Parallelizing Union-Find in Constraint Handling Rules Using Confluence Analysis

Thom Frühwirth

Faculty of Computer Science, University of Ulm, Germany
www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/

Abstract. Constraint Handling Rules is a logical concurrent committed-choice rule-based language. Recently it was shown that the classical union-find algorithm can be implemented in CHR with optimal time complexity. Here we investigate if a parallel implementation of this algorithm is also possible in CHR. The problem is hard for several reasons:
- Up to now, no parallel computation model for CHR was defined.
- Tarjan's optimal union-find is known to be hard to parallelize.
- The parallel code should be as close as possible to the sequential one.
It turns out that confluence analysis of the sequential implementation gives almost all the information needed to parallelize the union-find algorithm under a rather general parallel computation model for CHR.

1 Introduction

Constraint Handling Rules (CHR) [7] is a concurrent committed-choice constraint logic programming language consisting of guarded rules that transform multi-sets of constraints (atomic formulae) until a fixpoint is reached.

CHR was initially developed for writing constraint solvers, but is more and more used as a general-purpose programming language. Recent applications of CHR range from type systems and time tabling to ray tracing and cancer diagnosis [2, 15]. In these applications, conjunctions of constraints are best regarded as interacting collections of concurrent agents or processes.

So far, there have been several operational semantics for CHR (standard, refined and compositional), but none of them explicitely addresses *parallelism*. We develop a general parallel execution model for CHR relying on a monotonicity property (applicable rules cannot become in-applicable during a computation). Analogous concurrency constructions were suggested for other (constraint) logic programming languages, e.g. [12].

The clean semantics of CHR facilitates non-trivial program analysis and transformation. In particular, confluence analysis is an issue in CHR, since rule application is committed-choice, it is never undone (unlike Prolog). *Confluence* aks the question if a program produces the same result no matter which of the applicable rules are applied in which order. It turns out that this property facilitates the parallel execution of a CHR program. Since there is a decidable, sufficient and necessary criterion for confluence [3] that returns the problematic

cases of rules applications that rule out each other, we can use this analysis to construct a parallel program from a sequential one. Of course, some crucial insights in the nature of the algorithm to be implemented are still necessary. As a side-effect of our work, we implemented a practical confluence checker.

We will apply this methodology of *parallization to the classical union-find* (also: disjoint set union) algorithm of Tarjan [17]. This essential algorithm efficiently solves the problem of maintaining a collection of disjoint sets under the operation of union [9]. It is the basis for many graph algorithms and for dealing with equality, e.g. in unification algorithms. We have chosen union-find, because it was recently shown that it is possible to implement it with optimal time complexity in CHR [13, 14, 16], something that is not known to be possible in other pure logic programming languages. The bad news is that union-find is inherently sequential in most parts, and therefore hard to parallelize. Its worst case time performance can actually get worse upon parallelization if the sequential algorithm is used as a basis [4]. Often, other data structures and algorithms are used for the parallel union-find problem [10].

So this is our *challenge*: Can we come up with an optimal parallel union-find algorithm that is close to the sequential one? Can confluence analysis help? This paper gives a preliminary answer tending towards positive.

Outline of the Paper In the next two sections, we introduce union-find algorithms and CHR. In Section 4 we introduce a parallel execution model for CHR. In the next section we give a sequential CHR program for the basic union-find algorithm. Section 6 uses confluence analysis to parallize this implementation. The next two sections carry this approach over to optimized union-find. At the end of Section 8, correctness of the parallel version is argued by simulating the sequential one and vice versa. Section 9 concludes with future work.

2 The Union-Find Algorithm

The union-find algorithm maintains disjoint sets under union. Each set is represented by a rooted tree, whose nodes are the elements of the set. The root is called the *representative* of the set. The representative may change when the tree is updated by a union operation. With the algorithm come three operations:

- `make(X)`: generate a new tree with the only node `X`, i.e. `X` is the root.
- `find(X)`: follow the path from the node `X` to the root of the tree by repeatedly going to the parent node of the current node until the root is reached. Return the root as representative.
- `union(X,Y)`: to join the two trees, find the representatives of `X` and `Y` (they are roots). Then `link` them by making one point to the other.

The basic algorithm requires $\mathcal{O}(N)$ time per find (and union) in the worst case, where N is the number of elements (make operations). With two independent optimizations that keep the tree shallow and balanced, one can achieve logarithmic worst-case and quasi-constant (i.e. almost constant) amortized running time per operation.

The first optimization is *path compression* for find. It moves nodes closer to the root after a find. After `find(X)` returned the root of the tree, we make every node on the path from `X` to the root point directly to the root.

The second optimization is *union-by-rank*. It keeps the tree shallow by pointing the root of the smaller tree to the root of the larger tree. *Rank* refers to an upper bound of the tree depth (tree height). If the two trees have the same rank, either direction of pointing is chosen but the rank is increased by one. With this optimization, the height of the tree can be bound by $log(N)$. Thus the worst case time complexity for a single find or union operation is $\mathcal{O}(log(N))$.

Parallelization can worsen the performance of optimized union-find, because the find operation is inherently sequential and parallel tree updates can counteract the effects of path compression and union-by-rank [4] so that deep trees (with long paths) are generated. In order to achieve logarithmic worst case time complexity per operation, one has to restrict the parallelism, use special auxiliary data and operations [4] or has to rely on different special-purpose data structures and algorithms altogether [10, 5].

3 Constraint Handling Rules (CHR)

In this section we give an overview of syntax and semantics for Constraint Handling Rules (CHR) [7, 8].

Syntax of CHR We use two disjoint sets of predicate symbols for two different kinds of constraints: built-in (pre-defined) constraint symbols which are solved by a given constraint solver, and CHR (user-defined) constraint symbols which are defined by the rules in a CHR program. There are three kinds of rules:

$$\begin{array}{ll} \textit{Simplification rule:} & \textit{Name} \ @ \ H \Leftrightarrow C \mid B, \\ \textit{Propagation rule:} & \textit{Name} \ @ \ H \Rightarrow C \mid B, \\ \textit{Simpagation rule:} & \textit{Name} \ @ \ H \setminus H' \Leftrightarrow C \mid B, \end{array}$$

where *Name* is an optional, unique identifier of a rule, the *head* H, H' is a non-empty comma-separated conjunction of CHR constraints, the *guard* C is a conjunction of built-in constraints, and the *body* B is a goal. A *goal (query, problem)* is a conjunction of built-in and CHR constraints. A trivial guard expression "`true |`" can be omitted from a rule.

Simpagation rules abbreviate simplification rules of the form

$$\textit{Name} \ @ \ H \wedge H' \Leftrightarrow C \mid H \wedge B.$$

Standard Operational Semantics of CHR The operational semantics of CHR is given by a transition system (Fig. 1) . States are goals, i.e. conjunctions of built-in and CHR constraints. In the figure, all upper case letters are meta-variables that stand for conjunctions of constraints. CT is the constraint theory for the built-in constraints. G_{bi} denotes the built-in constraints of G, which is the remainder of the state/goal.

Simplify

If	$(H \Leftrightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\,(G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto (B \wedge G \wedge H{=}H' \wedge C)$

Propagate

If	$(H \Rightarrow C \mid B)$ is a fresh variant of a rule with variables $\bar{x}$
and	$CT \models \forall\,(G_{bi} \rightarrow \exists \bar{x}(H{=}H' \wedge C))$
then	$(H' \wedge G) \mapsto (H' \wedge B \wedge G \wedge H{=}H' \wedge C)$

Fig. 1. Computation Steps of Constraint Handling Rules

CHR rules are applied exhaustively, until a fixed-point is reached, to the initial state. A simplification rule $H \Leftrightarrow C \mid B$ *replaces* instances of the CHR constraints H by B provided the guard C holds. A propagation rule $H \Rightarrow C \mid B$ instead *adds* B to H. If new constraints arrive, rule applications are restarted.

A rule is *applicable*, if its head constraints are matched by constraints in the current goal one-by-one and if, under this matching, the guard of the rule is implied by the built-in constraints in the goal. Any of the applicable rules can be applied, and the application cannot be undone, it is committed-choice (in contrast to Prolog).

When a simplification rule is applied, the matched constraints in the current goal are replaced by the body of the rule. When a propagation rule is applied, the body of the rule is added to the goal without removing any constraints. When a simpagation rule is applied, all constraints to the right of the backslash are replaced by the body of the rule.

To avoid trivial non-termination, a CHR propagation rule is never applied a second time to the same constraints. A *final state* is one where either no computation step is possible anymore or where the built-in constraints are inconsistent.

The *refined operational semantics* [6] specializes the CHR standard semantics as given here to the one that is usally implemented: Similar to Prolog, constraints in a state are evaluated depth-first from left to right and rules are applied in textual program order. (The refined semantics thus rules out some computations that are possible in the standard semantics.)

4 Parallelism for CHR

Intuitively, we expect that in a parallel execution of a CHR program, rules can be applied to separate parts of the problem in parallel without interference. We will interpret conjunction as parallel operator and we will use an *interleaving semantics* for parallelism in CHR. It means that a parallel computation step can be performed by a sequence of sequential computation steps. A similar approach was taken for other concurrent constraint/logic languages, e.g. [11, 12].

To avoid the technicalities of special cases, we relax the operational semantics of CHR with regard to final states. We allow a finite, bounded number of additional computation steps from inconsistent states (they will stay inconsistent).

$$\begin{array}{lll} \text{If} & A & \mapsto B \\ \text{and} & C & \mapsto D \\ \text{then} & A \wedge C & \mapsto B \wedge D \end{array} \qquad \begin{array}{lll} \text{If} & A \wedge E & \mapsto B \wedge E \\ \text{and} & C \wedge E & \mapsto D \wedge E \\ \text{then} & A \wedge E \wedge C & \mapsto B \wedge E \wedge D \end{array}$$

Fig. 2. Weak Parallelism of CHR Strong Parallelism of CHR

We now define two notions of parallelism, weak and strong (Fig. 2). (In the figures, A, B, C, D and E are conjunctions of arbitrary constraints.) Such straightforward interleaving semantics for parallelism are not possible for imperative languages, where computations may give raise to conflicting (over-)writes. However, it is possible for many (constraint) logic programming languages, due to the *monotonicity (or stability) property* (left of Fig. 3): Adding constraints to a state cannot inhibit the applicability of a rule. Monotonicity of CHR was proven in [3]. The property also implies that constraints can be processed incrementally in CHR, giving rise to an online algorithm behavior.

We can now justify *weak parallelism* by a consequence of monotonicity which we call *trivial confluence* (right of Fig. 3), because independent of the intermediate state, we will arrive at the same successor state.

$$\begin{array}{lll} \text{If} & A & \mapsto B \\ \text{then} & A \wedge C & \mapsto B \wedge C \end{array} \qquad \begin{array}{lll} \text{If} & A & \mapsto B \\ \text{and} & C & \mapsto D \\ \text{then} & A \wedge C & \mapsto S \mapsto B \wedge D \\ & (S \text{ is either } A \wedge D & \text{or } B \wedge C) \end{array}$$

Fig. 3. Monotonicity of CHR Trivial Confluence of CHR

The definition of *strong parallelism* (right of Fig. 2) shows that there is more potential for parallelism in CHR than working on separate parts of the problem. Constraints in E may be necessary for rule application, but since both rules do not alter these constraints, we can still apply them in parallel. With strong parallelism, rules may work on *common* constraints at the same time if they do not change them. (With weak parallelism, we would need two copies of the constraints E.) Clearly all built-in constraints are common. Propagation rules only add CHR constraints, so any CHR constraints they match can be common. Simpagation rules do not remove some of the constraints they match, so these can be common as well. We will assume strong parallelism in the rest of the paper.

We assume for now that rule applications (hence, computation steps) are *instantaneous*, i.e. the removal and addition of constraints caused by the application of a rule is an atomic action. With this requirement, a rule can still take arbitrary time to check its applicability to constraints in (a snapshot of) the current state. When a rule is to apply, it will first flag the constraints it matched. If some are not there anymore, the rule application simply is not done and flags are reset.

5 Implementing Basic Union-Find in CHR

The CHR program `ufd_basic` (in concrete ASCII syntax) implements the operations and data structures of the basic union-find algorithm optimizations as CHR constraints [16].

ufd_basic

```
make      @ make(A) <=> root(A).
union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B  \ find(A,X) <=> find(B,X).
findRoot  @ root(A) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
link      @ link(A,B), root(A), root(B) <=> B ~> A, root(A).
```

The constraints `make/1`, `union/2`, `find/2` and `link/2` define the *operations*. The find operation is implemented as a relation `find/2` whose second argument returns the result. `link/2` is an auxiliary operation for performing union of two roots. The *tree (data) constraints* `root/1` and `~>/2` ("points to") represent the tree data structure. This program is operationally equivalent for allowed queries to implementations in imperative languages under the refined sequential CHR semantics [16]. An *allowed query* is - as usual for union-find - a sequence of `make`, `find` and `union` operations. The second argument of a `find` is a new variable. Nodes are typically constants. Each node is introduced by one `make`.

Now we discuss *parallel execution* of the above rules for allowed queries. We can accomodate different find operations on the same node, since the tree constraints are not altered by a find. The `link` rule replaces `root(B)` by `B~>A`. Since rule application is instantaneous and atomic in our model, there will always be a tree constraint for each node that has been introduced. So if one of the processes performing an operation fails, it can do no harm to the overall computation, since each rule defines exactly one operation. Actually, the operations make, union and find can always proceed at their own speed. The link operation obviously has to wait for the result of the find operations. Moreover, when we are about to apply the `link` rule, another link operation may remove one of the roots that we need for linking. The next section explains how we can detect and avoid such problematic situations using confluence analysis and additional rules.

6 Confluence for Parallelism

We already have used trivial confluence to justify our model of parallelism. But the relationship is deeper. The interleaving semantics of a parallel execution can be given as the semantics of all its possible interleavings, i.e. sequential executions that lead to the same resulting state. For analysis, we then have to consider all possible sequential execution orders. These different orders of constraints in a goal may mean that different rules are applied. Confluence tells us that no matter which of the applicable rules we apply, we always can reach the same resulting state. In other words, a particular parallel execution cannot go "astray", resulting in a different state (that may well correspond to a *deadlock*). We will see that such deadlocks actually can occur in the basic union-find algorithm.

Confluence Analysis in CHR Before we discuss confluence of the union-find algorithm, we introduce the basic idea behind confluence analysis. The papers [1, 3] give a decidable, sufficient and necessary condition for confluence for terminating CHR programs.

For checking confluence, one takes two rules (not necessarily different) from the program. The heads of the rules are overlapped by equating at least one head constraint from one rule with one from the other rule. For each *overlap*, we consider the two states resulting from applying one or the other rule. These two states form a so-called *critical pair*. One tries to *join* the states in the critical pair by finding two computations starting from the states that reach a common state. If the critical pair is not joinable, we have found a counterexample for confluence of the program.

6.1 Confluence of Basic Sequential Union-Find for Parallelization

A detailed confluence analysis of the sequential union-find algorithms in CHR is in [13, 14]. Union-find is not confluent under the standard sequential operational semantics. The relative order of find and union (link) operations matters for the outcome of find. This behavior is inherent in the union-find algorithm due to its update of the tree structure and the resulting changes of the representatives.

But non-confluence can also be caused by *incompatible tree constraints* such as `root(A), A~>B` (that can be shown not to occur when computing with allowed queries), and due to *competing link operations* for the same roots (that cannot occur in the left-to-right execution order of the refined semantics, but in parallel execution). A *deadlock* means that an operation cannot finish. In the last case, link operations deadlock, and the restoration of confluence by adding proper rules can avoid or break these deadlocks.

Since there is a combinatorial explosion in the number of critical pairs with program size, it is important to filter out "trivial" non-joinable critical pairs that either stem from overlaps that are not possible for allowed queries or that we would like to consider equivalent for our purposes. For union-find, the former means e.g. to detect incompatible tree constraints, the latter means to regard tree constraints that describe the same sets as equivalent. We revised the confluence

checker of [13] so that it performs the necessary additional checks before and normalization after computation of a critical pair. These program-specific filters are encoded as Prolog or CHR rules, e.g. `check, root(A,_), A~>_ <=> fail`. The confluence checker and its results for the union-find programs of this paper are available at `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/union-find/`.

For the union-find implementation `ufd_basic`, there are 8 non-joinable critical pairs [13, 14]. Three non-joinable critical pairs, between the pairs of rules `findNode-findNode`, `findNode-findRoot`, and `linkEq-link`, feature incompatible tree constraints. We avoid the remaining non-trivial critical pairs by modifying the given program.

The critical pair between `find` and `link` reveals that the *relative order of find and link* operations matters for the outcome of the find.

```
            find(B,A),root(B),root(C),link(C,B)
      findRoot                            link
root(C),B~>C,A=B                    root(C),B~>C,A=C
```

The first line gives an overlap of the two rules mentioned in the second line. The third line gives the critical pair, i.e. the two final states reachable when the corresponding rule from the second line is applied to the overlap.

The last four non-joinable critical pairs come from overlapping the `link` with itself. They feature *pending competing links*. Two link operations have at least one tree node in common. So when one link is performed, at least one node in the other link operation is not a root anymore, and so this link operation will *deadlock*, for example:

```
        root(A),root(B),link(B,A),link(A,B)
      link                          link
root(B),A~>B,link(A,B)      root(A),link(B,A),B~>A
```

Insight #1 To handle these non-confluences, we first concentrate on the critical pair between `findRoot` and `link`. We replace the culprit built-in equality constraint `=/2` by our own new CHR constraint `found/2`, that we can tailor to our needs. In the `findRoot` rule, `X=A` becomes `found(A,X)`. It holds the result of the find operation in the first argument. Now we can add a rule for `found` (given below) that joins the corresponding critical pair. (The rule mimics `findNode` so that the `found` constraint keeps track of the updates of the tree.)

```
            find(B,A),root(B),root(C),link(C,B)
      findRoot                            link
                root(C),B~>C,found(C,A)
```

The `link` rules are modified by replacing instances of `link(A,B)` in the head of a rule by the proper instances of `link(X,Y), found(A,X), found(B,Y)`. The resulting program is `ufd_basic1`. Also the critical pairs of the `link` rules can be joined now, because `found` can update itself so that its result argument is a root.

ufd_basic1

```
findNode   @ A ~> B  \ find(A,X) <=> find(B,X).
findRoot1  @ root(A) \ find(A,X) <=> found(A,X).

found      @ A ~> B \ found(A,X) <=> found(B,X).

linkEq1    @ link(X,Y), found(A,X), found(A,Y) <=> true.
link1      @ link(X,Y), found(A,X), found(B,Y),
                        root(A), root(B) <=> B ~> A, root(A).
```

Confluence of Basic Parallel Union-Find The confluence analysis lead us to a revised, parallel version `ufd_basic1` of the basic algorithm. There are now many more non-joinable critical pairs, because the introduction of the `found` constraints gives rise to many possible overlaps for the `link` rules. All the critical pairs are trivial, because they either cannot occur for allowed queries or they feature different tree constraints that represent the same set of nodes.

findNode	findNode	1
findNode	findRoot	1
linkEq	linkEq	6
link	linkEq	13
link	link	65
found	link	2
found	found	1

Number of crit. pairs between pairs of rules

7 Optimized Union-Find

The CHR program *ufd_rank* from [16] implements the optimized classical union-find algorithm, derived from the basic version by adding path compression for find and union-by-rank [17].

ufd_rank

```
make      @ make(A) <=> root(A,0).
union     @ union(A,B) <=> find(A,X), find(B,Y), link(X,Y).

findNode  @ A ~> B, find(A,X) <=> find(B,X), A ~> X.
findRoot  @ root(A,_) \ find(A,X) <=> X=A.

linkEq    @ link(A,A) <=> true.
linkLeft  @ link(A,B), root(A,N), root(B,M) <=> N>=M |
              B ~> A, N1 is max(N,M+1), root(A,N1).
linkRight @ link(B,A), root(A,N), root(B,M) <=> N>=M |
              B ~> A, N1 is max(N,M+1), root(A,N1).
```

When compared to the basic version `ufd_basic`, we see that `root` has been extended with a second argument that holds the rank of the root node. (The first two rules, `make` and `union`, will stay the same for all remaining programs in this paper, they are therefore omitted from now on.)

The rule `findNode` has been extended for immediate path compression: the logical variable `X` serves as a place holder for the result of the find operation. The `link` rule has been split into two rules `linkLeft` and `linkRight` to reflect the optimization of union-by-rank: The smaller ranked tree is added to the larger ranked tree without changing its rank. When the ranks are the same, either tree is chosen (both rules are applicable) and the rank is incremented.

Confluence The non-joinable critical pairs are in principle analogous to the ones discussed for `ufd_basic` in Section 6, but their numbers significantly increase due to the optimizations of path compression and union-by-rank that complicate the rules for the find and link operations. The confluence checker found 73 non-joinable critical pairs [13, 14]. The number of critical pairs is dominated by those 68 of the link rules.

Unlike the basic versions, in the optimized algorithm, two `findNode` rule applications on the same node will interact, because one will compress, and then the other cannot proceed until the first find operation has finished:

```
              find(B,A),B~>C,find(B,D)
        findNode                  findNode
find(A,D),find(C,A),B~>D    find(D,A),find(C,D),B~>A
```

The critical pairs for the find rules tell us that parallel finds have to wait for the result of path compression from one of the finds. In the worst case, if that find process fails, other finds will deadlock (which was not the case in the basic version of the algorithm). As a remedy we introduce an explicit compression operation that runs in parallel to the other operations.

8 Optimal Union-Find Parallelized

We first introduce `found` into the program of optimal union-find as for the basic algorithm.

Insight #2 We make compression explicit by a new operation `compr/2`. We modify the `findNode` rule to call `compr(A,X)` instead of immediately producing a tree data constraint that points to a yet free variable. As a consequence, the rule for explicit compression should take the found root node and the corresponding tree constraint to be compressed and replace it by the compressed tree constraint that points to the root. We should not forget to add compression also for `found`. The result are the following tentative rules.

```
findNode1?@ A ~> B \ find(A,X) <=> find(B,X), compr(A,X).
found1?   @ A ~> B \ found(A,X) <=> found(B,X), compr(A,X).
compress? @ root(C,_),found(C,X) \ A ~> B,compr(A,X) <=>A ~> C.
```

Compression is performed in parallel to the main part of the algorithm that performs the find and link operations. But since both linking and compression update the tree data structure, we may expect interferences. These are revealed by our confluence analysis. First of all, linking takes away `found`, so compression deadlocks after linking. We may compress to a root different from what has been used for linking. We may compress too early and thus too little: Consider parallel unions on different new nodes: all find and compress operations can immediately finish, because they are on roots. No compression is performed. But linking is sequentialized because roots change. `found` constraints handle these changes, but again no compression is performed.

The real problem is the *interference* between different compressions along the same sub-path, because roots change and because compression destroys paths. An old compress performed after a new one on the same node may "undo" the new compression, the overall result may be worse than without compression. Competing compressions may destroy the tree, even lead to cycles in the tree. These problems are well-known in the literature, different solutions have been proposed like comparing certain counters for nodes or time-stamps, or using a different compression technique like path halving [4]. We prefer a solution that does not introduce additional auxiliary operations or data and that is as close as possible to the original sequential optimal code. We do not want to be "too clever"[1]. This will also make it easier to verify the correctness of the implementation.

Insight #3 Our solution is to compress the nodes of a path to the root that was used for linking. So compression is performed *after* the corresponding linking operation. We do not think that this sequentiality is a disadvantage given the fact that before linking the roots of the tree may frequently change due to other link operations.

In the program `ufd_foundc_compr`, the rule `compress` now uses its own `found` named `foundc`. The program will leave the `foundc` constraints in the store. They represent the result of a find computation. If necessary, the `foundc` constraints can be *garbage collected*, when their second variable does not occur in any other constraint. Alternatively, their removal can be accommodated by keeping a counter on how often `foundc` will be used in path compression. When the counter is zero, the `foundc` constraint is removed.

ufd_foundc_compr

```
findNode1 @ A ~> B \ find(A,X) <=> find(B,X), compr(A,X).
findRoot1 @ root(A,_) \ find(A,X) <=> found(A,X).

found1    @ A ~> B \ found(A,X) <=> found(B,X), compr(A,X).

compress  @ foundc(C,X) \ A ~> B, compr(A,X) <=> A ~> C.
```

[1] Actually, we tried, investigated many variants, but confluence analysis usually revealed one of the problems mentioned here.

```
linkEq1c    @ found(A,X), found(A,Y), link(X,Y) <=>
                        foundc(A,X), foundc(A,Y).
linkLeft1c  @ found(A,X), found(B,Y), link(X,Y),
                        root(A,N), root(B,M) <=> N>=M |
                        foundc(A,X), foundc(B,Y),
                        B ~> A, N1 is max(N,M+1), root(A,N1).
linkRight1c @ found(A,X), found(B,Y), link(Y,X),
                        root(A,N), root(B,M) <=> N>=M |
                        foundc(A,X), foundc(B,Y),
                        B ~> A, N1 is max(N,M+1), root(A,N1).
```

8.1 Confluence of Parallelized Optimal Union-Find

The confluence analysis of this program finds several hundred non-joinable critical pairs, of the same nature as for the parallel basic version. The table shows that almost all critical pairs are between the rules for the link operations. 35 of them are joinable modulo equivalence of the nodes in the trees that are produced. All but one of the remaining critical pairs can be shown not to occur for allowed queries.

findNode1	findNode1	1
findNode1	findRoot1	1
findRoot1	findRoot1	1
found1	found1	1
found1	linkEq1c	2
found1	linkLeft1c	2
found1	linkRight1c	2
compress	findNode1	1
compress	found1	1
compress	compress	3

linkEq1c	linkEq1c	18
linkEq1c	linkLeft1c	39
linkEq1c	linkRight1c	39
linkLeft1c	linkLeft1c	191
linkLeft1c	linkRight1c	193
linkRight1c	linkRight1c	191

In the only non-trivial critical pair of the `compress` rule with itself, competing compressions may produce different trees, but the nodes are the same as the nodes `A, B, C` and `D` must be on the same path. In particular, trees are not destroyed and compression always improves the tree.

```
        foundc(C,X),compr(A,X), A~>B, foundc(D,Y),compr(A,Y)
    compress                                          compress
foundc(C,X),foundc(D,Y),A~>D          foundc(C,X),foundc(D,Y),A~>C
```

8.2 Correctness

We show that the new parallel CHR implementation and the optimal sequential implementation which was proven correct [16] *simulate each other* - to some

extent - by mapping computations between the two. We use the *refined operational semantics* of CHR for the sequential computations as in [13, 14, 16][2] and our proposed parallel semantics for CHR for the parallel computations.

We map states (constraints) and computation steps (rule applications). The *mapping* is inspired by the program transformation that we have performed to arrive at the parallel program: first, introduction of `found` that behaves like `find` until it is involved in a link (then it behaves like built-in equality), and second, replacement of implicit immediate path compression by an explicit one with `compr` that relies on `foundc` (which behaves like built-in equality) that is produced by linking.

In principle, the *rule applications* of parallel `ufd_foundc_compr`, i.e.
`make,union,findNode1,findRoot1,linkEq1c,linkLeft1c,linkRight1c`,
are mapped into the rule applications of sequential optimal `ufd_rank`, i.e.
`make,union,findNode, findRoot, linkEq, linkLeft, linkRight`,
and vice versa. (The rules `make` and `union` are identical in both programs, there is no need to further discuss them.)

Sequential to Parallel The mapping from a sequential to a parallel execution is as follows. Under the refined semantics, the sequential program for each union will do the two find operations and then the linking. A find operation is a sequence of `findNode` rule applications followed by a single application of `findRoot`. In the mapping, immediate path compression by rule `findNode` is replaced by explicit path compression with `compr` constraints. The built-in equality constraints produced by `findRoot` in the sequential computation are replaced by `found` constraints until they are involved in a link operation. From then on, the equalities are replaced by `foundc`. Immediately after linking, we have to insert applications of the `compress` rule into the resulting parallel computation, so that compression is actually performed (removing all `compr` constraints).

Parallel to Sequential Due to the interleaving semantics we have introduced for parallel CHR, any parallel computation can be described by a set of sequential computations involving the same rules and same result. Given such a sequentialized parallel computation, the following partial mapping will give us a computation of the sequential program.

By intended construction, not every execution of the parallel program can be mapped into one of the sequential program. Consider the critical pair for competing compressions (Subsect. 8.1): Since compression is immediate and implicit, only one of the computations can be simulated by the sequential program.

If we rule out these competing compressions (i.e. find or found constraints operating on the same nodes concurrently), parallel executions can be simulated by the sequential program: We map constraints `A~>B,compr(A,X)` into `A~>X` to achieve immediate compression. As a consequence, the `compress` rule applications become obsolete, because they do not change any constraints under the mapping.

We also map `found` into `find` constraints and thus applications of the rule `found1` into applications of `findNode`. Under the mapping, applications of the

[2] The correctness and optimality of the code was proven under the refined semantics.

rule `findRoot1` do not change constraints, they are therefore removed. Just before a link rule is applied, we insert two rule applications of the rule `findRoot` that apply to the two involved `find` constraints that come from mapping `found`. Finally, we map `foundc` constraints into built-in equalities. The result of the transformation is a correct computation of the sequential implementation of the optimal union-find algorithm in the standard semantics. Hence we claim that our parallel program for union-find is correct for computations without competing compressions.

9 Conclusion

In this exploratory paper, we introduced a parallel execution model for CHR. We parallelized basic and optimal sequential versions of the classical union-find algorithm with the help of confluence analysis and three insights. The resulting code is close to the original one and promises to be as efficient, even though it is acknowledged in the literature that this is hard to achieve due to the inherent sequential nature of the algorithm when it comes to tree updates.

The URL `www.informatik.uni-ulm.de/pm/mitarbeiter/fruehwirth/union-find` contains a list of programs, confluence checkers and results of confluence analysis used for this paper.

It was beyond the scope of the paper to give a time complexity analysis, but let us speculate shortly on the topic. We showed that each rule application in one program corresponds to a rule application in the other program, with exception of the `compress` rule applications that only occur in the parallel program. But their number is bounded by the number of `findNode` rule applications. Hence if rule applications cost the same in sequential and parallel CHR, the optimal worst-case time complexity is preserved. Since find operations can run in parallel (but not linking), we can expect a reduction in latency for simultaneous queries and updates.

The preliminary, exemplary findings in this paper can just be the starting point for a number of challenging research topics:

- parallel union-find correctness and time complexity analysis,
- parallel model for CHR, its implementation and empirical evaluation,
- more practical confluence analysis, including automatic detection of critical pairs that cannot occur for allowed queries,
- development of a confluence-based parallelization methodology and its application to other CHR programs, in particular constraint solvers.

As pointed out by a reviewer, our confluence-based parallelization could also be used to convert a program using the refined semantics to a program using the standard semantics (where parallelization is straightforward).

Acknowledgements We would like to thank Marc Meister and Tom Schrijvers for helpful discussions. We also thank the referees that provided us with detailed comments.

References

1. S. Abdennadher. Operational Semantics and Confluence of Constraint Propagation Rules. In *Third International Conference on Principles and Practice of Constraint Programming, CP97*, LNCS 1330. Springer, 1997.
2. S. Abdennadher, T. Frühwirth, and C. H. (Eds.). *Special Issue on Constraint Handling Rules, Journal of Theory and Practice of Logic Programming (TPLP)*. Cambridge University Press, to appear 2005.
3. S. Abdennadher, T. Frühwirth, and H. Meuss. Confluence and Semantics of Constraint Simplification Rules. *Constraints Journal*, 4(2), 1999.
4. R. J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *STOC '91: Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 370–380. ACM Press, Revision of November 1994.
5. M. J. Atallah, M. T. Goodrich, and S. R. Kosaraju. Parallel algorithms for evaluating sequences of set-manipulation operations. *J. ACM*, 41(6):1049–1088, 1994.
6. G. J. Duck, P. J. Stuckey, M. G. de la Banda, and C. Holzbaur. The Refined Operational Semantics of Constraint Handling Rules. In B. Demoen and V. Lifschitz, editors, *Proceedings of the 20th International Conference on Logic Programming*, 2004.
7. T. Frühwirth. Theory and Practice of Constraint Handling Rules, Special Issue on Constraint Logic Programming. *Journal of Logic Programming*, pages 95–138, October 1998.
8. T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Springer, 2003.
9. Z. Galil and G. F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Comp. Surveys*, 23(3):319ff, 1991.
10. M. C. Pinotti, V. A. Crupi, and S. K. Das. A parallel solution to the extended set union problem with unlimited backtracking. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 182–186, Washington, DC, USA, 1996. IEEE Computer Society.
11. V. A. Saraswat and M. Rinard. Concurrent constraint programming. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245, New York, NY, USA, 1990. ACM Press.
12. V. A. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–352, New York, NY, USA, 1991. ACM Press.
13. T. Schrijvers and T. Frühwirth. Union-Find in CHR. Technical Report CW389, Department of Computer Science, K.U.Leuven, Belgium, July 2004.
14. T. Schrijvers and T. Frühwirth. Analysing the CHR Implementation of Union-Find. In *19th Workshop on (Constraint) Logic Programming (W(C)LP 2005)*. Ulmer Informatik-Berichte 2005-01, University of Ulm, Germany, February 2005.
15. T. Schrijvers and T. Frühwirth. CHR Website, `www.cs.kuleuven.ac.be/~dtai/projects/CHR/`, May 2005.
16. T. Schrijvers and T. Frühwirth. Optimal Union-Find in Constraint Handling Rules, Programming Pearl. *Journal of Theory and Practice of Logic Programming (TPLP)*, to appear.
17. R. E. Tarjan and J. van Leeuwen. Worst-case Analysis of Set Union Algorithms. *J. ACM*, 31(2):245–281, 1984.