could *e.g* alternate on holding a particular lock while other tasks (readers) would be waiting endlessly to obtain their locks.

Another disadvantage of the commonly used RWL models is the fact that a lock is granted to a process (*e.g* file locks) or a thread (`pthread_rwlock_t`) and not to a data entity. Such a dissociation would be convenient when a task wants to reserve a resource pro-actively for an access in a near future ("*give me a lock on resource X whence it is available*"), to come back to that lock request later, and then only block on it when this has become unavoidable.

In this paper we try to overcome these inconveniences by first imposing that the locks are granted in the FIFO ordering of the requests. Second, we introduce *handles* as entities that require, acquire and release a lock (Section 2). As an additional benefit, lock-handles give the possibility that a task may issue a new requests on a particular lock while already holding one through another handle ("*give me a lock on resource X after all current requests have been satisfied*").

With these modifications we are able to introduce a convenient strategy (Section 3) for iterative computations with cyclic data dependencies that can detect (respectively avoid) deadlocks while still guaranteeing a high degree of parallelism and asynchronicity. In particular, in Section 4 we prove that the possible states of such a system of dependencies form a combinatorial lattice that only depends on the initial configuration (ordering) of the locks. As a consequence, we see then that a configuration may deadlock iff this lattice is finite (things that may go wrong will go wrong) and also that the computation then obeys a homogeneous progress, the "*homogeneity*" only depending on the connectivity properties between the tasks.

Finally, Section 5, we will briefly introduce an algorithm that allows for an efficient instantiation of such lock systems as well as another one that detects if a system that is in a given state would lead to a deadlock or not.

2 Model

In the following, we suppose we are given a set of computational tasks $\mathcal{T}$ that is to be performed and such that there are data dependencies between these tasks. As part of an iterative process, tasks may be recurrent, *i.e* we may want to executed them many times.

Data dependencies are modeled via read and write operations that are not necessarily *atomic*. So a data dependency of $v \in \mathcal{T}$ from $w \in \mathcal{T}$ then implies that v may only execute while w doesn't. Otherwise v could eventually read inconsistent parts of w's output, one from before an update and one from after.

In order to model parallelism between tasks, we introduce an auxiliary graph, the *conflict graph* $C(\mathcal{T})$. We draw an edge between $v, w \in \mathcal{T}$ if v reads the output of w or w reads the output of v. We say that a subset of tasks $T' \subseteq \mathcal{T}$ is *independent* if there is no data dependency between any pair of tasks $v, w \in T'$. Independent sets of tasks may be executed in parallel. Clearly T' is independent iff it forms an *independent set* in graph $C(\mathcal{T})$. The size of the *maximum independent set* in $C(\mathcal{T})$ is thus the maximal amount of parallelism that may be reached for the set of tasks $\mathcal{T}$.

We now describe the tools used in our model along with their basic properties.

Ordered Read-Write Locks. Our model targets applications with several tasks which have temporal and possibly data dependencies. Its goal is to allow easy and predictable synchronization between tasks and therefore relies on *ordered read-write locks* (ORWL). This type of locks has the same semantic of common read-write locks (Concurrent Read, Exclusive Write) but enforces the use of a FIFO policy for its waiting queue. Furthermore, these locks are *resource-oriented*. Acquisition is thus granted to *lock-handles* (LH) rather than threads following a two-step pattern.

1. Post a read- or write-request to the lock through a handle. This inserts the request in the FIFO queue for further use (and the requester is thus free to continue execution).

2. Require the previous request. This waits until the request moves first in the FIFO queue, thus granting read- or write-access to the associated resource.

This pattern is designed for a particular class of applications which require iterative computation (see Section 3). Obviously, for a comfortable use of ORWL other primitives would be convenient, in particular a non-blocking test that allows to know if a request has been achieved. But since they are not needed in the sequel, we will not go into details of these.

Synchronization overlay. For a given interdependent system of tasks, our model associates a synchronization overlay as an abstraction of the data access pattern. Its role is to give control to tasks whenever it is appropriate (based on the dependencies between them). More specifically, the data space is partitioned maximally according to the different dependencies of the tasks: to each primitive part of the data (called a *location*) corresponds an ORWL and to each individual request for that part of the data corresponds one (or later several) LH. Acquiring control for a task is done by requiring a write access to a data block (which can be viewed as being proprietary data of the task), while dependencies are modeled by requiring read accesses to other data blocks (viewed as owned by other tasks). Once these accesses are granted, the task is allowed to perform whatever action it is designed for on its data (with possible read-only use of other tasks' data).

This model is of course best suited for applications requiring parallel computation over a data space. The partitioning method for the data itself is application-dependent. As an example, take a matrix oriented computation where locations would correspond to blocks of the matrix, tasks would be responsible for writing an individual block. These tasks would then compete for write access to their own block and for read-access to 'neighboring' blocks.

Definition 2.1 (Overlay) *Consider a synchronization overlay used atop an executable system of n locations.*

- *Each of these locations can stack any number of exclusive and inclusive lock-requests (abbreviated Xreq and Ireq, respectively), numbered bottom-up. This numbering is called the* priority rank *of the lock request in the* location*.*
- *The request in a location (if any) that has the lowest priority is accounted as being* acquired*.*
- *Ireqs may correspond to several (but at least one) LH for which a read-request was issued for the location. Xreqs always correspond to exactly one LH for which was issued a write request for the location. In the priority order, Ireqs may only be directly followed by an Xreq; no such restriction applies to Xreqs which may be directly followed by an Xreq or Ireq.*
- *Locations which have no Xreqs are called* unconstrained *and can safely be removed from the system. Others are called* constrained*.*
- *An active* task *is represented by an Xreq along with all the Ireqs that it requested.*

We visualize such a lock overlay system by a directed data dependency graph as shown in Figure 1. Here Xreqs are symbolized by a □ and Ireqs by a ◯. The requests

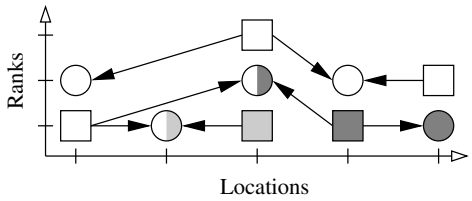


Figure 1: Sample overlay with an executable task (light grey) and a delayed task (dark grey).

are connected together following the data dependency of their defining tasks. Connections always connect an Xreq to an Ireq, thereby capturing the requirements for a given write request to hold read-locks on other locations.

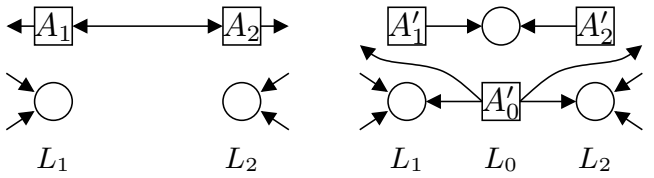
In the above definition we ask for a task to require exactly one Xreq. This is not a restriction in the modelling capacity since a job that has to place several Xreqs can be easily split into several tasks with one Xreq each such that the read-write semantics remain the same. Take Figure 2 as an example. Here the two coupled write requests A_1 and A_2 are replaced by one request A'_o that allows to pull the data from locations L_1 and L_2 and two requests A'_1 and A'_2 that allows to push the modified data back to L_1 and L_2.

We give now a few definitions to clearly identify the elements of our overlays and their attached semantic.

Definition 2.2 (Executability) *A task is* executable *if all its locks are acquired, otherwise it is* delayed*. For a given overlay S, the subset $R(S)$ of locations where the lowest lock is the Xreq of an executable task defines the set of* executable locations *of S. Locations that are not executable are* blocked*.*

Definition 2.3 (Deadlock) *An overlay is in a* deadlock *if all its locations are blocked.*

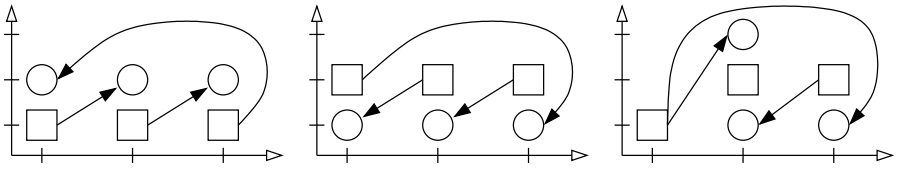
Definition 2.4 (Minimal Support) *A deadlock's* minimal support *in an overlay S is the smallest subset of S (both over the locations and over the priority ranks) so that the overlay restricted to that subset is in a deadlock situation.*



(a) Two coupled write requests

(b) Replacement by one pull task A'_0 and two push tasks A'_1, A'_2

Figure 2: Transforming coupled write requests



(a) Only Xreqs acquired (b) Only Ireqs acquired (c) Both types acquired

Figure 3: Deadlock situations

For an overlay to be in deadlock clearly implies that there must be some cycle among its dependencies (see also Lemma 2.1 below). Figure 3 shows different types of minimal supports of deadlocks.

In its general form, our model allows multiple tasks to operate on the same data. This is modeled by several Xreqs on the same location. From such a general overlay S, it is possible to derive an overlay where there is only one Xreq per location but still offers the same execution order of the tasks.

Definition 2.5 (Canonical Form) *An overlay is in* canonical form *if it has at most one Xreq per location.*

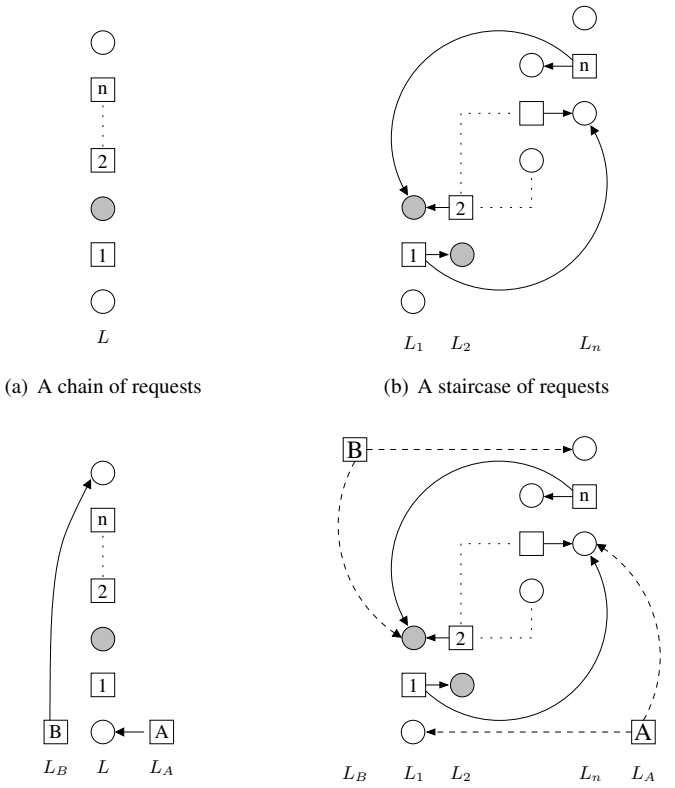
Figure 4 gives the rewrite pattern used to transform an overlay into canonical form. Here a location with n Xreqs is split into n different locations. The Xreqs are placed on their respective locations with just one new Ireq underneath and above. The Xreq on location L_i is then connected to the upper Ireq on L_{i-1} and to the lower one on L_{i+1}. Some slightly modified rules apply to the Xreqs 1 and n as indicated in the figure. This reordering from a chain on one location to a staircase on several locations does not change the relative ordering in which the Xreqs are acquired.

Other Xreqs that are connected to the Ireqs (e.g the grey one) are connected to the respective two copies. For Xreqs (as A or B), that connect to the top or the bottom of the chain, special connection rules apply.

It is important to note that these replacement rules ensure that the order of the acquisition of the requests remains stable, even if the analogous requests for completed task are re-issued after their termination: *e.g* B always gets acquired just after n is freed, and 1 only gets acquired just after B, whereas A and B are always executed in between n and 1 but their respective order is not fixed.

Let n be the size of the initial location and m be the number of external connection into that location. Then the number of added Ireqs is at most $2n$ and the number of added links is at most $2n + m$. Since each Xreq and each link appears at most once in such a replacement of a chain, the overall size of the overlay remains linear in size compared to the original one.

In the sequel we will usually assume that an overlay is given in canonical form. This means in particular that any given location may have at most three requests that are pending, the valid configurations are illustrated in Figure 5. Thus an implementation may restrict the length of the priority queue to three items.



(a) A chain of requests

(b) A staircase of requests

(c) External elements on locations L_A and L_B connected to the chain

(d) External elements on locations L_A and L_B connected to the staircase

Figure 4: Transforming a chain in location L to canonical form on locations $L_1, \ldots, L_n$.

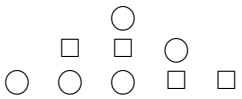


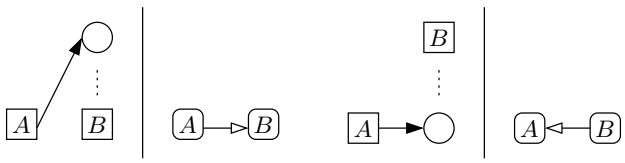
Figure 5: Non-trivial configurations in canonical form

As we have seen, this assumption does not change the modelling capacity nor does it increase the size of the overlay by more than a constant factor. It has the advantage that from now on we may identify three conceptually different types of objects, namely the set of *tasks*, the set of *lock locations* and the set of *Xreqs* : each task corresponds to exactly one (data) location and is the only one to have an Xreq for that location.

Definition 2.6 (Delay digraph) *For a given overlay S in canonical form, its* delay digraph *is constructed as follows:*

1. *Build an undirected graph whose vertices are the locations. One edge is added each time a link exists in the overlay between a pair of locations. This graph is isomorphic to the conflict graph $C(\mathcal{T})$ over tasks from above.*
2. *Add orientation to the edges depending on the link in the overlay:*
   - *If the Xreq A is connected to an Ireq which is above an Xreq B, the edge is oriented $A \rightarrow B$, as in Figure 6(a).*
   - *If the Xreq A is connected to an Ireq which is below an Xreq B, the edge is oriented $A \leftarrow B$, as in Figure 6(b).*

Lemma 2.1 *An overlay S in canonical form has a deadlock iff there is a cycle in its delay digraph.*



(a) priority of Xreq over Ireq (b) priority of Ireq over Xreq

Figure 6: Delays imposed by the priority order on the lock requests

Proof. In a delay digraph, a location L connected to a location L' means that L is blocked as long as L' is blocked. Therefore, if a cycle exists in the delay digraph, all the locations in the cycle are blocked and thus obviously fit the definition of a deadlock.

Contrariwise, if there is a deadlock then all the locations in the deadlock are blocked and are thus connected to a blocked location in the delay digraph. Suppose that all these connections do not form a cycle, then at least one of the location has to be a sink which means that it is not blocked and is thus not part of the deadlock, a contradiction. □

3 Iterative Computations

In this section, we explain how our model can be applied to iterative computations. This category of computations shows data dependencies that allow to run parallel execution by slicing data and feeding a pipeline. As applied here, the model performs per-slice synchronization, *i.e* it synchronizes between computations of data blocks. The use of ORWL and their FIFO policy perfectly suits the dependency requirements of such algorithms. As an iterative algorithm generally computes until stabilization or a predefined number of iterations, computation over a single data block is performed with the following loop: *wait* to acquire the ORWL for reading (or writing), then *perform* computation (if any), then *release* the ORWL.

Therefore a complete execution of an iterative algorithm over some data space requires one ORWL per sliced data block (a location). We now detail the execution model for such algorithms and give some of its properties.

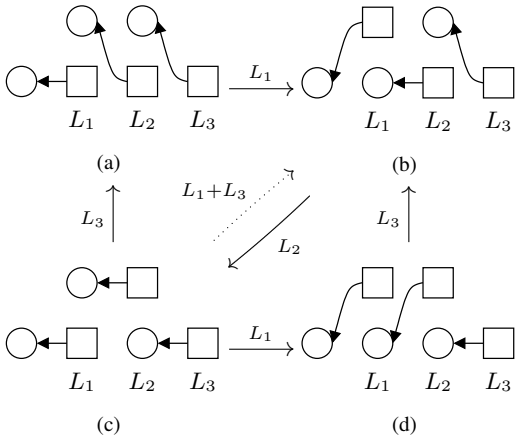


Figure 7: Evolutions of a linear chain of three tasks.

Definition 3.1 (Evolution) *In an overlay S, an executable location $L \in R(S)$ allows the overlay to evolve by executing the location L. This is done with the following steps:*

Setup.
- *A new Xreq is posted on top of L.*
- *For each Ireq to which the lowest Xreq in L is connected to, a new Ireq is posted on top of the corresponding location.*
- *For each connection between the lowest Xreq in L and an Ireq, a similar connection is established between the new Xreq and new Ireq that were previously added.*

Run.
- *The system is now ready to execute the computational task represented by L.*

Clean up.
- *All connections from the original Xreq in L are removed (including the target Ireq if it was the last connection to it).*
- *The original Xreq in L is removed.*

Less formally, execution of a location is done by reinserting the Xreq and its Ireqs (the task) back on top of their locations (for further execution) and then removing it (after the execution of the task took place). Figure 7 gives the evolution of a linear chain of three tasks.

The setup phase uses the delayed request feature of ORWL. Once a lock is acquired, a new request for the same lock is posted for the future iteration. The clean up phase is simply releasing of all acquired locks.

Iterative computations with our model involve an initial distribution of requests (see Section 5). Then each task only competes on a predefined subset of ORWL and runs according to the previously
defined loop. This means that posting a new request during the setup phase can only

Figure 8: New non-trivial configuration

be done if the corresponding (old) request is already hold.

Therefore once a new Xreq has been posted in the setup phase, the location is not in canonical form until the clean up phase. This introduces only one new possible configuration given by Figure 8. This situation is not problematic since the iterative execution model states that nothing can be added atop the location until the middle Ireq has been acquired. This can only happen after the lowest Xreq has been released, which leads back to a canonical form. In this new *half canonical* form, the lowest Xreq corresponds to the task performing the n^{th} iteration, while the highest Xreq corresponds to the computation of the $n + 1^{th}$ iteration by the same task.

Definition 3.2 (Parallelism) *The parallelism level of a given overlay S is given by:*

$$\|S\| = \frac{|R(S)|}{|S|} \quad (1)$$

A parallelism level of 0 *is a deadlock situation ($R(S) = \emptyset$, so there is no executable location).*
A parallelism level of 1 *is maximal ($R(S) = S$, so the entire overlay is executable).*

As defined, the parallelism level is computed on a snapshot of a single iteration. Iterative algorithms may show different parallelism levels, depending on the chosen iteration and data dependency pattern. For instance, the linear chain of three tasks has two possible configurations cycles of three configurations each (abc and bcd). On each of these cycles, one configuration has a parallelism level of $\frac{2}{3}$ and the other two have a parallelism level of $\frac{1}{3}$. The average parallelism level on the cycle is thus of $\frac{4}{9}$.

Definition 3.3 (Devolution) *The evolution rule can easily be reverted by considering the horizontally mirrored overlay and then applying the evolution rule. This is equivalent to the evolution rule with "lowest" replaced by "highest" and "on top of" by "below".*

The following lemma shows that our execution model for iterative algorithms cannot undo a deadlock, nor can it spawn a new one.

Lemma 3.1 *Whenever a deadlock exists in an overlay, then the minimal support for that overlay exists in all its ancestors and successors.*

Proof: Suppose we have an overlay which contains a deadlock. All the Xreqs in the deadlock's minimal support are either not the lowest in their corresponding locations or are connected to Ireqs that are not the lowest in their corresponding locations. This means that whenever an execution occurs, all the items that are removed from the overlay are not part of the deadlock's minimal support and so are the items added atop the overlay. Thus, execution does not modify the deadlock's minimal support, which is therefore present in all successors.

With the devolution rule, the same applies to all ancestors of a given overlay. □

4 Lattice Structure and Deadlocks

In this section, we will give the constructs and proofs of a lattice structure over the overlay configuration space and characterize the configuration spaces with deadlock.

Definition 4.1 (Overlay configuration) *Given an overlay I, an overlay S that can be obtained from I by i applications the evolution rule is said to be in configuration (i, S). The overlay I is in configuration $(0, I)$ and is said to be the* initial configuration*.*

Definition 4.2 *The relation $(i, A) \to (i+1, B)$ denotes the application of the evolution rule once from (i, A) to get $(i+1, B)$. The relation $\xrightarrow{*}$ is the reflexive transitive closure of the relation $\to$. The relation $\xrightarrow{+}$ is the transitive closure of the relation $\to$.*

Lemma 4.1 (Order) *For a given initial configuration $(0, I)$, the relation $\xrightarrow{+}$ defines a partial order.*

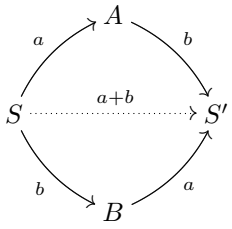
Proof: It is obvious that $(i, A) \xrightarrow{*} (i, A)$ by not applying the evolution rule once (*reflexivity*).

Let $(i, A) \xrightarrow{*} (j, B)$ and $(j, B) \xrightarrow{*} (k, C)$. It is obvious that $(i, A) \xrightarrow{*} (k, C)$, by applying the evolution rule along the two paths to and from (j, B) (*transitivity*).

Let $(i, A) \xrightarrow{*} (j, B)$ and $(j, B) \xrightarrow{*} (i, A)$. First, notice that $(i, A) \xrightarrow{+} (j, B) \implies i < j$. Suppose now that $(i, A) \neq (j, B)$. The evolution rule is thus applied at least once and we have $i < j$ and $j < i$, which is contradictory. Therefore $(i, A) = (j, B)$ (*antisymmetry*). □

Lemma 4.2 (Evolution Independence) *Whenever two locations a and b are independently executable in an overlay S, any order of execution of a and b leads to the same overlay S'.*

Proof. As a and b are independent, all the elements added and removed by the execution of a do not modify the location b. Contrariwise, all the elements added and removed by the execution of b do not modify the location a.



Therefore after both a and b have been executed, all the elements added and removed from S are the same, no matter the order in which the two locations were executed. □

Lemma 4.3 *Let (t, I_t) any common predecessor of (i, A_i) and (j, B_j) such that the two execution paths from (t, I_t) to (i, A_i) and (j, B_j) have a common executable location. Then (t, I_t) is not a maximal predecessor.*

Proof. Let $a = a_t, \ldots, a_{i-1}$ be the path from (t, I_t) to (i, A_i) and $b = b_t, \ldots, b_{j-1}$ be the path from (t, I_t) to (j, B_j). Let us assume that $L = a_\lambda$ is the first element in a with $L \in a \cap b$.

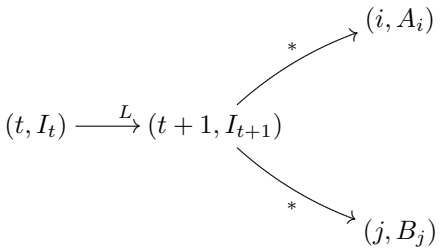


Figure 9: Inductive step.

If $\lambda = t$ then L is executable for (t, I_t). Otherwise, if $\lambda \neq t$ the locations $a_t, \ldots, a_{\lambda-1}$ are not executed in b. Then, since L is also executed in b, $a_t, \ldots, a_{\lambda-1}$ can't change the property of L of being executable or not. Thus L must already be executable for (t, I_t).

By the Lemma 4.2, we know that the execution order of initially executable elements along a path does not matter, thus if there is $L \in a \cap b$, we can permute the execution order so that L appears first in both a and b. Now denote by $(t+1, I_{t+1})$ the configuration that is obtained from (t, I_t) by executing L. Then we have the situation of Figure 9 and this completes the proof of the claim. □

Theorem 4.1 (Lattice structure) *The configuration space built from an overlay initial configuration and the evolution rule forms a lattice.*

Proof: Since any two configurations are derived from the initial configuration, by Lemma 4.3 we see that there is a unique maximal predecessor (p, P_p) to any two attainable configurations (i, A_i) and (j, B_j) with $i > p$ and $j > p$, and $(i, A_i) \not\xrightarrow{*} (j, B_j)$ and $(j, B_j) \not\xrightarrow{*} (i, A_i)$. It follows that the locations that are executed on the two path from (p, P_p) to (i, A_i) and (j, B_j) form two sets of locations that are mutually disjoint.

We now show by induction that (i, A_i) and (j, B_j) have a common successor and that the minimal such successor is unique.

Let $a = \{\!\{a_p, \ldots, a_{i-1}\}\!\}$ be the multiset containing the locations along the execution path from (p, P_p) to (i, A_i) and similarly let $b = \{\!\{b_p, \ldots, b_{j-1}\}\!\}$ be the multiset containing the locations along the execution path from (p, P_p) to (j, B_j).

By definition $a \cap b = \emptyset$ and either $a \neq \emptyset$ or $b \neq \emptyset$ (otherwise $A_i = B_j$). W.o.l.g, let assume that $a \neq \emptyset$. Lemma 4.2 states that, as a_p is executable in (p, P_p), it remains so along the path to (j, B_j). Thus $(j, B_j) \xrightarrow{a_p} (j + 1, B_{j+1})$.

Let $(p, P) \xrightarrow{a_p} (p + 1, P_{p+1})$. $(p + 1, P_{p+1})$ is the maximal predecessor of (i, A_i) and $(j + 1, B_{j+1})$. Now suppose that $A_i \not\xrightarrow{*} B_{j+1}$. Then obviously $(p + 1, P_{p+1})$, (i, A_i) and $(j + 1, B_{j+1})$ still fulfill the induction hypothesis. So by induction we may conclude that we may subsequently add elements to $B = B_j, B_{j+1}, \ldots, B_\ell$ until $(i, A_i) \xrightarrow{*} (\ell, B_\ell)$. This show that there is a common successor of (i, A_i) and (j, B_j). B_ℓ is minimal and unique with that property since by construction its set of locations is $a \cup b$ and $\ell = |a \cup b|$. □

Chip Firing Games. During the development of our model, we defined a morphism from overlays to chip firing games (CFG). CFG, as defined in [12], are directed graphs whose vertices are numbered by a certain amount of *chips*. Once a vertex acquires more chips
than its outgoing degree, it can be *fired*. Firing a vertex means to decrease its chips

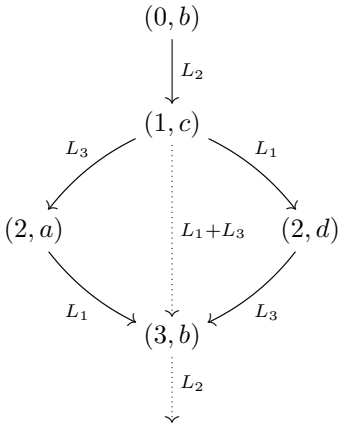


Figure 10: Lattice for the linear chain of three tasks.

by its outgoing degree and distribute one chip along each outgoing link, which is then added to the connected vertex.

Definition 4.3 (Evolution by a CFG) *The evolution of an overlay in canonical form can be modeled with a chip firing game (CFG). The support for the CFG of a given overlay S is built as follows:*

- *Each constrained location in the overlay gives a vertex in the support.*
- *Each location connected to a non-constrained location in the overlay gives a loop to its corresponding vertex.*
- *Each connection between an Xreq and an Ireq in a constrained location in the overlay gives a pair of arcs, one directed forward (according to the corresponding locations) and the other directed backward.*

With:

- $\delta_S(L)$ *be the number of connections originating from location* L*.*
- $\sigma_S(L)$ *be the total number of requests in location* L*.*
- $\omega_S(L)$ *be the rank of the Xreq in location* L*.*
- $\rho_S(L, L')$ *be the rank of the Ireq in location* L' *to which the Xreq in location* L *is connected (if such a connection exists,* 0 *otherwise).*

The initial chip distribution for a vertex v given by a constrained location L is:

$$chips(v) = \delta_S(L) + \sigma_S(L) - 1 - \omega_S(L) - \sum_{L' \in S} \rho_S(L, L')$$

Note that by construction, all connected components in the CFG are closed components.

In [11], an extended configuration for the CFG is given for the general case, in particular the ones with closed components. This new configuration space has the same properties of the original one, notably the equal length of paths between two configurations and the lattice structure.

Infinite Lattices and Deadlocks. In the general case of iterative computations, the configuration space is an infinite lattice. This is a desired state since this kind of applications typically require computation to be performed until some external event occurs (stabilization or user-defined termination). Figure 10 gives the beginning of the lattice structure for the linear chain of three tasks with initial configuration given by Figure 7(b).

Theorem 4.2 (Deadlock is finite) *An overlay which has a deadlock has a finite configuration space and the maximal configuration is a deadlock situation.*

Algorithm 1: Compute an initial request ordering

Input: A set of tasks $\mathcal{T}$, a set of lock locations $\mathcal{L}$ and for each task $T \in \mathcal{T}$ a list of Xreqs $(X_1, \ldots, X_w)$ and of Ireqs $(I_1, \ldots, I_r)$, where the X_i and I_j are locations in $\mathcal{L}$.

Output: For each $L \in \mathcal{L}$ a priority ordering of the requests for L such that the resulting overlay as a whole is deadlock free.

construct construct an implicit representation of the conflict graph $C(\mathcal{T})$;

color compute a coloring $\mathcal{T}_1, \ldots, \mathcal{T}_x$ of $C(\mathcal{T})$;

foreach *location* $L \in \mathcal{L}$ *of* T **do**
  initialize $p(L)$ to 0;

insert **foreach** *color* $c = 1, \ldots, x$ **do**
  foreach *task* $T \in \mathcal{T}_c$ **do in parallel**
    foreach *Xreq* X *of* T, $X = X_1, \ldots, X_w$ **do**
      increment $p(L)$, set the priority of X to the new value and increment $p(L)$ again;
    foreach *Ireq* I *of* T, $I = I_1, \ldots, I_r$ **do** set the priority of X to $p(L)$;

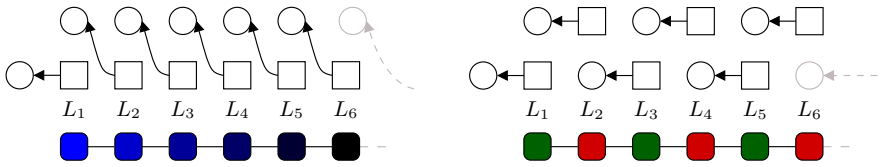
Proof: A finite configuration space (and thus a maximal configuration) obviously fits the definition of a deadlock (no further evolution possible).

Suppose we have an overlay which is known to have a deadlock. Let (i, D) be a configuration of a deadlock at minimum distance i from the initial configuration. Let (i, C) be another configuration at the same distance from the initial configuration. The configuration space is a lattice, so we have (j, B) with $(i, C) \xrightarrow{+} (j, B)$ and $(i, D) \xrightarrow{+} (j, B)$. Since (i, D) is in fact a deadlock, we must have that $j = i$ and thus that $B = D$. But then also $C = D = B$ and the two configurations are the same. Thus (i, D) is the only configuration that exists at distance i. It follows that there is no other attainable configuration at distance $j > i$. Hence the lattice is finite (with the maximal configuration being the deadlock configuration). □

5 Overlay Initialization and Deadlock Detection

In the first part of this section we will not be able to assume that an overlay would be in canonical form or that each task only has one Xreq. This is so, since the replacement techniques that we introduced already need a priority ordering as an input.

Deadlock Free Initialization. For a strategy to construct an initial overlay see Algorithm 1. It consists of first computing the conflict graph, finding a partition of it into independent sets (*i.e* a coloring) and then placing the lock requests according to the position of color class of the task. Since it only handles tasks in parallel that don't have conflicts, at most one Xreq X is inserted for a given location L during each of the x insertion phases. The following lemma is easily verified by observing that in the



(a) One color class per location (b) The result of a 2-coloring

Figure 11: Two different initializations of the same set of requests

corresponding delay digraph the orientations of the arcs always lead from a task with a higher number in the coloring to one with a lower number.

Lemma 5.1 *Algorithm 1 generates an overlay that is deadlock free.*

Lemma 5.2 *Besides the phases* **construct** *and* **color***, Algorithm 1 runs in sequential time that is proportional to the input and can be implemented in parallel to run in time x, where x is the coloring number that* **color** *computed.*

The two phases **construct** and **color** are not explained in Algorithm 1 (and thus in Lemma 5.2). In **construct**, we could be tempted to construct the entire graph $C(\mathcal{T})$. This graph could be quite large for the case that there are several Xreqs in one location; $C(\mathcal{T})$ then has a clique containing all the tasks that write to the same location. Such an explicit construction can be avoided by using a data structure that uses the knowledge about the shared locations. Details of that are left to the extended version of this paper.

With this implicit representation of $C(\mathcal{T})$ in **color** we then have to construct a coloring. Although this problem is NP-hard in general, there are good heuristics that provide colorings with $\Delta + 1$ colors, where Δ is the maximum degree of the graph. They can be computed in parallel and in a distributed setting, see [6], and large scale tests (especially for matrix computations) show that these algorithms behave quite well in practise, see [3]. We don't think these computations will be a bottleneck for real applications of our techniques. Also, usually we may suppose that we are in a setting where each processor will have to perform several tasks per iteration. Thus a rough estimate of the coloring number should suffice.

In addition, Algorithm 1 is only computing the initial setting, afterwards the system evolves asynchronously. The set of tasks that are active in parallel can be the set of executable locations of any of the reachable configurations. Consider the example in Figure 11. Both subfigures represent different initializations for the same system of tasks and that both could be produced by Algorithm 1. 11(a) corresponds to a non-optimal coloring where each task has its own color and these colors are taken in worst possible order. In contrast, 11(b) corresponds to an optimal coloring with just two colors. 11(b) has a much higher degree of parallelism than 11(a) and is therefore clearly preferable. Nevertheless, both configurations are part of the same configuration space and reachable from each other. Thus, the behavior of the system on the long run does not depend too much from initial setting but much more on the execution strategy that chooses among the executable tasks.

Transforming into Canonical Form. As we have seen in Section 2, a transformation of an overlay into canonical form can be done with local replacement rules that eventually introduce new locations and Ireqs, and may move some dependencies from one request to another. It is easy to see that such replacements can be computed in a distributed setting. Also observe that these replacements do not change the execution: no additional tasks are introduced and the relative order of tasks is strictly respected. We leave the details for a full version of this paper.

Deadlock Detection. If for some reason an overlay can't be initialized by Algorithm 1 we may use the tools of Section 2 to detect a deadlock. The idea would be to first construct the conflict graph and then the delay digraph $C(\mathcal{T})$ from it. Then from Lemma 2.1 it follows that a simple test for cycles will show whether or not the overlay contains a deadlock or not.

But with the structural results of Section 4 we even have a much easier strategy, which consists in a test execution of all tasks.

Theorem 5.1 *Suppose an overlay has a sequence of executions such that each location is executed at least once. Then the overlay is deadlock free.*

Proof: First assume that the overlay is connected. Let $L = (L_1, \ldots, L_k)$ be an execution path that executes all locations. Let $L' = (L'_1, \ldots, L'_{k'})$ the sub-sequence of the first occurrence of the locations. From Lemma 4.2 we know that L' is also executable. Thus L' leads back to the initial configuration and as a consequence the lattice of configurations is infinite. Theorem 4.2 then proves that the overlay must be deadlock-free.

If the overlay is not connected the argument applies to each connected component. □

The algorithmic details of such a deadlock detection routine are left to the reader.

6 Conclusion and Outlook

The analysis of our model shows valuable properties for parallel computation in general and iterative algorithms in particular. It is interesting to note that it allows for automated initialization of a system from an abstract representation of tasks and dependencies. The user has only to specify these dependencies and to implement the computational part of the different tasks. The lattice structure of the overlay configuration space then ensures that a deadlock-free initialized system will run as expected; namely is guaranteed to remain without deadlock and to execute all tasks evenly.

A first implementation of ORWL has been added to our PARXXL library [8]. It is based on request stamps (the priority ranks) that are accorded to lock-handles data structure and on condition variables to regulate congestion.

As ORWL are a general tool which is not bound to a particular class of applications, we plan some extensions. For instance, it could be used to model byte-range locking on files or other objects by tackling the problem of joint ranges. This would provide predictable access to shared objects without imputing to the user more overhead than

common read-write locks operations. Also, as ORWL are resource-oriented they can more easily be extended to distributed environments. This will enable us to implement the *data handover* API as it was described in [7].

As shown, the coloring method directly impacts the initialization of a given system. For some applications, different initializations can lead to disconnected configuration spaces. For instance, it is possible to initialize a linear cycle to produce either a token-ring or a half-homogeneous execution pattern. This makes it possible to also use ORWL as a temporal synchronizing tool.

References

[1] J. M. Bahi, S. Contassot-Vivier, and R. Couturier. Evaluation of the asynchronous iterative algorithms in the context of distant heterogeneous clusters. *Parallel Computing*, 31(5):439–461, 2005.

[2] 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.

[3] D. Bozdag, A. Gebremedhin, F. Manne, E. Boman, and U. Catalyurek. A framework for scalable greedy coloring on distributed memory parallel computers. *Journal of Parallel and Distributed Computing*, 68(4):515–535, 2008.

[4] P.-N. Clauss, J. Gustedt, and F. Suter. Out-of-Core Wavefront Computations with Reduced Synchronization. In Julien Bourgeois, François Spies, and Didier El Baz, editors, *16th Euromicro International Conference on Parallel, Distributed and network-based Processing*, page 8 pages, Toulouse France, 2008. IEEE Computer Society Press.

[5] G. Cong and D. A. Bader. Lock-free parallel algorithms: An experimental study. In L. Bougé and V. K. Prasanna, editors, *HiPC*, volume 3296 of *Lecture Notes in Computer Science*, pages 516–528. Springer, 2004.

[6] A. H. Gebremedhin, I. Guérin Lassous, J. Gustedt, and J. A. Telle. Graph Coloring on a Coarse Grained Multicomputers. *Discrete Applied Mathematics*, 131(1):179–198, Sept. 2003.

[7] J. Gustedt. Data Handover: Reconciling Message Passing and Shared Memory. Rapport de recherche, INRIA, Nov. 2004.

[8] J. Gustedt, S. Vialle, and A. De Vivo. parXXL: A fine grained development environment on coarse grained architectures. In *Workshop on State-of-the-Art in Scientific and Parallel Computing - PARA'06*, Umeå, Sweden, June 2006.

[9] A. Josey et al., editors. *The Open Group Base Specifications Issue 6 – IEEE Std 1003.1*. The IEEE and The Open Group, 2004.

[10] O. Krieger, M. Stumm, R. Unrau, and J. Hanna. A fair fast scalable reader-writer lock. In *International Conference on Parallel Processing (ICPP 1993)*, volume 2, pages 201–204. IEEE, 1993.

[11] M. Latapy and H. D. Phan. The lattice structure of chip firing games and related models. *Physica D*, 155:69–82, 2001.

[12] C. Magnien. *Étude du modèle du tas de sable abélien : points de vue algorithmique et algébrique*. PhD thesis, École Polytechnique, France, 2003.

[13] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9:21–65, 1991.

[14] C. Wagner and F. Mueller. Token-based read/write-locks for distributed mutual exclusion. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1185–1195, London, UK, 2000. Springer-Verlag.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399