Proceedings of PASCO'94

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
Wolfgang.Schreiner@risc.uni-linz.ac.at

Abstract: We describe the design and implementation of pD, a parallel variant of a small functional language that serves as a 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.

Keywords: Para-Functional Programming, Computer Algebra, Compilation.

1 Introduction

We present in this paper a small functional language D and its para-functional variant pD that is used as a parallel programming interface for the computer algebra package PACLIB. Parallelism may be expressed by several forms of annotations; the compiler generates strongly typed parallel C code with explicit task creation and synchronization statements.

While most computer algebra algorithms are based on the concept of mathematical functions, computer algebra programs that are intended for computationally intensive applications are usually written in a rather low-level imperative style. 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.

We discuss 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 three pD implementations of parallel computer algebra algorithms. These programs use external SACLIB [2] 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 explicitly parallel C programs that directly use the features of the PACLIB runtime system [20, 7].

There are several related projects going on in various research communities. Research groups on computer algebra have developed parallel va-

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

riants of computer algebra packages, e.g. PAC [14], PARSAC [11], Maple/Linda [3] all of which are based on parallel variants of C. ||MAPLE|| [19] is based on the (interpreted) guarded Horn clause language STRAND.

Research groups on functional programming have developed parallel variants of functional languages (for a bibliography, see e.g. [16]). Unfortunately, there is still a strong tradeoff between efficiency and flexibility. Among the more efficient approaches are SISAL [5], various data-parallel versions of functional languages [1], and several skeleton-based approaches [4]. 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" [6], dataflow languages [13], and para-functional variants of (typically lazy) functional languages [10]. These languages support dynamic task creation but in general rely on hardware support or runtime checks for transparent task synchronization.

We sketch in this paper a (we believe) 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.

Section 2 describes the PACLIB run-time kernel which is the oldest part of the system and has served as the basis of a set of parallel computer algebra programs. Section 3 describes D i.e. the "sequential" subset of the para-functional language; the parallel extensions to pD are discussed in Section 4. Section 5 sketches the compilation of pD into the explicitly parallel core; Section 6 presents several application results.

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 [8, 18]; in this paper we only give a short sketch:

The PACLIB kernel is a parallel variant of the runtime system of the computer algebra library SACLIB [2] for shared memory multi-processors. It provides automatic memory management with an interface that is upward compatible to the SACLIB kernel; consequently all 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* are scheduled among real tasks to allow even more fine-grained parallelism but especially to reduce the total memory consumption of a program. This concept has been developed by Küchlin [12] and was further refined in [18].

The PACLIB kernel has been implemented on a Sequent Symmetry shared memory multiprocessor. The system has been used for implementing a variety of parallel computer algebra algorithms e.g. [20, 7]. It is also the essential substrate of the multi-processor version of a new C++ library of computer algebra algorithms [9].

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 interface can be also implemented on *distributed memory* hardware (in fact, a PVM-based cluster port is currently under way).

3 The Functional Language

In this section, we sketch the design and implementation of D, a small functional language with HASKELL-like syntax but strict execution semantics. This language is the "sequential" subset of pD and is compiled into ANSI C code. We chose the functional framework because of our experience with the computer algebra library SACLIB developed at our institute.

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 [15], the definition of D and its compilation into C code are described in detail. 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 defining computational patterns without explicit recursion. This mechanism is 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 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 only that returns 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 (inlining of all non-recursive functions, 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. 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

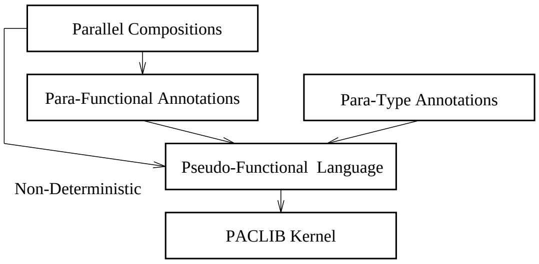

Figure 1: The Para-Functional Language

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 is a strict functional language whose compiler generates highly optimized code; current implementations of Haskell (which is non-strict) are considerably slower.

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

The results demonstrate the high quality of the generated target code.

4 The Parallel Language

In this section, we describe the constructs of pD that allow to express parallelism in several levels of abstraction. The compiler translates each level into the corresponding lower one. The relationships are described in Figure 1. In [17] the formal semantics of the individual constructions is described in detail.

This bottom-up construction allowed us to build every new form of abstractions on an already established sound basis. Furthermore, by the open definition of intermediate layers we maintain the freedom for the programmer to select for every program module the appropriate layer, a choice which is always a tradeoff between abstraction and flexibility.

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 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 making 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 task of the compiler to modify the program by inserting `start` and `wait` calls, 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 <- l | 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 <- l |@ y <- x..n ]
```

only the outer iteration level is parallelized.

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

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

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

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

Figure 2: A Type Relation

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. The compiler translates this expression into a pseudo-functional program with local non-determinism where the expression d controls the task granularity as follows: the parent task sequentially traverses the iterator I recording the generated variable bindings until the expression d becomes true.

Then a parallel task is created that sequentially evaluates e in these binding and combines the results. The parent task puts this task into a bag and continues execution until it has processed all bindings. Then it non-deterministically waits for any pair of terminating tasks, starts a new task that combines the results and puts this task into the bag. The combination of the last task result with b yields the final result. If the (optional) predicate p holds for some subresult, the combination may be aborted e.g.

```
ndcomp(and,True,not) [ p(x) | x <- l |@ ]
```

denotes the predicate $\forall x \in l.p(x)$.

Likewise the composer *tree composition*

```
tcomp(c,b,p) [ 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. We omit the details here.

Currently, the compiler does not provide special support for data parallelism as expressed by the form

```
list [ ... | ... |@ ... ]
```

i.e. this expression is not treated different from general `comp` forms. This reflects the fact that pure data parallelism is not so dominant in symbolic applications as in numerical programs. We therefore rely in this form on virtual threads for granularity control and do not (yet) provide an annotation `|@`d in combination with loop tiling.

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 [17].

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 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 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 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 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:

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

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*):

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

Any application now uses the extended result:

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

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.

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

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 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 clause in parentheses is necessary because functional programs with cyclic constant definitions must be properly annotated in order not to run into a deadlock.

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.

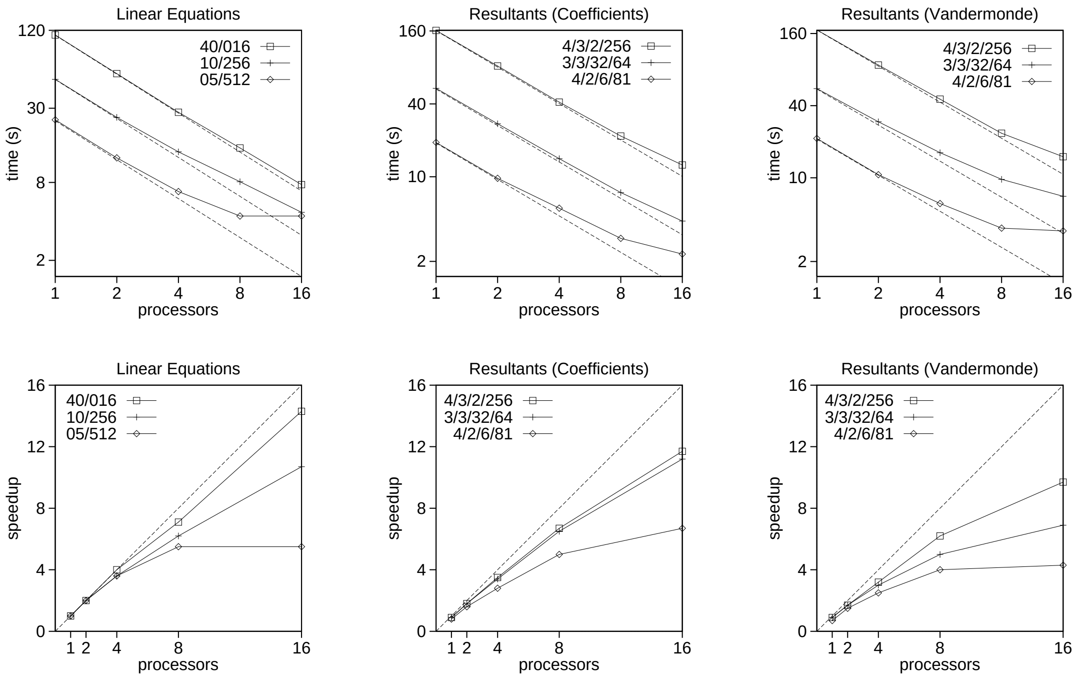

Figure 3: Application Results

6 Application Results

We are now going to sketch para-functional solutions to two computer algebra problems: the (exact) solution of linear equation systems over the integers and the computation of multivariate polynomial resultants. For both problems various parallel algorithms were already implemented in PACLIB [20, 7]. The timings were performed on a Sequent Symmetry computer with 20 processors i386 running at 20 MHz. All speedups are relative to the best available SACLIB programs.

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: we take k primes p_j (for some particular bound k) and start k tasks. Each task maps (A, b) into the homomorphic image (A_j, b_j) over $\mathbb{Z}_{p_j}$ and computes $n+1$ determinants d_j, y_{ij} (for $1 \leq i \leq n$) over $\mathbb{Z}_{p_j}$. $n+1$ tasks receive these results to compute by the Chinese remainder algorithm d from the d_j and y_i from the y_{ij}. n new tasks receive these results to compute $x_i = y_i/d$.

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

```
(dj_yij,pj,Pj,Ij) =
  comp(...) [ (d_yi,...) |
    p <- PRIMES, ... ; ... ; ...
    d_yi = dets(A, b, n, p)@ ]
d = cra_d(pj,Ij,Pj,dj_yij)@
y = list [ cra_y(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 the task `d` that executes `cra_d` and to the n tasks that execute `cra_y` and that 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.

Multivariate Polynomial Resultants. Let A and B be polynomials in $\mathbb{Z}[x_1, \ldots, x_{r-1}][x_r]$. The Sylvester matrix of A and B is a $(m+n) \times (m+n)$ matrix $M_{A,B}$ containing the coefficients $a_i, b_i \in \mathbb{Z}[x_1, \ldots, x_{r-1}]$ arranged in some special manner. The resultant of A and B is then $\det(M_{A,B}) \in \mathbb{Z}[x_1, \ldots, x_{r-1}]$.

An efficient parallel solution starts k tasks that map A and B into the homomorphic images $A_p, B_p \in \mathbb{Z}_p[x_1, \ldots, x_{r-1}][x_r]$ and compute $\det(A_p, B_p)$. The Chinese remainder algorithm is applied to construct $\det(A, B)$. 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 algorithmic structure is essentially expressed by the parallel iterator

```
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. Here we use two algorithms:

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 = mpint0(As, as)
```

where `mpint0` traverses the monomial lists As and as of two polynomials to construct the result monomial list.

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(...), ... ]
```

The diagrams given in Figure 3 show almost the same execution times for the functional programs as the C solutions described in [7] ($\pm 5\%$). 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.

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

7 Conclusions

The pD programming interface has been useful for parallelizing a few non-trivial computer algebra algorithms. We believe that in this application area where algorithms are essentially functional pD programs are efficient but much more compact and allow experiments with different parallelization strategies more easily. Non-functional elements that are essential for the performance (non-determinism, destructive operations) may be encapsulated by functional language constructs or external functions.

The main problem is that there is no 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. Here is much room for improvements.

Another pending issue is the introduction of arrays (by an `array` form of iterated compositions) and of some notion of overloading for use with the C++ based STURM library [9]. When the port of the runtime kernel to workstation clusters is completed, it will be interesting to study the coarse grain parallelization of computer algebra algorithms in the functional framework. Also we intend to work on some compiler support for automatic parallelization by inserting para-functional annotations in promising places.

References

[1] G. E. Blelloch. Nesl: A Nested Data-Parallel Language (Version 2.6). Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

[2] B. Buchberger, G. Collins, et al. A SACLIB Primer. Technical Report 92-34, RISC-Linz, Johannes Kepler Univ., Linz, Austria, 1992.

[3] B. W. Char. Progress Report on a System for General-Purpose Parallel Symbolic Algebraic Computation. In *Int. Symp. on Symb. and Alg. Comp. — ISSAC'90*, Tokyo, Japan, Aug. 20–24, 1990.

[4] J. Darlington, A. J. Field, et al. Parallel Programming Using Skeleton Functions. In *PARLE 93: Parallel Languages and Architectures Europe*, Munich, Germany, June 14–18, 1993.

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

[6] R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM TOPLAS*, 7(4):501–538, 1985.

[7] H. Hong and H.-W. Loidl. Parallel Computation of Modular Multivariate Polynomial Resultants on Shared Memory Machine. In *CONPAR 94 - VAPP VI International Conference on Parallel and Vector Processing*, Linz, Austria, September 6–8, 1994.

[8] H. Hong, A. Neubacher, and W. Schreiner. The Design of the SACLIB/PACLIB Kernels. In *DISCO 93 — Int. Symposium on Design and Implementation of Symbolic Computation System*, vol. 722 of *LNCS*, Gmunden, Austria, Sept. 15–17, 1993.

[9] H. Hong, A. Neubacher, W. Schreiner, and V. Stahl. The C++ Interface to the STURM Multi-Processor Kernel. Technical Report 94-32, RISC-Linz, Johannes Kepler University, Linz, Austria, April 1994.

[10] P. Hudak. Para-Functional Programming in Haskell. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers*, Frontier Series, chapter 5, pages 159–196. ACM Press, New York, 1991.

[11] W. Küchlin. PARSAC-2: A Parallel SAC-2 Based on Threads. In *Eighth International Symposium on Applied Algebra, Algebraic Algorithms, and Error Correcting Codes (AAECC-8)*, volume 508 of *LNCS*, Tokyo, Japan, August 1990.

[12] W. Küchlin and J. Ward. Experiments with Virtual C Threads. In *4th IEEE Symp. on Par. and Distr. Processing*, Arlington, TX, December 1992.

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

[14] J. L. Roch. PAC: Towards a Parallel Computer Algebra Co-Processor. In *Computer Algebra and Parallelism*, pages 33–50. Academic Press, 1989.

[15] W. Schreiner. Compiling a Functional Language to Efficient SACLIB C. Technical Report 93-49, RISC-Linz, Johannes Kepler University, Linz, Austria, 1993.

[16] W. Schreiner. Parallel Functional Programming — An Annotated Bibliography. Technical Report 93-24, RISC-Linz, Johannes Kepler University, Austria, 1993.

[17] W. Schreiner. Compiling a Para-Functional Language to Parallel PACLIB C. Technical Report 94-44, RISC-Linz, Johannes Kepler University, Linz, Austria, 1994.

[18] W. Schreiner. Virtual Tasks for the PACLIB Kernel. In *CONPAR 94 - VAPP VI International Conference on Parallel and Vector Processing*, Linz, Austria, September 6–8, 1994. Springer, LNCS.

[19] K. Siegl. Parallelizing Algorithms for Symbolic Computation Using ||MAPLE||. In *Fourth ACM SIGPLAN Symp. on Principles and Practice of Par. Prog.*, San Diego, CA, May 19-22, 1993.

[20] V. Stahl. Solving Systems of Linear Equations with Modular Arithmetic on a MIMD Computer. Tech. Rep. 92-62, RISC-Linz, Johannes Kepler Univ., Linz, Austria, 1992.