A Fine-Grained Parallel Completion Procedure

Reinhard Bündgen Manfred Göbel Wolfgang Küchlin
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
D-72076 Tübingen, Germany
⟨{buendgen,goebel,kuechlin}@informatik.uni-tuebingen.de⟩

Abstract

We present a parallel Knuth-Bendix completion algorithm where the inner loop, deriving the consequences of adding a new rule to the system, is multi-threaded. The selection of the best new rule in the outer loop, and hence the completion strategy, is exactly the same as for the sequential algorithm. Our implementation, which is within the PARSAC-2 parallel symbolic computation system, exhibits good parallel speed-ups on a standard multi-processor workstation.

1 Introduction

1.1 Overview

This paper discusses data-structures and algorithms for parallel Knuth-Bendix completion of plain term-rewriting systems [17]. We work under the general restriction that no change of the completion strategy is allowed when going parallel. We thus attack a facet of the completion process which is known to be hard to parallelise, but whose parallel form can be used by other versions which follow other strategies. Our main contributions are that we achieve overall speed-ups of about 3 on 4 processors and that we use a systematic high-level parallel divide-and-conquer approach.

Our work is within the well-defined framework of the PARSAC-2 parallel symbolic computation system [21] which also contains a number of parallel algebraic algorithms (cf. [22]). All parallel constructs are provided by the *S-threads* parallelisation environment [23], which is itself built upon a standard threads (lightweight processes) interface supported by most modern operating systems. Our hardware architecture is a shared memory multiprocessor such as a typical parallel workstation. The parallel algorithms use no further application 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.

In order to achieve meaningful results we started with the high-quality sequential implementation of completion in the ReDuX system [8], which we parallelised gradually.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association of Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
ISAAC 94 - 7/94 Oxford England UK
© 1994 ACM 0-89791-638-7/94/0007..$3.50

Despite these restrictions we obtained good speed-ups on a standard 4 processor SPARC server. We think that our results are significant because it had been argued before [31, 37] that the inner completion loop could not be profitably parallelised and that the parallelisation must include low-level code such as a specific scheduler. In addition we re-used most sequential code and worked on a portable interface relying largely on standard UNIX.

We now proceed as follows. After motivating our work and relating it to other work in the area, we give an account of the completion process and of our parallelisation framework in Section 2. Section 3 states our parallel completion algorithm. Section 4 explains how our sequential term data-structures were modified to allow their simultaneous use in multiple threads. Section 5 contains empirical data about the performance of our parallel completion algorithm. Section 6 presents our conclusion of the work.

1.2 Motivation

Completion, both in its term-rewriting and its polynomial ideal form, is an important computational process [4]. Incidentally, the relationship between both forms is by now well understood [6, 5, 7] so that advances with one version can frequently be transferred to the other; for parallelisations this has already been done to some extent in [10, 37]. The completion procedure is however notoriously hard to parallelise. 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 highly tuned sequential strategies do exist. Since parallel forms are likely to change the completion strategy, possibly dependent on dynamic scheduling decisions, it is difficult to tell what portion of a parallel speed-up or slow-down is to be attributed to the change in strategy as opposed to parallel processing per se. It is particularly easy to produce artificial speed-ups by starting with a bad sequential strategy and implicitly changing to a better strategy in the parallel form. Any parallel algorithm should be measured against a high-quality sequential version and it should gracefully degrade into this sequential form as processors are taken away from under it.

Third, completion is at its core a closure computation (cf. [31]) and therefore is inherently sequential in the sense that the n-th generation of consequences necessarily depends on the $n - 1$st generation. The amount of parallelism in the process 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[1]. If the process converges, the number of consequences (and hence the amount of parallelism) must somehow be limited. As we shall see in Section 5, 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.

Still, because completion is an important computational method, it is interesting to explore to what degree it can be speeded up in practice through parallelisation. Because of the many facets of completion, it is important to parallelise 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. 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.

The most salient advantage of our *strategy compliant* parallel completion is that it produces predictable and deterministic speed-ups, independent of the completion strategy. 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. The same strategy is followed regardless of the scheduler, the number of parallel processes, the number of other processes on the system, or the number of its processors. All parallel experiments are therefore reproducible. This is important for evaluating parallel data structures and for investigating optimal parallelisation grain sizes.

Also, completion attempts can be broken off and restarted with predictable behavior. In addition our fine-grained parallelisation speeds up those completion cycles in which a large number of consequences must be processed. Therefore interactive completion of new unknown problems is particularly supported by our method.

1.3 Other Work

Slaney and Lusk [31] have examined the parallelisation of the general closure computation and have experimented with coarse-grained outer loop parallelisation. Their findings provide a useful framework and reference point for work on parallel completion, but completion is also significantly more complex than plain closure (cf. Section 3).

Even more coarse-grained, Avenhaus and Denzinger [1] run several incarnations of the completion procedure concurrently. Each uses a different strategy and periodically all of them compare their progress and exchange useful results. The combined process frequently exhibits large super-linear speed-ups due to the strategy refinement.

Yelick and Garland [37] have obtained significant speedups with a parallel completion procedure on a 6 processor workstation. They concluded that, in order to get good results, it was impossible to start with a sequential implementation. They assembled a parallel implementation by clustering the application of transition rules (cf. Figure 1) until they had tasks of a suitable grain-size. Then they executed the tasks using an application-level scheduler. Optimisations, such as locking tasks onto processors, were responsible for a significant portion of their speed-ups.

Other research has focussed on algorithms for extremely fine-grained tasks such as parallel matching [28] or rewriting a term by a set of rules [15]. The parallelism in these procedures is in all likelihood far below a profitable grain size on our set of examples.

Our work is placed in between coarse [1] and very fine grained parallelism [28]. Note that all these results, including ours, can in principle be combined to an algorithm that is parallel on several levels of granularity.

For the related Gröbner Basis algorithm this has been done to some extent by Schwab [30]. He combined the coarse-grained parallelisation done by Vidal [35] with the fine-grained parallel polynomial reduction technique developed by Melenk and Neun [26] and achieved better speedups than each individually. Both Schwab and Vidal use tools that are similar to ours (viz. a shared memory machine with C Threads under Mach), but unlike ours their coarse-grained parallelisation is not strategy compliant. It is not known to what extent precisely their speed-ups are due to strategy changes, but their data show significant super-linear effects.

2 Background

2.1 Term Rewriting Systems

We assume the reader is familiar with term-rewriting systems and completion procedures. For general introductions to term rewriting systems see [14, 27, 16].

A term is made up of *variables*, *constants* and *function symbols (operators)*. Each operator f has a fixed arity associated with it. *Terms* are defined recursively: All variables and constants are terms. If f is a n-ary operator and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term. Nothing else is a term. The set of constants and function symbols is called the *signature*.

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 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 $l \rightarrow r$ of terms. It may be applied to *reduce* a term t if t contains an instance of l. Then t reduces to t' where in t' this instance of l is replaced by the corresponding instance of r. A set of rewrite rules is a *term-rewriting system (TRS)* $\mathcal{R}$. A term rewriting system $\mathcal{R}$ should have the *termination property*. That is, starting with any term t there is no infinite sequence of reductions using rules from $\mathcal{R}$. The termination property of a term-rewriting system is undecidable in general, but there are powerful criteria to ensure this property; see [13] for an overview. A second important property is *confluence*. When reducing a term t by a confluent term rewriting system, the sequence of the rule applications does not matter: if a term is reachable by one sequence, it is reachable by any other sequence as well. A terminating *and* confluent term rewriting system is called *canonical* because it eventually reduces each term to

[1]Gröbner basis completion always terminates in theory, but frequently does not terminate in practice.

a unique and irreducible *normal form.*

An *equation* is a pair $s \leftrightarrow t$ of terms. Equations may be applied to terms both from left to right and from right to left. Thus $s \leftrightarrow t$ corresponds to the (non-terminating) term rewriting system $\{s \rightarrow t,\ t \rightarrow s\}$. A set of equations together with a signature is called an *algebraic specification.* A *term completion procedure* compiles on success a set of equations $\mathcal{E}$ into a canonical term rewriting system $\mathcal{R}$. It does so by repeatedly applying the inference rules in Figure 1 [2] to a pair $(\mathcal{E};\mathcal{R})$ of equations and rules. It succeeds if starting with $(\mathcal{E};\emptyset)$ a pair $(\emptyset;\mathcal{R})$ can be derived where $\mathcal{R}$ is canonical.

Delete: $\dfrac{(\mathcal{E}\cup\{s\leftrightarrow s\};\mathcal{R})}{(\mathcal{E};\mathcal{R})}$

Simplify: $\dfrac{(\mathcal{E}\cup\{s\leftrightarrow t\};\mathcal{R})}{(\mathcal{E}\cup\{s\leftrightarrow u\};\mathcal{R})}$ if $t \rightarrow_{\mathcal{R}} u$

Orient: $\dfrac{(\mathcal{E}\cup\{s\leftrightarrow t\};\mathcal{R})}{(\mathcal{E};\mathcal{R}\cup\{s\rightarrow t\})}$ if $s \succ t$[a]

Compose: $\dfrac{(\mathcal{E};\mathcal{R}\cup\{s\rightarrow t\})}{(\mathcal{E};\mathcal{R}\cup\{s\rightarrow u\})}$ if $t \rightarrow_{\mathcal{R}} u$

Collapse: $\dfrac{(\mathcal{E};\mathcal{R}\cup\{s\rightarrow t\})}{(\mathcal{E}\cup\{u\leftrightarrow t\};\mathcal{R})}$ if $\left\{ \begin{array}{c} s \rightarrow_{\mathcal{R}} u \text{ by } l \rightarrow r \in \mathcal{R} \\ \text{where } (s,t) \rhd (l,r) \end{array} \right.$[b]

Deduce: $\dfrac{(\mathcal{E};\mathcal{R})}{(\mathcal{E}\cup\{s\leftrightarrow t\};\mathcal{R})}$ if $s \leftarrow_{\mathcal{R}} u \rightarrow_{\mathcal{R}} t$

[a] $\succ \supseteq \rightarrow$ is a terminating term ordering.
[b] $\rhd$ is a terminating ordering on term pairs.

Figure 1: Completion inference rules

Let $l \rightarrow r$ and $l' \rightarrow r'$ be two rules where l contains a subterm s which unifies with l' such that the 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 [17] showed that a terminating term rewriting system is confluent iff all critical pairs are confluent and that only critical pairs must be considered as new equations in the *deduce* inference rule of Figure 1.

2.2 Multi-Threaded Symbolic Programming

In traditional operating systems, each process has an address space and a single thread of control. The thread of control is an active entity, moving from statement to statement, calling and returning from procedures. 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. A threads system can be implemented at the user level, but most modern operating systems, such as Mach, Solaris 2.x or OS/2, provide kernel threads [34].

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 (if any).

PARSAC-2 [21] is built upon a minimal threads abstraction called C Threads [12]. C Threads are directly supported by Mach and (at least to the extent that we need them) can be easily implemented using Solaris 2 or Posix threads, or, in our case, using the threads provided by PCR [36] which in turn needs only System V UNIX.

A C Thread is an execution context for a C procedure, providing a private register file and a private C 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 [23] which is a one-to-one extension of C Threads. S-threads has been designed such that most sequential SAC-2 algorithms will execute unmodified as a single S-thread; however the parallel list-processing context imposes a slight execution penalty. S-threads has been successfully employed to parallelise a number of algebraic algorithms (cf. [22]).

The original S-threads memory management scheme distributes the SAC-2 heap to threads as paged segments of cells. Heap transfer between threads is possible on the page level only.

The corresponding S-threads garbage collection method is very independent of the underlying threads system because it does not assume that all threads can be stopped by a user. Its effectiveness relies critically on a novel scheme called preventive garbage collection [24]. If a functional programming style is used and global side-effects are avoided, then upon procedure exit preventive garbage collection is able to collect the garbage produced inside the procedure on its own heap segment.

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.

Therefore, the memory management of S-threads was changed to use the PACLIB [29] scheme. Since it works on the cell level, cells allocated by one thread can be weaved into a global data-structure via side-effects. The PACLIB scheme assumes that all threads can be stopped by a user.

Since each S-thread is mapped one-to-one onto a C thread, limitations of different C Threads 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. In order to insulate the application from much of the vagaries of the C Threads system, S-threads was enhanced once more to *virtual S-threads* [25]. VS-threads is a user-level threads system with lazy threads creation. Its goal is to provide a virtually unlimited amount of extremely light-weight threads that are multiplexed on top of kernel-supported threads with real concurrency. The architecture allows the programmer to separate logical concurrency of the algorithm from the real concurrency of the parallel system.

VS-threads has the same semantics as S-threads. However, VS-threads keeps its own run-queues of micro-tasks, and it manages a small pool of C 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 number of available threads is virtually unlimited, and at the same time there is a significant reduction in the number of C-thread context switches through lazy evaluation of virtual threads. Furthermore, an abundant amount of logical (virtual) parallelism is dynamically reduced to a small amount of real parallelism that the kernel can support, and the grain-size of the remaining threads is increased by executing child tasks as procedures.

An abstract rendition of our parallelisation methodology which can be described as divide-and-conquer in combination with virtual tasks, is roughly as in the piece of code in Figure 2. 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.

```
Word example(Word args[])
{
  Word C, C1, C2, ...;
  Word args1[...], args2[...], r1, r2;
  sthread_t sth;

  C = args[0]; ...;        /* unpack input parameters */
  if Large(C)              /* check grain-size: fork? */
    {
      SPLIT_LIST(C, ..., &C1, &C2);       /* split C */
      args1[0] = C1; ... /* pack parameters for C1 */
      args2[0] = C2; ... /* pack parameters for C2 */
                         /* fork thread sth for C2 */
      sth = sthread_fork(example, args2, ...);
      r1  = example(args1);     /* rec. call for C1 */
      r2  = sthread_join(sth);  /* join thread sth */
      r = MERGE_LIST(r1, r2,...); /* merge results */
    }
  else { ...; }                        /* process C */
  return (r);                      /* return result */
}
```

Figure 2: Parallelisation scheme

Together with the VS-threads environment, this divide-and-conquer approach to parallelisation 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 an order of magnitude lower overhead. Thus we enjoy a dynamic adjustment of grain-size, with mostly large-grain S-threads executing concurrently when there is much work to do, and fine-grain S-threads executing only when workers would remain idle otherwise. Note well that all grain-size adjustment and task scheduling is done automatically within VS-threads and remains transparent to the application programmer.

3 Parallel Knuth-Bendix Completion

The inference rule characterisation of Knuth-Bendix completion leaves many decisions open that determine how to actually perform the completion. A regime which fixes the order and the manner in which the inference rules are to be applied is called a *completion methodology.* Figure 3 shows a first abstraction of the one we use.

Steps (2)–(8) form an *outer loop* in which the best equation is selected and turned into a new rule. Steps (4)–(8) contain *inner loops* in which new equations are derived and old rules and equations are reduced. The exact order in which equations are turned into rules is of the utmost importance both in the term case and for polynomials. This is called the *completion strategy* and in our case it is encapsulated in the exact method after which the minimum of the equations is determined. Minute changes in this strategy can have huge effects (positive or negative) on the duration of completion. E. g. we must find the minimum of the equations (critical pairs) in $\mathcal{E}$ 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 $\mathcal{E}$ which are equivalent w. r. t. the quasi-ordering (e. g. have the same number of symbols) may result in a different completion behaviour.

This completion methodology was chosen in ReDuX because the completion strategy can be fixed easily in procedure *Find_Min* and because ordering decisions (which may be very difficult) are minimised which favours interactive use.

The procedure can be parallelised on the level of the outer loop (adding equations concurrently), on the level of any of the inner loops, or below (e.g. by parallelising the reduction of a single term). As a general rule of thumb, the greater the grain-size of parallel tasks, the greater the efficiency of the parallelisation. It is therefore clear that, given a choice, the outer loop rather than the inner loops should be parallelised; this has been argued by Slaney and Lusk [31].

However, when equations are added concurrently by several tasks, before the consequences of the last addition are known, the completion strategy depends on the order in which the tasks are scheduled which is in general irreproducible and beyond the control of the programmer. In particular, it is virtually certain that the same code will execute a different strategy when run on a different number of processors. There are many reasons to keep the strategy fixed between completion runs. Completion of a new TRS frequently succeeds only after a suitable ordering and strategy have been developed in several (aborted) completion experiments. Having the strategy change uncontrollably between the experiments would be disruptive. Furthermore, it is difficult to pinpoint the reason for speed-ups (or slow-downs) if strategy effects interfere with parallelisation effects.

We therefore parallelised on the inner loop level, since low level parallelisation is too fine-grained for our examples in our environment.

The algorithm *COMPLETE* can of course be further improved if the following loop invariant is enforced in step (2): All rules in $\mathcal{R}$ and equations in $\mathcal{E}$ are fully (inter)reduced. Then *collapse*, *compose* and *simplify* apply only to those rules and old equations which are reducible by the newly oriented rule $l \to r$. Also only simplified equations may be deletable. A further improvement which is also implemented in our plain completion procedure is a simple application of the subconnectedness criterion [20] which allows to remove all equations in $\mathcal{E}$ which were derived from a collapsed rule.

To enforce a common strategy in the sequential and the parallel case we need step (3) as a synchronisation point. I. e. after each round, $\mathcal{E}$ and $\mathcal{R}$ must be the same for the two versions of the algorithm. Experimental experience showed that 90% of sequential completion time is spent in steps (6)–(8) (cf. section 5). The time spent in steps (2) and (3) is generally much below the minimal amount needed for an effective parallelisation. If we parallelised steps (4) and (5) we would face the following difficulties: (a) we must either synchronise the use of $\mathcal{R}$ or (b) we may use outdated copies of $\mathcal{R}$ which might lead to a different strategy because $\mathcal{R}$ is

```
R ← COMPLETE(ℰ, ≻)

[Completion procedure.
ℰ is a set of equations and ≻ is a terminating term ordering. Then upon success R is a canonical term rewriting system with =_R = =_ℰ.]

(1) [Initialise.] $\mathcal{R} := \emptyset$.
(2) [Stop?] if $\mathcal{E} = \emptyset$ then return $\mathcal{R}$ and stop.
(3) [Orient.] $\mathcal{E} := Find\_Min(\mathcal{E})$; $a \leftrightarrow b := First(\mathcal{E})$;
    $\mathcal{E} := Red(\mathcal{E})$; if $a \succ b$ then $\{l := a; r := b\}$
    elsif $b \succ a$ then $\{l := b; r := a\}$ else stop with failure;
    $\mathcal{R} := \mathcal{R} \cup \{l \to r\}$.
(4) [Collapse.] while the collapse-inference rule applies do
    $(\mathcal{E};\mathcal{R}) := Collapse((\mathcal{E};\mathcal{R}))$.
(5) [Compose.] while the compose-inference rule applies do
    $(\mathcal{E};\mathcal{R}) := Compose((\mathcal{E};\mathcal{R}))$.
(6) [Deduce.] Compute all critical pairs $\mathcal{P}$ of $l \to r$ and rules in $\mathcal{R}$. $\mathcal{E} := \mathcal{E} \cup \mathcal{P}$[a].
(7) [Simplify.] while the simplify-inference rule applies do
    $(\mathcal{E};\mathcal{R}) := Simplify((\mathcal{E};\mathcal{R}))$.
(8) [Delete.] while the delete-inference rule applies do
    $(\mathcal{E};\mathcal{R}) := Delete((\mathcal{E};\mathcal{R}))$;
    continue with step 2.                                   □
```

[a]Here we identify pairs and equations.

Figure 3: Algorithm *COMPLETE*

not confluent yet. Therefore we decided to parallelise only steps (6)–(8) of the completion.

For the parallel procedure we reorganised steps (6)–(8) into two independent procedures:

1. computing a list of non-trivial normalised critical pairs between a copy of $l \to r$ and $\mathcal{R}$ and
2. updating the old equation list w. r. t the subconnectedness criterion, simplification and deletion.

Both procedures are started simultaneously and each is parallelised using the divide-and-conquer scheme sketched at the end of Section 2 in Figure 2. Note that during the divide and conquer processes the minimal elements of the resulting critical pairs can be found in parallel too. Then only the minimal pairs of the two procedures need to be compared in step (3).

4 Data Structures for Parallel Term Rewriting

In this section, we first explain the most important aspects of the data structures of the sequential ReDuX System which were first designed in [18] and 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 denoting this term. 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, unification and is particular well-suited for efficient normalisations [19, 33].

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 synchronisation 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 normalisations and critical pair computations.

Since the minimal grain size for efficient parallelisation is much larger than the time spent in a single normalisation or critical pair computation, we perform several normalisation (or critical pair computation) tasks in parallel. In the sequential normalisation 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 behaviour. 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 realised by introducing an additional level of indirection for variable bindings: Each variable in a rule (or term) is labelled in its binding field by a fixed integer which is unique within the rule and indexes an entry in a *binding table*. 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 realised as a local array declared in the normalisation 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 the variable x (with index 3) of the term $f(x, g(x))$ is bound to $f(y, z)$. Instead of a direct pointer from x_3 to the bound term $f(y, z)$ as it is conveniently used in the sequential implementation, we now consider variable bindings relative to the context in which the variable is used. Thus terms of a rule consist of *term schemes* rather than actual terms and they 'materialise' only when they are applied (or used in a context like critical pair computations).

With this technique we can avoid copying global data if only one global object is used (substituted) at a time in a parallel thread. In case a thread accesses and modifies the variable bindings of more than one global item at a time, all

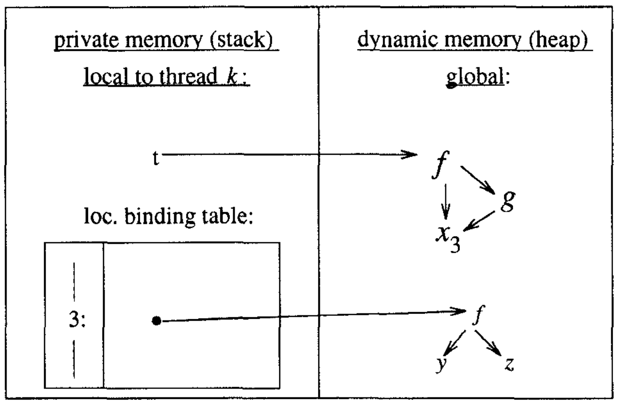

Figure 4: Variable bindings for $t\{x_3 \mapsto f(y,z)\}$

but one of the items must be 'coloured' in order to associate one binding table (of corresponding 'colour') to each item. Since 'colouring' is a real change of global data, items which are to be coloured must be copied.

Luckily, during the 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) coloured copy of the newly oriented rule $l \to r$ and the original rules in $\mathcal{R}$. Note that this does not lead to extra copy-overhead compared to the sequential procedure if we copy the newly oriented rule because it must be copied anyway to obtain the critical pairs of the rule and itself.

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.

5 Experimental Results

We implemented the parallel completion procedure on a Solbourne 5/704 with 48 Mbyte of main memory and four 33MHz SPARC processors on a common bus. The Solbourne operating system is an enhancement of SunOS 4.1.1 to allow for parallel processing but it does not support kernel threads. We implemented the C Threads interface on top of the user level PCR environment [36] and loaded VS-threads [25] on top. For our experiments we ported ReDuX to the SACLIB [3] and PARSAC-2 environments. ReDuX was translated to C using the ALDES-to-C Compiler produced by Michael Sperber [32].

In addition to plain completion, ReDuX contains extensions for inductive completion and for rewriting modulo equational theories (such as associativity-commutativity). Our parallel code still contains the data-structures and hooks necessary for these extensions.

Tables 1 and 2 and Figure 5 present the results (times in sec) for several completion experiments. Columns 2 of Tables 1 and 2 show the times of the sequential ReDuX implementation and columns 3–6 give the timings for the parallelised code run on 1–4 processors. Comparing columns 2 and 3 reveals a 3–7% penalty for using the parallel environment. According to our experiments the following grain sizes resulted in the best speed-ups: A single thread performed the normalisation of at least two equations or computed the critical pairs of a new rule and at least six old rules.

Figure 5 shows the runtimes of all experiments normalised to one with regard to the sequential implementation which is denoted as '0' processors.

	seq.	parallel on			
TRS		1 proc.	2 proc.	3 proc.	4 proc.
P6	111.8	119.6	72.9	56.1	47.4
P7	423.7	434.8	262.2	192.7	158.6
D16	25.3	26.9	15.7	11.8	10.1
Z22	83.8	87.0	50.4	36.5	30.0
Z22W	1432.6	1470.7	832.2	584.4	461.1
Z22t	3037.4	3145.8	1729.0	1215.1	960.9
M14	551.2	580.5	343.8	253.5	211.7
M15	753.8	790.1	468.3	352.1	287.4

Table 1: Total completion procedure (times in sec)

	seq.	parallel on			
TRS		1 proc.	2 proc.	3 proc.	4 proc.
P6	102.4	109.7	63.8	46.7	38.6
P7	394.1	404.1	233.9	165.0	131.5
D16	24.2	25.7	14.6	10.8	9.0
Z22	79.8	82.9	46.2	32.5	25.8
Z22W	1381.5	1418.2	782.0	537.0	415.7
Z22t	2971.3	3075.4	1662.9	1147.0	893.2
M14	501.2	530.1	294.5	204.6	162.4
M15	687.0	722.3	401.9	286.1	219.0

Table 2: Parallelised part of completion (times in sec)

The overall speed-ups for the total completion (Table 1) are 1.5–1.8 for two, 2.0–2.5 for three and 2.4–3.2 for four processors compared to the sequential implementation. Looking only at the parallelised part (Table 2), we get speed-ups of 1.7–1.9 for two, 2.4–2.7 for three and 2.8–3.4 for four processors compared to one processor.

The TRSs for experiments P6 and P7 are taken from [11] and that for Z22 is taken from [1][2]. D16 contains the three group equations and the relations of the dihedral group $\langle a, b;\ a^{16}, b^2, ba = \bar{a}b \rangle$. Z22W is the same group as Z22 but specified using an explicit binary group operator and an inversion operator. Z22t is the extension of

[2]Note that we used different strategies than the other authors.

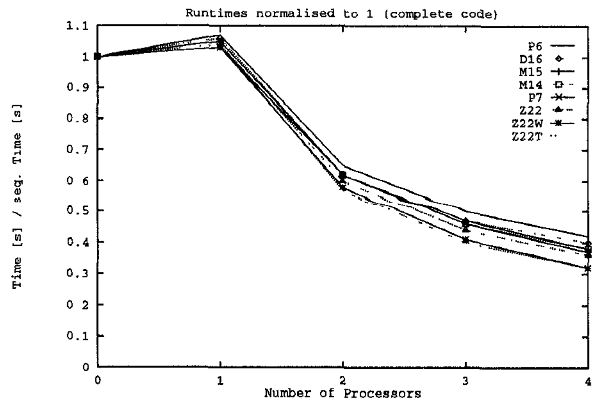

Figure 5: Normalised overall completion times

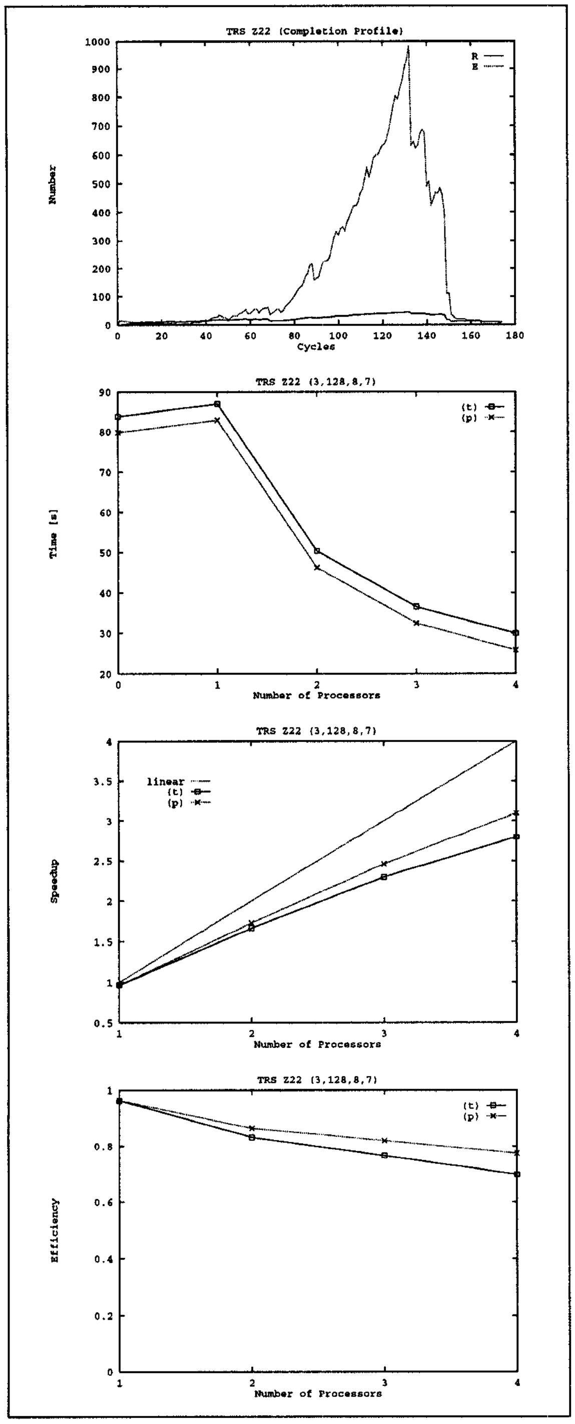

Figure 6: Experiment Z22

Z22 with an additional generator. M14 and M15 are instances of a scalable TRS Mn with $n+1$ unary operators and one binary operator and the equations $g(g(x,y),j(z)) \leftrightarrow g(x,g(y,z))$, $g(g(x,y),z) \leftrightarrow g(x,z)$, $g(f_i(x),y) \leftrightarrow f_i(y)$, $g(x,j(f_i(f_{i+1 \bmod n}(x)))) \leftrightarrow f_{i+1 \bmod n}(x)$ for $0 \leq i \leq n-1$.

Figure 6 describes the experiment Z22. The topmost plot shows a completion profile: The upper graph denotes $|\mathcal{E}|$ (y-axis) and the lower graph is $|\mathcal{R}|$ (y-axis) which are measured at step 2 in each round (x-axis) of the outer loop of COMPLETE. The profile given for Z22 is in our experience typical. We see that only a rather small portion of the completion procedure provides the potential for good inner loop parallelisation. Therefore we cannot expect optimal speed-ups. The other plots show runtimes, speed-ups and efficiencies. Again '0' processors denote the sequential implementation. The graphs marked with (t) describe the whole completion process and the graphs marked with (p) describe the prallelised part.

For a detailed description of the experiments and additional experimental data the reader is referred to [9].

6 Conclusion

We have shown that the inner loops of the completion procedure can be parallelised with parallelised speed-ups of up to three on a four processor workstation. Our programs use the fork/join paradigm in a threads environment. In our implementation, the parallel inner loops do not affect the completion strategy; they can be used in outer loop parallelisations which may give greater speed-ups but do affect the strategy. Our results present some hope that similar speed-ups are also possible for the inner loop of Buchberger's algorithm.

Our speed-ups can probably be improved further by (1) specialising the code for plain completion, (2) switching to an operating system with native threads, (3) writing an application specific scheduler, (4) using our parallel inner loop within multiple parallel outer loops.

7 Acknowledgements

This paper is based upon work supported by a grant from Deutsche Forschungsgemeinschaft. We thank Hans Boehm and Mark Weiser of Xerox Parc for letting us use their fine PCR system and for answering our questions. We also thank Michael Sperber for his implementation of the ALDES-to-C Compiler and Nick Nevin and Jeff Ward for their contributions to the S-threads system.

References

[1] J. Avenhaus and J. Denzinger. Distributing equational theorem proving. In C. Kirchner, editor, *Rewriting Techniques and Applications (LNCS 690)*, pages 62–76. Springer-Verlag, 1993. (Proc. RTA'93, Montreal, Canada, June 1993).

[2] L. Bachmair and N. Dershowitz. Critical pair criteria for completion. *Journal of Symbolic Computation*, 6:1–18, 1988.

[3] Buchberger, Collins, Encarnación, Hong, Johnson, Krandick, Loos, Mandache, Neubacher, and Vielhaber. Saclib user's guide, 1993. On-line software documentation.

[4] B. Buchberger and R. Loos. Algebraic simplification. In *Computer Algebra*, pages 14–43. Springer-Verlag, 1982.

[5] R. Bündgen. Completion of integral polynomials by AC-term completion. In S. M. Watt, editor, *International Symposium on Symbolic and Algebraic Computation*, pages 70 – 78, 1991. (Proc. ISSAC'91, Bonn, Germany, July 1991).

[6] R. Bündgen. Simulating Buchberger's algorithm by Knuth-Bendix completion. In R. V. Book, editor, *Rewriting Techniques and Applications (LNCS 488)*, pages 386–397. Springer-Verlag, 1991. (Proc. RTA'91, Como, Italy, April 1991).

[7] R. Bündgen. Buchberger's algorithm: The term rewriter's point of view. In G. Kuich, editor, *Automata, Languages and Programming (LNCS 623)*, pages 380 –391, 1992. (Proc. ICALP'92, Vienna, Austria, July 1992).

[8] R. Bündgen. Reduce the redex → ReDuX. In C. Kirchner, editor, *Rewriting Techniques and Applications (LNCS 690)*, pages 446–450. Springer-Verlag, 1993. (Proc. RTA'93, Montreal, Canada, June 1993).

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

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

[11] J. Christian. Fast Knuth-Bendix completion: Summary. In N. Dershowitz, editor, *Rewriting Techniques and Applications (LNCS 355)*, pages 551–555. Springer-Verlag, 1989. (Proc. RTA'89, Chapel Hill, NC, USA, April 1989).

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

[13] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–115, 1987.

[14] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuven, editor, *Formal Models and Semantics*, volume B of *Handbook of Theoretical Computer Science*, chapter 6. Elsevier, 1990.

[15] N. Dershowitz and N. Lindenstrauss. An abstract concurrent machine for rewriting. In H. Kirchner and W. Wechler, editors, *Algebraic and Logic Programming*. Springer-Verlag, 1990. (Proc. ALP'90, Nancy, France, October 1990).

[16] J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational Strcutures*, volume 2 of *Handbook of Logic in Computer Science*, chapter 1. Oxford University Press, 1992.

[17] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational Problems in Abstract Algebra*. Pergamon Press, 1970. (Proc. of a conference held in Oxford, England, 1967).

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

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

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

[21] W. W. Küchlin. PARSAC-2: A parallel SAC-2 based on threads. In S. Sakata, editor, *Applied Algebra, Algebraic Algorithms, and Error-Correcting Codes: 8th International Conference, AAECC-8*, volume 508 of *LNCS*, pages 341–353, Tokyo, Japan, Aug. 1990. Springer-Verlag.

[22] W. W. Küchlin. On the multi-threaded computation of integral polynomial greatest common divisors. In S. M. Watt, editor, *Proc. 1991 Internatl. Symp. on Symbolic and Algebraic Computation: ISSAC'91*, pages 333–342, Bonn, Germany, July 1991. ACM Press. (Also OSU-CISRC-1/91-TR2).

[23] W. W. Küchlin. The S-threads environment for parallel symbolic computation. In R. Zippel, editor, *Computer Algebra and Parallelism*, volume 584 of *LNCS*, pages 1–18, Ithaca, NY, Mar. 1992. Springer-Verlag. (Proc. CAP'90, Ithaca, NY, May 1990).

[24] W. W. Küchlin and N. J. Nevin. On multi-threaded list-processing and garbage collection. In *Proc. Third IEEE Symp. on Parallel and Distributed Processing*, pages 894–897, Dallas, TX, Dec. 1991. IEEE Press.

[25] W. W. Küchlin and J. A. Ward. Experiments with virtual C Threads. In *Proc. Fourth IEEE Symp. on Parallel and Distributed Processing*, pages 50–55, Dallas, TX, Dec. 1992. IEEE Press.

[26] H. Melenk and W. Neun. Parallel polynomial operations in the large Buchberger algorithm. In J. Della Dora and J. Fitch, editors, *Computer Algebra and Parallelism*, Computational Mathematics and Applications, pages 143–158, London, 1989. Academic Press. (Proc. CAP'88, Grenoble, France, June 1988).

[27] D. Plaisted. Equational reasoning and term rewriting systems. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Logical Foundations*, volume 1 of *Handbook of Logic in Artificial Intelligence and Logic Programming*, chapter 5. Oxford University Press, 1993.

[28] R. Ramesh and I. V. Ramakrishnan. Optimal speedups for parallel pattern matching in trees. In P. Lescanne, editor, *Rewriting Techniques and Applications (Proc. RTA'87)*, volume 256 of *Lecture Notes in Computer Science*, pages 274–285, Bordeaux, France, May 1987. Springer-Verlag.

[29] W. Schreiner and H. Hong. The design of the PACLIB kernel for parallel algebraic computation. In J. Volkert, editor, *Parallel Computation (LNCS 734)*, pages 204–218. Springer-Verlag, 1993. (Second International ACPC Conference, Gmunden, Austria, October 1993).

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

[31] J. K. Slaney and E. L. Lusk. Parallelizing the closure computation in automated deduction. In M. E. Stickel, editor, *10th International Conference on Automated Deduction, (LNCS 449)*, pages 28–39. Springer-Verlag, 1990. (Proc. CADE'90, Kaiserslautern , Germany, July 1990).

[32] M. Sperber. Mørk: A generator for preprocessors. Master's thesis, Universität Tübingen, 1994.

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

[34] A. S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[35] J.-P. Vidal. The computation of Gröbner bases on a shared memory multiprocessor. In A. Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 429 of *LNCS*, pages 81–90, Capri, Italy, Apr. 1990. Springer-Verlag. Int. Symp. DISCO'90.

[36] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. In *12th ACM SOSP*, pages 114–122, 1989.

[37] K. A. Yelick and S. J. Garland. A parallel completion procedure for term rewriting systems. In D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *LNCS*, pages 109–123, Saratoga Springs, NY, June 1992. Springer-Verlag.