J. Symbolic Computation (1996) **11**, 1–22

A Para-Functional Programming Interface for a Parallel Computer Algebra Package†

WOLFGANG SCHREINER

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, A-4040 Linz, Austria
Email: Wolfgang.Schreiner@risc.uni-linz.ac.at

(Received 30 May 1995, Revised 27 February 1996)

We describe the application of pD, a small para-functional language that we have developed as a high-level programming interface for the parallel computer algebra package PACLIB. pD provides several facilities to express parallel algorithms in a flexible way on different levels of abstraction. The compiler translates a pD module into statically typed parallel C code with explicit task creation and synchronization constructs. This target code can be linked with the PACLIB kernel, the multi-processor runtime system of the computer algebra library SACLIB. The parallelization of several computer algebra algorithms on a shared memory multi-processor demonstrates the elegance and efficiency of this approach.

1. Introduction

In our Ph.D. thesis (Schreiner 1994b) we have developed a para-functional language pD as a high-level programming interface for the parallel computer algebra kernel PACLIB (Hong *et al.* 1993). PACLIB represents a multiprocessor variant of the runtime system for SACLIB (Buchberger *et al.* 1992), a widely used C library of about 650 functions that efficiently implements many computer-algebra algorithms. In pD, parallelism may be expressed by several forms of annotations; the compiler generates parallel PACLIB C code with explicit task creation and synchronization statements.

While most computer algebra algorithms (Buchberger *et al.* 1982) are based on the concept of pure functions, real-world computer algebra programs are for performance reasons usually written in a low-level imperative style (typically in C). Consequently, there is little resemblance between the notation in which the mathematical theory is formulated and the notation in which the code is actually programmed. In particular, it is difficult to perform the restructuring necessary to exhibit the parallelism contained in a problem i.e. to transform a sequential algorithm into a parallel one. Because of low-level programming details, it is a rather time-consuming process to experiment with different parallelization strategies or different variants of a parallel algorithm.

† Supported by the FWF grant S5302-PHY "Parallel Symbolic Computation".

0747–7171/90/000000 + 00 $03.00/0 © 1996 Academic Press Limited

We sketch in this paper a programming style where the programmer is freed from some of these technical details and may concentrate on the essential tasks. The part of the programmer is to formulate a parallel algorithm in a functional notation and mark some critical spots. The part of the compiler is to create the necessary task creation and synchronization statements such that the denoted parallelism is efficiently utilized.

We present performance results for four pD implementations of parallel computer algebra algorithms. These programs use external SACLIB functions for integer and polynomial arithmetic (and in two cases a sequential C function for a specific subproblem). The performance is in all examples basically the same as for manually written C programs that directly use the parallel programming features of the PACLIB kernel (Stahl 1992, Hong and Loidl 1994).

There are several related projects going on in various research communities. Research groups on computer algebra have developed parallel variants of computer algebra packages, e.g. PAC (Roch 1990), PARSAC (Küchlin 1990), Linda/Maple (Char 1990) all of which are based on parallel variants of C. ||MAPLE|| (Siegl 1993) is based on the (interpreted) guarded Horn clause language STRAND.

Research groups on functional programming have developed parallel variants of functional languages (a comprehensive bibliography has been compiled by Schreiner (1993), more recent work in this area is presented by Böhm and Feo (1995)). Unfortunately, there is still a strong tradeoff between efficiency and flexibility:

Among the more efficient approaches are SISAL (Feo *et al.* 1990), various data-parallel versions of functional languages (Blelloch *et al.* 1993), and several skeleton-based approaches (Darlington *et al.* 1993). These languages are often intended for numerical ("scientific") applications.

Among the approaches that put more emphasis on flexibility are various dialects of Lisp based on "futures" (Halstead, Jr. 1985), dataflow languages (Nikhil 1991), and para-functional variants of (typically lazy) functional languages (Hudak 1991). These languages support dynamic task creation but in general rely on hardware support or runtime checks for transparent task synchronization.

In contrast to all these approaches, the pD compiler utilizes a new scheme where the user may program in a para-functional (or future-based) style and the compiler analyzes the program to automatically insert the synchronization operations. Synchronization is deferred as far as possible (beyond function and task boundaries) to promote parallelism. The imperative target code is statically typed and runtime type checks can be avoided.

In this paper, we want to concentrate on the application of the pD system for particular computer algebra problems (Section 6). However, in order to make the paper self-contained, in Sections 2–5 we provide an overview on the runtime system, the programming language and the compilation process, respectively. This presentation is an extended version of a PASCO '94 paper (Schreiner 1994a).

2. The Runtime System

The generated target code is linked with a runtime system that provides first-order tasks and automatic memory management (garbage collection). Currently, we use the PACLIB kernel that has been described elsewhere in detail (Hong *et al.* 1993); in this paper we only give a short sketch:

The PACLIB kernel is a shared memory multi-processor variant of the runtime system of the computer algebra library SACLIB developed at our institute (Buchberger *et*

al. 1992). It provides automatic memory management with an interface that is upward compatible to the SACLIB kernel; consequently all sequential SACLIB functions (implemented in C) run without change and without loss of efficiency when linked with the PACLIB kernel.

The parallel runtime system is based on *virtual processors* (implemented by Unix processes) among which *tasks* (light-weight threads of execution) are scheduled. *Virtual tasks* (Küchlin and Ward 1992) are scheduled among real tasks to allow even more fine-grained parallelism but especially to reduce the total memory consumption of a program (Schreiner 1994c).

The PACLIB kernel has been implemented on a Sequent Symmetry shared memory multi-processor. The system has been used for implementing a variety of parallel computer algebra algorithms e.g. (Stahl 1992, Hong and Loidl 1994). It is also the predecessor of the multi-processor version of the C++-based symbolic computation library STURM (Hong *et al.* 1994).

Please note that while the PACLIB kernel is implemented on shared memory, its interface is essentially *functional* i.e. tasks communicate only by arguments and results. Consequently this programming model can be also implemented on *distributed memory* hardware (in fact, the STURM library supports this model on massively parallel processors and on networks of workstations (Bertoli *et al.* 1996)).

3. The Functional Language

pD is based on a small functional language with HASKELL-like syntax but strict execution semantics. This language called D is the "sequential" core of pD and is compiled into standard ANSI C code. We chose the functional framework because of our experience with the SACLIB library.

While the SACLIB algorithms are written in C, they are essentially functional in spirit. A SACLIB function constructs from its input arguments an object that is returned as the result without any side effects. Moreover, all SACLIB objects are based on the data structure "list" (and fixed-size tuples implemented as lists); thus many functions operate by iterating over lists and constructing new lists.

The motivation for designing and implementing D was to express the SACLIB algorithms currently formulated in C on a higher level but without loss of efficiency. The generated C code should match in structure and quality the existing code. In short, D provides the following functionality:

Polymorphic Types. D supports a simplified polymorphic type system where a type signature must be explicitly given for every function. For every application of a polymorphic function f to an object of type T, a monomorphic function f_T is generated. Thus the target code is monomorphically typed (which allows to operate on objects of different size).

Higher Order Functions. D supports functions as arguments to other functions (but not as function results). For every application of a higher-order function f to a (possibly curried) function g, a first-order version f_g is generated by instantiating the higher-order parameter. This restricted form of higher order functions serves in D as templates for computational patterns.

Iterated Compositions. D provides a general mechanism for expressing computational patterns without the use of recursion. This mechanism has been inspired by the concept of list comprehensions in modern functional languages but has a different flavor.

The general syntax of an "iterated composition" is

C `[` e `|` I_1 `|` ... `|` I_n `]`

where the iterators I_i generate a sequence of variable bindings in which the expression e is evaluated. The composer C describes in which way the individual objects are combined. Instead of defining the general (rather retrieved) syntax, we give several examples of its application:

```
list [ f(x,y) | x <- l | y <- 1..x ]
```

constructs the list of $f(x, y)$ where x runs over the elements of l and, for every x, y runs over the integers from 1 to x. The composer `rlist` correspondingly constructs the result list in the reverse order.

```
comp(c,b) [ x | x = 1 ; p(x) ; x' = x+1 ]
```

lets x iterate from 1 to x_n where $x_n + 1$ is the smallest value larger than 1 for which $p(x)$ does not hold. The result is $c(x_1, c(x_2, \ldots c(x_n, b)))$ (right fold) where c is some user-defined function.

```
rcomp(c,b) [ x*y | y <- l, x = 1 ; p(x) ; x' = x+1 ]
```

lets y iterate over l and *simultaneously* x from x_1 to x_n; the result is $c(b, c(z_n, \ldots c(z_1, c)))$ (left fold) where $z_i = x_i * y_i$. Furthermore, there exists the composer `loop` that returns only the last generated value of the sequence.

The composers `comp` and `rcomp` describe two fundamental patterns of recursion (iterated right fold and left fold) from which the other composers are derived e.g. `list` = `comp(cons,[ ])`. This is different from most functional languages where list construction is considered as the fundamental concept and a fold operation may be applied to the result list. We thus automatically avoid the construction of the intermediate list without relying on further optimizations.

External Declarations. D provides external function and type declarations. A declaration `extern f :: S -> T` allows to call C functions from the functional layer. A declaration `extern proc f :: S -> (T1, T2)` allows to call a procedure `void f(S, T1*, T2*)` whose results are returned in an "unboxed" way (i.e. not as a tuple but by multiple output arguments).

A declaration `extern type T` declares an external type `T` whose structure is not known to the functional layer. This is required since several SACLIB objects do not match any object in the functional type system (e.g. SACLIB polynomials are flat lists $(e_0, c_0, e_1, c_1, \ldots)$ where the exponents e_i and the coefficients c_i are of different types). The compiler checks the correct typing of the program, but only external C functions can operate on the components of such structures.

Code Optimizations. The compiler applies several code optimizations in a rather aggressive way (e.g. inlining of all non-recursive functions and constant folding). Tail-recursive functions are transformed into loops. Likewise (non-tail)recursive list constructions are transformed into loops and result tuples are transformed into multiple output parameters if lists are involved.

Table 1. Benchmark Results

Benchmark	n	D	SML/NJ
primes	15000	2580 ms	2550 ms
queens	11	1690 ms	2750 ms
qsort	2500	5350 ms	9150 ms

For example, the functional program

```
split(e, l) =
   case l of
     [ ] => ([ ], [ ])
     h:t => let (a,b) = split(e, t)
            in if h < e
              then (h:a, b)
              else (a, h:b)
```

is compiled into the (beautified) procedure

```
split(int e, List l, List *a, List *b)
{
  while (!ISNIL(l)) {
    INT h = HEAD(l);
    if (h < e) {
      a = RCONS(h, a);
      l = TAIL(l);
    }
    else {
      b = RCONS(h, b);
      l = TAIL(l);
    }
  }
  *a = NIL;
  *b = NIL;
}
```

which is called as `split(e,l,&a,&b)` with output parameters `a` and `b`. The allocation function `RCONS` is a variant of the more familiar `CONS` operator: `RCONS(`h`,`t`)` allocates a new list cell, writes the element h into the cell head and the address of the cell into the location t; the address of the cell tail is returned as its result.

We have compared the execution times of several benchmark programs to corresponding programs implemented in the New Jersey variant of Standard ML (version 0.93, i386@20MHz, all optimizations on). SML/NJ (Milner *et al.* 1990) is a strict functional language whose compiler generates highly optimized code; most current implementations of lazy languages are considerably slower. The results listed in Table 1 demonstrate the high quality of the generated target code.

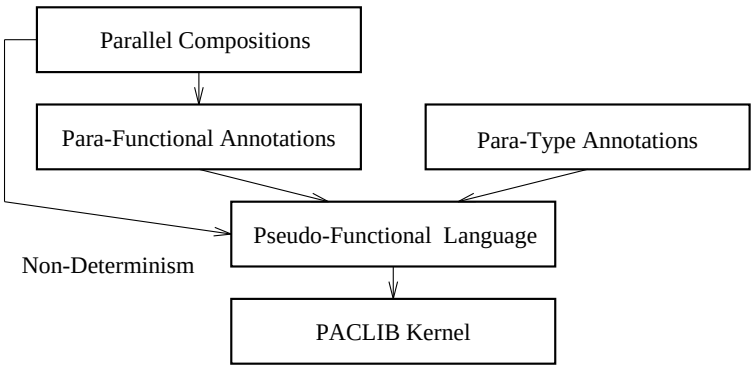

Figure 1. The pD Language Layers

4. The Parallel Language

In this section, we introduce the constructs of pD that allow to express parallelism in several levels of abstraction. The compiler translates each level into the corresponding lower one (see Figure 1).

4.1. Pseudo-Functional Operations

At the lowest functional level, we introduce the types `Task` and `Stream` and corresponding conversion functions.

Task Creation and Synchronization. The operator `start(`f`,`a_i`)` returns an object t of type `Task(`T`)` for $f : S \to T$. Analogously, `wait(`t`)` returns an object of type T for t : `Task(`T`)` such that `wait(start(`f`,`a_i`))` $= f(a_i)$.

Semantically, `start` denotes non-strict function application in contrast to the strict "operator" `( )` i.e. `start` terminates whenever the a_i terminate.

Stream Construction and Selection. The constant `snil` : `Stream(*a)` denotes the empty stream. `scons(`h`,`t`)` returns a stream of type `Stream(`T`)` for $h : T$ and t : `Task(Stream(`T`))`. The selection operator `scase` s `of {[ ] => `a`; `h`:`t` => `b`}` decomposes s : `Stream(`T`)` into $h : T$ and t : `Task(Stream(`T`))`, respectively.

Semantically, streams are not different from lists except for the type interface. Also technically, a stream cell is just a list cell whose tail field holds a task that returns either `nil` or a reference to the next stream cell.

Non-Determinism and Task Abortion. `any(`l`)` takes a list l : `[Task(`T`)]` and returns a pair t : `(`T`,[Task(`T`)])` whose first element is the result of any task in l and whose second component is the list without this task. `stop(`l`)` asserts that l : `[Task(`T`)]` is only referenced once, that any task in l is only referenced by l and that the current task will not use l any more.

Both constructs impose semantic difficulties. First of all, `any` is non-deterministic and thus destroys the referential transparency of the (therefore pseudo-) functional language. The same expression may return different results on different occasions, which makes reasoning much more difficult. On the other hand, its application allows in some cases to drastically reduce synchronization delays leading to more efficient parallel programs.

Likewise, `stop` is an annotation that states some program property by which the compiler may optimize the program but that it may not verify in general. Its incorrect application may yield erroneous programs. We tackle this problem by defining a restricted context (non-deterministic compositions) in which `any` and `stop` can be safely applied but that is referential transparent for the observer (see Subsection 4.3).

The pseudo-functional language may be directly used by the programmer e.g. in

```
pmap :: (*a -> *b, [*a]) -> [*b]
pmap(f,l) = list [ wait(y) | y <- r ]
  where r <- list [ start(f,x) | x <- l ]
```

Its primary purpose however is to serve as the intermediate language between the imperative target code and the para-functional annotation language.

4.2. Para-Functional Annotations

At the pseudo-functional level, the programmer deals with `Task` and `Stream` objects and has to care for correct typing by inserting synchronization functions and modifying function interfaces (with possible source code duplications) in case that objects that contain tasks migrate beyond function boundaries.

The goal of the para-functional layer is to relieve the programmer from this tedious work. This allows him to concentrate on the essential part i.e. the formulation of the parallel algorithm. For this purpose, there exist three basic forms of annotations:

1. **Task Annotations.** An expression e may be annotated as e`@` denoting that e shall be evaluated by a parallel task.
2. **Stream Annotations.** A list construction h`:`t may be annotated as h`:@`t denoting that the result is a stream cell. Hence t has to be extended to a parallel task that returns a stream as well.
3. **Type Annotations.** A constant c may be annotated as `para` c `::` T where T may contain `Task` and `Stream` subtypes. The compiler will extend c to the denoted type.

Neither does the result semantics of the program change by these annotations, nor does its observable type. It is the role of the compiler to modify the program by inserting `start` and `wait` calls for task creation and synchronization, changing function interfaces and duplicating code. A well-typed program in the pseudo-functional language is created that implements the parallel algorithm indicated by the programmer in a more or less efficient fashion. For instance, we may write

```
list [ 1+x | x <- pmap(f,l) ] where
  pmap :: (*a->*b, [*a]) -> [*b]
  pmap(f,l) = list [ f(x)@ | x <- l ]
```

which the compiler essentially transforms into

```
list [ 1+wait(x) | x <- pm(f,l) ] where
  pm :: (*a->*b, [*a]) -> [Task(*b)]
  pm(f,l) = list [ start(f,x) | x <- l ]
```

Using para-functional (type) annotations, the programmer may also write cyclic constant definitions as in the well-known program for prime number computation

```
para plist :: Task(Stream(Int))
plist = 3 : sieve(n, 5, plist)
```

where the result list is defined in terms of itself. The compiler will transform `plist` into the result of a forward-declared task t that returns a stream and receives t itself as its argument.

4.3. Parallel Compositions

The language supports parallel variants of iterated compositions that are translated into para-annotated recursive functions. For instance,

```
stream [ f(x,y) | x <- 1 | y <- 1..x ]
```

creates a stream of tasks where `stream` is simply defined as `comp(:@,[ ])`.

More general, every composition form may be annotated as in

```
comp(sum,0) [ prod(Ai,B) | Ai <- A |@ ]
```

which is translated into

```
iter(A) where
  iter(A) =
    case A of
      [ ] => 0
      Ai:Ar => sum(prod(Ai,B),iter(Ar)@)
```

denoting that the recursion is transformed into a chain of task creations and corresponding sequential synchronizations. The annotated form

```
rcomp(sum,0) [ prod(Ai,B) | Ai <- A |@ ]
```

is correspondingly interpreted as

```
iter(A,0) where
  iter(A,R) =
    case A of
      [ ] => R
      Ai:Ar => iter(Ar,sum(prod(Ai,B),R)@)
```

where the compositions are performed in the reverse direction. In nested iterations the programmer may explicitly annotate the desired parallelization level e.g. in

```
comp(sum,0) [ x*y | x <- 1 |@ y <- x..n ]
```

$$\frac{U \to_{\mathrm{T}} V}{U \to_{\mathrm{T}} \mathrm{Task}(V)} \qquad \frac{U_1 \to_{\mathrm{T}} V_1 \ldots U_n \to_{\mathrm{T}} V_n}{(U_1 \ldots U_n) \to_{\mathrm{T}} (V_1 \ldots V_n)}$$

$$\frac{U \to_{\mathrm{T}} V}{[U] \to_{\mathrm{T}} [V]} \qquad \frac{U \to_{\mathrm{T}} V}{[U] \to_{\mathrm{T}} \mathrm{Stream}(V)}$$

Figure 2. A Type Relation

only the outer iteration level is parallelized.

Furthermore, we have the new composer *non-deterministic combination*

```
ndcomp(c,b) [ e | I |@d ]
```

which indicates that the function $c : (T, T) \to T$ is commutative and associative such that *any* permutation of the combination order gives the same result. Likewise the composer *tree composition*

```
tcomp(c,b) [ e | I |@d ]
```

asserts that $c : (T, T) \to T$ is associative such that all generated tasks may be combined in a binary tree fashion.

5. Compilation

In this section, we sketch the compilation of a program in the para-functional annotation language into a well-typed pseudo-functional program with explicit task creation and synchronization. The formal details can be found in the Ph.D. thesis (Schreiner 1994b).

The core problem is that the translation of an annotation `@` into a `start` call introduces a type disorder that has to be eventually resolved by a corresponding `wait`. The goal is to defer this synchronization as much as possible such that parallel execution may take place.

Type Relation. To describe the effects of the translation, we assume a type system containing atomic types (integers, booleans), lists, tuples, tasks and streams. We then define a type relation $\to_{\mathrm{T}}$ as the reflexive transitive closure of the derivation rules given in Figure 2. The motivation for this definition is stated by the following

Lemma 5.1. (Synchronization/Extension) *If $U \to_{\mathrm{T}} V$, then there exists an extension function* $\mathrm{ext}[U, V] = e : U \to V$ *and a synchronization function* $\mathrm{sync}[V, U] = s : V \to U$ *such that e does not use* `wait`*, s does not use* `start`*, and for any $u : U$, $s(e(u)) = u$ and, for any $v : V$, $e(s(v)) = v$.*

Definition 5.1. (Intersection) *The intersection $T_1 \sqcap_{\mathrm{T}} T_2$ of two types T_1 and T_2 is*

that type U such that

$$U \to_{\mathrm{T}} T_1, U \to_{\mathrm{T}} T_2,$$
$$\forall V.(V \to_{\mathrm{T}} T_1, V \to_{\mathrm{T}} T_2) \to U \not\to_{\mathrm{T}} V$$

(if such a U exists).

Likewise, we define the union of two types $T_1 \sqcup_{\mathrm{T}} T_2$. If $T_1 \sqcap_{\mathrm{T}} T_2$ (respectively $T_1 \sqcup_{\mathrm{T}} T_2$) exists, it is unique.

Definition 5.2. (Maximum Extension) *The maximum extension* $\max_{\mathrm{T}}(U, V)$ *of a type U in direction V (with $U \to_{\mathrm{T}} V$) is the largest type T with $U \to_{\mathrm{T}} T \to_{\mathrm{T}} V$ such that T does not contain any subtype* $\mathrm{Task}(\mathrm{Task}(T'))$*. If already U contains such a subtype, we define $T := U$.*

The typing algorithm proceeds by the application of a set of transformation rules with the property

Lemma 5.2. (Transformation) *The application of a rule to an expression $e : U$ yields an expression $d : V$ where*

1 $U \to_{\mathrm{T}} V$ (types are only extended).
2 d is in the parallel core language and well typed,
3 $\mathbf{S}[\![sync[V, U](d)]\!] = \mathbf{S}[\![e]\!]$ (the semantics is consistent),
4 the interfaces of functions do not change.

We now give a selection of these rules where $e \vdash d : T$ reads as "e is transformed into d of type T". The transformation of `@` introduces a `start` call that changes the type of the expression:

$$\frac{e \vdash d : T}{e\texttt{@} \vdash \begin{array}{l}(\texttt{let}\ f(x_1, \ldots, x_n)\ \texttt{=}\ d \\ \texttt{in start}(f, x_1, \ldots, x_n)) : \mathit{Task}(T)\end{array}}$$

At strict basic operators, tasks have to be synchronized:

$$\frac{e_1 \vdash d_1 : T_1, e_2 \vdash d_2 : T_2}{e_1\ \texttt{+}\ e_2 \vdash \begin{array}{l}(\mathrm{sync}[T_1, \mathrm{Int}](d_1) \\ \texttt{+}\ \mathrm{sync}[T_2, \mathrm{Int}](d_2)) : \mathrm{Int}\end{array}}$$

In list constructions, both head and tail are constrained to the same base type. We extend both to the minimum common supertype (if it exists, else they are synchronized to the maximum common subtype):

$$\frac{\begin{array}{c}e_h \vdash d_h : T_h, e_t \vdash d_t : T_t \\ T_t = \mathrm{Task}^i(S), S = [U], V = T_h \sqcup_{\mathrm{T}} U\end{array}}{e_h\ \texttt{:}\ e_t \vdash (\mathrm{ext}[T_h, V](d_h)\ \texttt{:}\ \mathrm{ext}[T_t, [V]](d_t)) : [V]}$$

The result of a function definition may have an extended type. We synchronize the function result to the maximum allowed type and introduce a new function with the extended interface (recorded in *syncret*):

$$\frac{\begin{array}{c} e_i \vdash d_i : T_i, e \vdash d : T \\ U_i = \max_{\mathrm{T}}(S_i, T_i) \end{array}}{\begin{array}{c} \texttt{let } {c_i}^{R_i \to S_i}(x_j) \texttt{ = } e_i \texttt{ in } e \ \vdash \\ (\texttt{let } {c'_i}^{R_i \to U_i}(x_j) \texttt{ = } \mathit{sync}[T_i, U_i](d_i), \\ {c_i}^{R_i \to S_i}(x_j) \texttt{ = sync}[U_i, S_i]({c'_i}(x_j)) \texttt{ in } d) : T \\ \mathit{syncret}[U_i, S_i, {c'_i}] = {c_i}^{R_i \to S_i} \end{array}}$$

Any application now uses the extended result:

$$\frac{c^{S_i \to S} = \mathit{syncret}[T, S, c']}{id^{S_i \to S}(e_i) \ \vdash \ {id'}^{S_i \to T}(e_i) : T}$$

The application of a function to arguments of extended types introduces a function that synchronizes these arguments before calling the original one. Later, this function may be replaced by a copy of the original where synchronization takes place *inside* the body.

$$\frac{\begin{array}{c} e_i \vdash d_i : T_i \\ U_i = \max_{\mathrm{T}}(S_i, T_i) \end{array}}{\begin{array}{c} id^{S_i \to S}(e_i) \ \vdash \ {c'}^{U_i \to S}(\mathit{sync}[T_i, U_i](d_i)) : S \\ \mathit{syncarg}[U_i, S_i, c] = {c'}^{U_i \to S} \end{array}}$$

The Transformation Algorithm

1. **Transformation.** Apply the rules resolving applications of *syncret* to defer the synchronization of function results from the callee to the caller.
2. **Argument Extension.** Resolve applications of *syncarg* to defer the synchronization of function arguments from the caller to the callee.
3. **Iteration.** Repeat the algorithm until no more applications of *syncret* and *syncarg* functions are generated.

In the argument extension phase, argument synchronization is deferred only as far as it can be verified that the argument is only used in non-recursive branches of the function (and the execution of the task can thus overlap with the recursive evaluation of the function).

Please note that a function definition in the para-annotated source code may yield multiple pseudo-functional definitions: different applications of this function may pass arguments of different extended task types; also the return type of a function may be extended in different ways.

Lemma 5.3. (Correctness) *Let P be a program and P' the same program with all para-functional annotations removed. The transformation algorithm terminates on P and generates a well-typed pseudo-functional program P'' that produces the same result as P' (provided that P' terminates).*

The last clause in parentheses is necessary because parallel functional programs with cyclic constant definitions (see Section 6.3) run into deadlock if the parallelization annotations are removed.

The transformation algorithm terminates because function arguments and results are

never extended beyond the finite set of maximum extension types. For all of our parallel programs, we have only observed 3–5 iterations.

Finally, a set of optimizations are applied to avoid multiple synchronizations of the same task and to define explicitly the order of task creation and synchronization statements and function applications. For instance, the expression

```
wait(start(f,x))+wait(start(g,y))
```

must be transformed into

```
let a = start(f,x)
    b = start(g,y)
in wait(a)+wait(b)
```

to exploit inherent parallelism.

6. Application Examples

We are now going to describe para-functional implementations to three computer algebra problems: the exact solution of linear equation systems with multiprecision integer coefficients, the computation of multivariate polynomial resultants (two programs), and the computation of the Q-matrix in Berlekamp's algorithm for polynomial factorization. For these problems various parallel algorithms have been formerly implemented in C using the features of the PACLIB kernel (Stahl 1992, Hong and Loidl 1994)†.

For each program, Figure 3 displays three diagrams denoting the actual execution time, the relative speedup, and the visualized runtime behavior. The execution times were measured on a Sequent Symmetry computer with 20 processors i386 running at 20 MHz. The speedups are relative to the execution times of the corresponding C functions of the SACLIB library. The program behavior is illustrated by a time/task diagram where each horizontal line represents the execution of a concurrent task; the total program runtime is displayed in the right lower corner. This visualization is based on profiling information generated by the PACLIB kernel.

The size of the source code of the para-functional programs is 1/3 to 1/2 of the size of the corresponding PACLIB C source (without comments and without the main program in charge of input/output). The generated C target is about twice the size of the manually coded program; roughly the same ratio applies to the corresponding object files. The increase in code size is mainly a consequence of inlining.

6.1. Solution of Linear Equation Systems

Let A be a regular $n \times n$ matrix over $\mathbb{Z}$ and b a n vector over $\mathbb{Z}$. We want to find the unique n vector x over $\mathbb{Q}$ with $Ax = b$.

An efficient parallel solution to this problem proceeds as follows (see Figure 4): we take k primes p_j (for some bound k determined from n and the maximum entry in (A, b)) and

† Please note that, while the presented para-functional programs are independent solutions, they are based on the same *parallel algorithms* as the corresponding PACLIB programs. The algorithmic ideas for parallelization have to *precede* the implementation.

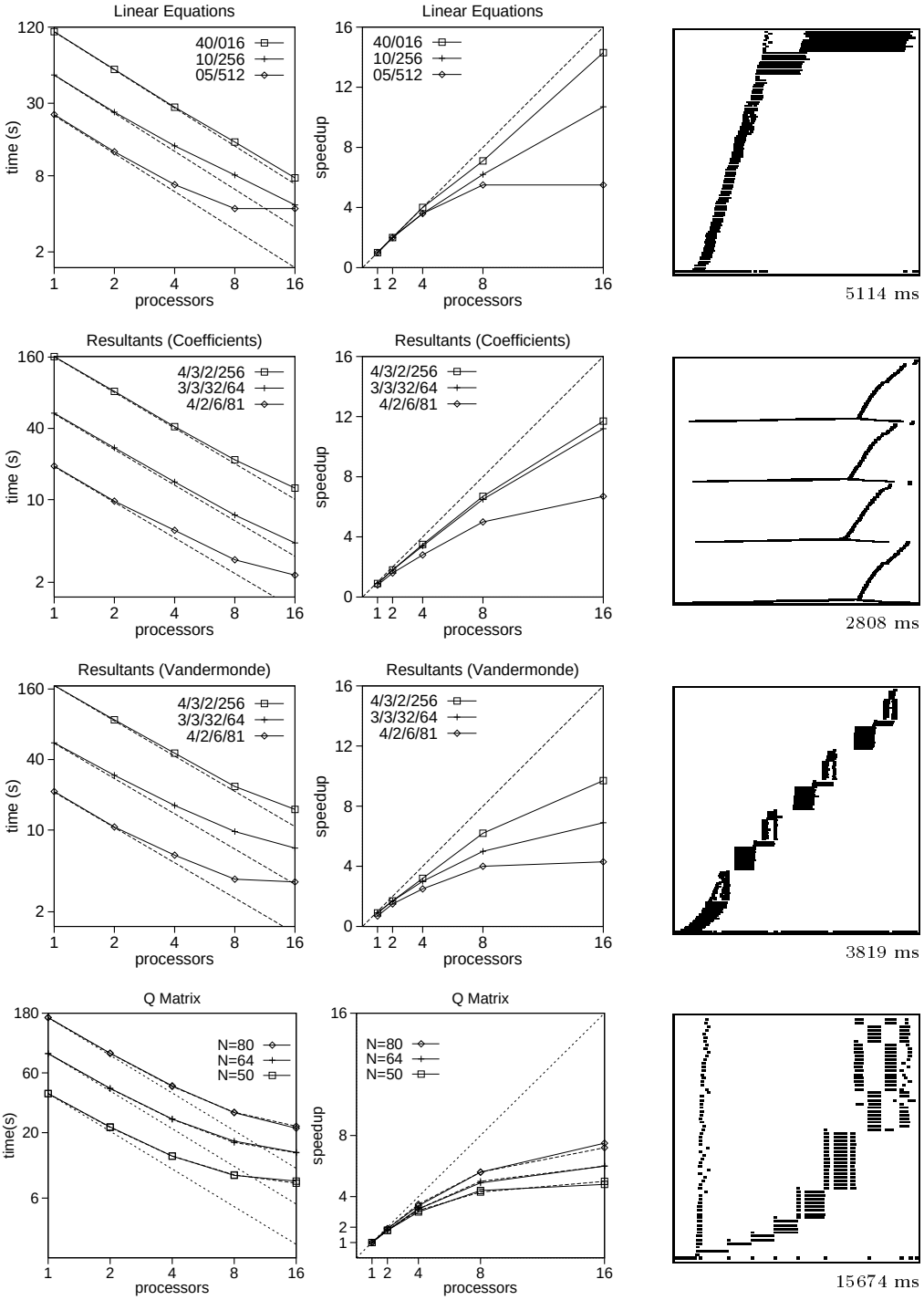

Figure 3. Application Results

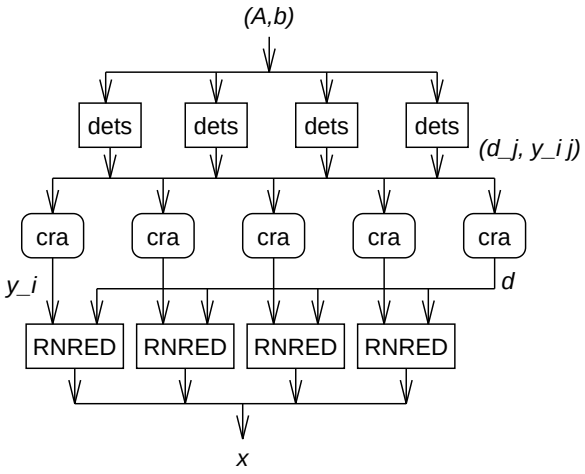

Figure 4. Linear Equation Systems

start k tasks. Each task concurrently maps (A, b) into the homomorphic image (A_j, b_j) over the finite field $\mathbb{Z}/p_j$ and computes $n+1$ determinants $d_j, y_{ij} \in \mathbb{Z}/p_j$ (with $1 \leq i \leq n$). $n+1$ other tasks receive these results and use the Chinese remainder algorithm to combine the $d_j \in \mathbb{Z}/p_j$ into $d \in \mathbb{Z}$ and the $y_{ij} \in \mathbb{Z}/p_j$ into n values $y_i \in \mathbb{Z}$, respectively. n new tasks receive these results to compute in parallel $x_i = y_i/d \in \mathbb{Q}$.

The para-functional solution uses the SACLIB C code for determinant computation in an external function `dets` as a "black box". The structure of the program is essentially as follows:

```
(dj_yij,pj,Pj,Ij) = comp(...) [ (d_yi,...) |
  p <- PRIMES, ... ; ... ; ...
  d_yi = dets(A, b, n, p)@ ]
d = cra(pj,Ij,Pj,dj_yij)@
y = list [ cra(pj,Ij,Pj,dj_yij,i)@ | i <- 1..n ]
x = list [ RNRED(yi,d)@ | yi <- y ]
```

The tasks that compute the determinants by executing `dets` are stored in the list `dj_yij`; their exact number has to be determined by a dynamic criterion at runtime. `dj_yij` is passed to a task that executes `cra` to compute `d` and to n tasks that execute `cra` and are stored in `y`. `x` represents the tasks that compute the final results from `d` and from the result of every task in `y`. Please note that all synchronizations are performed *within* the tasks that demand the results of other tasks; the main task just waits for the results of the tasks recorded in `x`.

Figure 3 shows the results of the parallelization for various pairs n/l (where l is the bit length of the integer coefficients in A, b). The execution times are virtually ($\pm 5\%$) the same as for the corresponding PACLIB C implementation of this method. The sublinear speedups for small n are not a consequence of the parallelization overhead but simply of the limited parallelism in the final reduction phase where only n tasks are active.

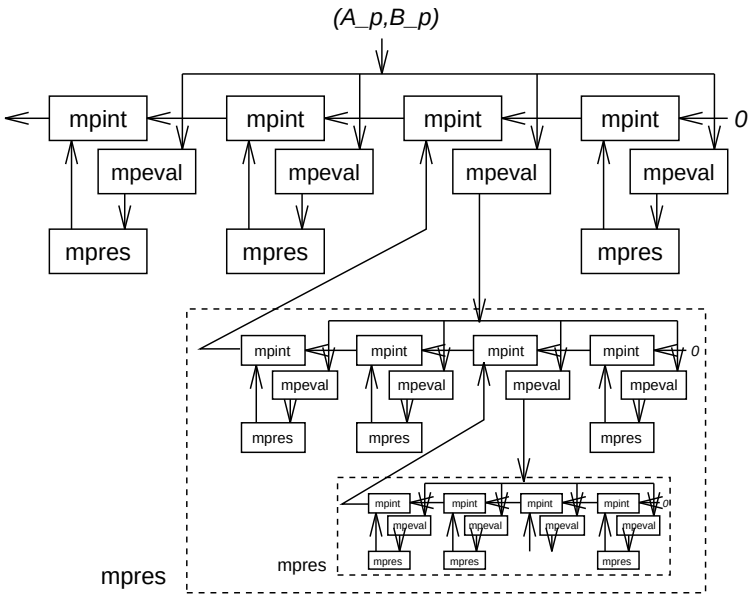

Figure 5. Multivariate Polynomial Resultants

6.2. Multivariate Polynomial Resultants

Let A and B be polynomials in $\mathbb{Z}[x_1, \ldots, x_{r-1}][x_r]$ i.e. with r variables and integer coefficients. The Sylvester matrix of A and B is a $(m+n) \times (m+n)$ matrix $M_{A,B}$ containing the coefficients $a_i, b_i \in \mathbb{Z}[x_1, \ldots, x_{r-1}]$ (i.e. polynomials with $r-1$ variables) arranged in some special manner. We want to compute the *resultant* of A and B which is the determinant $\det(M_{A,B}) \in \mathbb{Z}[x_1, \ldots, x_{r-1}]$ (i.e. a polynomial in $r-1$ variables).

One of the most efficient solutions of this problem utilizes parallelism on *three different levels*:

On the outermost parallelization level, we start k tasks that map A and B into the homomorphic images $A_p, B_p \in \mathbb{Z}/p[x_1, \ldots, x_{r-1}][x_r]$ (i.e. polynomials with coefficients from a finite field) and compute $\det(A_p, B_p)$. The Chinese remainder algorithm (which is also parallelized) then constructs $\det(A, B)$.

On the second level, the computation of $\det(A_p, B_p)$ in r variables proceeds recursively by computing in parallel k determinants (for some k) in $r-1$ variables and combining the sub-results by interpolation. If r falls below some bound, a sequential algorithm is called. This algorithm gives rise to a k-ary tree of tasks as sketched in Figure 5; its algorithmic structure is essentially expressed by the parallel composition

```
mpres(r,p,A,B) = ...
  comp(mpint,...) [ mpres(r-1,p,As,Bs) |
    ...; ...; ...
    As = mpeval(r,p,A,i),
    Bs = mpeval(r,p,B,i) |@ ]
```

However, since the combination phase `mpint` is very time consuming, it is important to parallelize it as well (third parallelization level). Here we attempt two algorithms:

6.2.1. Coefficient-wise Combine

This approach parallelizes the interpolation algorithm itself by simultaneously processing the coefficients of the polynomials. We express this by the annotation

```
mpint(As,Bs) = ... Cs where
  ...
  para Cs :: Stream(IntMon)
  Cs = cra(As, as)
```

where `cra` traverses the monomial lists As and as of two polynomials to construct the result monomial list.

The actual pD source code of this variant of the algorithm (including all three parallelization levels) is presented in the appendix.

6.2.2. Vandermonde Combine

Here the main program is rewritten to use the SACLIB C program `IVMMD` for computing the inverse Vandermonde matrix which is a $k \times k$ matrix constructed from the evaluation points used for the k determinants in $r-1$ variables:

```
mpres(r,p,A,B) = ... C where
  D = stream [ mpres(r-1,p,As,Bs) |
    ...; ...; ...
    As = mpeval(r,p,A,i),
    Bs = mpeval(r,p,B,i) ]
  C = rlist [ (i,MDMPVP(r-2,p,Pi,D))@ |
    Pi <- IVMMD(...), ... ]
```

6.2.3. Comparison

The diagrams given in Figure 3 show almost the same execution times for the functional programs as the C solutions described in (Hong and Loidl 1994) ($\pm 5\%$). The keys in these diagrams describe the characteristics of the randomly generated polynomials A and B, e.g. 4/2/6/81 denotes polynomials with 81 monomials in 4 variables of maximum degree 2 with integer coefficient of at most 6 bits.

The sublinear speedups are a consequence of the parallelization overhead and of the algorithm which demands for minimum time complexity the sequential iteration of `mpint` phases (where every iteration is parallelized as described above). Like in the C versions, the Vandermonde combine turns out to be less efficient than the coefficient-wise combine.

These factors that limit the parallelization do less depend on the language and its compilation but more on the algorithm; they affect the C implementation in the very same way. Actually we achieve in the coefficient-wise combine solution a slightly higher speedup than the C program by the application of virtual tasks that allow finer grained parallelism.

6.3. Q-Matrix Computation

Let p be a prime and A be a monic univariate polynomial over $\mathbb{Z}_p[x]$ of degree n. Then the matrix $Q := (q_{ij})_{i=0}^{n-1}$ is uniquely defined by the following relation (for all $0 \leq i \leq n-1$)

$$\sum_{j=0}^{n-1} q_{ij} x^j \equiv x^{ip} \pmod{A}$$

i.e. row i of the matrix consists of the coefficients of the polynomial $x^{ip} \pmod{A}$. The Q matrix is required as an intermediate result in several computer algebra algorithms e.g. in Berlekamp's method for factorizing univariate polynomials.

In the sequential algorithm, an auxiliary table of remainders $x^k \pmod{A}$ is constructed for $k = n \ldots 2n-2$. In the subsequent computations of polynomial products, powers of x are replaced by the corresponding remainders stored in the table:

Row 0 of the Q matrix is $x^0 = 1$. If $p \leq 2n-2$, row 1 is determined by the auxiliary table. If $p > 2n-2$, row 1 is computed by iteratively squaring $x^{2n-2} \pmod{A}$ until the exponent of the polynomial is as close to p as possible. This intermediate result is multiplied by x until $x^p \pmod{A}$ is available. All polynomial products are simplified using the auxiliary table.

Every subsequent row $x^{kp} \pmod{A}$ of the matrix is determined by multiplying its predecessor row $x^{(k-1)p} \pmod{A}$ with x using again the auxiliary table for simplifications.

Minimair (1992) describes a parallel PACLIB variant where the auxiliary table and the two first rows of the matrix are computed as in the sequential version. The parallelization idea is that, if rows i and j (i.e. the polynomials x^{ip} and x^{jp}) are available, row $i+j$ can be computed as the polynomial product $x^{(i+j)p} = x^{ip} * x^{jp}$.

The pD version of this algorithm starts a task for every row of the matrix. Rows 0 and 1 are computed as in the sequential algorithm. Task $k > 1$ computes its result by squaring the result of task $k/2$ (if k is even) or multiplying the results of tasks $\lfloor k/2 \rfloor$ and $\lfloor k/2 \rfloor + 1$ (if k is odd). Every task k depends therefore on at most two tasks i with $i < k$. Beginning with a single task, the number of tasks ready for execution grows exponentially.

This idea is expressed by a a cyclic construction of the polynomial sequence `QP`; the coefficients of every polynomial in this sequence denote the entries of one row of the Q-matrix (see Figure 6):

```
para QP :: Task([Task(ModPol)])
Y = first(A)
QP = ModPolOne : Y :
   list [ row(i, QP) | i <- 2..n-1 ]
QM = list [ coeffs(P) | P <- QP ]
```

A single `para` annotation suffices for the compiler to extend every polynomial computation to a task and to create the necessary task synchronization operations within the functions `row` and `coeffs`. While pD is basically a *strict* language (i.e. objects have to be fully evaluated before they can be used), the creation of tasks is *non-strict* (i.e. synchronizations on task results only take place when required), which allows to apply cyclic constructions as that shown above.

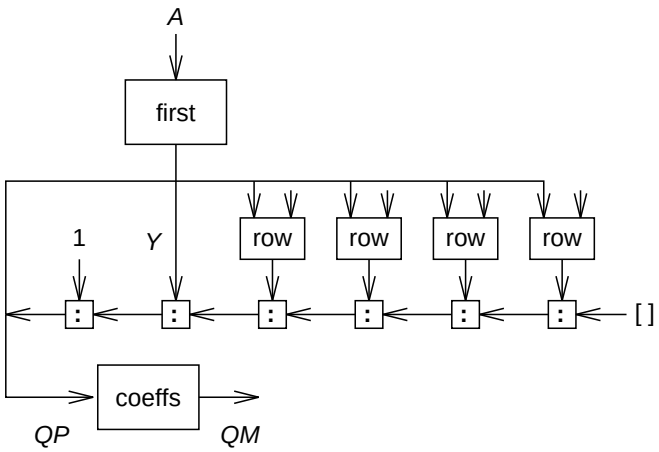

Figure 6. Q-Matrix Computation

7. Conclusions

The pD system shows that para-functional programs may be as efficient as C programs but are much more compact and thus more amenable to experiments with different parallelization strategies. Non-functional elements that are essential for the performance (non-determinism, destructive operations) may be encapsulated by referentially transparent functional language constructs or external functions.

Our goal is *not* automatic parallelization in the sense that a compiler extracts parallelism from the implementation of a sequential algorithm. This is a very difficult task (even when expressed in a functional language, a sequential algorithm remains sequential) and in general beyond the capabilities of automatic tools. Instead our goal is to find a suitable framework for *expressing parallel algorithms*, such that the source code is elegant and the target code is efficent. We believe that in an application area like computer algebra functional programming languages may be the right basis.

The main problem we have encountered during our experiments is the lack of an obvious relationship between the denotation of a program and its dynamical behavior in the parallel context. It is often difficult to anticipate the effects of different formulations of an algorithm and/or of different forms of para-functional annotations. Another pending issue is the introduction of arrays and of some notion of overloading for use with the new C++ based symbolic computation library STURM (Hong *et al.* 1994). Based on the port of the STURM runtime kernel to workstation clusters, we will study the coarse grain parallelization of computer algebra algorithms in the functional framework.

References

Bertoli, Piergiorgio, Hong, Hoon, Podisor, Ovidiu (1996). The Design of the Sturm Parallel Kernel. Technical Report 96-11, RISC-Linz, Johannes Kepler University, Linz, Austria, May.

Blelloch, Guy E., Chatterjee, Siddharta, Jonathan, C. Hardwick, Sipelstein, Jay, Zagha, Marco (1993). Implementation of a Portable Nested Data-Parallel Language. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP*, pages 102–112, San Diego, California, May 19–22.

Böhm, A. P., Feo, T., John, editors (1995). *High Performance Functional Computing Conference*, Denver, Colorado, April 9–11. Lawrence Livermore National Laboratory Report CONF-9504126.

Buchberger, B., Collins, G. E., Loos, R., Albrecht, R., editors (1982). *Computer Algebra — Symbolic and Algebraic Computation*. Springer, Vienna, New York.

Buchberger, Bruno, Collins, George, Encarnación, Mark, Hong, Hoon, Johnson, Jeremy, Krandick, Werner, Loos, Rüdiger, Mandache, Ana, Neubacher, Andreas, Vielhaber, Herbert (1992). A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler University, Linz, Austria.

Char, Bruce W. (1990). Progress Report on a System for General-Purpose Parallel Symbolic Algebraic Computation. In *International Symposium on Symbolic and Algebraic Computation — ISSAC'90*, pages 96–103, Tokyo, Japan, August 20–24. ACM Press.

Darlington, J., Field, A. J., Harrison, P. G., Kelly, P. H. J., Sharp, D. W. N., Wu, Q. (1993). Parallel Programming Using Skeleton Functions. In Bode, Arndt, editor, *PARLE 93: Parallel Languages and Architectures Europe*, volume 694 of *Lecture Notes in Computer Science*, Munich, Germany, June 14–18. Springer, Berlin.

Feo, J. T., Cann, D. C., Oldehoeft, R. R. (1990). A Report on the SISAL Language Project. *Journal of Parallel and Distributed Computing*, 10(4):349–366.

Halstead, Jr., Robert H. (1985). Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538.

Hong, Hoon, Neubacher, Andreas, Schreiner, Wolfgang (1993). The Design of the SACLIB/PACLIB Kernels. In Miola, Alfonso, editor, *DISCO '93 — International Symposium on Design and Implementation of Symbolic Computation Systems* (invited paper), volume 722 of *Lecture Notes in Computer Science*, Gmunden, Austria, September 15–17. Springer, Berlin.

Hong, Hoon, Loidl, Hans-Wolfgang (1994). Parallel Computation of Modular Multivariate Polynomial Resultants on Shared Memory Machine. In Buchberger, Bruno, Volkert, Jens, editors, *Parallel Processing: CONPAR 94 — VAPP VI International Conference on Parallel and Vector Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 325–336, Linz, Austria, September 6–8. Springer, Berlin.

Hong, Hoon, Neubacher, Andreas, Stahl, Volker (1994). The STURM Library Manual. Technical Report 94-31, RISC-Linz, Johannes Kepler University, Linz, Austria, April.

Hudak, Paul (1991). Para-Functional Programming in Haskell. In Szymanski, Boleslaw K., editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 5, pages 159–196. ACM Press, New York.

Küchlin, Wolfgang, Ward, Jeffrey (1992). Experiments with Virtual C Threads. In *4th IEEE Symposium on Parallel and Distributed Processing*, Arlington, Texas, December. IEEE Press.

Küchlin, Wolfgang (1990). PARSAC-2: A Parallel SAC-2 Based on Threads. In Sakata, S., editor, *Eighth International Symposium on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC-8)*, volume 508 of *Lecture Notes in Computer Science*, pages 206 –217, Tokyo, Japan, August. Springer, Berlin.

Milner, R., Tofte, M., Harper, R. (1990). *The Definition of Standard ML*. MIT Press, Cambridge, MA.

Minimair, Manfred (1992). Computation of the Q-Matrix. Internal report, RISC-Linz, Johannes Kepler University, Linz, Austria, June.

Nikhil, R. S. (1991). Id (Version 90.1) Reference Manual. CSG Memo 284-2, MIT Laboratory for Computer Science, Cambridge, MA, July.

Roch, J. L. (1990). An Environment for Parallel Algebraic Computation. In Zippel, R. E., editor, *Computer Algebra and Parallelism, Second International Workshop on Parallel Algebraic Computation*, volume 584 of *Lecture Notes in Computer Science*, pages 33–50, Ithaca, USA, May. Springer, Berlin.

Schreiner, Wolfgang, Hong, Hoon (1993). A New Library for Parallel Algebraic Computation. In Sincovec, Richard F. et al., editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume II, pages 776–783, Norfolk, Virginia, March 22–24. SIAM.

Schreiner, Wolfgang (1993). Parallel Functional Programming — An Annotated Bibliography (2nd Edition). Technical Report 93-24, RISC-Linz, Johannes Kepler University, Linz, Austria, May.

Schreiner, Wolfgang (1994a). A Para-Functional Programming Interface for a Parallel Computer Algebra Package. In Hong, Hoon, editor, *PASCO 94 International Symposium on Parallel Symbolic Computation*, Hagenberg/Linz, Austria, September 26–28. World Scientific Publishing, Singapore.

Schreiner, Wolfgang (1994b). *Parallel Functional Programming for Computer Algebra*. PhD thesis, Johannes Kepler University, Linz, Austria, September. RISC-Linz Report 94-61.

Schreiner, Wolfgang (1994c). Virtual Tasks for the PACLIB Kernel. In Buchberger, Bruno, Volkert, Jens, editors, *Parallel Processing: CONPAR 94 — VAPP VI International Conference on Parallel and Vector Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 533–544, Linz, Austria, September 6–8. Springer, Berlin.

Siegl, Kurt (1993). Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–186, San Diego, California, May 19-22. ACM Press.

Stahl, Volker (1992). Solving Systems of Linear Equations with Modular Arithmetic on a MIMD Computer. Technical Report 92-62, RISC-Linz, Johannes Kepler University, Linz, Austria, October.

Appendix: Multivariate Polynomial Resultants

```
------------------------------------------------------------------------------
--
-- Integral Multivariate Polynomial Resultant Computation
-- Based on PACLIB version by Hoon Hong
--
-- Arguments:
--   r:    a digit, r > 0
--   A, B: in Z[X1,...,Xr], A and B of positive degree in the main variable.
--
-- Result:
--   C: in Z[X1,...,X_{r-1}], the resultant of A and B w.r.t. the main variable
--
------------------------------------------------------------------------------
pIPRES(r, A, B) where

  -- expression is compiled to C function with free variables as parameters
  extern r :: Digit
  extern A :: IntPol
  extern B :: IntPol

  -- interface to SACLIB C library (external functions and types)
  #include "saclib.i"


  ------------------------------------------------------------------------------
  -- integral polynomial resultant, parallel version
  ------------------------------------------------------------------------------
  pIPRES :: (Digit, IntPol, IntPol) -> IntPol
  pIPRES(r, A, B) = C where

    m = IPDEG(A); n = IPDEG(B)
    d = IEXP(IPMAXN(r, A), n)
    e = IEXP(IPMAXN(r, B), m)
    f = IDPR(IPROD(IFACTL(m+n), IPROD(d, e)), 2)

    C = rcomp(ICOMB, IntPolZero) [
          (pMPRES(r, p, As, Bs), Q, p)
        |
          Q = IntOne, p <- PRIME
        ;
          ICOMP(Q, f) < 0
        ;
          Q' = if D then IPROD(Q, DTOI(p)) else Q,
          D = MPDEG(As) == m && MPDEG(Bs) == n,
          As = MPHOM(r, p, A), Bs = MPHOM(r, p, B) ----------------------------
        |@                                          -- 1. parallelization level
        ; D ]                                      ----------------------------

    ICOMB :: ((ModPol, Integer, Digit), IntPol) -> IntPol
    ICOMB((Cs, Q, p), C) = C0 where
      q = MDINV(p, MDHOM(p, Q))
      C0 = pIPCRA(Q, p, q, r-1, C, Cs)


  ------------------------------------------------------------------------------
  -- integral polynomial Chinese remainder algorithm, parallel version.
  ------------------------------------------------------------------------------
  pIPCRA :: (Integer, Digit, Digit, Digit, IntPol, ModPol) -> IntPol
  pIPCRA(M, m, mp, r, A, a) =
    if r == 0
      then ITOIP(MIDCRA(M, m, mp, IPTOI(A), MPTOD(a)))
      else IPStdRep(R)
```

```
  where                                          ----------------------------
    para R :: Stream(IntMon)                     -- 2. parallelization level
    R = cra(IPMonoList(A), MPMonoList(a))        ----------------------------

    cra :: (IntMonList, ModMonList) -> IntMonList
    cra(Ap, ap) =
      case Ap of
        [ ] =>
          case ap of
            [ ] => [ ]
            a:ap0 =>
              let (e, c) = a
              in (e, IPCRA(M, m, mp, r-1, IntPolZero, c)) : cra(Ap, ap0)
        A:Ap0 =>
          let (E, C) = A
          in case ap of
            [ ] => (E, IPCRA(M, m, mp, r-1, C, ModPolZero)) : cra(Ap0, ap)
            a:ap0 =>
              let (e, c) = a
              in if E > e then
                (E, IPCRA(M, m, mp, r-1, C, ModPolZero)) : cra(Ap0, ap)
              else if E < e then
                (e, IPCRA(M, m, mp, r-1, IntPolZero, c)) : cra(Ap, ap0)
              else
                (E, IPCRA(M, m, mp, r-1, C, c)) : cra(Ap0, ap0)


------------------------------------------------------------------------------
-- modular polynomial resultant, parallel version.
------------------------------------------------------------------------------
pMPRES :: (Digit, Digit, ModPol, ModPol) -> ModPol
pMPRES(r, p, A, B) =

  if r == 1 then
    MUPRES(p, A, B)
  else if (r <= CUT) then
    MPRES(r, p, A, B)
  else C where
    m = MPDEG(A)
    n = MPDEG(B)
    m1 = MPDEGSV(r, A, 1)
    n1 = MPDEGSV(r, B, 1)
    k = m*n1+m1*n

    (C, D) =
      comp(MCOMB, (ModPolZero, ModPolOne)) [
        (pMPRES(r-1, p, As, Bs), i)
      |
        i = 0
      ; i <= k
      ; i' = if D then i+1 else i,
        D = MPDEG(As) == m && MPDEG(Bs) == n,
        As = MPEVAL(r, p, A, 1, i),
        Bs = MPEVAL(r, p, B, 1, i)               ----------------------------
      |@                                         -- 2. parallelization level
      ; D ]                                      ----------------------------

    MCOMB :: ((ModPol, Digit), (ModPol, ModPol)) -> (ModPol, ModPol)
    MCOMB ((Cs, i), (C, D)) = (C0, D0) where
      b = MDINV(p, MPTOD(MPEMV(1, p, D, i)))
      C0 = pMPINT(p, D, i, b, r-1, C, Cs)
      Dp = MPCONST4(1, 1, 0, p-i)
      D0 = MPPROD(1, p, D, Dp)
```

```
-------------------------------------------------------------------------------
-- modular polynomial interpolation, coefficient-wise parallel version.
-------------------------------------------------------------------------------
pMPINT :: (Digit, ModPol, Digit, Digit, Digit, ModPol, ModPol) -> ModPol
pMPINT(p, B, b, bp, r, A, a) =
  if MPDEG(B) == 0 then
    MPINV(r-1, a, 1)
  else if r == 1 then
    let d = MDDIF(p, MPTOD(a), MPTOD(MPEMV(1, p, A, b)))
    in if (d == 0)
      then A
      else let c = MDPROD(p, d, bp)
           in MPSUM(1, p, MPMDP(1, p, c, B), A)
  else
    MPStdRep(R)
  where                                          ----------------------------
    para R :: Stream(ModMon)                     -- 3. parallelization level
    R = cra(MPMonoList(A), MPMonoList(a))        ----------------------------
    cra :: (ModMonList, ModMonList) -> ModMonList
    cra(Ap, ap) =
      case Ap of
        [ ] =>
          case ap of
            [ ] => [ ]
            a:ap0 =>
              let (e, c) = a
              in (e, MPINT(p, B, b, bp, r-1, ModPolZero, c)) : cra(Ap, ap0)
        A:Ap0 =>
          let (E, C) = A
          in case ap of
            [ ] => (E, MPINT(p, B, b, bp, r-1, C, ModPolZero)) : cra(Ap0, ap)
            a:ap0 =>
              let (e, c) = a
              in if E > e then
                (E, MPINT(p, B, b, bp, r-1, C, ModPolZero)) : cra(Ap0, ap)
              else if E < e then
                (e, MPINT(p, B, b, bp, r-1, ModPolZero, c)) : cra(Ap, ap0)
              else
                (E, MPINT(p, B, b, bp, r-1, C, c)) : cra(Ap0, ap0)
```