J. Symbolic Computation (1996) **21**, 475–505

Strategy Compliant Multi-Threaded Term Completion

REINHARD BÜNDGEN[‡] MANFRED GÖBEL[†] AND WOLFGANG KÜCHLIN[†]

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany

(Received 2 June 1995)

We report on the design, implementation, and performance, of the parallel term-rewriting system PaReDuX. We discuss the parallelization of three term completion procedures: Knuth–Bendix completion, completion modulo AC, and unfailing completion. Our parallelization is strategy-compliant, i.e., the parallel code performs exactly the same work as the sequential code, but the work load is shared by many processors. PaReDuX is designed for shared memory parallel architectures, such as multi-processor workstations, where it shows good performance on a variety of examples.

1. Introduction

The advent of shared memory multi-processor workstations on the desk-top poses new challenges and opportunities for symbolic computation. Parallel workstations support programs that are multi-threaded; traditional single-threaded programs can utilize only a fraction of their total computation power. Hence we must learn how to bring symbolic computation into the multi-threaded future. More specifically, we must acquire know-how in the areas of parallel data-structure design, parallel programming techniques, and parallel systems environments, which permit us to build and run parallel symbolic computation systems as our standard tools on the desk-top.

The specific topic of this paper is to provide an answer for the area of term-rewriting and completion. We discuss the design, implementation, and performance, of PaReDuX, a system evolving, by multi-threading, from the sequential ReDuX term-rewriting laboratory (Bündgen, 1993; Bündgen *et al.* 1995).

1.1. OVERVIEW

The contributions of this paper are in several areas. We parallelized plain Knuth–Bendix completion, completion modulo AC, and unfailing completion, in a single system and obtained a wealth of empirical performance data. Our parallelizations show good

[†] E-mail: {buendgen,goebel,kuechlin}@informatik.uni-tuebingen.de
[‡] URL: http://www-sr.informatik.uni-tuebingen.de

0747–7171/96/040475 + 31 $18.00/0

overall speed-ups, roughly of the order of 2–3.8, on 4 processors of a parallel workstation. We also developed a parallel term data-structure which permits several processes to use terms simultaneously in matching or unification attempts.

We employed a high-level divide-and-conquer programming style which generates parallelism through calls to a well-defined kernel. Our parallel completion algorithms are strategy compliant, i.e., are guaranteed to adhere to a statically selected strategy. Their performance is stable, reproducible and predictable. It had been an open problem whether significant speed-ups could be achieved using relatively fine-grained shared-memory techniques on these algorithms.

The construction of PaReDuX started with the high-quality sequential implementation of completion in the ReDuX system (Bündgen, 1993), which we parallelized gradually. PaReDuX now contains data-structures and algorithms for parallel Knuth–Bendix completion of plain term-rewriting systems (Knuth and Bendix, 1970), of completion modulo associativity and commutativity (AC) (Peterson and Stickel, 1981), and unfailing completion (Bachmair *et al.*, 1989)[†].

In order to develop PaReDuX, we must first 'parallelize' the ReDuX data structures in such a way that concurrent access is possible by multiple threads of control with minimal synchronization, and then we must parallelize the algorithms. We achieved this in a very systematic way, using a high-level divide-and-conquer approach to parallel programming, and hiding the key change in data structures behind an existing access abstraction.

We work under the general restriction of *strategy compliance*, i.e., no change of the completion strategy is allowed when going parallel. Thus the parallel work is exactly the same as the sequential work, and no new correctness proofs are needed. All speed-ups come from spreading this work over the processors of a parallel workstation. Our parallel algorithm library can be used as a building block within other, more coarse-grained, completion algorithms which may derive additional speed-ups from parallel search.

This work is closely related to the design of PARSAC-2 (Küchlin, 1990, 1995), a system which focusses on multi-threaded algebraic computation. PARSAC-2 is a parallel extension of the sequential SAC-2 system in its SACLIB form (Buchberger *et al.*, 1993), whose algorithms form its algebraic kernel. All parallel constructs are provided by its *S-threads* parallelization environment (Küchlin, 1992), which is an extension of the threads (lightweight processes) environment supported by most modern operating systems.

PaReDuX is also built on the S-threads system of PARSAC-2, and applies the same high-level parallelization style. This was made possible because ReDuX also uses the list processing module of SAC-2, which could then be replaced by S-threads. Within PARSAC-2, similar parallel symbolic programming techniques have already been used to develop algorithms for long integer multiplication, polynomial real root isolation, and multivariate polynomial g.c.d. computation. Therefore, beyond their significance of speeding up term-completion in practice, the results presented here contribute towards our overall goal of developing the know-how for practically useful parallel symbolic computation.

Our hardware architecture is a shared memory multiprocessor such as a typical parallel workstation. S-threads acts as a resource broker between the demands of the algorithms and the capacities of the hardware. The parallel algorithms themselves use no application

[†] ReDuX also contains a module for inductive completion whose parallelization is left for future work.

level assumptions on the architecture (such as the number of processors etc.) and do not contain low-level code such as explicit task schedulers or assignment of tasks to processors.

We now proceed as follows. In the remainder of this section we give a more detailed motivation for our work. In Section 2, we present the context of completion terminology and of our parallelization approach as far as it is necessary to understand our results. In Section 3, we present our parallelization philosophy. Section 4 proposes a classification of parallel completion procedures and shows how known parallelization approaches fit into that framework. Section 5 explains the parallel algorithms, and Section 6 the parallel data-structures used in PaReDuX. Section 7 contains empirical data about the performance of our parallel completion procedures on the problems whose specification is given in the appendix. Finally, Section 8 presents our conclusions.

1.2. motivation

Since multi-processors have appeared on the desk-top, the programs we work with on a daily basis must be parallel. For practical reasons, they must not be radically different from the standard sequential programs. They must be upwardly compatible in functionality, speed, and programming language. Therefore, we seek parallelizations which start with proven sequential code and value reliability, predictability, and ease of programming, over other aspects.

The general question motivating this work is: *how can we routinely program shared memory parallel workstations to perform standard symbolic computation tasks equally well, but faster than before?* It is particularly desirable to develop a discipline of parallel programming which can be applied uniformly to many aspects of symbolic computation, from Computer Algebra to Theorem Proving.

This paper contributes towards an answer for the important and complex computational process of *completion* (Buchberger and Loos, 1982). Completion has both a term-rewriting form, in the tradition of Knuth and Bendix (1970), and a polynomial ideal form, in the tradition of Buchberger (1965). Incidentally, the relationship between both forms is by now well understood (Buchberger,1985; Bündgen, 1991, 1992), so that advances with one version can frequently be transferred to the other.

Hence the question arises to what extent completion can be speeded up by parallelization and by what means this can be accomplished. Completion procedures are, however, notoriously hard to parallelize. This is due to several overlapping effects.

First, completion is a chaotic process in the sense that it is extremely data-dependent and its course of action, and running time, are impossible to predict from the input data. Any parallel form must cope with unpredictable and dynamically changing amounts of parallelism and memory, and no fixed schedules or processor allocations are possible.

Second, completion is extremely strategy dependent and parallel forms are likely to change the completion strategy, possibly dependent on dynamic scheduling decisions[†]. Strategy induced speed-ups (or slow-downs) may therefore superpose with speed-ups from parallel work and contort the picture.

Third, completion is at its core a closure computation [cf. Slaney and Lusk (1990)] and therefore is inherently sequential in the sense that the nth generation of consequences necessarily depends on the $(n-1)$st generation. The amount of parallelism in the process

[†] Thus parallelization even risks slow-downs, if highly tuned sequential strategies exist.

is essentially limited by the size of the generations of consequences. This applies particularly to converging processes. If the number of consequences (and hence the amount of parallelism) is great, the completion process may be diverging. If the process converges, the number of consequences (and hence the amount of parallelism) must somehow be limited. As we shall see in Section 7, the number of consequences to be considered is typically small during large stretches of a converging completion run, but is several orders of magnitude larger in the remaining tight spots.

Because of the many facets of completion, it is important to parallelize the process at all levels of granularity. Most theoretical work has so far focussed on extremely fine-grained subproblems such as parallel matching. Practical work has focussed on the coarse-grained end of the spectrum, where it is easier to get speed-ups, but where strategy effects may come into play. In this paper, we attack the middle ground where speed-ups are already difficult to obtain, especially when programming at a high level of abstraction.

So far, all our parallelizations are *strategy compliant*, i.e., they are guaranteed to adhere to the same completion strategy independent of task scheduling or the number of processors or the number of other processes in the system. Strategy induced super-linear speed-ups, but also slow-downs, are therefore impossible. We believe that strategy effects should be isolated and exploited separately, and then combined with this work in an orthogonal way.

Our strategy compliant design has a number of advantages. It produces predictable and deterministic speed-ups. If processors are added, the code will run faster as long as there is enough parallelism; if processors are taken away it will gracefully degrade to sequential performance. Further, all parallel experiments are reproducible. This is important for evaluating parallel data structures and for investigating optimal parallelization grain sizes. Also, completion attempts can be broken off and restarted with predictable behavior. Since our design also speeds up the main completion loop, it is well suited for the interactive completion of new unknown problems.

We are not aware of other work that specifically attempts strategy compliant parallel term completion. In the Gröbner basis case, Faugère's (1994) parallelization is strategy compliant, but it is based on modular computations that have no analogue in term completion. Attardi and Traverso (1994) have designed a strategy compliant parallel Gröbner basis algorithm similar to ours, but so far the empirical results are limited[†].

In principle, 'cross fertilization' between term completion techniques and polynomial ideal completion techniques is possible, due to the similarity of the procedures. For parallelizations this has been done to some extent by Chakrabarti and Yelick (1993) and Yelick and Garland (1992). However, polynomial completion seems to be harder to parallelize than term completion. On the coarse grained level, significant speed-ups are difficult because very good sequential strategies exist. Put the other way, the practical significance of even large speed-ups is limited if the base line of measurement is an inferior sequential strategy. On the fine-grained level, parallel reduction of critical pairs (i.e., S-polynomials) performs much excess work because most critical pairs can be deleted through confluence criteria without reduction. So far, only Faugère's parallelization seems to be derived from, and to achieve speed-ups over, a state-of-the-art sequential implementation[†].

It should be noted that our strategy compliant parallelizations can be used within

[†] *Note added in proof*: for another parallelization with significant speed-ups see Amrhein *et al.* (1966).

other, more coarse-grained parallelizations, e.g., on the multi-processor nodes of a workstation network.

2. Background

2.1. parallel computation with virtual S-threads

In traditional operating systems, each process has an address space and a single *thread of control.* A thread of control is an execution context for a procedure, much as a process is an execution context for a complete program. A *threads system* allows several threads, i.e., procedures, to be active concurrently. Hence multi-threading achieves finer grained parallelism than multi-processing. A threads system can be implemented at the user level, but most modern operating systems (Tanenbaum, 1992), such as Mach, Solaris 2.x, Windows NT or OS/2, provide kernel threads.

New threads can be created by a *fork operation.* Forking a new thread is similar to calling a procedure, except that the caller does not wait for the procedure to return. Instead, the parent continues to execute concurrently with the newly forked child; on a multiprocessor system this may result in true parallelism. At some later time, the parent may rendezvous with the child by means of a *join operation* and retrieve its results.

A thread provides a procedure with a private register file and a private stack. For efficient parallel *symbolic* computation this is not enough: a private portion of the heap is needed together with an appropriate (parallel) garbage collection facility. In PARSAC-2 this is provided by the *S-threads* system (Küchlin, 1992). S-threads was originally modeled after C Threads (Cooper and Draves, 1988), the interface to the Mach operating system. It assumes that a minimal standard threads interface, such as POSIX threads, is provided by some kernel below. Each kernel thread is then extended to an S-thread capable of concurrent list processing. In this way, virtually all sequential C programs in SACLIB will execute unmodified as a single S-thread, with a slight (say 2–5%) execution penalty imposed by the parallel list-processing context. S-threads runs on the Mach and Solaris 2.x kernel threads, but also on the user-level threads provided by PCR (Weiser *et al.*, 1989), which in turn runs on UNIX System V.

The original S-threads memory management scheme distributes the SAC-2 heap to threads as paged segments of cells, and it uses *preventive garbage collection* (Küchlin and Nevin, 1991). If a side-effect free functional programming style is used, then the result of a function can be copied into its parent's heap segment, and its own heap segment, containing all garbage, can be recycled in bulk. This scheme is efficient, naturally concurrent, and does not assume that first all threads be stopped by a user.

During Knuth–Bendix completion, however, partial modifications are made to very large critical pair queues. For reasons of efficiency, these queues are updated via side-effects and preventive garbage collection is not applicable in this case. Therefore, the memory management of S-threads was changed to use the PACLIB (Schreiner and Hong, 1993) scheme. Since it works on the cell level, cells allocated by one thread can be woven into a global data-structure via side-effects. The PACLIB scheme assumes however that all threads can be stopped by a user. Since we did not parallelize nor optimize our implementation of this scheme yet, garbage collection times are always excluded from our measurements.

S-threads has been successfully employed to parallelize a number of algebraic algorithms (Küchlin, 1995). It was found, however, that kernel threads do not cope well with large amounts of dynamically created parallelism. Since each S-thread is mapped one-to-one onto a kernel thread, limitations of kernel implementations may show up in S-threads. This might include high fork/join times or a strict limitation on the number of concurrently active threads. It has been observed elsewhere that differences in kernel thread efficiency may destroy all parallel speed-ups on a new machine, making the system effectively non-portable (Morisse and Oevel, 1995). As a consequence, either the application must become involved in thread management decisions, or the threads system must be improved.

S-threads was enhanced to *virtual S-threads* (Küchlin and Ward, 1992). The rôle of a *virtual* thread is to document *logical* concurrency, i.e., it represents a task which can possibly run in parallel on a separate processor. It is then up to the underlying thread scheduler to decide whether a virtual thread is executed as a subroutine call, or whether it is executed on a separate kernel thread leading to *real* concurrency.

VS-threads not only add application-level efficiency but also insulate the application from the idiosyncrasies and limitations of the underlying threads implementation. VS-threads manage fork requests using *lazy task creation* (Mohr *et al.*, 1991), handling most threads system calls itself and passing only a tiny fraction to the OS kernel. It keeps its own run-queues of micro-tasks, and it manages a small pool of kernel threads which it employs as *workers*. On each fork, a record containing the fork parameters is put in the run-queue. These tasks can be asynchronously stolen by idle workers and executed as S-threads. However, if a join finds that the task was not yet stolen, the parent S-thread executes the task as a procedure call.

Thus, the virtually unbounded logical concurrency of the application is dynamically reduced to the bounded amount of real parallelism that the kernel can support. At the same time there is a significant reduction in the number of kernel thread context switches, and the grain-size of the remaining threads is increased by executing child tasks as procedures. Virtual threads also have a much lower overhead than kernel threads, so that on average thread overhead drops and becomes easier to handle.

The system now manages the tasks at a small cost in execution overhead and a great savings in programming complexity. VS-threads allow us to generate routinely tens to hundreds of thousands of parallel tasks while maintaining execution efficiency. Thus, to a great extent, virtual threads take the pain out of parallel symbolic programming in practice.

2.2. TERM REWRITING AND COMPLETION

In this section, we will describe a term completion procedure to a level of detail which is necessary to understand its parallelization. For a more detailed discussion and alternative approaches the reader is referred to Dershowitz and Jouannaud (1990), Klop (1992) and Plaisted (1993).

Terms are constructed from variables, constants and function symbols in the usual way. The basic operations on terms are instantiation and tests for (structural) equality, matching and unification. A term t' is an *instance* of t if it can be obtained by substituting terms for the variables in t; we write $t' = t\sigma$ where σ is the instantiating *substitution.* A term s *matches* another term t if all variables in s can be *substituted* by terms such that the new *instance* of s is equal to t. Two terms s and t *unify* if their respective variables can be substituted in such a manner that s and t have a common instance. A substitution

Delete: $$\frac{(\mathcal{P} \cup \{s = s\}; \mathcal{R})}{(\mathcal{P}; \mathcal{R})}.$$

Simplify: $$\frac{(\mathcal{P} \cup \{s = t\}; \mathcal{R})}{(\mathcal{P} \cup \{s = u\}; \mathcal{R})} \quad \text{if } t \rightarrow_{\mathcal{R}} u.$$

Orient: $$\frac{(\mathcal{P} \cup \{s = t\}; \mathcal{R})}{(\mathcal{P}; \mathcal{R} \cup \{s \rightarrow t\})} \quad \text{if } s \succ t \text{ for terminating term ordering } \succ \supseteq \rightarrow_{\mathcal{R}} .$$

Compose: $$\frac{(\mathcal{P}; \mathcal{R} \cup \{s \rightarrow t\})}{(\mathcal{P}; \mathcal{R} \cup \{s \rightarrow u\})} \quad \text{if } t \rightarrow_{\mathcal{R}} u.$$

Collapse: $$\frac{(\mathcal{P}; \mathcal{R} \cup \{s \rightarrow t\})}{(\mathcal{P} \cup \{u = t\}; \mathcal{R})} \quad \text{if } s \rightarrow_{\mathcal{R}} u \text{ by } l \rightarrow r \in \mathcal{R} \text{ where } (s,t) \rhd (l,r)^{a}.$$

Deduce: $$\frac{(\mathcal{P}; \mathcal{R})}{(\mathcal{P} \cup \{s = t\}; \mathcal{R})} \quad \text{if } (s,t) \text{ is a critical pair of } \mathcal{R}.$$

[a] $\rhd$ is a terminating ordering on term pairs.

Figure 1. Completion inference rules.

μ is a *most general unifier* of s and t if $s\mu = t\mu$ and all other common instances of s and t are also instances of $s\mu$.

A *rewrite rule* is a pair of terms. It may be applied to *reduce* a term t if t contains an instance l' of l. Then t reduces to t', where t' is t with l' replaced by a corresponding instance of r. A set of rewrite rules is a *term rewriting system (TRS).* A TRS $\mathcal{R}$ presents a reduction relation such that $s \rightarrow_{\mathcal{R}} t$ if there is a rule in $\mathcal{R}$ that reduces s to t. Computing the (reflexive) transitive closure of the $\mathcal{R}$-reduction of a term t is called *normalizing* t. We assume that the normalization relation of the TRS considered in this paper is a terminating procedure.

Let $l \rightarrow r$ and $l' \rightarrow r'$ be two rules where l contains a subterm s which unifies with l' such that a most general unifier of s and l' is μ. Then $l\mu$ can be reduced by each of the two rules, and the two terms resulting from the two different one-step reductions are called a *critical pair*. Knuth and Bendix (1970) showed that a terminating TRS computes unique normal forms iff all critical pairs have a common normal form.

A TRS is *complete* if for any term the result of the normalization procedure is uniquely determined. Given a congruence relation on terms presented by a finite set of equations $\mathcal{P}$, a *completion procedure* computes a complete TRS $\mathcal{R}$ for $\mathcal{P}$ that can decide $\mathcal{P}$-equivalences: two terms are equal modulo $\mathcal{P}$ if their respective $\mathcal{R}$-normal forms are equal. Such a completion procedure was discovered by Knuth and Bendix (1970).

A *term completion procedure* compiles on success a set of equations $\mathcal{P}$ into a complete TRS $\mathcal{R}$. It does so by repeatedly applying the inference rules in Figure 1 (Bachmair and Dershowitz, 1988) to a pair $(\mathcal{P}; \mathcal{R})$ of equations and rules. It succeeds if starting with $(\mathcal{P}; \emptyset)$ a pair $(\emptyset; \mathcal{R})$ can be derived such that $\mathcal{R}$ is complete.

In many interesting applications some binary operators are known to be both associative and commutative (AC). In these cases term rewriting and completion is performed on AC-equivalence classes of terms in order to avoid infinite rewrites. Technically this means that tests for equality, matches, and unifications, must be performed modulo AC. Ex-

```
$\mathcal{R} \leftarrow$ COMPLETE($\mathcal{P}$,$\succ$)

Inputs: a set of equations $\mathcal{P} = \{s_i = t_i \mid 1 \leq i \leq m\}$, and a term ordering $\succ$
Output: a complete TRS $\mathcal{R} = \{l_i \rightarrow r_i \mid 1 \leq i \leq n\}$

$\mathcal{R} := \emptyset$;
while $\mathcal{P} \neq \emptyset$ do
    (1) [Orient.] Select the best equation from $\mathcal{P}$; remove it from $\mathcal{P}$ and add it as a rule $l \rightarrow r$
        to $\mathcal{R}$, provided that $l \succ r$, else stop with failure;
    (2) [Extend.][a] If needed attach its extension rule to $l \rightarrow r$;
    (3) [Collapse.] for each $l' \rightarrow r' \in \mathcal{R}$ do
        if $l'$ is reducible then remove $l' \rightarrow r'$ from $\mathcal{R}$ and put $l' = r'$ back to $\mathcal{P}$;
    (4) [Compose.] for each $l' \rightarrow r' \in \mathcal{R}$ do normalize $r'$ w.r.t. $\mathcal{R}$;
    (5) [Deduce.] for each $l' \rightarrow r' \in \mathcal{R}$ do add critical pairs of $l \rightarrow r$ and $l' \rightarrow r'$ to $\mathcal{P}$;
    (6) [Simplify.] for each $s = t \in \mathcal{P}$ do normalize $s$ and $t$ w.r.t. $\mathcal{R}$;
    (7) [Delete.] for each $s = t \in \mathcal{P}$ do if $s = t$ is trivial then remove it from $\mathcal{P}$;
od
```

[a] This step is only needed in AC-completion.

Figure 2. Procedure *COMPLETE*.

tended completion procedures to deal with terms containing AC-symbols are described by Lankford and Ballantyne (1977), Peterson and Stickel (1981) and Jouannaud and Kirchner (1986). Besides the replacement of the basic operations by their AC-variants, two modifications to standard completion are necessary for AC-completion. First, AC-equal pairs may be *deleted* from $\mathcal{P}$ and second there is a new *extension* inference rule that essentially computes 'critical pairs' between a rule in $\mathcal{R}$ and equations specifying the AC-theory.

The inference rule characterization of Knuth–Bendix completion leaves many decisions open that determine how to perform the completion. In particular it does not prohibit *unfair* inference chains that delay crucial reductions for ever. A regime which fixes the order and the manner in which the inference rules are to be applied is called a *completion methodology.* Figure 2 shows a first abstraction of the one used in ReDuX and also in our parallelization experiments.

Note that COMPLETE fixes the strategy up to two decisions: the reduction strategy used and the definition of the 'best equation' in Step 1. If the reduction strategy is normalizing and every equation in $\mathcal{P}$ is eventually either deleted or considered for orientation then COMPLETE is fair.

Knuth–Bendix completion and AC-completion are used to compute a complete TRS. There is an alternative application of completion methods as proof procedures. In that case the completion procedure stops as soon as the derived TRS computes a common normal form for a pair of input terms (s, t) to be proven equal modulo the set of input equations, i.e., $s =_{\mathcal{P}_0} t$ if $(\mathcal{P}_0; \emptyset) \vdash^* (\mathcal{P}_i; \mathcal{R}_i)$ and $s \rightarrow^*_{\mathcal{R}_i} n \leftarrow^*_{\mathcal{R}_i} t$.

Unfailing completion based on ordered rewriting (Bachmair *et al.*, 1989) is such a completion procedure meant to prove the validity of an equation modulo an equational theory. Unfailing completion extends the standard Knuth–Bendix completion in that it allows us to reduce terms w.r.t. ordered instances of equations: Let $\succ$ be a term ordering and $a \leftrightarrow b$ be an equation, then s reduces to t by applying $a \leftrightarrow b$ if there is an instance $a\sigma \leftrightarrow b\sigma$ of $a \leftrightarrow b$ such that $a\sigma \succ b\sigma$ and $s \rightarrow_{\{a\sigma \rightarrow b\sigma\}} t$, or $b\sigma \succ a\sigma$ and $s \rightarrow_{\{b\sigma \rightarrow a\sigma\}} t$.

Consequently, unfailing completion does not fail if a non-orientable equation has been selected in the orientation step. Further, the equations that have been selected to define the reduction relation can be split into a set of orientable rules $\mathcal{R}$, and a set of non-orientable equations $\mathcal{E}$ of which only orientable instances may be used in reductions. Thus in the deduction step we must distinguish between the case where a rule $l \rightarrow r$ or an unorientable equation $s = t$ has been selected. In the first case, critical pairs must be computed between the rule $l \rightarrow r$ and both $\mathcal{R}$ and $\mathcal{E}$, and in the second case we have pairs between the equation $s = t$ and both $\mathcal{R}$ and $\mathcal{E}$. For more details on unfailing completion see Bachmair *et al.* (1989).

3. The Parallelization Philosophy

In the following, we list some key decisions in constructing PaReDuX, since they had a substantial influence on the shape of the parallel system: we always attempt to parallelize existing well proven sequential code before we radically alter the system. We program in C and parallelize our code by including calls to VS-threads. Thus our PaReDuX code is compatible with the existing sequential ReDuX code, and it contains many calls to the sequential system.

The VS-threads system supports a programming style that focusses on concurrent *procedures*, by making sure that they can be used very much like sequential procedures. They can be forked in parallel largely (but within reason) where program logic permits rather than where system load or application grain-size dictate. They can be called in sequential or parallel code, sequentially or in parallel. They can be put into libraries and linked and combined in the usual way, without worry of overloading the system.

Whenever possible we follow the *divide and conquer* paradigm when parallelizing code. Where ReDuX normalizes a list of terms in greedy fashion, PaReDuX splits the list and forks two threads with recursive calls to the reduction procedure. As an optimization, we usually do stop dividing when the list is short. The point is that with the low overhead of virtual threads it is often possible to determine by intuition rather than extensive tests when short is short enough. Some of this intuition must be acquired for the parallel case, so that testing is necessary in a first phase for a first application. We built this intuition for plain completion first (Bündgen *et al.*, 1994a), and then used it for AC and unfailing completion.

An abstract rendition of our parallelization methodology is roughly as follows: a given list C of uniform data, e.g., a list of critical pairs or rewrite rules, is either processed sequentially if it is too short and the work it represents falls below a predetermined grain-size, or it is split into two equal parts C_1, C_2 with $C = C_1 \circ C_2$. In the latter case, one recursive call is forked in parallel for the list C_2, and one recursive call is done by the parent thread itself for the list C_1. After computing the result for C_1, the result for C_2 is joined and both results are merged.

Together with the VS-threads environment, this divide-and-conquer approach to parallelization has a most desirable effect. Tasks generated early on have large grain-sizes and these are the tasks that are stolen by initially idle workers. Tasks generated later on have smaller grain-sizes, but those tasks are likely to be executed as procedure calls, with a substantially lower overhead. Thus we enjoy a dynamic adjustment of grain-size, with mostly large-grain VS-threads executing in parallel when there is much work to do, and fine-grain VS-threads executing in parallel only when workers would remain idle otherwise. Note well that task scheduling is done automatically within VS-threads and remains transparent to the application programmer.

4. Parallel Completion Schemes

The inference rule characterization of Knuth–Bendix completion is already completely parallel, because the rules are logically independent. However, a straightforward parallel implementation would also be horrendously inefficient, because most work would be redundant. The sequential completion challenge is to organize the completion steps in such a way that little redundant work is performed. The parallel completion challenge is to improve upon the best sequential algorithm in such a way that the parallel algorithm defaults to the best sequential one when there is only one processor, and that it achieves good speed-ups as processors are added.

The procedure COMPLETE can be parallelized on the level of the outer while loop (adding equations concurrently), on the level of any of the inner foreach loops, or below (e.g., by parallelizing the reduction of a single term). Several independent inner loops may also be computed in parallel. As a general rule of thumb, the greater the grain-size of parallel tasks, the greater the efficiency of the parallelization. It is therefore clear that, given a choice, the outer loop rather than the inner loops should be parallelized; this has been argued by Slaney and Lusk (1990).

When new rules are selected concurrently by several tasks in the outer loop, the completion strategy depends on the order in which the tasks are scheduled. In effect, a new *parallel non-deterministic search strategy* for good rules is executed. This may lead to large super-linear speed-ups, but in general it is unpredictable and irreproducible. In any case, a proof is required that the new strategy is fair in all cases. In contrast, inner loop parallelizations are *strategy compliant* because they do not change the completion strategy. However, they maintain a synchronization point, and hence a sequential bottleneck, in Step 1 which can only be executed by a single thread.

The *most coarse grained* parallelization scheme exploits or-parallelism of non-deterministic procedures: It starts several completion processes with different strategies in parallel. Intermediate results may be communicated. Such a scheme is appropriate for distributed computation and has been presented by Avenhaus and Denzinger (1993), where each process uses a different selection function in Step 1, and a referee process selects the best intermediate results and communicates them to all workers.

A *less coarse grained* scheme parallelizes the outer loop of COMPLETE, letting several workers perform concurrent completion cycles on common $\mathcal{P}$ and $\mathcal{R}$, but such that each worker computes a different fragment of $\mathcal{P}$. But since $\mathcal{P}$ and $\mathcal{R}$ of round $i + 1$ depend on the respective sets of round i, synchronization overhead may be high. In particular the need for backward subsumption may spoil potential parallelization gains. Therefore, in distributed implementations, various schemes are employed to relax the consistency assumptions between the distributed copies of $\mathcal{P}$ and $\mathcal{R}$, so that computation can be overlapped with communication. Two approaches for the polynomial case are to let each worker select the best rule independently and then communicate it to all others (Chakrabarti and Yelick, 1993), or to let a central referee select the best of the workers' choices and communicate it back to the workers (Sawada *et al.*, 1994). Bonacina and Hsiang (1993), and Bonacina and McCune (1994), present distributed parallelizations of OTTER, including Knuth–Bendix completion, based on the scheme of Clause

Diffusion, which falls into this category. In Clause Diffusion, a worker either retains a newly derived clause or sends it off to another worker. Using the switches of OTTER, the workers can even be instructed to follow different strategies (but without refereeing). In the shared memory polynomial case, this scheme was used by Vidal (1990). It has also been applied to general closure computations by Slaney and Lusk (1990), and Lusk and McCune (1990) report a shared memory parallelization of OTTER including Knuth–Bendix completion.

By a *medium grained* parallelization, we understand the parallelization of the inner loops of COMPLETE†. In particular, lists of terms (equations) may be normalized in parallel; in the same way the critical pairs of a rule and a list of rules may be computed in parallel. Depending on the hardware and the problem, *grain-size decisions* must be made fixing the minimal number of terms to be normalized and the minimal number of superpositions to be computed in a single thread of control. Using VS-threads these parameters may be controlled (to some extent) automatically by the machine load.

By a *fine grained* parallelization, we understand the parallelization of operations inside the inner lops, such as parallel reduction of a single term or polynomial. In the term completion case, Dershowitz and Lindenstrauss (1990) proposed the parallel reduction of different subterms. In the polynomial case a parallel reduction algorithm was developed by Melenk and Neun (1989) on a vector computer and it was later multi-threaded by Schwab (1992).

5. Parallel Completion in PaReDuX

We parallelized the Knuth–Bendix completion procedure, the Peterson–Stickel completion procedure for TRS with AC-operators, and the unfailing completion procedure of the ReDuX system. In all three cases we exploited medium grained parallelism by parallelizing the inner loops of the completion procedure (cf. Figure 2). This leaves us with a sequential synchronization phase in each cycle of the outer loop comprising the orientation, extension, collapse, and composition steps.

The methodology of COMPLETE is certainly not the only possible choice [cf. Huet (1981)]. The reasons why we decided to choose a parallelization based on this methodology are manifold. First it turned out that the time spent in the synchronization phases makes up only a small portion of the total completion time. Second, our completion scheme keeps all critical pairs in normalized form. This allows us to avoid unnecessary reductions (see below) and to use better heuristics in selecting the best equation in the orientation step. A further advantage of our approach is that it allows us to keep the overall completion strategy fixed by fixing the selection function and the normalization strategy only. This is important for several reasons: the measurements are reliable, i.e., timings do not change if an experiment is repeated. Thus improvements during the development of the parallel program can be contributed to better parallelization techniques rather than to hazardous strategy effects. Similar arguments hold for the search of good non-problem specific grain size settings. Comparisons between sequential and parallel code is fair if both use the same strategy. For fixed strategies parallelization is scalable. That is, for problems that contain enough parallelism, timings improve if processors are added. In addition the correctness proof of a parallelization comes for free if it uses the

[†] In our parallelization of TC (Bündgen *et al.*, 1994b) we characterized this as *fine grained*, because for TC this is the finest level profitable in our system.

same strategy as a correct sequential program. A last very practical argument in favor of our completion scheme is that it minimizes ordering decisions which is important for research in completing new equational theories. In these cases an appropriate term ordering is rarely known *a priori* and completion is run in semi-automatic mode where ordering decisions are made interactively, controlled by the intuition of the researcher. In this setting it is of utmost importance that repeated experiments yield the same results, so that a wrong decision at one point can be corrected in a new trial and that the (human idle) time between two ordering decisions is as short as possible.

Before we describe our parallelization of COMPLETE, we must describe the completion procedure implemented in ReDuX more precisely. As mentioned before, COMPLETE maintains all rules in $\mathcal{R}$ and equations in $\mathcal{P}$ in fully (inter)reduced form. Thus *collapse*, *compose*, and *simplify*, apply only to those rules and old equations which are reducible by the newly oriented rule $l \to r$. Also, *delete* need only be tried on newly simplified equations. A further improvement which is also included in ReDuX and PaReDuX is a special case of the subconnectedness criterion (Küchlin, 1985, 1986) which allows to remove all equations in $\mathcal{P}$ which were derived from a collapsed rule.

Let us now investigate the synchronization requirements for the access to $\mathcal{P}$ and $\mathcal{R}$ in the steps of COMPLETE. Steps 1 and 3 modify both $\mathcal{P}$ and $\mathcal{R}$. Steps 2 and 4 read and modify $\mathcal{R}$. Steps 5–7 modify $\mathcal{P}$ while requiring read-only access to $\mathcal{R}$. Further, note that *simplification* and *deletion* of old equations is independent of *deduction*, *simplification*, and *deletion*, of new critical pairs. Therefore these two tasks can be performed in parallel. Figure 3 gives an outline of the resulting general parallel completion scheme realized in PaReDuX. Besides computing independent loops of COMPLETE in parallel, each of the loops depicted in doubly framed boxes is parallelized using the divide and conquer scheme described in Section 3. To keep the tasks as coarse-grained as possible the normalization and deletion procedures are called within the reducibility tests and critical pair computations, respectively. Thus the parallelization of the reduction tests and critical pair computations act as filters for the parallelization of the normalization and deletion procedures.

This kind of parallelization requires that all parallel threads have simultaneous access to $\mathcal{R}$. In Section 6, we will describe data structures for terms that support sharing $\mathcal{R}$ without the need of copying rules.

Since most of the operations depend on $\mathcal{R}$, modifications of $\mathcal{R}$ have a strong limiting influence on the parallelization of COMPLETE. The problem caused by changing and deleting rules (Steps 3 and 4) has become known as the *backward subsumption* (Slaney and Lusk, 1990) or *backward contraction* problem (Bonacina and Hsiang, 1993). In general, there are two ways to overcome these problems: we may permit reductions with outdated copies of $\mathcal{R}$, which is correct but may possibly be inefficient, or we must synchronize accesses to $\mathcal{R}$ by locking. As we will see in Section 7, the time spent in steps 1–4 makes up less than 10% of the whole completion time—in most cases even less than 5%. Therefore it is not worth putting much effort into parallelizing this portion of the code.

All in all, the sequential ReDuX code had to be modified in the following ways:

1. Iterative loops had to be reorganized as divide and conquer style procedures,
2. VS-thread-fork and VS-thread-join instructions had to be inserted and
3. the term data structure had to be modified to allow for rule sharing in reductions and critical pair computations (cf. Section 6).

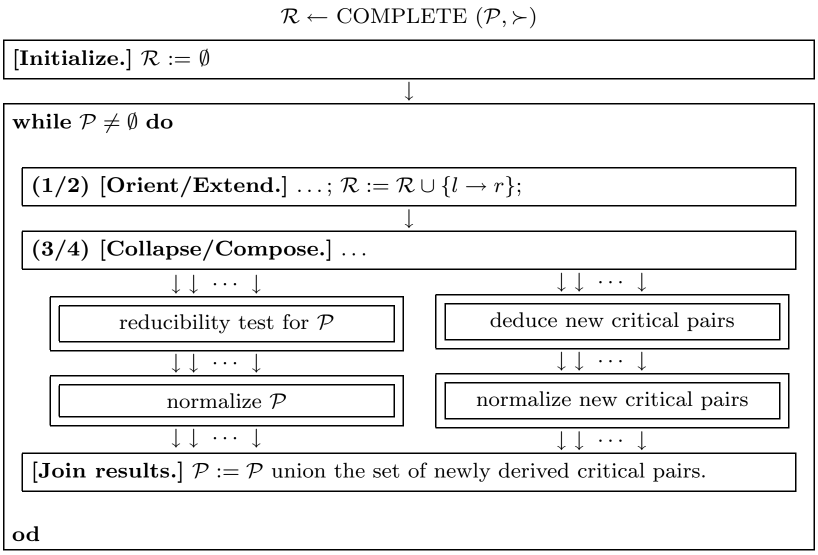
$\mathcal{R} \leftarrow$ COMPLETE $(\mathcal{P}, \succ)$
[Initialize.] $\mathcal{R} := \emptyset$
while $\mathcal{P} \neq \emptyset$ **do**
(1/2) [Orient/Extend.] $\ldots$; $\mathcal{R} := \mathcal{R} \cup \{l \to r\}$;
(3/4) [Collapse/Compose.] $\ldots$
reducibility test for $\mathcal{P}$
deduce new critical pairs
normalize $\mathcal{P}$
normalize new critical pairs
[Join results.] $\mathcal{P} := \mathcal{P}$ union the set of newly derived critical pairs.
od

Figure 3. Overall structure of parallel completion in PaReDuX.

The exact order in which equations are turned into rules is known to be of the utmost importance for both term completion procedures and for Buchberger's algorithm. This is called the *completion strategy* and in our case it is encapsulated in the exact method after which the *best* of the equations is determined. Minute changes in this strategy can have huge effects (positive or negative) on the duration of completion. For example, we must find the best of the equations (critical pairs) in $\mathcal{P}$ w.r.t. an ordering which is total on the equations. Simple quasi orderings (like those based on counting the symbols in each pair) are not sufficient: changing the ordering of pairs in $\mathcal{P}$ which are equivalent w.r.t. the quasi-ordering (e.g., have the same number of symbols) may result in a different completion behavior. Our parallel completion procedures are designed to ensure that the priority queues containing $\mathcal{P}$ at each cycle exactly correspond to the queues in the respective cycle of the sequential completion.

5.1. parallel Knuth–Bendix completion

The parallelization of the plain Knuth–Bendix completion procedure follows exactly the scheme described above. Depending on our experiments, the parallelized portion of the code took 91–98% of the total completion time. Our experiments indicate that for problems of a similar class the sequential bottleneck decreases with the size of the problem. We obtained speed-ups of 2.4–3.2 on four processors[†].

[†] For problems that take more than 100s sequentially!

5.2. parallel AC completion

Even though the differences between standard completion and AC completion seem moderate, their effect is tremendous from a complexity point of view. Searching for a most general AC-unifier of two terms results in general in a non-singleton set of substitutions. Thus more than one critical pair can be computed for each two rules with fixed superposition position. In addition, standard matches and unification can be computed in linear time. To test for AC-matchability or AC-unifiability however is NP-complete, and to compute a complete set of AC-unifiers is even of doubly exponential cost. This rise in complexity had to be taken into account when deciding on the parallelization grain size.

Parallelization of normalization tasks on the critical pair queue level turned out to be too coarse. We had to allow for normalizing the individual terms of an equation in parallel and we even allowed for fine grained parallelism, normalizing single terms in parallel.

For AC completion the sequential bottleneck was in general less than 2%. We obtained speed-ups of 2.4–3.5 on four processors[†].

5.3. parallel unfailing completion

Ordered rewriting in the unfailing completion procedure is either done w.r.t. a TRS $\mathcal{R}$ or a set of unorientable equations $\mathcal{E}$. Thus in the orientation step either $\mathcal{R}$ or $\mathcal{E}$ is updated. The critical pairs between the new rule or equation on the one hand, and $\mathcal{R}$ or $\mathcal{E}$ on the other hand, can be computed and normalized independently.

The computation of ordered reductions and critical pairs for ordered rewriting is more expensive than the corresponding operations for plain Knuth–Bendix completion because the application of equations in $\mathcal{E}$ involves a term comparison w.r.t. $\succ$, and, by symmetry, equations may be applied in both directions. The complexity is, however, much lower than the complexity of the corresponding AC-operations. Clearly, the minimal parallelization grain sizes for operations based on reductions by equations in $\mathcal{E}$ must be lower than the corresponding operations based on rewrite rules in $\mathcal{R}$.

Unfailing completion is mainly intended to prove a single equation. Thus it may be an overkill to keep all critical pairs normalized. Keeping this consideration into account we implemented a second variant of the unfailing completion procedure that normalizes a critical pair only immediately after creation and just before selection in the orientation step. This eliminates the left column of tasks in Figure 3.

Unless unfailing completion is used to complete a specification, both procedures spend less than 4% of the total completion time in the sequential bottleneck, with slight advantages for the fully normalizing procedure. Surprisingly, none of the two procedures is uniformly better than the other. With both we obtained speed-ups of 3.0–3.8 on four processors[†].

[†] For problems that take more than 100s sequentially!

6. Parallel Data Structures in PaReDuX

6.1. parallel Knuth–Bendix completion

In this section, we first explain the most important aspects of the data structures of the sequential ReDuX System which were first designed by Küchlin (1982a) and further extended during the development of ReDuX. Then we describe the modifications necessary for the parallel implementation.

ReDuX terms are represented as directed acyclic graphs (DAGs) with unique representation of variables (and constants). The representation of variables and operators is based on scoped property lists. The 'most local' properties of an object occur at the front of the list, and the 'global' properties are at its end. Therefore the argument list of an operator occurrence is stored in the first field of the list. The symbol (e.g., the operator, constant, variable), together with all signature information, is stored in later fields. Thus the signature information of operators, constants, and variables, can be shared by all occurrences of these symbols. Likewise, each *incarnation of a variable* (i.e., a variable *occurring* in a rule or term) starts with a *binding field* representing the binding property. This field indicates whether the variable is currently bound (by a substitution), and if this is the case the field points to the bound term. This accounts for an implicit representation of substitutions and allows for efficient equality tests, matching, and unification, and is particularly well-suited for efficient normalizations (Küchlin, 1982b; Stickel, 1983). In particular, matches and unifications do not consume memory.

Data structures for parallel programs should support easy access to shared resources from several parallel tasks. The access to these resources should be granted with as little synchronization overhead as possible. The solution to this problem is very easy if we can enforce a functional programming discipline which does not allow the modification of (shared) input parameters.

During the parallel completion procedure described in the last section the rule set $\mathcal{R}$ is shared by *all* parallel threads. This creates a problem, because efficient algorithms for the base operations like matching, unification, and subterm replacement, temporarily modify the rules as they are applied to terms, by changing the binding property of variables. The tasks we want to perform in parallel are normalizations and critical pair computations. In the sequential normalization and critical pair computation procedures all side-effects are hidden to the outside by undoing all temporary substitutions, and thus these procedures have a functional behavior. However, in a parallel environment also temporary side-effects which modify global shared memory must be hidden from other threads with access to the same global data.

In particular, we must change the representation of substituted variables. This is realized by introducing an additional level of indirection for variable bindings: instead of storing in the binding field a pointer to a term, we now store a variable specific index into a *binding table* which contains the pointer. The table associates a variable (index) with (a pointer to) a term. The table is now a parameter to the matching and unification procedures. It is realized as a local array declared in the normalization and critical pair computation procedures calling Match or Unify. Thus the table is allocated on the C stack in private thread memory.

Figure 4 depicts the situation where in thread j the variable y (with index 2) of the term $t = f(f(x,y),g(y))$ is bound to $f(v,w)$, and in thread k the variable y of the subterm $s = g(y)$ of the same term $f(f(x,y),g(y))$ is bound to $g(z)$ simultaneously. Instead of

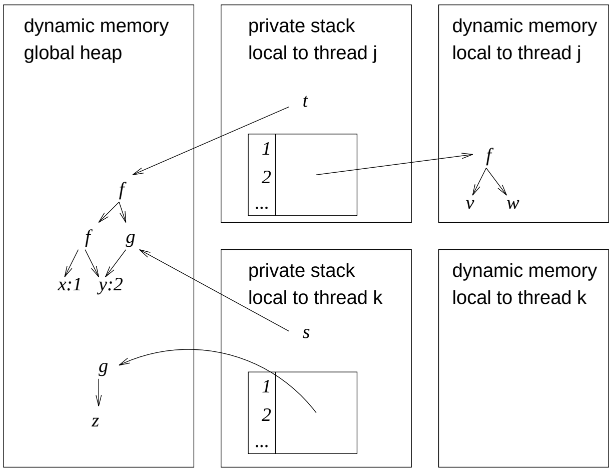

Figure 4. Variable bindings in PaReDuX.

a direct pointer from y to the bound term $f(v, w)$ (or $g(z)$ resp.) as it is conveniently used in the sequential implementation, we now consider variable bindings relative to the context in which the variable is used.

With this technique we can avoid copying global data, provided only one global object per parallel thread is used (substituted) at a time. In case a thread accesses and modifies the variable bindings of more than one global item at a time, all but one of the items must be 'colored' in order to associate one binding table (of corresponding 'color') with each item. Since 'coloring' is a real change of global data, items which are to be colored must be copied.

Luckily, during Knuth–Bendix completion this situation occurs only in the critical pair computation process when the subterms of two rules are to be unified. We decided to always work with a (single) colored copy of the newly oriented rule $l \to r$, and with the original uncolored rules in $\mathcal{R}$ including the uncolored original $l \to r$. Note that copying a newly oriented rule does not lead to extra copy-overhead compared to the sequential procedure because it must be copied anyway to obtain the critical pairs of the rule and itself. Again both the colored and the uncolored copy of $l \to r$ can be shared by all threads computing the critical pairs deduced from this rule.

Using the modifications described above, we could reuse all software for the basic operations from the sequential system after changing the macros to access the variable bindings.

6.2. parallel AC completion

As opposed to the standard case, the matching and unification procedures compute in general more than one substitution in AC theories. Even for AC matching, where only one matcher is needed to perform a reduction, it may be necessary to compute several substitutions as intermediate results for non-linear matching terms. Therefore a new data structure to represent substitutions, (an association list of variables and their substitution values) had to be introduced into ReDuX. However the mechanism to apply (and undo) a substitution [by (un)setting a pointer in the binding field of a variable] remained unmodified because only one binding at a time is needed in the sequential procedures. Thus the parallel binding mechanism described in the last section could be carried over to the AC case.

A major difference between standard matching and unification and the respective AC operations is the fact that AC matching and AC unification require to construct new objects (substitutions, terms) that must be allocated in the dynamic memory. Using VS-threads, new objects can be constructed independently by several VS-threads in parallel since each VS-thread has a private portion of the heap allotted to it (Cf. Figure 4 where the term $f(v, w)$ is stored in the private heap segment of thread j.). Note that each local heap segment may contain references into the global heap of a VS-thread but not vice-versa, and the local heap of a VS-thread is global to all its child threads. Potential read or write conflicts are avoided by the functional programming style at the level of parallelization. Thus the problem of constructing new objects is solved automatically by the VS-thread system.

A second difference between standard unification and AC unification is that the number of variables needed in an AC unification procedure is not known *a priori* on entrance into the AC unification procedure. Therefore we had to provide for dynamically extensible binding tables. No further changes were necessary.

6.3. parallel unfailing completion

The parallel data structures designed for Knuth–Bendix completion could be reused without change in the unfailing completion procedure.

7. Experiments with PaReDuX

7.1. hard- and software

We have implemented our parallel completion procedures on two different platforms:

1. Solbourne 5/704, 48 Mbyte of main memory, four 33 MHz SPARC processors, common bus, SunOS 4.1.1C, threads provided by the user level PCR environment.
2. Sun SPARCstation 10, 64 Mbyte of main memory, two 66 MHz ROSS hyperSPARC double processor modules (4 processors total), 2×512 Kbyte secondary level cache, OS Solaris 2.4, Solaris 2.4 threads (4 LWP's, max. 12 Solaris threads).

The experiments were run on platform 1 for Knuth–Bendix completion and AC completion, and on platform 2 for unfailing completion. All parallel experiments were run with the VS-thread system (Küchlin and Ward, 1992). We ported ReDuX to the SACLIB (Buchberger *et al.*, 1993) and PARSAC-2 environments. ReDuX was translated to C using the ALDES-to-C Compiler by Sperber (1994).

All runtimes given in the following are mean values of three instances of the same experiments measured in wall-clock units. All experiments were performed with a minimum amount of 4 000 000 SAC-2 list cells occupying 32 MByte of memory. Notice that in our parallelization approach the sequential and parallel programs perform the same work and use the same number of list cells.

Table 1. General statistics for Knuth–Bendix completion.

Knuth–Bendix completion (platform 1)					
TRS	cyc.	max. $\lvert\mathcal{P}\rvert$	$\Sigma\lvert\mathcal{P}\rvert$	max. $\lvert\mathcal{R}\rvert$	final $\lvert\mathcal{R}\rvert$
Z_{22W}	507	2519	5696	345	188
Z_{22T}	576	1461	13 643	80	27
Z_{22}	174	983	2745	45	10
M_{15}	547	12 367	17 425	529	19
M_{13}	423	8301	11 775	407	17
M_{12}	367	6655	9478	352	16
M_{10}	267	4047	5820	254	14
P_8	1305	13 392	16 263	1202	79
P_7	617	3936	5283	546	55
P_6	369	1656	2457	314	43
P_5	269	960	1542	222	37
D_{64}	160	323	760	111	87
D_{32}	94	77	326	64	55
D_{16}	62	34	190	42	39
Q_4	55	56	178	44	44

7.2. parallel Knuth–Bendix completion

To evaluate our parallel implementation we used a set of input specifications that were meant to cover a wide range of specification types. The full details of those specifications are given in the appendix. Here we just give an informal description of the specifications. $D_n, Q_4, Z_{22}, Z_{22W}$ and Z_{22T} specify finitely presented groups. The term rewriting systems can be classified into ordinary or ‘true’ term rewriting systems with at least one binary operator and possibly non-linear rules $(D_n, M_n, P_n, Q_4, Z_{22W})$ and into simulated string rewriting systems (Z_{22}, Z_{22T}). Most of our examples are so-called parametric or scalable data-types (D_n, P_n, M_n). Note, that Z_{22} and Z_{22W} specify the same group.

The sequential completion behavior of each specification is presented in Table 1. The second column contains the number of outer loop cycles needed to complete the specification, the third column shows the maximal length of the critical pair queue during the completion and column four sums up the lengths of the critical pair queues for all cycles. Column five shows the maximal size of the TRS during the completion and the last column shows its final size. These figures give a rough estimate on the potential parallelism inherent to each completion task. E.g., if there are only a few critical pairs and rules created over many cycles then our approach may not exploit much parallelism.

Our implementation of PaReDuX is parameterized by several parameters intended to influence the parallelization behavior of the program. Three of those parameters are *grain size parameters*: NF is the maximal length of a critical pair list not worth parallelizing, CL is the maximal length of a critical pair list not worth to be checked for reducibility and subconnectedness criterion application in parallel and AX determines the maximal size of a rule set to be used in a non-parallelized critical pair computation procedure. Remember that both CL and AX act as filters for NF. The last parameter, DTH, sets

Table 2. Grain size parameter analysis for Z_{22W}.

Knuth–Bendix completion (platform 1)						
Grain size parameter				Parallel on 4 proc.		VS-
NF	CL	AX	DTH	time [s]	speed-up	threads
3	128	8	7	461.1	3.1	24 005
6	128	8	7	476.1	3.0	21 688
9	128	8	7	481.0	3.0	21 314
12	128	8	7	495.9	2.9	21 207
3	64	8	7	465.9	3.1	28 656
3	96	8	7	469.9	3.1	25 291
3	160	8	7	469.3	3.1	22 869
3	128	4	7	467.3	3.1	38 215
3	128	6	7	470.6	3.0	28 119
3	128	10	7	474.0	3.0	20 667
3	128	8	3	511.9	2.8	6566
3	128	8	5	474.4	3.0	18 940
3	128	8	9	463.1	3.1	24 193

an upper bound to the number of levels to be parallelized using the divide and conquer scheme.

Given that degree of freedom in tailoring our parallel completion procedure the question is to find an optimal set of assignments to the parallelization parameters. This requires very extensive measuring experiments. Table 2 illustrates a small excerpt of our efforts to determine optimal or at least good parameter settings. For several combinations of parameter settings the times and speed-ups measured for completing Z_{22W} on four processors are given. The last row depicts the number of virtual threads created during the whole completion. The results for Z_{22W} are typical in that they show that our approach to parallelizing completion is rather robust w.r.t. variations of grain sizes.

Only too small a bound for the divide-and-conquer approach leads to a significant loss of speed-ups, since without generating enough threads of control there is not enough parallelism to keep all processors busy. Thus divide-and-conquer parallel programming with lazy task generation appears to be a suitable programming model for parallel symbolic computation because it is (within broad margins) efficient, robust and scalable. NF is the only grain size parameter that seems to influence the runtimes. The variation of CL and AX affects the runtimes less significantly. The reason for this is that CL and AX act as filters for the normal form algorithm.

Many experiments with different specifications and a lot of parameter combinations have shown that the combination NF = 3, CL = 128, AX = 8 and DTH = 7 performs very well and is the best choice for many term rewriting systems for parallel Knuth–Bendix completion (Bündgen *et al.*, 1994a). Table 3 contains the runtimes and the percentage of parallelized code needed by the sequential program in column 2, the speed-ups of the parallelized program on one to four processors w.r.t. the sequential / 1 processor implementation in the next four columns. The last column contains the number of VS-threads. All experiments were run using the parameter settings stated above.

Table 3. Results for parallel Knuth–Bendix completion.

Knuth–Bendix completion (platform 1)						
	Sequential	Parallel speed-ups on				VS-
TRS	time [s]/pp	1 proc.	2 proc.	3 proc.	4 proc.	threads
Z_{22W}	1432.6/96%	1.0/1.0	1.7/1.8	2.5/2.5	3.1/3.2	24 005
Z_{22T}	3037.4/98%	1.0/1.0	1.8/1.8	2.5/2.6	3.2/3.3	20 829
Z_{22}	83.3/96%	1.0/1.0	1.7/1.7	2.3/2.4	2.8/2.9	3112
M_{15}	753.8/91%	1.0/1.0	1.6/1.7	2.1/2.2	2.6/2.8	54 806
M_{13}	404.5/91%	1.0/1.0	1.6/1.7	2.2/2.3	2.7/2.8	31 233
M_{12}	276.0/91%	1.0/1.0	1.6/1.7	2.1/2.2	2.5/2.7	23 598
M_{10}	125.2/90%	0.9/1.0	1.6/1.7	2.0/2.2	2.4/2.6	11 209
P_8	2994.9/94%	1.0/1.0	1.6/1.7	2.2/2.3	2.7/2.7	158 710
P_7	423.7/93%	1.0/1.0	1.6/1.7	2.2/2.3	2.7/2.7	34 565
P_6	111.8/92%	0.9/1.0	1.5/1.6	2.0/2.1	2.4/2.5	13 761
P_5	51.7/91%	0.9/1.0	1.5/1.6	2.0/2.1	2.2/2.4	7236
D_{64}	2675.0/98%	1.0/1.0	1.7/1.7	2.3/2.3	2.7/2.7	4010
D_{32}	208.2/97%	1.0/1.0	1.6/1.7	2.2/2.3	2.7/2.7	1389
D_{16}	25.3/96%	0.9/1.0	1.6/1.7	2.1/2.3	2.5/2.7	637
Q_4	6.4/95%	0.9/1.0	1.4/1.6	1.8/2.0	2.0/2.2	446

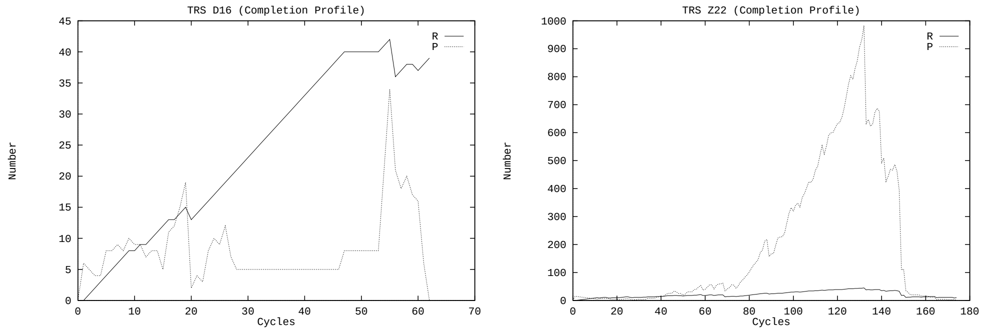

Figure 5. TRS D_{16} and TRS Z_{22}.

There is an up to 10% penalty for using the parallel implementation. The overall speed-ups for the Knuth–Bendix completion are 1.4–1.8 for two, 1.8–2.5 for three and 2.0–3.2 for four processors compared to the sequential runtimes. One reason for this behavior is given in the completion profiles for our examples which show that normally only a rather small portion of the completion procedure provides the potential for good inner loop parallelization. Figure 5 contains completion profiles for D_{16} and Z_{22}. P (R) denotes the length of the list of critical pairs $\mathcal{P}$ (set of rewrite rules $\mathcal{R}$) during the completion at a given cycle of the outer loop.

Another interesting aspect of our parallel completion procedures is illustrated by the analysis for our parametric term rewriting systems in dependence of the parameter as shown in Figure 6. This figure presents a run time and efficiency analysis for P_n depending of the parameter n. The runtime plots contain the sequential runtime labeled by (s)

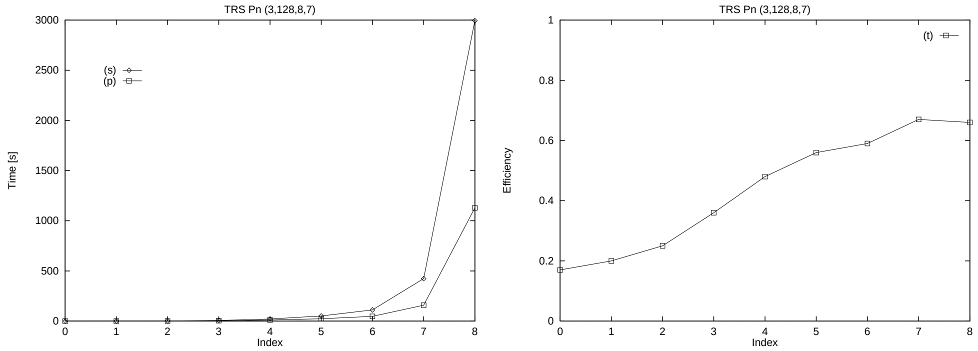

Figure 6. TRS P_n, $n = 0, \ldots, 8$.

Table 4. General statistics for AC completion.

AC completion (platform 1)											
		max.		max.	final	normalization		matching		unification	
TRS	cyc.	$\lvert\mathcal{P}\rvert$	$\Sigma\lvert\mathcal{P}\rvert$	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{R}\rvert$	#	%	#	%	#	%
AGR	7	63	92	5	5	871	58	12 596	49	80	35
DLT	6	43	46	4	4	844	75	10 434	70	76	16
RX	26	110	251	21	21	2680	76	74 964	63	1522	17
BRG	64	741	1678	63	63	25 557	91	4 465 983	77	12 149	5
R_3	18	103	220	13	13	1752	79	62 202	67	576	14
R_7	17	97	224	13	13	1797	93	84 188	89	542	5
R_{23}	33	113	482	25	18	3723	87	220 849	77	2202	8
R_{35}	30	144	461	23	18	3382	92	230 565	85	1866	5
R_{235}	58	512	1169	41	24	8694	95	1 129 449	93	7398	2
AZ_{22}	51	461	1015	38	25	7868	78	720 366	67	5288	14
Z_{22}	174	982	2745	45	10	9906	66	1 561 258	40	37 904	1

and the runtime on four processors labeled by (p). The efficiency plots for P_n show an increasing efficiency w.r.t. a growing parameter n. The results suggest that our speed-ups scale with the size of the input problem.

7.3. PARALLEL AC COMPLETION

Our parallel AC-completion was evaluated using the following input specifications: AGR presents the free Abelian group. DLT is the free distributive lattice. RX is a many sorted specification of multivariate polynomials over commutative rings. BRG defines a Boolean ring together with six atoms denoting $a > 0$, $a < 0$, $a = 0$, $a \geq 0$, $a \leq 0$ and $a \neq 0$. R_3, R_7, R_{23}, R_{35}, and R_{235}, are finitely presented commutative rings, respectively specifying $\{\frac{1}{3}\}$, $\{\frac{1}{7}\}$, $\{\frac{1}{2}, \frac{1}{3}\}$, $\{\frac{1}{3}, \frac{1}{5}\}$ and $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{5}\}$ adjoined to the integers. AZ_{22} is Z_{22W} with the additional information that the binary operator is associative and commutative. Z_{22} has already be used with the Knuth–Bendix completion. It does not contain any AC-operator. For more details including the term orderings used see the appendix.

In analogy to Tables 1 and 3, the left half of Table 4 characterizes the sequential

Table 5. Results for parallel AC completion.

AC completion (platform 1)						
	Sequential	Parallel speed-ups on				VS-
TRS	time [s]/pp	1 proc.	2 proc.	3 proc.	4 proc.	threads
AGR	8.9/99%	0.8/1.0	1.2/1.5	1.4/1.7	1.5/1.7	2702
DLT	10.0/99%	0.9/1.0	1.4/1.7	1.8/2.1	1.9/2.2	3966
RX	43.7/99%	0.9/1.0	1.5/1.8	1.9/2.3	2.1/2.4	9480
BRG	1598.3/99%	1.0/1.0	1.9/2.0	2.8/2.9	3.5/3.6	101 222
R_3	31.7/99%	0.9/1.0	1.4/1.7	1.9/2.1	2.2/2.5	6588
R_7	194.9/99%	0.9/1.0	1.7/2.0	2.5/2.8	3.0/3.3	17 934
R_{23}	117.6/99%	0.9/1.0	1.7/1.9	2.4/2.6	2.8/3.1	16 094
R_{35}	214.7/99%	0.9/1.0	1.8/2.0	2.5/2.7	3.0/3.3	20 006
R_{235}	1923.2/99%	0.9/1.0	1.6/1.8	2.3/2.5	2.4/2.7	65 165
AZ_{22}	232.5/98%	0.9/1.0	1.7/1.9	2.5/2.7	3.0/3.3	47 412
Z_{22}	138.7/97%	0.9/1.0	1.7/1.9	2.3/2.6	2.8/3.1	19 048

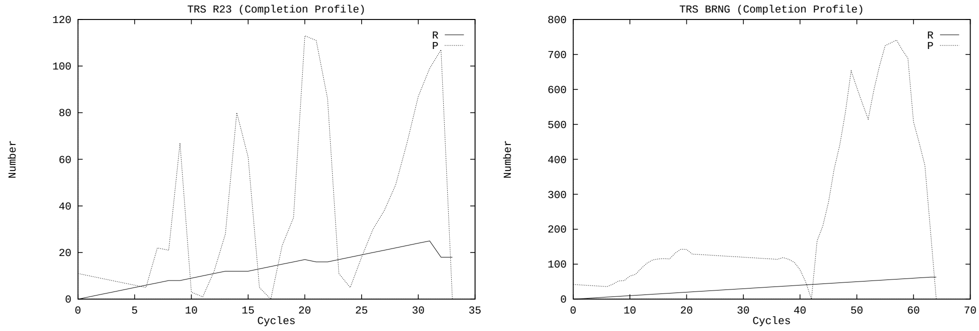

Figure 7. Completion profile of TRS R_{23} and TRS BRG.

completions and Table 5 summarizes the results of the parallel experiments[†]. For the AC experiments the purely sequential part of the code comprises only 1–3% of the completion time. Thus according to Amdahl's law, the speed-ups for four processors are theoretically limited to 3.6–3.9. The overall speed-ups for the AC completion are 1.2–1.9 for two, 1.4–2.8 for three and 1.5–3.5 for four processors compared to the sequential runtimes. Speed-ups are better for the larger examples. Some of the speed-ups of our parallel completion were not as good as expected. In particular, the speed-up of R_{235} is not very high considering its long sequential runtime and large parallelizable part. One reason for this may be seen in the completion profiles, which, like those in Figure 7, always exhibit phases with a dearth of parallelism. A probably more serious problem is the extreme irregularity in the costs of the operations to be computed in parallel. Therefore we want to analyse the behavior and costs of the key procedures of the AC-completion. The right half of Table 4 presents statistics about the number of normalizations, matchings

[†] The experiments include parallel term normalization after scheme (1) described later in this section.

Table 6. Distribution of costs for AC normalization tasks.

AC completion of R_{235} (platform 1)										
	Number of normalization tasks and % of total completion time									
Cycle	0.0–0.1 [s]		0.1–1.0 [s]		1,0–10 [s]		10–100 [s]		> 100 [s]	
...										
33	123	0%	6	0%	0	0%	0	0%	0	0%
34	24	0%	16	0%	0	0%	0	0%	0	0%
35	20	0%	12	0%	1	0%	1	0%	1	7%
36	207	0%	57	0%	17	1%	5	22%	0	0%
...										
Σ	6504	10%	2039	26%	131	14%	19	35%	1	7%

and unifications performed during the AC completion process. Moreover, it contains the percentage of the sequential runtime spent in these operations.

The percentage of time spent in matching (and thus in normalization) is always significantly greater than in unification. With only one exception, unification consumes less than 20% of the AC completion time. Most of the time (up to 95%!) is spent for normalizing terms. Note that most matches belong to the normalizations. There are only a few independent reducibility tests. Only Z_{22}, which does not contain AC operators, spends more than 10% of its completion time outside the listed procedures. The costs of single normalizations may vary largely. In extreme cases a single normalization may outweigh all other normalizations in the same completion cycle. Such a case is documented in Table 6 that shows that there are a few extremely time consuming normalizations during the completion of R_{235}. The table lists for a few cycles how many normalizations finish within certain intervals of time. In addition, for each time interval, the percentage of the completion time needed by all normalizations within that interval is given.

These findings encouraged us to investigate the parallelization of normalization of critical pairs beyond normalizing one term per thread.

7.4. parallel AC normalization

We tried to parallelize the normalization in a fine grained fashion as proposed by Dershowitz and Lindenstrauss (1990). Tests with single random terms were rather discouraging. For example, normalizing the term $t = \frac{1}{2} \cdot (\frac{1}{2} \cdot \frac{1}{2} \cdot (-\frac{1}{3} + b) + 0) \cdot (-\frac{1}{5})$ by the complete TRS for R_{235} takes 84s with an innermost reduction strategy. The longest sequential subtask takes at least 26s, from which 12s are spent in top reductions and thus form a pure sequential bottleneck. Applying Amdahl's law, the theoretical bound for the speed-up is 2.8 for four, and 7 for infinitely many processors. In practice, we obtained a speed-up of 2 on four processors.

We can create yet more concurrency when the parallel normalization is applied to lists of terms, but we have to beware of over-saturation. We have compared the parallel normalizations of lists of 128 random terms using two different parallelization schemes.

1. Medium grained parallelism plus parallel normalization of the principal subterms of each term, and

Table 7. AC normalization statistics and speed-ups for scheme (1)/(2).

AC completion (platform 1)								
	max.	seq.	Parallel speed-ups on				VS-	
TRS	$\|t\|$	time [s]	1 proc.	2 proc.	3 proc.	4 proc.	threads	
AGR	36	542.3	0.9/0.9	1.7/1.7	2.3/2.3	3.0/3.0	3839	10 186
DLT	96	40.1	1.0/0.9	1.9/1.7	2.6/2.3	3.4/2.9	1310	33 040
RX	32	204.4	0.9/0.9	1.0/1.0	1.1/1.1	1.1/1.1	623	10 631
BRG	60	98.2	0.9/0.9	1.8/1.8	2.4/2.5	3.2/3.1	852	10 754
R_3	16	59.2	0.9/0.9	1.7/1.6	2.2/2.2	2.5/2.3	618	6421
R_7	13	73.8	0.9/0.9	1.7/1.7	2.3/2.3	2.8/2.5	529	10 051
R_{23}	15	68.7	0.9/0.9	1.7/1.6	2.1/2.0	2.3/2.1	805	5618
R_{35}	11	16.1	1.0/1.0	1.8/1.8	2.5/2.5	2.8/3.0	493	3793
R_{235}	10	45.6	0.9/0.9	1.5/1.4	1.9/1.8	2.1/2.0	639	5582
AZ_{22}	56	227.5	0.9/0.9	1.8/1.8	2.7/2.5	3.4/3.4	14 184	31 278
Z_{22}	64	34.7	1.0/0.9	1.9/1.8	2.8/2.6	3.3/3.3	128	128

2. medium grained parallelism plus fine grained parallelism (independent subterms of each term are reduced in parallel using an innermost reduction strategy).

To compute the random terms we used the ReDuX random term generator which computes pseudo random terms with up to p positions where p is a positive integer. Table 7 describes the experiments. Column 1 contains the completed TRS used for the normalizations, column 2 shows the maximal size of the terms in the lists and column 3 lists the times needed for the sequential normalization. The next four columns compares the speed-ups of the two parallelization schemes relative to the sequential implementation. The last column contains the number of VS-threads produced by the parallelization schemes (1) and (2), respectively. Surprisingly, the speed-ups for the two schemes do not differ much, and in some cases they are even worse for scheme (2)[†]. The extra overhead for forking VS-threads may consume any additional speed-ups, or the costs of the basic operations used in the normalizations (in particular the AC matches) may differ too much and the parallelization granularity may still be too large. Declining speed-ups of (2) against (1) point to the former, while the good speed-ups of experiment Z_{22}, which does not contain any AC operators, point to the latter. This question certainly deserves further investigation.

7.5. Parallel unfailing completion

The following set of specifications and proof obligations was used to evaluate our unfailing completion procedure. *CGroup* is the problem to show that in a commutative group the inversion is a homomorphism. The *Luka* specifications are an equational axiomatization for propositional calculus by Frege. Lukasiewicz gave another set of axioms, which are the goals in our examples $Luka_1$, $Luka_2$, and $Luka_3$. $Lusk_3$ proves that a ring with $x^2 = x$ is commutative. $Lusk_4$ shows that the commutator $h(h(x, y), y) = e$

[†] This is why we used scheme (1) in our AC completion experiments.

Table 8. General statistics for unfailing completion.

	Unfailing completion (platform 2)									
	With normalization					Without normalization				
TRS	cyc.	max. $\|\mathcal{P}\|$	$\Sigma\|\mathcal{P}\|$	max. $\|\mathcal{R}\|$	max. $\|\mathcal{E}\|$	cyc.	max. $\|\mathcal{P}\|$	$\Sigma\|\mathcal{P}\|$	max. $\|\mathcal{R}\|$	max. $\|\mathcal{E}\|$
$Luka_1$	107	3023	17 030	99	0	107	3376	17 030	99	0
$Luka_2$	129	5449	25 965	121	0	129	5796	25 965	121	0
$Luka_3$	382	51 746	240 629	373	0	382	54 757	240 629	373	0
$Lusk_3$	83	6263	10 592	40	34	112	12 964	18 495	39	50
$Lusk_4$	96	3408	7319	53	23	171	27 483	40 680	85	47
$Lusk_5$	26	494	1081	21	5	26	494	1081	21	5
$CGroup$	29	2530	4228	9	20	29	2909	4228	9	20
Z_{22}	193	632	3029	49	0	251	7407	8995	52	0

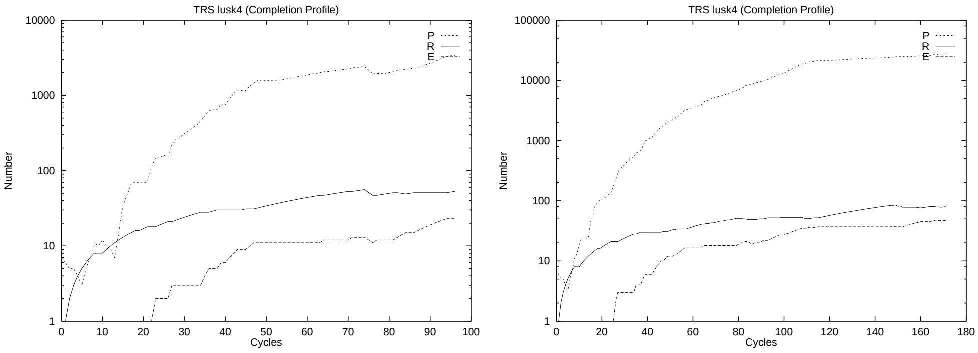

Figure 8. Completion profiles for unfailing completion with (left) and without normalization (right).

holds in a group with $x^3 = e$, and $Lusk_5$ proves that $f(x, g(x), y) = y$ holds in a ternary algebra where the third axiom is omitted. Note that for completing Z_{22} the unfailing completion procedure uses a critical pair selection strategy different from the two previous procedures. For more detailed information see the appendix.

As mentioned before, we have implemented two versions of the unfailing completion procedure. The first one keeps all critical pairs normalized and the second one normalizes critical pairs only immediately after their creation and upon selection for orientation. In analogy to Table 1, Table 8 characterizes the sequential completions by the two procedures. $|\mathcal{E}|$ denotes the maximal number of non-orientable equations that may be used for ordered rewriting.

Figure 8 presents a typical completion profile for unfailing completion with (left col.) and unfailing completion without normalization (right col.), respectively. R, E, and P denote the lengths of the list of rewrite rules $\mathcal{R}$, the list of equations $\mathcal{E}$, and the list of critical pairs $\mathcal{P}$ during the proof process at any given cycle of the outer loop.

The left half of Table 9 summarizes our parallel experiments with unfailing completion that strictly normalizes all pairs (cf. Table 5 for explanations). Again all experiments were conducted with a common set of grain size settings that proved favorable in our grain size experiments: The minimal load of a single thread was either (1) normalizing at

Table 9. Results for parallel unfailing completion.

	Unfailing completion (platform 2)								
	With normalization					Without normalization			
	sequential	parallel speed-ups on			VS-	sequential		speed-ups	
TRS	time [s]/pp	1 proc.	2 proc.	4 proc.	threads	time [s]/pp		on 4 proc.	
$Luka_1$	118.7/99%	0.9/1.0	1.8/2.0	3.4/3.7	18 488	98.8/97%	(1.2)	3.3	(1.0)
$Luka_2$	232.4/99%	0.9/1.0	1.9/2.0	3.5/3.8	28 678	195.0/98%	(1.2)	3.5	(1.0)
$Luka_3$	8197.7/99%	1.0/1.0	1.9/2.0	3.8/3.9	260 119	6779.8/99%	(1.2)	3.7	(1.0)
$Lusk_3$	765.8/99%	0.8/1.0	1.5/1.9	3.2/4.1	14 450	1181.4/98%	(0.7)	3.2	(1.0)
$Lusk_4$	189.8/99%	0.9/1.0	1.9/2.0	3.6/3.8	10 660	2358.2/97%	(0.1)	3.6	(1.0)
$Lusk_5$	9.3/99%	0.7/1.0	1.1/1.6	2.0/2.8	1144	8.1/98%	(1.2)	2.1	(1.0)
CGroup	309.1/99%	0.7/1.0	1.1/1.7	3.0/4.5	4942	250.8/98%	(1.2)	3.0	(1.0)
Z_{22}	62.3/98%	0.9/1.0	1.7/1.9	3.0/3.3	4130	745.4/26%	(0.1)	1.1	(2.7)

least three terms, or (2) testing at least 96 terms for reducibility w.r.t. a rewrite rule, or (3) testing at least 32 terms for reducibility w.r.t. an equation, or (4) computing critical pairs between a rule and at least eight other rules, or (5) computing critical pairs between a rule and at least four equations, or (6) computing critical pairs between an equation and at least four rules, or (7) computing critical pairs between an equation and at least two other equations.

Note that speed-ups of 4.1 and 4.5 measured against one processor parallel run times are unrealistic as well as 0.7 and 0.8 slow down for one processor parallel code compared to sequential code. This suggests that those parallel runtimes measured on one processor are to high[†]. The remaining problems of appropriate size $Luka_1$, $Luka_2$, $Luka_3$, $Lusk_4$ and Z_{22} show good and regular speed-ups, where the largest problem $Luka_3$ also provides the best speed-ups.

The right half of Table 9 describes the result for the parallel implementation of unfailing completion procedure that does not keep all critical pairs normalized. It contains the sequential runtimes and percentages of the parallelized code in column 7 and the speed-ups on four processors w.r.t. the sequential implementation in the last column. The figures in parentheses show the corresponding ratios for runtimes and speed-ups between unfailing completion with normalization and without normalization.

We recognize that there are different kinds of problems. Table 8 shows that *CGroup*, the *Luka* examples and $Lusk_5$ take the same number of completion rounds, no matter whether we use unfailing completion with or without normalization. This is also mirrored in the respective completion profiles. These hints lead us to conjecture that both versions do roughly the same, where unfailing completion without normalization is about 1.2 times faster than unfailing completion, because it has to do less work.

The other kind of problems are those where unfailing completion without normalization takes more completion rounds than unfailing completion. In case of $Lusk_3$, $Lusk_4$ and Z_{22}, simplifying all critical pairs before choosing the best one in the orientation step must lead to a more intelligent choice and thus to a short cut in the proof or completion

[†] It is possible that the system was employed by something else than our job, while we measured these runtimes, which are wall-clock times.

process. Unfailing completion without normalization is significantly slower on standard completion tasks.

Note that for all problems except Z_{22} speed-ups are roughly the same, no matter whether we use unfailing completion with or without normalization. This is rather astonishing, because unfailing completion without normalization does less work in parallel than unfailing completion and therefore was expected to produce worse speed-ups generally. For more details on our experiments with unfailing completion see Maier *et al.* (1995).

8. Conclusions

Completion and parallel computation each have a great many facets, and the combination of both opens a complex decision space. Currently, we think it is unlikely that a single optimal solution exists. Our work represents an approach geared towards everyday operation on shared memory multiprocessors of the workstation class, such as the 4-processor SPARCstation we use. In addition, we are working towards a discipline of parallel programming which can be applied uniformly to many aspects of symbolic computation, from Computer Algebra to Theorem Proving.

In this setting, we have provided a strategy compliant parallelization which shows reliable and predictable speed-ups. These have been achieved by a high-level parallel programming technique relying on the support of a well-defined user-level threads system. We have thus demonstrated that this approach, which had proved successful in algebraic applications before, is valid across the spectrum of symbolic computation. These results are significant because it had been argued before that the inner completion loop could not be profitably parallelized and that the parallelization must include low-level code.

Reviewing all our parallelization experiments, we notice that the speed-ups mainly depend on two factors. The first factor is the amount of independent work intrinsic to a problem. Long total runtimes and completion profiles showing large rule and equation sets during most completion cycles can be seen as indicators for the parallelism available. The second factor is the (ir)regularity of the costs of basic operations. For very irregular costs as in AC completion the indicators of much parallelism may be misleading. In summary, we successfully parallelized all three completion procedures obtaining peak speed-ups of over 3 on four processors in all three cases. We have thus shown that for parallel completion on the desk-top good to excellent speed-ups can be achieved with an inner loop parallelization that preserves the usual behavior and functionality of sequential completion, but exploits the parallel hardware.

Acknowledgements

This paper is based upon work supported by grant Ku 966/1-2 from the Deutsche Forschungsgemeinschaft within the Schwerpunkt Deduktion. We thank Patrick Maier for his contribution to implementing the parallel unfailing completion procedure. Jeff Ward and Nicholas Nevin contributed to the design and development of the VS-threads system; their work was supported by a grant from NSF.

References

Amrhein, B., Gloor, O., Küchlin, W. (1996). A case-study of multi-threaded Gröbner basis completion. Proc. ISSAC'96, ACM Press, New York.

Attardi, G., Traverso, C. (1994). A strategy-accurate parallel Buchberger algorithm. In Hong, H. (1994), pp. 12–21.

Avenhaus, J., Denzinger, J. (1993). Distributing equational theorem proving. In Kirchner, C., (1993), pp. 62–76.

Bachmair, L., Dershowitz, N. (1988). Critical pair criteria for completion. *J. Symbolic Computation*, **6**(1):1–18.

Bachmair, L., Dershowitz, N., Plaisted, D.A. (1989). Completion without failure. In Aït-Kaci, Nivat, (eds), *Resolution of Equations in Algebraic Structures*, volume 2 of *Rewriting Techniques*, chapter 1. Boston, Academic Press.

Bonacina, M.P., Hsiang, J. (1993). Distributed deduction by clause-diffusion: the Aquarius prover. In Miola, (ed), *Design and Implementation of Symbolic Computation Systems*, pp. 272–287, Gmunden, Austria, September. Berlin, Springer-Verlag.

Bonacina, M.P., McCune, W.W. (1994). Distributed theorem proving by *peers*. In Bundy, (ed), *Automated Deduction—CADE-12*, pp. 841–846, Nancy, France, June. Berlin, Springer-Verlag.

Buchberger, B. (1965). *Ein Algorithmus zum Auffinden der Basiselemente des Restklassenringes nach einem nulldimensionalen Polynomideal.* PhD thesis, Universität Innsbruck.

Buchberger, B. (1985). Basic features and development of the Critical-Pair/Completion procedure. In Jouannaud, (ed), *Rewriting Techniques and Applications*, pp. 1–45, Dijon, France, May. Berlin, Springer-Verlag.

Buchberger, B., *et al.* (1993). Saclib user's guide. On-line software documentation.

Buchberger, B., Loos, R. (1982). Algebraic simplification. In *Computer Algebra: Symbolic and Algebraic Computation*, pp. 11–43. Berlin, Springer-Verlag, 2nd edition.

Bündgen, R., Göbel, M., Küchlin, W. (1994a). Experiments with multi-threaded Knuth–Bendix completion. Technical Report 94–05, Wilhelm-Schickard-Institut, Universität Tübingen, D-72076 Tübingen.

Bündgen, R., Göbel, M., Küchlin, W. (1994b). A fine-grained parallel completion procedure. In von zur Gathen, Giesbrecht, (eds), *Proc.* 1994 *International Symposium on Symbolic and Algebraic Computation: ISSAC'94*, pp. 269–277, Oxford, England, July. New York, ACM Press.

Bündgen, R., Göbel, M., Küchlin, W. (1995). Parallel ReDuX → PaReDuX. In Hsiang, J, (ed), *Rewriting Techniques and Applications, 6th Intl. Conf., RTA-95*, pp. 408–413, Kaiserslautern, Germany, April. Berlin, Springer-Verlag.

Bündgen, R. (1991). Completion of integral polynomials by AC-term completion. In Watt, (ed), *Proc.* 1991 *International Symposium on Symbolic and Algebraic Computation: ISSAC'91*, pp. 70–78, Bonn, Germany, July. New York, ACM Press.

Bündgen, R. (1992). Buchberger's algorithm: The term rewriter's point of view. In Kuich, (ed), *Automata, Languages and Programming*, pp. 380–391, Vienna, Austria, July. EATCS, Berlin, Springer-Verlag.

Bündgen, R. (1993). Reduce the redex → ReDuX. In Kirchner (1993), pp. 446–450.

Chakrabarti, S., Yelick, K. (1993). Implementing an irregular application on a distributed memory multiprocessor. In *Fourth ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pp. 169–178, San Diego, CA, May. New York, ACM Press. (Also SIGPLAN Notices **28**(7)).

Christian, J. (1989). Fast Knuth–Bendix completion: Summary. In Dershowitz, (ed), *Rewriting Techniques and Applications*, pp. 551–555, Chapel Hill, North Carolina, USA, April. Berlin, Springer-Verlag.

Cooper, E.C., Draves, R.P. (1988). C threads. Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, June.

Dershowitz, N., Jouannaud, J.-P. (1990). Rewrite systems. In *Formal Models and Semantics*, volume 2 of *Handbook of Theoretical Computer Science*, chapter 6. Amsterdam, Elsevier.

Dershowitz, N., Lindenstrauss, N. (1990). An abstract concurrent machine for rewriting. In Kirchner, Wechler, (eds), *Algebraic and logic programming: Second international conference*, pp. 318–331, Nancy, France, October. Berlin, Springer-Verlag.

Faugère, J.C. (1994). Parallelization of Gröbner basis. In Hong (1994), pp. 124–132.

Fleischer, J., Grabmeier, J., Hehl, F., Küchlin, W., (eds) (1995). *Computer Algebra in Science and Engineering*, Singapore, World Scientific.

Hong, H. (ed) (1994). *First Intl. Symp. Parallel Symbolic Computation PASCO'94*, Linz, Austria, September, Singapore, World Scientific.

Huet, G. (1981). A complete proof of correctness of the Knuth–Bendix completion algorithm. *J. Computer and System Sciences*, **23**:11–21.

Jouannaud, J.-P., Kirchner, H. (1986). Completion of a set of rules modulo a set of equations. *SIAM J. Computation*, **15**:1155–1194.

Kirchner, C. (ed) (1993). *Rewriting Techniques and Applications*, Montreal, Canada, June. Berlin, Springer-Verlag.

Klop, J.W. (1992). Term rewriting systems. In Abramsky, Gabbay, Maibaum, (eds), *Handbook of Logic in Computer Science*, volume 2: Background: Mathematical Structures, chapter 1, pp. 1–116. Oxford, Clarendon Press.

Knuth, D.E., Bendix, P.B. (1970). Simple word problems in universal algebra. In Leech, (ed), *Computational Problems in Abstract Algebra*. Oxford, Pergamon Press.

Küchlin, W., Nevin, N.J. (1991). On multi-threaded list-processing and garbage collection. In *Proc. Third IEEE Symp. on Parallel and Distributed Processing*, pp. 894–897, Dallas, TX, December. Los Alamitos, CA, IEEE Press.

Küchlin, W., Ward, J.A. (1992). Experiments with virtual C Threads. In *Proc. Fourth IEEE Symp. on Parallel and Distributed Processing*, pp. 50–55, Dallas, TX, December. Los Alamitos, CA, IEEE Press.

Küchlin, W. (1982a). An implementation and investigation of the Knuth–Bendix completion algorithm. Master's thesis, Informatik I, Universität Karlsruhe, D-7500 Karlsruhe, W-Germany. (Reprinted as Report 17/82.).

Küchlin, W. (1982b). Some reduction strategies for algebraic term rewriting. *ACM SIGSAM Bull.*, **16**(4):13–23.

Küchlin, W. (1985). A confluence criterion based on the generalised Newman Lemma. In Caviness, B. (ed), *Eurocal'85*, pp. 390–399, Linz, Austria, April. Berlin, Springer-Verlag.

Küchlin, W. (1986). A generalized Knuth–Bendix algorithm. Technical Report 86-01, Mathematics, Swiss Federal Institute of Technology (ETH), CH-8092 Zürich, Switzerland, January.

Küchlin, W. (1990). PARSAC-2: A parallel SAC-2 based on threads. In Sakata, (ed), *Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes: 8th International Conference, AAECC-8*, pp. 341–353, Tokyo, Japan, August. Berlin, Springer-Verlag.

Küchlin, W. (1992). The S-threads environment for parallel symbolic computation. In Zippel, (ed), *Computer Algebra and Parallelism*, pp. 1–18, Ithaca, NY, March. Berlin, Springer-Verlag.

Küchlin, W. (1995). PARSAC-2: Parallel computer algebra on the desk-top. In Fleischer, J., *et al.* (1995), pp. 24–43.

Lankford, D., Ballantyne, A.M. (1977). Decision procedures for simple equational theories with commutative-associative axioms: Complete sets of commutative-associative reductions. Technical Report Report ATP-39, Department of Mathematics and Computer Sciences, University of Texas, Austin, August.

Lusk, E.L., McCune, W.W. (1990). Experiments with ROO, a parallel automated deduction system. In Fronhöfer, Wrightson, (eds), *Parallelization in Inference Systems*, pp. 139–162, Dagstuhl Castle, Germany, December. Berlin, Springer-Verlag.

Lusk, E.L., Overbeek, R.A. (1985). Reasoning about equality. *J. Automated Reasoning*, **1**:209–228.

Maier, P., Göbel, M., Bündgen, R. (1995). Multi-threaded unfailing completion. Technical Report WSI 95–06, Wilhelm Schickard-Institut, Universität Tübingen, D-72076 Tübingen.

Melenk, H., Neun, W. (1989). Parallel polynomial operations in the large Buchberger algorithm. In Della Dora, Fitch, (eds), *Computer Algebra and Parallelism*, Computational Mathematics and Applications, pp. 143–158, London, Academic Press.

Mohr, E., Kranz, D.A., Halstead, R.H., Jr. (1991). Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. on Parallel and Distributed Systems*, **2**(3):264–280.

Morisse, K., Oevel, G. (1995). New developments in MuPAD. In Fleischer, J., *et al.* (1995).

Peterson, G., Stickel, M. (1981). Complete sets of reductions for some equational theories. *J. ACM*, **28**:223–264.

Plaisted, D. (1993). Equational reasoning and term rewriting systems. In Gabbay, Hogger, Robinson, (eds), *Logical Foundations*, volume 1 of *Handbook of Logic in Artificial Intelligence and Logic Programming*, chapter 5. Oxford, Oxford University Press.

Sawada, H., Terasaki, S., Aiba, A. (1994). Parallel computation of Gröbner Bases on distributed memory machines. *J. Symbolic Computation*, **18**(3):207–222.

Schreiner, W., Hong, H. (1993). The design of the PACLIB kernel for parallel algebraic computation. In Volkert, (ed), *Parallel Computation (2nd Internatl. ACPC Conf.)*, pp. 204–218, Gmunden, Austria, October. Berlin, Springer-Verlag.

Schwab, St. A. (1992). Extended parallelism in the Gröbner Basis algorithm. *Int. J. of Parallel Programming*, **21**(1):39–66.

Slaney, J.K., Lusk, E.L. (1990). Parallelizing the closure computation in automated deduction. In Stickel, (ed), *10th International Conference on Automated Deduction*, pp. 28–39, Kaiserslautern, Germany, July. Berlin, Springer-Verlag.

Sperber, M. (1994). Mørk: A generator for preprocessors. Master's thesis, Universität Tübingen.

Stickel, M.E. (1983). A note on leftmost innermost term reduction. *ACM SIGSAM Bull.*,**17**(1):19–20.

Tanenbaum, A.S. (1992). *Modern Operating Systems*. Englewood Cliffs, NJ: Prentice Hall.

Tarski, A. (1956). *Logic, Semantics, Metamathematics*. Oxford University Press.

Vidal, J.-P. (1990). The computation of Gröbner bases on a shared memory multiprocessor. In Miola, (ed), *Design and Implementation of Symbolic Computation Systems*, pp. 81–90, Capri, Italy, April. Berlin, Springer-Verlag.

Weiser, M., Demers, A., Hauser, C. (1989). The portable common runtime approach to interoperability. In *12th ACM SOSP*, pp. 114–122.

Yelick, K.A., Garland, S.J. (1992). A parallel completion procedure for term rewriting systems. In Kapur, (ed), *Automated Deduction—CADE-11*, pp. 109–123, Saratoga Springs, NY, June. Berlin, Springer-Verlag.

Appendix. Problem Specifications

Algebraic specifications and term orderings for Knuth–Bendix completion.

Q_4: $1 \cdot X = 1$, $X^{-1} \cdot X = 1$, $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$, $a^4 = b^2$, $a^{-1} = b \cdot (a \cdot b^{-1})$
Knuth–Bendix order: $()^{-1} : 0 > \cdot : 2 > b : 1 > a : 1 > 1 : 1$

D_n: (Dihedral group) $1 \cdot X = 1$, $X^{-1} \cdot X = 1$, $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$, $a^n = 1$, $b^2 = 1$, $b \cdot a = a^{-1} \cdot b$
Knuth–Bendix order: $()^{-1} : 0 > \cdot : 2 > b : 1 > a : 1 > 1 : 1$

P_n: (k equals 0, 2, 4, 6, 8, 10, 12, 16, 24 for P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7 and P_8, cf. Christian (1989)) $f(f(X,Y),X) = f(X, f(Y,X))$, $f(e_i, X) = X$, $f(X, j_i(X)) = e_i$ for $1 \leq i \leq k$
Knuth–Bendix order: $j_k > \cdots > j_1 > f > e_k > \cdots > e_1$, $w(e_i) = 1$ for $1 \leq i < k$, $w(e_k) = 2$, $w(f) = 0$, $w(j_i) = 2(n - i)$ for $1 \leq i \leq k$.

M_n: $g(X, j(f_i(f_{i+1 \bmod n}(X)))) = f_{i+1 \bmod n}(X)$ for $1 \leq i \leq n$, $g(f_i(X), Y) = f_i(Y)$ for $1 \leq i \leq n$, $g(g(X,Y), j(Z)) = g(X, g(Y,Z))$, $g(g(X,Y),Z) = g(X,Z)$
Knuth–Bendix order: $j : 0 > g : (n+1) > f_1 : n > \cdots > f_n : 1$

Z_{22}: (Cyclic group of order 22, cf. Avenhaus and Denzinger (1993)) $a(b(c(X))) = d(X)$, $b(c(d(X))) = e(X)$, $c(d(e(X))) = a(X)$, $d(e(a(X))) = b(X)$, $e(a(b(X))) = c(X)$, $a(a_1(X)) = X$, $a_1(a(X)) = X$, $b(b_1(X)) = X$, $b_1(b(X)) = X$, $c(c_1(X)) = X$, $c_1(c(X)) = X$, $d(d_1(X)) = X$, $d_1(d(X)) = X$, $e(e_1(X)) = X$, $e_1(e(X)) = X$
Path order: $e_1 > e > \cdots > a_1 > a$

Z_{22T}: (cf. Avenhaus and Denzinger (1993), modified) $a(b(c(X))) = d(X)$, $b(c(d(X))) = e(X)$, $c(d(e(X))) = f(X)$, $d(e(f(X))) = a(X)$, $e(f(a(X))) = b(X)$, $f(a(b(X))) = c(X)$, $a(a_1(X)) = X$, $a_1(a(X)) = X$, $b(b_1(X)) = X$, $b_1(b(X)) = X$, $c(c_1(X)) = X$, $c_1(c(X)) = X$, $d(d_1(X)) = X$, $d_1(d(X)) = X$, $e(e_1(X)) = X$, $e_1(e(X)) = X$, $f(f_1(X)) = X$, $f_1(f(X)) = X$
Path order: $f_1 > f > \cdots > a_1 > a$

Z_{22W}: (cf. Avenhaus and Denzinger (1993), modified) $X \cdot 1 = X$, $X \cdot X^{-1} = 1$, $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$, $a \cdot (b \cdot c) = d$, $b \cdot (c \cdot d) = e$, $c \cdot (d \cdot e) = a$, $d \cdot (e \cdot a) = b$, $e \cdot (a \cdot b) = c$
Knuth–Bendix order: $()^{-1} : 0 > \cdot : 3 > e : 2 > \cdots > a : 2 > 1 : 1$

Algebraic specifications and term orderings for AC completion.

AGR: (Abelian group, $\{+\} \in \mathcal{F}_{AC}$) $0 + X = X$, $X^{-1} + X = 0$
Path order: $()^{-1} > + > 1 > 0$

DLT: (Distributive lattice, $\{\wedge, \vee\} \in \mathcal{F}_{AC}$) $(X \wedge (X \vee Y)) = X$, $(X \vee (X \wedge Y)) = X$, $(X \vee (X \wedge Y)) = ((X \vee Y) \wedge (X \vee Y))$
Path order: $\vee > \wedge$

RX: (Multivariate polynomials over commutative rings with unit elements, cf. Bündgen (1991), $\{+, \cdot, ., \odot, \oplus\} \in \mathcal{F}_{AC}$) $x + 0 = x$, $x + -x = 0$, $x \cdot 1 = x$, $x \cdot (y + z) = (x \cdot y) + (x \cdot z)$, $X.I = X$, $f \oplus \Omega = f$, $f \odot \Omega = \Omega$, $f \odot M(1, I) = f$, $f \odot (g \oplus h) = (f \odot g) \oplus (f \odot h)$, $f \oplus (f \odot M(-1, I)) = \Omega$, $M(x, X) \oplus M(y, X) = M(x + y, X)$, $M(x, X) \odot M(y, Y) = M(x \cdot y, X.Y)$
Polynomial Interpretation: $0 : 6$, $1 : 2$, $I : 6$, $\Omega : 6$, $X : 6$, $Y : 6$, $+ : X_1 + X_2 + 5$, $\cdot : X_1 X_2$, $. : X_1 X_2$, $- : 10X_1 + 2$, $M : (X_1 + 5) * X_2$, $\oplus : X_1 + X_2 + 5$, $\odot : X_1 X_2$

BRG: (Boolean ring, $\{\oplus, \wedge\} \in \mathcal{F}_{AC}$) $X \oplus F = X$, $X \oplus X = F$, $X \wedge T = X$, $X \wedge X = X$, $X \wedge F = F$, $X \wedge (Y \oplus Z) = (X \wedge Y) \oplus (X \wedge Z)$, $a_1 \oplus a_2 = a_6$, $a_1 \oplus a_3 = a_4$, $a_1 \oplus a_4 = a_3$, $a_1 \oplus a_5 = T$, $a_1 \oplus a_6 = a_2$, $a_2 \oplus a_3 = a_5$, $a_2 \oplus a_4 = T$, $a_2 \oplus a_5 = a_3$, $a_2 \oplus a_6 = a_1$, $a_3 \oplus a_4 = a_1$, $a_3 \oplus a_5 = a_2$, $a_3 \oplus a_6 = T$, $a_4 \oplus a_5 = a_6$, $a_4 \oplus a_6 = a_5$, $a_5 \oplus a_6 = a_4$, $a_1 \wedge a_2 = F$, $a_1 \wedge a_3 = F$, $a_1 \wedge a_4 = a_1$, $a_1 \wedge a_5 = F$, $a_1 \wedge a_6 = a_1$, $a_2 \wedge a_3 = F$, $a_2 \wedge a_4 = F$, $a_2 \wedge a_5 = a_2$, $a_2 \wedge a_6 = a_2$, $a_3 \wedge a_4 = a_3$, $a_3 \wedge a_5 = a_3$, $a_3 \wedge a_6 = F$, $a_4 \wedge a_5 = a_3$, $a_4 \wedge a_6 = a_1$, $a_5 \wedge a_6 = a_2$, $a_1 \oplus T = a_5$, $a_2 \oplus T = a_4$, $a_3 \oplus T = a_6$, $a_4 \oplus T = a_2$, $a_5 \oplus T = a_1$, $a_6 \oplus T = a_3$
Path order: $\wedge > \oplus > a_1 \sim a_2 \sim a_3 \sim a_4 \sim a_5 \sim a_6 > T > F$

AZ_{22}: (cf. Avenhaus and Denzinger (1993), modified, $\{\cdot\} \in \mathcal{F}_{AC}$) $X \cdot 1 = X$, $X \cdot X^{-1} = 1$, $a \cdot (b \cdot c) = d$, $b \cdot (c \cdot d) = e$, $c \cdot (d \cdot e) = a$, $d \cdot (e \cdot a) = b$, $e \cdot (a \cdot b) = c$
Path order: $()^{-1} > \cdot > e > \cdots > a > 1$

Z_{22}: (see above)

$R_{ijk\ldots}$**:** (Finitely presented rings, $\{+,\cdot\} \in \mathcal{F}_{AC}$ cf. Bündgen (1992)) $0+X=X$, $0\cdot X=0$, $1\cdot X=X$, $-0=0$, $-(-X)=X$, $X+-X=0$, $-(X+Y)=-X+-Y$, $X\cdot -Y=-(X\cdot Y)$, $X\cdot(Y+Z)=(X\cdot Y)+(X\cdot Z)$, $(\underbrace{1+\cdots+1}_{i})\cdot\frac{1}{i}=1$, $(\underbrace{1+\cdots+1}_{j})\cdot\frac{1}{j}=1$, $(\underbrace{1+\cdots+1}_{k})\cdot\frac{1}{k}=1,\ldots$
Polynomial Interpretation: $0: 6$, $1: 2$, $\frac{1}{i}: 1000$, $\frac{1}{j}: 10000$, $\frac{1}{k}: 100000$, $\ldots$, $+: X_1+X_2+5$, $\cdot: X_1X_2$, $-: 10X_1+2$

Algebraic specifications and term orderings for unfailing completion.

*CGroup***:** $g(e)=e$, $g(g(x))=x$, $g(f(x,y))=f(g(y),g(x))$, $f(f(x,y),z)=f(x,f(y,z))$, $f(e,x)=x$, $f(g(x),x)=e$, $f(g(x),f(x,y))=y$, $f(x,y)=f(y,x)$, $f(x,e)=x$, $f(x,g(x))=e$, $f(x,f(g(x),y))=y$
Knuth–Bendix order: $g:0>f:2>e:1$ Task: prove $g(f(x,y))=f(g(x),g(y))$

$Luka_1$**:** (cf. Tarski (1956)) $C(C(p,q),C(N(q),N(p)))=T$, $C(T,x)=x$, $C(p,C(q,p))=T$, $C(N(N(p)),p)=T$, $C(p,N(N(p)))=T$, $C(C(p,C(q,r)),C(C(p,q),C(p,r)))=T$, $C(C(p,C(q,r)),C(q,C(p,r)))=T$
Path order: $C>N>T>Ap>Aq>Ar$ Task: prove $C(C(Ap,Aq),C(C(Ap,Aq),C(Ap,Ar)))=T$

$Luka_2$**:** (cf. Tarski (1956)) $C(T,x)=x$, $C(p,C(q,p))=T$, $C(p,N(N(p)))=T$, $C(N(N(p)),p)=T$, $C(C(p,C(q,r)),C(C(p,q),C(p,r)))=T$, $C(C(p,C(q,r)),C(q,C(p,r)))=T$, $C(C(p,q),C(N(q),N(p)))=T$
Path order: $C>N>T>Ap$ Task: prove $C(C(N(Ap),Ap),Ap)=T$

$Luka_3$**:** (cf. Tarski (1956)) $C(C(p,C(q,r)),C(C(p,q),C(p,r)))=T$, $C(T,x)=x$, $C(p,C(q,p))=T$, $C(C(p,C(q,r)),C(q,C(p,r)))=T$, $C(N(N(p)),p)=T$, $C(p,N(N(p)))=T$, $C(C(p,q),C(N(q),N(p)))=T$
Path order: $C>N>T>Ap$ Task: prove $C(Ap,C(N(Ap),Aq))=T$

$Lusk_3$**:** (cf. Lusk and Overbeek (1985)) $f(f(x,y),z)=f(x,f(y,z))$, $j(0,x)=x$, $j(x,0)=x$, $j(g(x),x)=0$, $f(x,j(y,z))=j(f(x,y),f(x,z))$, $j(x,g(x))=0$, $j(x,y)=j(y,x)$, $j(j(x,y),z)=j(x,j(y,z))$, $f(j(x,y),z)=j(f(x,z),f(y,z))$, $f(x,x)=x$
Knuth–Bendix order: $f:5>j:4>g:3>0:1>b:1>a:1$ Task: prove $f(a,b)=f(b,a)$

$Lusk_4$**:** (cf. Lusk and Overbeek (1985)) $f(f(x,y),z)=f(x,f(y,z))$, $f(g(x),x)=e$, $f(e,x)=x$, $f(x,f(x,x))=e$, $h(x,y)=f(x,f(y,f(g(x),g(y))))$, $f(x,g(x))=e$, $f(x,e)=x$
Knuth–Bendix order: $f:4>g:3>h:5>b:1>a:1>e:1$ Task: prove $h(h(a,b),b)=e$

$Lusk_5$**:** (cf. Lusk and Overbeek (1985)) $f(f(v,w,x),y,f(v,w,z))=f(v,w,f(x,y,z))$, $f(y,x,x)=x$, $f(x,x,y)=x$, $f(g(y),y,x)=x$
Path order: $g>f>b>a$ Task: prove $f(a,g(a),b)=b$

Z_{22}**:** (see above) Task: completion