Parallel Execution of Multi-Set Constraint Rewrite Rules

Martin Sulzmann
Programming, Logics and Semantics Group,
IT University of Copenhagen
Rued Langgaards Vej 7, 2300 Copenhagen S Denmark
martin.sulzmann@gmail.com

Edmund S. L. Lam
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
lamsoonl@comp.nus.edu.sg

Abstract

Multi-set constraint rewriting allows for a highly parallel computational model and has been used in a multitude of application domains such as constraint solving, agent specification etc. Rewriting steps can be applied simultaneously as long as they do not interfere with each other. We wish that the underlying constraint rewrite implementation executes rewrite steps in parallel on increasingly popular becoming multi-core architectures. We design and implement efficient algorithms which allow for the parallel execution of multi-set constraint rewrite rules. Our experiments show that we obtain some significant speed-ups on multi-core architectures.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

General Terms Languages, Performance

1. Introduction

Rewriting is a powerful discipline to specify the semantics of programming languages and to perform automated deduction. There are numerous flavors of rewriting such as term, graph rewriting etc. We consider here exhaustive, forward chaining, multi-set constraint rewriting as found in Constraint Handling Rules (CHR) [5]. Rewriting steps are specified via CHR rules which replace a multi-set of constraints matching the left-hand side of a rule (also known as rule head) by the rule's right-hand side (also known as rule body). Production rule systems are related to CHR. For concreteness, we base our investigations here on CHR, mainly because CHR have a well-established formal basis [5] and are used in a multitude of applications such as general purpose constraint programming, type system design, agent specification and planning etc [7].

CHR support a very fine-grained form of parallelism. CHR rules can be applied in parallel if the rewriting steps they imply do not interfere with each other. An interesting feature of CHR is that the left-hand side of a CHR rule can have a mix of simplified and propagated constraint patterns. This provides the opportunity for further parallelism. We can execute CHR rules in parallel as long as only their propagated parts overlap. We wish that the implementation of our rewrite language takes advantage of additional processor power (more cores) on increasingly popular and more accessible becoming shared memory, multi-core architectures.

PPDP'08, July 16–18, 2008, Valencia, Spain.

In the CHR context, there is practically no prior work which addresses this important issue. The main focus of previous research is on the efficient execution of deterministic variants of the CHR semantics on single-threaded computing architectures. For example, see [3, 18, 10] and the references therein. In contrast, our starting point is the abstract CHR semantics [5] which is highly indeterministic and thus naturally supports a high degree of parallelism.

In this paper, we address the issue of parallel execution of CHR. Our contributions are:

- We design and implement algorithms which allow for the parallel execution of CHR (Sections 3 and 4):
  - One of the key aspects of any parallel implementation is the protection of critical regions. In our case, we need to protect the atomic rewriting of the left-hand side of a CHR rule by its right-hand side. We report our experiences in using different implementation methods such as traditional locks versus the more recent concept of Software Transactional Memory (Section 3.3).
  - Any rule can fire in parallel as long as there are no interferences with another rule firing. However, we cannot directly exploit this form of parallelism because system resources (processor cores) are limited and naively firing rules leads to conflicts which may even result in an increase in the running time of the program. Therefore, we investigate more systematic execution methods which achieve desired speed-ups (Section 3.2).
- Our experimental results show that our parallel implementation of CHR provides significant speed-ups on shared memory, multi-core architectures (Section 5).

We postpone a discussion of related work until Section 6. Section 7 concludes. A preliminary version of this paper was presented at [15]. Since then we have significantly revised the paper and improved the implementation. Our implementation can be downloaded via `http://code.google.com/p/parallel-chr/`.

2. Constraint Handling Rules

We review the basics of CHR rule systems. First, we consider a few CHR examples programs. Then, we define the CHR semantics and discuss possible approaches on how to parallelize CHR.

2.1 Examples

Figure 1 contains several examples of CHR rules and derivations. We adopt the convention that lower-case symbols refer to variables and upper-case symbols refer to constraints. The notation $rulename@$ assigns distinct labels to CHR rules.

The first example simulates a simple communication channel. The $Get(x)$ constraint represents the action of writing a value from the communication channel into the variable x, while the $Put(y)$ constraint represents the action of putting the value y into the

Communication channel:

$$get \ @ \ Get(x), Put(y) \iff x = y$$

$$\frac{\{Get(m), Put(1)\} \rightarrowtail_{get} \{m = 1\} \parallel \{Get(n), Put(8)\} \rightarrowtail_{get} \{n = 8\}}{\{Get(m), Put(1), Get(n), Put(8)\} \rightarrowtail^* \{m = 1, n = 8\}}$$

Greatest common divisor:

$$gcd1 \ @ \ Gcd(0) \iff True$$
$$gcd2 \ @ \ Gcd(n) \backslash Gcd(m) \iff m >= n \&\& n > 0 \mid Gcd(m - n)$$

$$\frac{\begin{array}{c}\{Gcd(3), Gcd(9)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(6)\} \parallel \{Gcd(3), Gcd(3)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(0)\} \\ \hline \{Gcd(3), Gcd(3), Gcd(9)\} \quad \rightarrowtail_{gcd2,gcd2} \quad \{Gcd(3), Gcd(0), Gcd(6)\} \\ \qquad\qquad\qquad\qquad\quad \rightarrowtail^* \qquad\quad \{Gcd(0)\}\end{array}}{\{Gcd(3), Gcd(3), Gcd(9)\} \rightarrowtail^* \{Gcd(0)\}}$$

Merge sort:

$$merge1 \ @ \ Leq(x, a) \backslash Leq(x, b) \iff a < b \mid Leq(a, b)$$
$$merge2 \ @ \ Merge(n, a), Merge(n, b) \iff a < b \mid Leq(a, b), Merge(n + 1, a)$$

Shorthands: $L = Leq$ and $M = Merge$

$$\begin{array}{lll} & M(1,a), M(1,c), M(1,e), M(1,g) & \\ \rightarrowtail_{merge2} & M(2,a), M(1,c), M(1,e), L(a,g) & \\ \rightarrowtail_{merge2} & M(2,a), M(2,c), L(a,g), L(c,e) & \parallel \quad M(1,b), M(1,d), M(1,f), M(1,h) \\ \rightarrowtail_{merge2} & M(3,a), L(a,g), L(c,e), L(a,c) & \parallel \quad \rightarrowtail^* \quad M(3,b), L(b,d), L(d,f), L(f,h) \\ \rightarrowtail_{merge1} & M(3,a), L(a,c), L(c,g), L(c,e) & \\ \rightarrowtail_{merge1} & M(3,a), L(a,c), L(c,e), L(e,g) & \end{array}$$

$$\begin{array}{lll} & M(3,a), L(a,c), L(c,e), L(e,g), M(3,b), L(b,d), L(d,f), L(f,h) \\ \rightarrowtail_{merge2} & M(4,a), L(a,c), L(a,b), L(c,e), L(e,g), L(b,d), L(d,f), L(f,h) \\ \rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,e), L(e,g), L(b,d), L(d,f), L(f,h) \\ \rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(c,e), L(e,g), L(d,f), L(f,h) \\ \rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,g), L(d,f), L(f,h) \\ \rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,f), L(e,g), L(f,h) \\ \rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,f), L(f,g), L(f,h) \\ \rightarrowtail_{merge1} & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,f), L(f,g), L(g,h) \end{array}$$

$$\begin{array}{lll} & M(1,a), M(1,c), M(1,e), M(1,g), M(1,b), M(1,d), M(1,f), M(1,h) \\ \rightarrowtail^* & M(4,a), L(a,b), L(b,c), L(c,d), L(d,e), L(e,f), L(f,g), L(g,h) \end{array}$$

Figure 1. CHR examples

channel. The interaction between both constraints is specified via the CHR rule get which specifies the replacement of constraints matching $Get(x)$ and $Put(y)$ by $x = y$. The point to note is that in contrast to Prolog, we use matching and not unification to trigger rules.

For example, the constraint store $\{Get(m), Put(1)\}$ matches the left-hand side of the get rule by instantiating x by m and y by 1. Hence, $\{Get(m), Put(1)\}$ rewrites to the answer $\{m = 1\}$. We write $\{Get(m), Put(1)\} \rightarrowtail_{get} \{m = 1\}$ to denote this derivation step. Similarly, we find that $\{Get(n), Put(8)\} \rightarrowtail_{get} \{n = 8\}$. Rules can be applied concurrently as long as they do not interfere. In our case, the two derivations above can be executed concurrently, indicated by the symbol $\parallel$, and we can straightforwardly combine both derivations which leads to the final answer $\{m = 1, n = 8\}$. We write $\rightarrowtail^*$ to denote exhaustive rule application.

The answer $\{m = 8, n = 1\}$ is also possible but the CHR rewrite semantics is committed-choice. We can guarantee a unique answer if the CHR rules are confluent which means that rewritings applicable on overlapping constraint sets are always joinable. In general, (non)confluence is not of a concern to us here and is left to the programmer (if desired). We follow here the abstract CHR semantics [5] which is inherently indeterministic. Rewrite rules can be applied in any order and thus enables us with a high degree of parallelism.

The second CHR example computes the greatest common divisor among a set of numbers by applying Euclid's algorithm. The left-hand side of rule $gcd2$ is interesting because it uses a mix of simplified and propagated constraint patterns. We replace (simplify) $Gcd(m)$ by $Gcd(m - n)$ but keep (propagate) $Gcd(n)$ if the guard $m >= n \&\& n > 0$ holds. For example, we find that

$$\{Gcd(3), Gcd(9)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(6)\}$$

and $\{Gcd(3), Gcd(3)\} \rightarrowtail_{gcd2} \{Gcd(3), Gcd(0)\}$. The point to note is the above rule applications only overlap on the propagated part. Hence, we can execute both rewrite derivations simultane-

Notation	$\uplus$		Multi-set union
	$\models$		Model-theoretic entailment
	ϕ		Substitution
Function symbols	f	::=	$0 \mid 1 \mid + \mid > \mid >= \mid \&\& \mid ...$
Terms	t	::=	$x \mid f\ t...t$
Guard	t_g	::=	t
Predicate symbols	$Pred$	::=	$Gcd \mid Leq \mid ...$
Constraint	C	::=	$Pred(t,..,t) \mid t = t$
	S, P	::=	$Pred(t,..,t)$
Store	St	::=	$\{C\} \mid St \uplus St$
	St_e	::=	$\{t = t\} \mid St_e \uplus St_e$
	St_c	::=	$\{Pred(t,...,t)\} \mid St_c \uplus St_c$
Rule	R	::=	$r \ @\ P_1,...,P_l \backslash S_1,...,S_m \Longleftrightarrow t_g \mid C_1,...,C_n$
Rule system	$\mathcal{P}$	::=	$\{R_1,...,R_k\}$

$$\text{(Rewrite)}\quad \frac{\begin{array}{c} r \ @\ P_1,...,P_l \backslash S_1,...,S_m \Longleftrightarrow t_g \mid C_1,...,C_n \in \mathcal{P} \\ St = St_e \uplus St_c \uplus \{\phi(P_1),...,\phi(P_l),\phi(S_1),...,\phi(S_m)\} \quad St_e \models \phi(t_g) \\ St' = St_e \uplus St_c \uplus \{\phi(P_1),...,\phi(P_l),\phi(C_1),...,\phi(C_n)\} \end{array}}{St \rightarrowtail_r St'}$$

$$\text{(Solve)}\quad \frac{\begin{array}{c} St = St_e \uplus St_c \quad \phi \text{ m.g.u. of } St_e \\ St' = St_e \uplus \phi(St_c) \end{array}}{St \rightarrowtail_{Solve} St'} \qquad \frac{St \rightarrowtail_{Solve} St'}{St \rightarrowtail^* St'} \qquad \frac{St \rightarrowtail_r St'}{St \rightarrowtail^* St'} \qquad \frac{St \rightarrowtail^* St' \quad St' \rightarrowtail^* St''}{St \rightarrowtail^* St''}$$

Figure 2. CHR abstract rewrite semantics

ously

$$\{Gcd(3), Gcd(3)Gcd(9)\} \rightarrowtail_{2\times gcd2} \{Gcd(3), Gcd(0), Gcd(6)\}$$

The last example is a CHR encoding of the well-known merge sort algorithm. To sort a sequence of (distinct) elements $e_1, ..., e_m$ where m is a power of 2, we apply the rules to the initial constraint store

$$Merge(1, e_1), ..., Merge(1, e_m)$$

Constraint $Merge(n, e)$ refers to a sorted sequence of numbers at level n whose smallest element is e. Constraint $Leq(a, b)$ denotes that a is less than b. Rule $merge2$ initiates the merging of two sorted lists and creates a new sorted list at the next level. The actual merging is performed by rule $merge1$. Sorting of sublists belonging to different mergers can be performed simultaneously. See the example derivation in Figure 1 where we simultaneously sort the characters a, c, e, g and b, d, f, h.

2.2 CHR Semantics

In Figure 2, we review the essentials of the abstract CHR semantics [5]. The general form of CHR rules contains propagated components P_i and simplified components S_j as well as a guard t_g

$$r \ @\ P_1, ..., P_l \backslash S_1, ..., S_m \Longleftrightarrow t_g \mid C_1, ..., C_n$$

In CHR terminology, a rule with simplified components only ($l = 0$) is referred to as a *simplification* rule, a rule with propagated components only ($m = 0$) is referred to as a *propagation* rule. The general form is referred to as a *simpagation* rule.

CHR rules manipulate a global constraint store which is a multi-set of constraints. We execute CHRs by exhaustive rewriting of constraints in the store with respect to the given rule system (a finite set of CHR rules).

Rule (Rewrite) describes application of a CHR rule r at some instance ϕ. We simply (remove from the store) the matching copies of $\phi(S_j)$ and propagate (keep in the store) the matching copies of $\phi(P_i)$. But this only happens if the instantiated guard $\phi(t_g)$ is entailed by the equations present in the store, written $St_e \models \phi(t_g)$. In case of a propagation rule we need to avoid infinite re-propagation. We refer to [1] for details. In (Solve), we normalize the store by building the most general unifier. The remaining rewrite rules specify the exhaustive application of CHR rules.

The examples from the previous section show that if there is no interference, we can apply CHR rules simultaneously. However, the formal description of the CHR semantics in Figure 2 tells us little how to derive an efficient parallel CHR implementation. Hence, our next task is to explore possible ways how to parallelize CHR.

2.3 Parallel CHR Execution Models

There are two natural approaches towards parallelizing CHR execution: Rule-based [9] and goal-based CHR execution [3]. In the first approach, each CHR rule attempts to rewrite matching left-hand sides by the rule's right-hand side. The second approach views the store as a multi-set of goals (active constraints). Each goal seeks some missing partners to build a complete match with the left-hand side of a CHR rule.

We choose a goal-based execution scheme which in our opinion, matches fairly well our hardware assumption. We assume that we have available a shared-memory, multi-core architecture with a reasonable number of cores, say 4-64 cores. [1] The goal-based execution model supports a natural and adaptive parallel execution strategy, where given n cores we can execute n active goals (if available) in parallel. Parallel executions are guaranteed to be non-redundant, as they originate from unique runtime goal constraints. Hence, the goal-based approach scales well with the number of available cores. We will back-up this claim via some experimental results in Section 5.

Another issue is the computation of matches, i.e. constraints matching the left-hand side of a rule. There are again two possible approaches: Eager [4] and lazy matching [3]. The first approach finds

[1] At the moment, 2 cores are standard and 4 cores become more frequent. The prediction is that within the next few years 8 and even 16 cores become standard.

```
1  goal_based_thread:
2      while ∃ goal
3         select goal G
4         if ∃ r@ P_1,...,P_l \ S_1,...,S_m ⟺ t_g | C_1,...,C_n ∈ P and
5            ∃ φ such that
6            St ≡ St_c ⊎ {φ(P_1),...,φ(P_l),φ(S_1),...,φ(S_m)} and
7            ⊨ φ(t_g) and
8            either (G ≡ φ(P_i) for some i ∈ {1,...,l}) or
9                   (G ≡ φ(S_j) for some j ∈ {1,...,m})
10        then let ψ be m.g.u. of all equations in C_1,...,C_n
11             St := St_c ⊎ { φ(P_1),...,φ(P_l), φ∘ψ(C_1),...,φ∘ψ(C_n)}
```

Table 1. Goal-based lazy match rewrite algorithm for ground CHR

all matches. This is only sensible if we consider propagation rules (i.e. rules which will not remove constraints from the store). The reason is that matches for propagation rules will not conflict with each other. However, in case of simplification rules we may encounter conflicts among matches. That is, constraints may take part in several matches. Hence, only one of those matches can be applied. Lazy matching on the other hand computes only one match at a time. There is still the possibility that a parallel thread computes an overlapping match (hence a conflict arises). But by only computing one match at a time the likelihood of conflicts is clearly reduced.

To summarize, for the problem setting we consider, CHR systems containing rules with simplified components, and our hardware assumptions, 4-64 core machines, goal-based lazy match CHR execution is our preferred choice. Table 1 lays out the basic structure of a goal-based lazy match rewrite algorithm. We select a goal G which then finds its matching partners. Lines 8 and 9 ensure that the goal must be part of the left-hand side. On each core we have a thread running `goal_based_thread` where the store St is shared among all threads. Our formulation assumes that the CHR rule system is *ground*. That is, equations on right-hand side of rules can immediately be eliminated by applying the m.g.u. This ensures that any derivation starting from a ground constraint store (none of the constraints contains any free variables) can only lead to another ground constraint store. In our experience, the restriction to ground CHR is not onerous because most examples either satisfy this condition, or it is fairly straightforward to program unification/instantiation on top of CHR (e.g. see our encoding of union-find in the upcoming Section 5).

The algorithm in Table 1 gives a high-level specification of a goal-based lazy match rewrite algorithm. Interestingly, such a rewrite algorithm (or variants of it) is employed in all major CHR implementations [10, 14]. However, these works consider a single-threaded setting. Hence, our task is to design and implement an efficient parallel implementation. This is what we discuss in the following two sections.

3. Goal-Based Lazy Match CHR Parallelism

The main task of each `goal_based_thread` is to seek matching partner constraints in the store for a given goal. For n cores we have n threads which execute in parallel. We need to address the following implementation details.

- Shared store representation and constraint lookup.
- Goal selection, execution, ordering and storage.
- Finding matches in parallel and atomic rule execution.

Each of the implementation details comes with a design space. We will back-up our decisions with experimental data which we report in Section 5. Details regarding the actual implementation are given in Section 4.

3.1 Shared Store Representation and Constraint Lookup

The kind of store representation affects the number of conflicts among threads when searching for partner constraints. For example, suppose we use naively (linked) lists as a store representation.

$get \ @ \ Get(x), Put(y) \iff x = y$

Shared Store: $[Get(x), Get(y), Put(2), Put(8)]$

Thread 1	`Goal:` $Get(x)$	`Found partner constraint:` $Put(2)$
Thread 2	`Goal:` $Get(y)$	`Found partner constraint:` $Put(2)$

Both threads scan the shared store (a list) from left to right and therefore find the same partner constraint $Put(2)$. Because of the overlap among the partner, only one of the threads can execute the get rule whereas the other thread needs to continue scanning for a partner. To avoid conflicts, we wish that threads try matching partner constraints in their unique ordering, rather than all trying the same ordering.

We achieve this by viewing the store as a collection of n linked lists where n is the number of threads. Each list can be accessed by any of the threads. To reduce the likelihood of conflicts, each thread uses a unique ordering in which the linked lists are looked up for partners, or in which new constraints are appended to. For example, if the store constraints $Put(2)$ and $Put(8)$ appear in different (sub)lists then thread 1 and thread 2 can execute rule get in parallel.

To guarantee that $Put(2)$ and $Put(8)$ actually end up in different (sub)lists we need to ensure that $Put(2)$ and $Put(8)$ will be selected and executed by two different threads. How threads select goals will be discussed next.

We employ standard optimization methods [10, 18] such as **constraint indexing** to lookup constraints more efficiently. Suppose the goal $Leq(x, b)$ seeks the partner $Leq(x, a)$. By storing Leq constraints in hash-tables indexed by their first arguments, the task of looking up the matching partners $Leq(x, b)$ is a simple constant time operation rather than an iterative search through the store.

3.2 Goal Selection, Ordering, Execution and Storage

Each thread selects a goal which drives the execution process by finding partner constraints in the store which match the left-hand side of a CHR rule.

Goal selection, execution and ordering. Each store constraint is a potential goal. However, the number of such constraints will be in general larger than the number of threads. Hence, we need a method which systematically selects active goals, i.e. goals which are executed by threads. The constraints on the right-hand side of a CHR rule are all potential goal constraints. Hence, we could simply distribute these constraints (after the rule has fired) among the n

threads. However, the order in which we select and execute these constraints can have an impact on the performance.

For example, recall the merge sort example.

$$merge1 \ @ \ Leq(x,a) \backslash Leq(x,b) \iff a < b \mid Leq(a,b)$$
$$merge2 \ @ \ Merge(n,a), Merge(n,b) \iff a < b \mid Leq(a,b), Merge(n+1,a)$$

Merging of sorted (sub)lists via rule $merge1$ is clearly a sequential operation. Hence, there is no point in having one thread selecting $Leq(a,b)$ and another thread selecting $Leq(a,c)$. This only leads to unnecessary conflicts. To avoid such situations, we select and execute Leq goals before selecting and executing $Merge$ goals, thus, eagerly performing the merge of sorted sequences. In general, the optimal selection and execution ordering will depend on the specific problem and will require program annotations to guide the selection and execution order of constraints.

Our current implementation supports annotations to distinguish between stacked and queued goals. By specifying $Head \iff C_1 \backslash C_2$ the programmer indicates to put C_1 at the front of the list of goals and to put C_2 at the end of the list of goals. Thus, we can support depth-first and breadth-first goal execution. To support more complex priorities we plan to support shared priority queues for goals and use bags instead of lists to avoid the likelihood of conflicts (like in the case of the store). We first exhaustively process goals from the highest priority bag before moving on to next lower priority bag.

Another idea is to provide annotations to specify the execution order among constraints in the same priority class. For example, by specifying $Head \iff C_1 \parallel C_2$ the programmer indicates to execute the goal constraints C_1 and C_2 in parallel. That is, have one thread processing C_1 and another thread processing C_2. Via $Head \iff C_1; C_2$ the programmer indicates to first process C_1 before processing C_2. Our current implementation does not yet support the above execution control annotations. But it reasonably straightforward to integrate such features in a future version.

Goal storage. Besides the execution order among goals, there is also the issue when we actually store goals. That is, when executing the constraints on the right-hand side of rules shall we add these constraints eagerly into the store? The issue is that storing of constraints is costly for several reasons. We occupy more memory and building of constraint indices takes time. In addition, more constraints in the store means that the operation to find matches takes longer.

Hence, a possible strategy is to store goals lazily, i.e. only when they are actually executed. In fact, we must store goals before they are executed. Otherwise, we risk that CHR rules are not applied exhaustively. For example, suppose we have the rule

$$r1 \ @ \ A, B \iff rhs$$

and two threads, one executing goal A and the other executing goal B. Both constraints are not available in the store yet. Goal constraints seek matching partners in the store only. Hence, both threads fail to fire rule $r1$. On the other hand, if we store A and B before they become active then one of the threads will be able to fire rule $r1$.

There are cases where eager storage has an advantage over lazy storage. We explain this point via the following contrived example. There are two rules

$$r2 \ @ \ A(x), B(x) \iff rhs$$
$$r3 \ @ \ C(0) \iff D(1), A(1), ..., D(n), A(n)$$

and the initial store is $\{B(1), ..., B(n), C(0)\}$. We assume there are $n+1$ threads, n threads executing goals $B(1)$,...,$B(n)$ and one thread executing $C(0)$. To improve performance A constraints are stored eagerly after firing of rule $r3$, thus allowing threads 1,...,n to fire rule $r2$ in parallel. There is no need to store D constraints immediately. Hence, we use lazy storage for D constraints. Like in the case of goal priorities, the choice among eager or lazy storage depends on the problem setting. By default, we apply lazy storage and stack goals.

Dropping goals. If a goal cannot fire a rule we will drop the goal by deleting it from the list of goals. Because we assume ground CHR, we do not have to deal with reactivation of dropped goals. The situation is different for non-ground CHR where firing of a rule may further instantiate a constraint in the store. The instantiated constraint can fire a rule which was earlier impossible. For example, consider the rule $r@ \ A(1) \iff rhs$. We drop the goal $A(x)$ because the left-hand side cannot be matched. Suppose that the firing of some other rule instantiates $A(x)$ to $A(1)$ which means that the rule can fire now. We therefore need to reactivate $A(1)$ by re-adding this constraint to the set of goals. In our setting of ground CHR, reactivation is not necessary because once a goal is deactivated it will not be further instantiated.

3.3 Parallel Multi-Set Constraint Matching and Execution

We take a look at finding matches in parallel and atomic rule execution. The challenge we face in a parallel setting is that the partners found by several threads may overlap. This is harmless in case they only overlap in propagated components. However, we need to protect partner constraints which will be simplified.

Lock-based parallel matching is the standard approach where we incrementally lock partner constraints. However, we must be careful to avoid deadlocks. For example, suppose that thread 1 and 2 seek partners A and B to fire any of the rules $A, B, C \iff rhs_1$ and $A, B, D \iff rhs_2$. We assume that C is thread 1's goal constraint and D is the goal constraint of thread 2. Suppose that thread 1 first encounters A and locks this constraint. By chance, thread 2 finds B and imposes his lock on B. But then none of the two threads can proceed because thread 1 waits for thread 2 to release the lock imposed on B and thread 2 waits for the release of the locked constraint A.

This is a classic (deadlock) problem when programming with locks. The recently popular becoming concept of Software Transactional Memory (STM) is meant to avoid such issues. Instead of using locks directly, the programmer declares that certain program regions are executed atomically. The idea is that atomic program regions are executed optimistically. That is, any read/write operations performed by the program are recorded locally and will only be made visible at the end of the program. Before making the changes visible, the underlying STM protocol will check for read/write conflicts with other atomically executed program regions. If there are conflicts, the STM protocol will then (usually randomly) commit one of the atomic regions and rollback the other conflicting regions. Committing means that the programs updates become globally visible. Rollback means that we restart the program. The upshot is that the optimistic form of program execution by STM avoids the typical form of deadlocks caused by locks. In our setting, we can protect critical regions via STM as follows.

STM-based parallel matching means that we perform the search for partner constraints and their removal from the store atomically. For the above example, where both threads attempt to remove constraints A and B as well as their specific goal constraints we find that only one of the threads will commit whereas the other has to rollback, i.e. restart the search for partners.

The downside of STM is that unnecessary rollbacks can happen due to the conservative conflict resolution strategy. Here is an example

Abstract Data Types

Integer Value:	`Int`	Boolean Value:	`Bool`	List of a's:	`[a]`	Substitution:	`Subst`
CHR Constraint:	`Cons`	Rule Guard:	`Guard`	CHR Store:	`Store`	CHR Goals:	`Goals`

Rule Occurrence Data Types

```
Head Type:        data Head = Simp | Prop
Match Task:       data MatchTask = LpHead Head Cons | SchdGrd Guard
Rule Occurrence:  type Occ = ([MatchTask],[Cons])
CHR Program:      type Prog = [Occ]
```

CHR Solver Sub-routines

- `isAlive :: Cons -> Bool`
  Given CHR constraint `c`, returns true if and only if `c` is still stored.
- `match :: Subst -> Cons -> Cons -> IO (Maybe Subst)`
  Given a substitution and two CHR constraints `c` and `c'`, returns resultant substitution of matching `c` with `c'`, if they match. Otherwise return nothing.
- `consApply :: Subst -> [Cons] -> [Cons]`
  Given a substitution and a list of CHR constraints, apply the substitution on each constraint of the list and return the results.
- `grdApply :: Subst -> Guard -> Bool`
  Given a substitution and a guard condition, apply the substitution on the guard and return true iff guard condition is satisfiable.
- `emptySub :: Subst`
  Returns the empty substitution.
- `addToStore :: Store -> Cons -> IO ()`
  Given a CHR store `st` and a CHR constraint `c`, add `c` into `st`.
- `getCandidates :: Store -> Cons -> IO [Cons]`
  Given a CHR Store `st` and a CHR constraint `c`, return a list of constraints from the `st` that matches `c`.
- `getGoal :: Goals -> IO (Maybe Cons)`
  Given CHR goals, returns the next goal if one exists, otherwise returns nothing.
- `addGoals :: Goals -> [Cons] -> IO ()`
  A Given CHR goals `gs` and a list of CHR constraints `cs`, add `cs` into `gs`.
- `notRepeat :: [(Head,Cons)] -> Cons -> Bool`
  Given a list of matching heads, and a constraint **c** returns true if **c** is not already found in the list of heads.
- `verifyAndCommit :: Store -> [Head] -> STM Bool`
  Given the CHR store and a list of matching constraints `hds`, do the following atomically: verify that all constraints in `hds` are still in store and delete matching simplification heads in `hds`. Returns true if and only if this is successfully executed.
- `atomically :: STM a -> IO a`
  Given a `STM` operation, execute it atomically in the `IO` monad.

Figure 3. Interfaces of CHR data types

to explain this point. Suppose that thread 1 seeks partner A and thread 2 seeks partner B. There is clearly no conflict. However, during the search for A, thread 1 reads B as well. This can happen in case we perform a linear search and no constraint indexing is possible or the hash-table has many conflicts. Suppose that thread 2 commits first and removes B from the store. The problem is that thread 1 is forced to rollback because there is a read/write conflict. The read constraint B is not present anymore. STM does not know that this constraint is irrelevant for thread 1 and therefore conservatively forces thread 1 to rollback.

We have experimented with a pure STM-based implementation of atomic search for partners and rule execution. The implementation is elegant but unnecessary rollbacks happen frequently which in our experience results in some severe performance penalties. We provide concrete evidence in the upcoming Section 5. In our current implementation, we use a **hybrid STM-based** scheme which partially abandons STM. The search for matching partner constraints is performed "outside" STM (to avoid unnecessary rollbacks). Once a complete match is found, we perform an atomic re-verification step before we actually execute the rule. In our experience, this approach scales well.

4. Implementation

We have fully implemented the system as described so far. In our implementation, we use the Glasgow Haskell Compiler [8] because of its good support for shared memory, multi-core architectures. We also found Haskell's high-level abstraction facilities (polymorphic types higher-order functions etc) and its clean separation between pure and impure computations invaluable in the development of our system. In principle, our system can of course be re-implemented in other languages such as C and Java.

We briefly discuss our parallel implementation of the goal-based lazy match rewrite algorithm. See Table 2. Figure 3 highlights the data types and sub-routines of the implementation. **Abstract Data Type** shows the Haskell data type representation of CHR language elements, like constraints, substitution, store etc. For brevity, we

```
1  goal_based_thread :: Goals -> Store -> Prog -> IO ()
2  goal_based_thread gs st prog =
3     rewrite_loop
4     where
5       rewrite_loop = do
6         { mb <- getGoal gs
7         ; case mb of
8            Just g -> do { a <- addToStore st g
9                         ; execute_goal a prog
10                        ; rewrite_loop }
11           Nothing -> return () }
12      execute_goal a (occ:occs) = do
13        { match_goal gs st a occ
14        ; if isAlive a then execute_goal a occs
15                       else return () }
16      execute_goal _ [] = return ()
17
18 match_goal :: Goals -> Store -> Cons -> Occ -> IO ()
19 match_goal gs st g (mtasks,body) = do
20  { let (LpHead hd c):rest = mtasks
21  ; mb <- match emptySub c g
22  ; case mb of
23     Just sub -> exec_match [(hd,g)] sub rest
24     Nothing  -> return () }
25  where
26   exec_match hds sub ((SchdGrd grd):mts) =
27    if grdApply sub grd then exec_match hds sub mts
28                        else return ()
29
30   exec_match hds sub ((LpHead hd c):mts) =
31    let exec_match_candidates (nc:ncs) =
32         if (notRepeat hds nc) && (isAlive a)
33         then do
34          { mb <- match sub c nc
35          ; case mb of
36             Just sub' -> do
37              { exec_match ((h,nc):hds) sub' mts
38              ; exec_match_candidates ncs }
39             Nothing   -> exec_match_candidates ncs }
40         else exec_match_candidates ncs
41       exec_match_candidates [ ] = return ()
42    in do { cans <- getCandidates st c
43          ; exec_match_candidate cans }
44
45   exec_match hds sub [ ] = do
46    { b <- atomically (verifyAndCommit st hds)
47    ; if b then let body' = consApply sub body
48                in addGoals goals body'
49           else return () }
```

Table 2. Top Level CHR Rewriting Routine

leave the details of the shared data structures to hold goals and store constraints abstract. **Rule Occurrence Data Types** represent the internal compilation of CHR rules and we give the type signatures and brief descriptions of some basic **CHR Solver Sub-routines**.

We use a CHR compilation scheme which is comparable with those used in existing CHR systems [10]. Each CHR rule is compiled into a set of CHR rule occurrences (one for each of it's head constraints), essentially consisting of a list of match tasks (rule heads and guard) and a list of constraints (rule body). Match tasks describe the matching sequence to complete the rule head match, which can be either checking of guards (`SchdGrd`) or looking up of rule heads (`LpHead`). For example, the following CHR rule:

$$r@A(x)\backslash B(x,y) \Longleftrightarrow x > y \mid C(x,y)$$

is compiled into the two rule occurrences:

```
occR = [(mt1,body),(mt2,body)]
 where
   mt1 = [LpHead Prop A(x), LpHead Simp B(x,y),
          SchdGrd (x > y)]
   mt2 = [LpHead Simp B(x,y),LpHead Prop A(x),
          SchdGrd (x > y)]
   body = [C(x,y)]
```

We omit the details of how we deal with variable bindings etc.

In general, a rule with n heads is compiled into n rule occurrence compilations, one for each of it's head constraints being the first lookup head match task. A CHR program therefore compiles to a list of all rule occurrence compilations derived from the CHR rules in the program. Rule occurrence compilations are executed by the top level goal-based rewrite loop given in Table 2. This implementation corresponds to the hybrid STM-based parallel matching scheme (Section 3.3) where most operations are performed within `IO` and only the atomic re-verification step is performed within `STM` (see `verifyAndCommit` in Figure 3).

Function `goal_based_thread` is intended and designed to be executed asynchronously in parallel by multiple solver threads. This top level procedure is is given the references to the shared goals `gs` and store `st`, and the CHR program `prog`. Goals are exhaustively executed via the `rewriting_loop` procedure, which terminates only when the goals are empty (line 11). For this presentation, we assume an incremental goal storage scheme (Section 3) where goals are only stored when executed (line 8). Procedure `execute_goal` attempts to match the active goal `g` with each of the occurrence compilations (via the `match_goal` operation at line 13, whose definition is from line 18 onwards). Procedure `execute_goal` stops when the goal is no longer alive (line 15) or all occurrence have been tried (line 16).

Procedure `match_goal` implements the parallel matching algorithm described earlier. We assume that the first match task is the lookup of the active goal pattern (line 20). This is a valid assumption because CHR rules have at least one head and therefore this lookup task must exist. If the active goal successfully matches the head pattern (line 23) we call `exec_match`. If matching fails, we abort the procedure(line 24). Procedure `exec_match` checks for the remaining match tasks. Lines 26-28 implements the scheduling of a guard constraint `grd`. We proceed with matching if the guard evaluates to true. In case of a head constraint `c` (line 30) we first collect all possible candidates `cans` matching `c` (line 42). Then, we call `exec_match_candidate` (line 43) which tries to find a complete match for the entire rule head by iterating over the set of candidates. In case we find a complete match (line 45), we fire the rule. This step happens in parallel with our rules firing. Hence, we must atomically verify and commit this match via `verifyAndCommit`. This procedure checks that all heads are still alive and deletes the simplified heads. All these operation are done in one atomic transactional step. That is, if any of the intermediate steps fails the entire transaction fails with no visible side effect.

5. Experimental Results

We report experimental results of our parallel CHR implementation. In our implementation, we make use of the Glasgow Haskell Compiler (GHC) [8] run on a quad-core Intel Xeon processor with 1 GB of memory. At the time of writing, we have no access to an 8 core machine. But measurements in [17] performed on a 8 core machine (using an earlier less efficient implementation) support our claim that more cores imply that programs run faster.

Parallelized Union Find:

$$union \ @ \ Union(a,b), Fresh(x) \iff Fresh(x+2), Find(a,x), Find(b,x+1), Link(x,x+1)$$
$$findNode \ @ \ Edge(a,b) \setminus Find(a,x) \iff Find(b,x)$$
$$findRoot \ @ \ Root(a) \setminus Find(a,x) \iff Found(a,x)$$
$$found \ @ \ Edge(a,b) \setminus Found(a,x) \iff Found(b,x)$$
$$linkeq \ @ \ Link(x,y), Found(a,x), Found(a,y) \iff True$$
$$link \ @ \ Link(x,y), Found(a,x), Found(b,y), Root(a), Root(b) \iff Edge(b,a), Root(a)$$

Blockworld:

$$grab \ @ \ Grab(r,x), Empty(r), Clear(x), On(x,y) \iff Hold(r,x), Clear(y)$$
$$puton \ @ \ PutOn(r,y), Hold(r,x), Clear(y) \iff Empty(r), Clear(x), On(x,y)$$

Dining Philosophers:

$$grabforks \ @ \ Think(c,0,x,y), Fork(x), Fork(y) \iff Eat(c,20,x,y)$$
$$thinking \ @ \ Think(c,n,x,y) \iff n > 0 \mid Think(c,n-1,x,y)$$
$$putforks1 \ @ \ Eat(0,0,x,y) \iff Fork(x), Fork(y)$$
$$putforks2 \ @ \ Eat(c,0,x,y) \iff Fork(x), Fork(y), Think(c-1,20,x,y)$$
$$eating \ @ \ Eat(c,n,x,y) \iff Eat(c,n-1,x,y)$$

Prime Numbers:

$$prime1 \ @ \ Candidate(1) \iff True$$
$$prime2 \ @ \ Candidate(x) \iff x > 1 \mid Prime(x), Candidate(x-1)$$
$$prime3 \ @ \ Prime(y) \setminus Prime(x) \iff x \ mod \ y == 0 \mid True$$

Fibonacci Numbers:

$$fibo1 \ @ \ FindFibo(0) \iff Fibo(1)$$
$$fibo2 \ @ \ FindFibo(1) \iff Fibo(1)$$
$$fibo3 \ @ \ FindFibo(x) \iff FindFibo(x-1), FindFibo(x-2)$$
$$fibo4 \ @ \ Fibo(x), Fibo(y) \iff Fibo(x+y)$$

Turing Machine:

$$delta\_left \quad @ \ Delta(qs,ts,qs',ts',LEFT) \setminus CurrState(i,qs), TapePos(i,ts) \iff CurrState(i-1,qs'), TapePos(i,ts')$$
$$delta\_right \quad @ \ Delta(qs,ts,qs',ts',RIGHT) \setminus CurrState(i,qs), TapePos(i,ts) \iff CurrState(i+1,qs'), TapePos(i,ts')$$

Figure 4. More CHR Examples:

Figure 5 shows the experimental results. To avoid interferences with GHC's single-threaded garbage collector, we run experiments with larger heap-sizes (GHC's garbage collector has been recently extended [16] but we have not had a chance yet to perform new measurements). Results shown are averaged over multiple test runs. We use an optimal configuration based on the following parameters:

- **Constraint Indexing** for shared variable patterns in rule head.
- **Bag** implementation of CHR store.
- **Hybrid STM-Based parallel matching** of constraint multisets.
- **Goal ordering/storage** domain specific optimizations.

We measure the performance of a single-threaded execution of the `goal_based_thread` routine against 2, 4 and 8 `goal_based_threads` on the 4 core machine. We test a selection of CHR programs. Implementation of the first two are given in the earlier Figure 1. The rest are found in Figure 4

- **Merge Sort**: We merge sort 1024 integers. We can compare different pairs of integers in parallel.
- **Gcd**: We find the greatest common divisor of 1000 integers. Finding the Gcds of distinct pairs of integers can be executed in parallel.
- **Parallelized Union Find**: Adapted from [6], Union find is basically a data structure which maintains the union relationship among disjoint sets. Sets are represented by trees ($Edge(x,y)$) in which root notes ($Root(x)$) are the representatives of the sets. The union operation between two sets of a and b ($Union(a,b)$) is executed by finding the representatives x and y of the sets a and b ($Find(a,x)$ and $Find(b,y)$), and then linking them together ($Link(x,y)$). The $union$ rule initiates the union operation. The constraint $Fresh(x)$ introduces "fresh variables" since our current prototype only supports ground CHR rules/stores. Rule $findNode$ traverses edges until we reach the root in rule $foundRoot$. Rule $found$ re-executes a find if the tree structure has changed. This is necessary since union find operations can be executed in parallel. Rule $linkeq$ removes redundant link operations and rule $link$ performs the actual linking of two distinct trees. In experiments, we test an instance of parallelized union find, where 300 union operations are issued in parallel to unite 301 disjoint sets (binary trees) of depth 5.
- **Blockworld**: A simple simulation of robot arms re-arranging stacks of blocks. $Grab(r,x)$ specifies that robot r grabs block x, only if r is empty and block x is clear on top and on y ($On(x,y)$). The result is that robot r will be holding block x ($Hold(r,x)$) and block x is no longer on block y, thus y is clear. $PutOn(r,y)$ specifies that robot r places a block on block y, if r is holding some block x and y is clear. In our experiments, we simulate 4 agents each moving a unique stack of 1000 blocks. Robots can be executed in parallel as long as their actions do not interfere.
- **Dining Philosophers**: The classic dining philosopher problem, simulating a group of philosophers thinking and eating on a round table, and sharing a fork with each of her neighbors. In our implementation, Forks are represented by the constraints $Fork(x)$ where x is a unique fork identifier. A thinking and eating philosopher is represented by the constraints

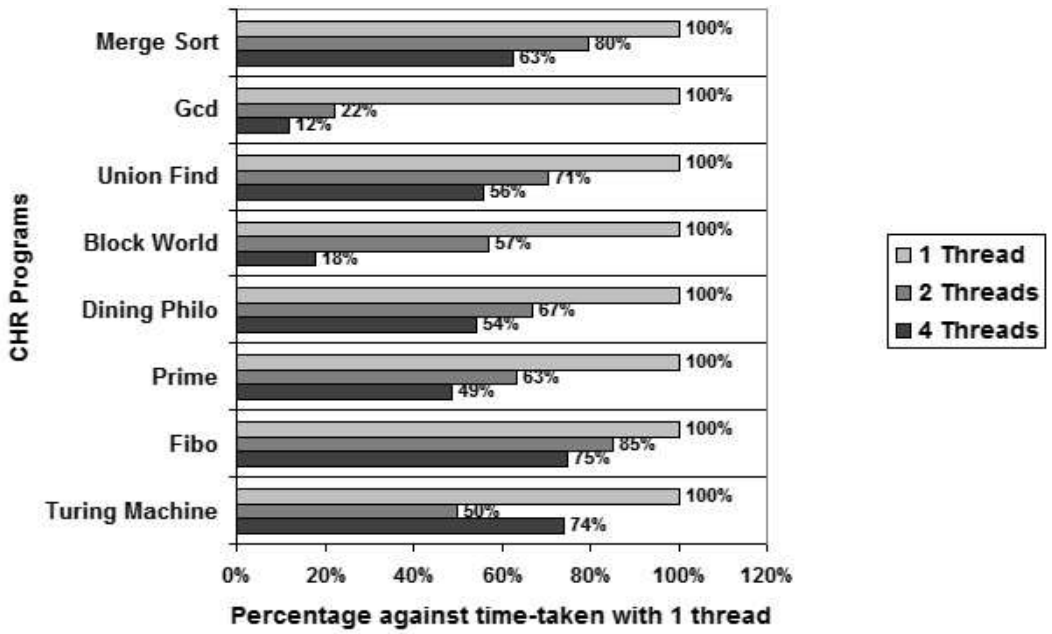

Figure 5. Experimental Results of optimal configuration on 4 core machine

$Think(c, n, x, y)$ and $Eat(c, n, x, y)$ where x and y are the fork identifiers, c represents the number of eat/think cycles left and n a counter that simulates the delay of thinking/eating process. Rules $thinking$ and $easting$ delay thinking and eating. If there any think/eat cycles left, we return both forks and issue a new thinking process. See rule $putforks2$. Otherwise, we only return both forks. See rule $putforks1$. In our experiments, we simulated the dining philosopher problem with 150 philosophers, each eating and thinking for 50 cycles with a delay of 20 steps.

- **Prime**: Identifying the first n prime numbers. We find the first 1500 prime numbers. Parallelism comes in the form of parallel comparison of distinct pairs of candidate numbers.
- **Fibonacci**: Identifying the value of the n^{th} Fibonacci number. We find the 25^{th} Fibonacci number. Parallelism is present when evaluating different parts of the Fibonacci tree.
- **Turing Machine:** A simple formulation of the classic Turing machine. In our implementation, $delta\_left$ and $delta\_right$ define the state transitions of the Turing machine. The constraint $Delta(qs, ts, qs', ts', dir)$ specifies the state transition mapping $(qs, ts) \mapsto (qs', ts', dir)$ where qs, qs' are state symbol and ts, ts' are tape symbols and dir is the direction which the tape is moved. $CurrState(i, qs)$ states that the current state of the machine is qs at tape position i. $TapePos(i, ts)$ states that tape position i has the symbol ts. In our experiments, we tested a Turing machine instance which determines if a tape (string of 0's and 1's) of length 200 is of the form $\{0^n 1^n \mid n > 1\}$. The Turing machine simulator is inherently single thread (rules cannot fire in parallel), as it involves state transitions of a single state machine. This serves to investigate the effects of parallel rewriting applied to a single threaded problem.

Results show the percentage of the original (1 thread) time-taken when the specified number of threads are used. Results show that optimal performance is generally achieved when we match the number of `goal_based_threads` with the number of processors. In our experience, having more threads than actual processors available does not improve the performance. In fact, the performance may get worse because we encounter unnecessary conflicts and overheads.

One interesting result that our experiment uncovered is the presence of super-linear speed-up for certain examples, like Gcd. The reasons for this is often very subtle and domain specific. Figure 6 illustrates why we get super-linear speed-up for the Gcd example. For presentation purpose, we annotate each constraint with a unique identifier and each derivation with the rule name parameterized by the constraints that fired it and the number of times it fired. For instance $g2(x, y) \times t$ denotes that rule $g2$ fired on constraints x and y for t number of times. We examine derivations of the Gcd example from the initial store

$$\{Gcd(30), Gcd(2), Gcd(45), Gcd(15)\}$$

Derivation A shows the single threaded case where we get a total of 57 derivation steps to reach the final store. Derivation B shows the parallel derivation of 2 threads which yield the expected results (linear speed-up of 26 sequential derivation steps). This assumes an unlikely scenario where derivations between 2 pairs of Gcd constraints do not overlap (i.e. interfere with each other). Derivation C shows the actual result which yields super-linear speed-up. Derivations overlap, that is, there can be rule firings across parallel derivations. [2] This allows Gcd constraints of higher values to be matched together, cutting down tediously long derivations initiated by Gcd constraints of lower values (which is typical in the single threaded case). Derivation C shows only one of many indeterministic derivations possible. Our experimental results in Figure 5 confirm that we still get super-linear speed-up in general.

[2] Of course, this behavior is also possible in a sequential execution scheme where we interleave the execution of goal constraints, thus, effectively simulating the parallel execution scheme.

Gcd Example:

$$gcd1 \ @ \ Gcd(0) \iff True$$
$$gcd2 \ @ \ Gcd(n)\backslash Gcd(m) \iff m >= n \&\& n > 0 \mid Gcd(m-n)$$

Derivation A: Single Threaded (Shorthands: $G = Gcd$, $g1 = gcd1$ and $g2 = gcd2$)

$$\begin{array}{ll} & \{G(30)_1, G(2)_2, G(45)_3, G(15)_4\} \\ \rightarrowtail_{g2(2,1)\times 15} & \{G(0)_1, G(2)_2, G(45)_3, G(15)_4\} \\ \rightarrowtail_{g1(1)\times 1} & \{G(2)_1, G(45)_2, G(15)_3\} \\ \rightarrowtail_{g2(1,2)\times 22} & \{G(2)_1, G(1)_2, G(15)_3\} \\ \rightarrowtail_{g2(2,1)\times 2} & \{G(0)_1, G(1)_2, G(15)_3\} \\ \rightarrowtail_{g1(1)\times 1} & \{G(1)_1, G(15)_2\} \\ \rightarrowtail_{g2(1,2)\times 15} & \{G(1)_1, G(0)_2\} \\ \rightarrowtail_{g1(1)\times 1} & \{G(1)_1\} \end{array}$$

Total Number of Sequential Derivations: 57 Steps

Derivation B: 2 Distinct Parallel Derivation (Expected Results)

$$\begin{array}{ccc} \{G(30)_1, G(2)_2\} & & \{G(45)_3, G(15)_4\} \\ \rightarrowtail_{g2(2,1)\times 15} & & \rightarrowtail_{g2(4,3)\times 3} \\ \{G(0)_1, G(2)_2\} & \| & \{G(0)_3, G(15)_4\} \\ \rightarrowtail_{g1(1)\times 1} & & \rightarrowtail_{g1(3)\times 1} \\ \{G(2)_1\} & & \{G(15)_3\} \\ \hline & & \{G(2)_1, G(15)_2\} \\ & \rightarrowtail_{g2(1,2)\times 7} & \{G(2)_1, G(1)_2\} \\ & \rightarrowtail_{g2(2,1)\times 2} & \{G(0)_1, G(1)_2\} \\ & \rightarrowtail_{g1(1)\times 1} & G(1)_1 \end{array}$$

Total Number of Sequential Derivations: 26 Steps ($\approx$ linear speed-up)

Derivation C: 2 Overlapping Parallel Derivations (Actual Results)

$$\begin{array}{ll} & \{G(30)_1, G(2)_2, G(45)_3, G(15)_4\} \\ \rightarrowtail_{(g2(2,1)\| g2(4,3))\ \times\ 1} & \{G(28)_1, G(2)_2, G(30)_3, G(15)_4\} \\ \rightarrowtail_{(g2(1,3)\| g2(2,4))\ \times\ 1} & \{G(28)_1, G(2)_2, G(2)_3, G(13)_4\} \\ \rightarrowtail_{(g2(4,1)\| g2(2,3))\ \times\ 1} & \{G(15)_1, G(2)_2, G(0)_3, G(13)_4\} \\ \rightarrowtail_{(g2(4,1)\| g1(3))\ \times\ 1} & \{G(2)_1, G(2)_2, G(13)_3\} \\ \rightarrowtail_{g2(1,2)\times 1} & \{G(0)_1, G(2)_2, G(13)_3\} \\ \rightarrowtail_{g1(1)\times 1} & \{G(2)_1, G(13)_2\} \\ \rightarrowtail_{g2(1,2)\times 6} & \{G(2)_1, G(1)_2\} \\ \rightarrowtail_{g2(2,1)\times 2} & \{G(0)_1, G(1)_2\} \\ \rightarrowtail_{g1(1)\times 1} & \{G(1)_1\} \end{array}$$

Total Number of Sequential Derivations: 15 Steps (super-linear speed-up)

Figure 6. Why Super-Linear Speed-up in Gcd

If we use a more refined variant of the Gcd program, where we replace rule *gcd2* by

$$gcd2' \ @ \ Gcd(n)\backslash Gcd(m) \iff m >= n \&\& n > 0 \mid Gcd(m \ mod \ n)$$

we "only" get linear speed-up because by using $gcd2'$ instead of *gcd2* we effectively "compress" the derivation steps even for the single threaded case. Therefore, there are practically no rule firings across parallel derivations.

The Block World example also shows some super-linear speed-up in case of 4 cores but for a different reason. Each robot works independently on a fragment of the Block World. Hence, there cannot be any rule firings across parallel derivations. However, we arranged the store for optimal access by 4 threads (robots). In essence, this reduces the time for each robot to search and store constraints by a quarter and this is the reason for the super-linear speed-up.

The Turing machine example is inherently single-threaded. Interestingly, we obtain improvements from parallel execution of administrative procedures (for example dropping of goals, due to failed matching). Relative drop in performance (between 2 and 4 threads) indicates a upper bound of such "administrative" parallelism.

In the next series of experiments, we are interested in investigating which of the four optimizations forming the optimal configuration of the parallel rewriting system is critical for parallelism. We deactivate the four optimizations one at a time and test their relative effects (between number of threads used) against the optimal control results. For brevity, we only explicitly show the respective results if the relative performance between number threads has significantly changed, and saturate all results above 200% of the single-thread time.

- **Deactivating constraint indexing**: With constraint indexing deactivated, CHR programs with shared variable patterns (Blockworld, Dining Philo, mergesort and unionfind) generally run slower. However, the performance ratio between number of threads used largely remains the same. This shows that constraint indexing is an orthogonal optimization and not intimately related to parallelism.
- **Linked list CHR store**: With the list implementation of the store, threads are more likely to finding conflicting constraint matchings. Experiment results (Figure 7) show that most of our test cases are not sensitive to the store implementation used. The exception being the Fibonacci example which shows worse performance with increasing number of cores in case a linked list store is used (Performances saturates above $>$ 200% of original time).
- **Pure STM-based parallel matching** In this configuration, parallel constraint matching is implemented as coarse-grained transactional steps. This corresponds to the original STM-based parallel matching discussed in Section 3.3. Hence we expect that entire rewriting steps run in interleaving steps rather than in parallel. Results shown in Figure 8 support this claim, illustrating that this approach generally does not scale with the number of threads.
- **Deactivating domain specific goal optimizations** We use an unordered implementation of goals. Threads are allowed to pick any goal with no form of ordering as discussed in Section 3.2. Our results show that most of our test cases are in-sensitive to

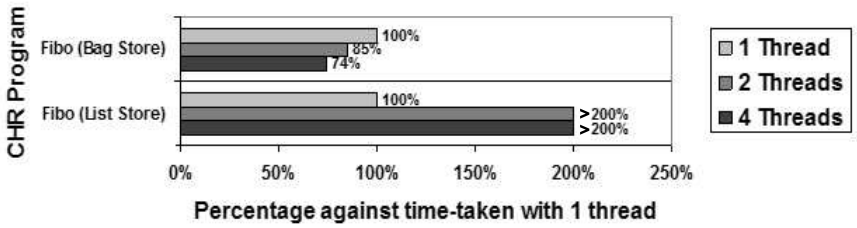

Figure 7. Result variations with linked list CHR store

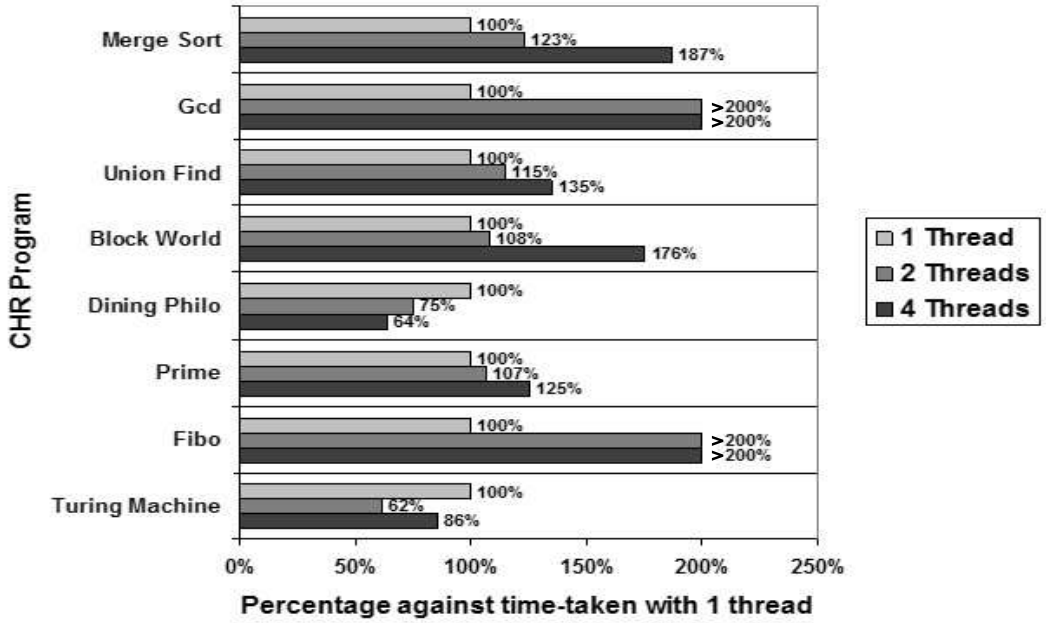

Figure 8. Result variations with STM-based parallel matching

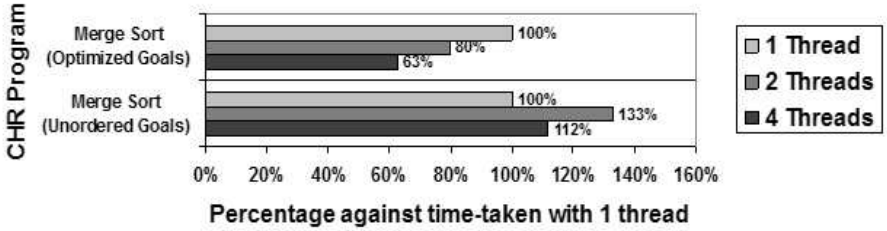

Figure 9. Result variations with domain specific goal-based optimizations

goal ordering/ storage schemes. Merge sort however exhibits high sensitivity to goal ordering. See Figure 9.

To summarize, optimal performance is obtained when the number of threads matches the number of processor cores available. Constraint indexing is an orthogonal optimization in the context of parallelism. Bag store implementation, lock-based parallel matching and domain specific goal optimizations can significantly affect the performance and scalability of parallel rewriting, even though scalability of many CHR programs is insensitive to their presence/absence.

In our current implementation, we make (overly) extensive use of transactional variables (TVars) which causes some unnecessary overhead. For example, our parallel implementation using one thread runs approximately four times slower compared to a sequential implementation where we use imperative mutable variables (IORefs). The overhead caused by the atomic re-verification step is insignificant. Some preliminary tests show that the overhead re-

duces to one to two times if instead of transactional variables we use synchronized mutable variables (MVars).

We yet have to measure the performance of our system against the state-of-the-art CHR implementations [14]. We believe that our implementation is competitive but there is still space for improvement. The main advantage of our system is that it scales with the number of processors whereas existing CHR systems practically ignore this additional computing power.

6. Related Work

Current state-of-the-art implementations of CHR rewriting systems [14, 20] are based on the CHR refined operational semantics [3] which dictates a strict stack-based ordering of goal constraints using a lazy match algorithm, thus, achieving a more deterministic semantics for which efficient compilation schemes exist [2, 10, 18]. Goal constraints are executed sequentially using a lazy storage scheme. Eager storage would only incur some unnecessary runtime overhead. We believe that to parallelize a strictly deterministic execution model such as the refined operational CHR semantics requires tremendous effort in synchronizing parallel execution threads to maintain the semantic determinism of the model and is highly likely to hamper the possibilities of any parallel executions. Therefore, our approach to parallelize CHR rewriting is based on the abstract CHR semantics [5] which is highly parallelizable. Previous theoretical work on reasoning about the parallel execution of CHR [6] critically rely on this semantics. In our implementation, we borrow the notion of goal constraints and the lazy match algorithm from the refined semantics model. But we are less strict when it comes to ordering the execution of goal constraints. This gives us much more freedom when it comes to exploiting parallelism.

Our techniques of ordering goals via priority queues is similar to the approach taken in the CHR rule priority semantics [12, 13], but motivated by entirely different purposes. While our approach prioritize goals for optimizing performance of parallel rewriting, the CHR rule priority semantics order goals to enforce prioritized rule executions.

The only related work on parallelizing CHR we are aware of is [19]. Like us the work described there follows the abstract CHR semantics but uses a mix of goal-based and rule-based execution which in our experience yields too many conflicts. No optimizations (such as the ones discussed in Sections 3.1, 3.3 and 3.2) are considered. Our experimental results show that such optimizations are crucial to obtain better performance results.

Parallel production rule systems [11] often use some form of parallel implementation of the Rete algorithm [4, 9] where matching multisets are eagerly computed and incrementally accumulated. Our approach to parallel matching follows closer the traditional CHR matching semantics, which computes matchings only on demand (lazily) and is guided by goal constraints. Lazy matching algorithms are proven to exhibit better memory efficiency than eager matching algorithms like Rete. Furthermore, eager matching algorithms are not suitable for general (simpagation) CHR rules since matchings involving simplified heads are not longer useful and should not have been accumulated. However, parallel eager matching could be considered for pure propagation rules.

7. Conclusion

We reported on an efficient parallel implementation of CHR which takes advantage of additional processor power as provided by today's modern shared memory, multi-core architectures. This is supported by our experiments which show that CHR programs run generally faster given more processor cores. Our results show that the CHR language is well-suited for parallel programming on multi-core architectures which is one of the big challenges in computer science today.

Acknowledgments

We thank Greg Duck, Tom Schrijvers, Michael Stahl, Peter Van Weert and PPDP'08, RTA'08 referees for their helpful comments on previous versions of this paper.

References

[1] S. Abdennadher. Operational semantics and confluence of constraint propagation rules. In *Proc. of CP'97*, LNCS, pages 252–266. Springer-Verlag, 1997.

[2] G. J. Duck. *Compilation of Constraint Handling Rules*. PhD thesis, The University of Melbourne, 2005.

[3] G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *Proc of ICLP'04*, volume 3132 of *LNCS*, pages 90–104. Springer-Verlag, 2004.

[4] C. Forgy. A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17–37, 1982.

[5] T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.

[6] T. Frühwirth. Parallelizing union-find in Constraint Handling Rules using confluence analysis. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 113–127. Springer-Verlag, 2005.

[7] T. Frühwirth. Constraint handling rules: the story so far. In *Proc. of PPDP '06*, pages 13–14. ACM Press, 2006.

[8] Glasgow haskell compiler home page. http://www.haskell.org/ghc/.

[9] A. Gupta, C. Forgy, A. Newell, and R. G. Wedig. Parallel algorithms and architectures for rule-based systems. In *Proc. of ISCA'86*, pages 28–37, 1986.

[10] C. Holzbaur, M. J. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *TPLP*, 5(4-5):503–531, 2005.

[11] T. Ishida. Parallel rule firing in production systems. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):11–17, 1991.

[12] L. De Koninck, T. Schrijvers, and B. Demoen. User-definable rule priorities for CHR. In *Proc. of PPDP'07*, pages 25–36. ACM, 2007.

[13] L. De Koninck, P.J. Stuckey, and G.J. Duck. Optimizing compilation of CHR with rule priorities. In *Proc. of FLOPS'08*, 2008. To appear.

[14] The K.U. Leuven CHR System. http://www.cs.kuleuven.be/~toms/Research/CHR.

[15] E. S. L. Lam and M. Sulzmann. A concurrent Constraint Handling Rules implementation in Haskell with software transactional memory. In *Proc. of ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP'07)*, pages 19–24, 2007.

[16] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proc. of International Symposium on Memory Management 2008*, 2008. To appear.

[17] C. Perfumo, N. Sonmez, O. S. Unsal, A. Cristal, M. Valero, and T. Harris. Dissecting transactional executions in Haskell. In *The Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2007.

[18] T. Schrijvers. Analyses, optimizations and extensions of Constraint Handling Rules: Ph.D. summary. In *Proc. of ICLP'05*, volume 3668 of *LNCS*, pages 435–436. Springer-Verlag, 2005.

[19] M. Stahl. Implementing CHR with STM, March 2007. personal communication.

[20] P. Van Weert, T. Schrijvers, and B. Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *Proc. of Second Workshop on Constraint Handling Rules*, pages 47–62, 2005.