An Automatic Parallelization Framework for Algebraic Computation Systems

Yue Li
Texas A&M University
yli@cse.tamu.edu

Gabriel Dos Reis
Texas A&M University
gdr@cse.tamu.edu

ABSTRACT

This paper proposes a non-intrusive automatic parallelization framework for typeful and property-aware computer algebra systems. Automatic parallelization remains a promising computer program transformation for exploiting ubiquitous concurrency facilities available in modern computers. The framework uses semantics-based static analysis to extract reductions in library components based on algebraic properties. An early implementation shows up to 5 times speed-up for library functions and homotopy-based polynomial system solver. The general framework is applicable to algebraic computation systems and programming languages with advanced type systems that support user-defined axioms or annotation systems.

Categories and Subject Descriptors

I.1.3 [**Symbolic and Algebraic Manipulation**]: Languages and Systems—*Special-purpose algebraic systems*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Algorithms, Performance, Languages, Experimentation

Keywords

Automatic parallelization, computer algebra, user-defined axioms

1. INTRODUCTION

Concurrency offered by modern computers has the potential of enabling efficient scientific computation. However, by and large, development of scalable concurrent software remains a challenge. Furthermore, manual modification of *existing* programs to benefit from ubiquitous concurrency is just as elusive, left to a few highly trained programmers. In this paper, we propose an automatic parallelization framework for computer algebra systems that take properties of algebraic entities they manipulate seriously.

We previously reported [12] on the rich opportunity for parallelization in algebraic libraries such as those of the AXIOM family systems. The intuition behind that work is that it is hard for a (well) structured algebraic algorithm not to reflect, in one way or the other, properties related to the entities it manipulates. Consequently, we built a semantics-based static analysis tool with the goal of detecting parallelizable reductions in algebraic libraries. In this paper, we show how to effectively exploit those findings, leading to an automatic parallelization framework.

The main thrust of this paper is a semantics-based source-to-source transformation. The transformation replaces sequential parallelizable reductions with their parallelized versions. It does not require the user to be an expert in parallel programming. Nor does it require write-access to the input program fragment, much less authorship. All that is needed is a description of algebraic properties of the input program fragment. The description language is an extension [12] of the "category" subset of the Spad programming language. For example, the content of a univariate polynomial with integer coefficients $P = \sum_0^n a_n X^n$ is the greatest common divisor of all the polynomial coefficients. It can be computed with a simple loop:

```
content(p: Polynomial(Integer)): Integer ==
  coefs : List(Integer) := coefficients(p)
  result : Integer := 0
  for c in  repeat
    result := gcd(result, c)
  return result
```

It is clear that this computation is a reduction of the *monoid operator* gcd over the coefficients of P. That suggests a computation strategy where pairs of coefficients are evaluated concurrently, and the results are themselves combined using the same divide-and-conquer pattern. That computation strategy corresponds to the following program:

```
content(p: Polynomial(Integer)): Integer ==
  coefs : List(Integer) := coefficients(p)
  result := parallelLeftReduce(gcd, coefs, 0)
  return result
```

where the function `parallelLeftReduce` performs reduction in parallel. Notice that while the example uses a polynomial with integer coefficients, all we need is a domain of computation where GCD computation makes sense and is effective, The function gcd remains a monoid operation, which we express in our extension to Spad [12] as:

```
forall(S: GcdDomain)
  assume MonoidOperator(S, gcd) with
    neutralValue = 0$S
```

The name `MonoidOperator` designates a category constructor that specifies what it means for an opetaor to be an monoid operator over a domain:

ISSAC'11, June 8–11, 2011, San Jose, California, USA.

```
MonoidOperator(T: BasicType, op: (T, T) -> T): Category
  == AssociativeOperator(T, op) with
    neutralValue: T
```

The parameterized assumption statement instructs the reduction detector of the program transformation framework that it can assume (without having to conduct a proof) that gcd is a monoid operation over any GCD domain.

The contributions of this paper include:

1. Specification and implementation of semantics-based program analysis algorithms for detecting parallel reductions in user library code (section 3). The detection is guided by user provided assumptions.

2. Specification and implementation of program transformation algorithms for generating parallel reductions from the reductions identified in 1 (section 4).

3. Performance evaluation of our automatic parallelization framework using multicore PC and cluster (section 6). Experimental results show up to 5 times speed-up for some programs (section 6.3).

2. INTERNAL DATA STRUCTURES

The analysis and transformation algorithms operate on an internal representation (IR) of the user input program. The source code is first elaborated to IR. The result of the parallelization phase is translated back to Spad syntax, which is then compiled as if it was the original program. The abstract syntax for a subset of the IR is defined as follows:

$$
\begin{array}{rrcl}
\textit{name} & \mathrm{x} & & \\
\textit{type} & \tau & ::= & \mathrm{x}(\tau^*) \mid (\tau^*) \rightarrow \tau \\
\textit{constant} & \mathrm{c} & ::= & \mathrm{Constant}(\mathrm{x}, \tau) \\
\textit{expression} & e & ::= & \mathrm{Funcall}(\mathrm{x}, e^*, \tau) \\
 & & \mid & \mathrm{Variable}(\mathrm{x}, \tau) \\
 & & \mid & \mathrm{Assign}(e, e) \\
 & & \mid & \mathrm{c} \mid \mathrm{a} \\
\textit{segment} & \mathrm{g} & ::= & \mathrm{Segment}(e, e, e) \\
\textit{unnamed function} & \mathrm{a} & ::= & \mathrm{Lambda}(\mathrm{Variable}(\mathrm{x}, \tau)^*, \mathrm{s}, \tau) \\
\textit{statement} & \mathrm{s} & ::= & e \mid \mathrm{s}^+ \\
 & & \mid & \mathrm{Declare}(\mathrm{x}, \tau) \\
 & & \mid & \mathrm{For}(\mathrm{x}, e, \mathrm{s}) \\
 & & \mid & \mathrm{For}(\mathrm{x}, \mathrm{g}, \mathrm{s}) \\
 & & \mid & \mathrm{While}(e, \mathrm{s}) \\
 & & \mid & \mathrm{Return}(e) \\
 & & \mid & \mathrm{If}(e, \mathrm{s}, \mathrm{s}) \\
\textit{function} & \mathrm{f} & ::= & \mathrm{FunDef}(\mathrm{x}, \mathrm{Variable}(\mathrm{x}, \tau)^*, \mathrm{S}, \tau)
\end{array}
$$

A function definition node carries a name, a list of parameters, a definition body and a signature. A function call has an operator and optional operands. An anonymous function is a function definition without name. A variable is declared with its type. The iteration range of a for-loop is a sequence represented by either a container expression or an integer segment.

The reduction detection algorithms and the Ir manipulation algorithms several internal operators:

- *getIterationVariable*: retrieves the iteration variable from a for-loop.
- *getIterationSequence*: obtains the iteration sequence of a for-loop.
- *getLoopBody*: retrieves the body of a for-loop.
- *getOperator*: obtains the operator of a function call.
- *getOperands*: returns the operand list of a function call.
- *genAssign*: constructs an assignment.
- *genAnonymousFunction*: creates a lambda expression.
- *genFunCall*: builds a function call.

Figure 1 shows the IR for the body of the iterative `content` function definition. Each IR node represents an expression. The directed edges between nodes indicate the control flow in a program. The undirected edges refer to the corresponding components of different IR nodes. The leaves of the IR nodes are expressions.

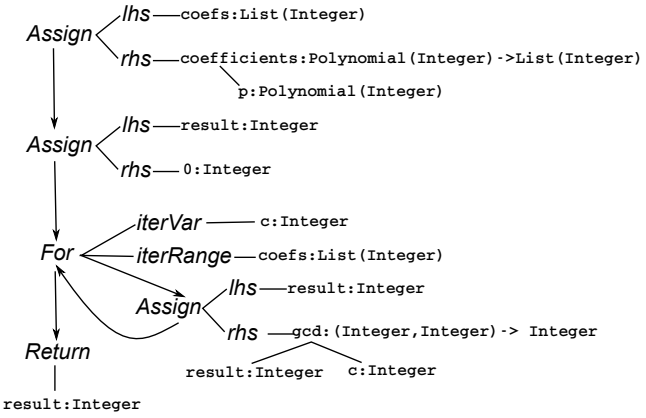

Figure 1: The internal representation of the `content` function body

Type and user assumption information is used by the static analysis and program transformation algorithms. These information are stored in environments defined as follows:

$$
\begin{array}{rrcl}
\textit{TypeEnv} & \Gamma & ::= & [\,] \mid (\mathrm{x} \mapsto \tau), \Gamma \\
\textit{property} & \mathrm{p} & ::= & [\,] \mid (\mathrm{x} \mapsto e), \mathrm{p} \\
\textit{PropEnv} & \mathrm{E} & ::= & [\,] \mid ((\mathrm{x}, \tau, \tau) \mapsto \mathrm{p}), \mathrm{E}
\end{array}
$$

A type environment maps identifiers to their types. An entry of a property environment is a collection of all properties attached to an operator. Each entry in a property environment has an operator name, a signature, a defining type, and a list of properties. Each property maps a property name to a value. For instance, the property $(\textit{neutralValue}, 1)$ means the property *neutralValue* has value 1.

3. REDUCTION DETECTION

The reduction detector looks for semigroup operators, and also monoid operators. A reduction can be expressed as a *accumulation loop*, or a *library function call* to `reduce`, or an application of the *built-in reduce operator*. The detection algorithm is a semantics-based static analysis performed in two steps: (1) properties collection; followed by (2) reduction extraction. We summarize the algorithm for detecting accumulation loops in this section. Algorithms for detecting reduce call and built-in reduce operator are simpler. More in-depth discussion may be found in our previous work [12].

The detector first examines every user written assumption. The purpose is to derive all the algebraic properties attached to the operators. The derivation instantiates property categories according to a specific assumption. Algebraic property information is collected

via the transitive closure of the instantiated categories. Consider the user assumption in section 1. Category `MonoidOperator` has two ancestor categories `AssociativeOperator` and `MagmaOperator`:

```
MagmaOperator(T: BasicType, op: (T,T) -> T): Category
  == Type

AssociativeOperator(T: BasicType,op: (T,T)->T): Category
  == MagmaOperator(T,op) with
    associativity:
      rule forall(a:T, b: T, c: T)
             op(a,op(b,c)) == op(op(a,b),c)
```

The definition of `MagmaOperator` means a binary operation on a domain satisfies the *magma operator* property. The `AssociativeOperator` category contains a logical statement of the property that an operator is associative if it is a magma operator and follows the associativity rule. Category `MonoidOperator` is instantiated with type `Polynomial Integer` and operator `gcd`. Variable `neutralValue` is given value polynomial `0`. The instantiated category has two ancestor categories `AssociativeOperator(Polynomial Integer,gcd)` and `MagmaOperator(Polynomial Integer,gcd)`. Therefore, the operator `gcd` carries all the properties of `MonoidOperator`, `AssociativeOperator` and `MagmaOperator`. At reduction extraction, the detector traverses the IR for the input source program. The extraction algorithm matches each IR node against the predefined parallel reduction patterns.

3.1 Detecting accumulation loop

One way for writing a reduction is to use an *accumulation loop*. The pattern of an accumulation loop is defined as a single for-loop of the following *recognizable form*:

$$\texttt{for i in l repeat } s^+$$

Variable i is the iteration variable of the loop, and l is a sequence. A new value from the sequence is yield at each iteration. The body of the loop is a sequence of statements s^+. Each statement s is an assignment a. An assignment a is of the form $v := f(X, e_1)$ or $v := f(e_2, X)$. Variable v is an accumulation variable. An accumulation variable has a linear occurrence on the right hand side of the assignment, and does not appear in sequence l. For each accumulation assignment, the same binary operator f has to be used consistently to accumulate values into v. The expression e is arbitrary, but must not mention v. The expression X is either the accumulation variable v, or of the form $f(X_1, e_1)$ or $f(e_1, X_1)$.

The pattern matching algorithm for accumulation loops is present in Algorithm 1. We start by preprocessing the input loop. The function *simplify* transforms an input loop using standard forward subexpression substitution [13]. The purpose of this preprocessing is to discover more "hidden" reductions. Consider the following code fragments:

```
a := 0
for i in 1..10 repeat
  r := i       -- local
  x := a       -- local
  a := x + r   -- global
```

```
a := 0
for i in 1..10 repeat
  a := a + i
```

the code on the left contains an accumulation loop. Without preprocessing, the loop is rejected during pattern matching. The reason is that the dependencies between each variable cause the violation of the recognizable form restriction. The preprocessed loop on the right is identified as an accumulation loop.

Each statement in an input loop is matched against the pattern of an accumulation assignment. This functionality is implemented

Algorithm 1 *isParallelAccumLoop*?(l: *Loop*, E: *PropEnv*, Γ: *TypeEnv*)

Require: l is an un-nested for-loop with body containing only assignments or definitions, E is a property environment, Γ is a type environment.
  $l' \leftarrow$ *simplify*(l)
  $v \leftarrow$ *getIterationVariable*(l')
  $r \leftarrow$ *getIterationSequence*(l')
  $b \leftarrow$ *getLoopBody*(l')}
  for s **in** b **do**
    if not *isAccumulationAssignment*?(s) **then**
      return false
  for s **in** b **do**
    $v \leftarrow$ *getAccumulationVariable*(s)
    if *dependsOn*(r, v) **then**
      return false
    for s' **in** $b - \{s\}$ **do**
      if *dependsOn*(s', v) **then**
        return false
  for s **in** b **do**
    op $\leftarrow$ *getAccumulationOperator*(s)
    if *associative*?(op, Γ, E) = unknown **then**
      return unknown
  return true

by the operator *isAccumulationAssignment*?. If each statement is an accumulation assignment, we continue checking that for each accumulation variable v, the iteration sequence and other assignments do not depend on v. The operator *getAccumulationVariable* takes out the variable at the left hand side of an accumulation statement. The operator *getAccumulationOperator* extracts the binary operator used in an accumulation statement, and *dependsOn* tests dependency between a statement and a variable, *i.e.*, whether the variable is read or written in the statement. Finally, the associativity of the accumulation operator in each statement is checked against user's assumption. The function *associative*? searches the property list of an accumulation operator for the associativity. If found, the function returns true, otherwise it returns unknown.

4. PROGRAM TRANSFORMATIONS FOR GENERATING PARALLEL REDUCTIONS

The transformation of an accumulation loop proceeds in several steps. First, we transform the body of the loop into a functional abstraction. This function computes the list of elements combined in the loop. Then parallel reduction is generated for combining the elements of the list. As an illustration, consider the following code for computing the n-th harmonic number:

```
harmonic(n: NonNegativeInteger): Fraction(Integer) ==
  h: Fraction(Integer) := 0
  for k in 1..n repeat
    h := h + 1/k
  return h
```

The transformation results in the following code:

```
harmonic(n: NonNegativeInteger): Fraction(Integer) ==
  h := parallelLeftReduce(+,_
         parallelMap((G768) +-> 1/G768,_
           [G769 for G769 in 1..n]),0)
  return h
```

In this example, a list of integer fractions `[1, 1/2, ..., 1/n]` are added to the accumulation variable `h`. In the transformed code, the list is computed in parallel with the function `parallelMap`. The

library function applies the unary function `(G768) +-> 1/G768` to each element of the sequence `[1..n]` in the loop iterator. The list is combined together by function `parallelLeftReduce`. The function performs a left associative reduction in parallel.

To parallelize reduce call and built-in reduce operator, we replace the reduce operator with parallel reduce function calls. Therefore, the parallelizations require to use parallel mapping and parallel reduction in the generated code.

4.1 A library for parallel mapping and reduction

We developed a small but sufficient Spad package `ParallelMapReduce` to provide the functionalities mentioned above. A package in the Spad language is a collection of function definitions. The interface of the package is shown below:

```
ParallelMapReduce(S:Type, R:Type):Public == Private where
  Public == with
    parallelMap   : (S -> R, List S) -> List R
    parallelRightReduce: ((S, R) -> R, List S, R) -> R
    parallelLeftReduce: ((R, S) -> R, List S, R) -> R
    ...
```

The package exports three functions `parallelMap`, `parallelRightReduce` and `parallelLeftReduce`. The definition of the package is parameterized by types `S` and `R`. The type parameter `S` is the element type of the list taken by parallel mapping and reduction. The type parameter `R` is the element type of the list returned by parallel mapping and the resultant type of the binary combination in a reduction. The expression `parallelMap(op, l)` applies `op` to the portions of `l` concurrently using threads. Functions `parallelRightReduce` and `parallelLeftReduce` are two parallel reduction operators. A parallel reduction takes a binary operator, a list and the neutral element of the binary operator. The elements of `l` are combined with the binary operator asynchronously. The function `parallelRightReduce` combines elements in a right associative way, *i.e.*, the accumulation variable is the right operand of the binary operator, and vice versa.

The input lists of parallel mapping and reduction are evenly split according to the number of working tasks. For parallel mapping, each task locally applies the same binary operator to the assigned piece of the original input list. In parallel reduction, each task performs sequential reduction on the assigned piece of the original input list. The local results are combined using a sequential reduction to produce the final result.

4.2 Transforming reductions

Different transformation algorithms are applied to reductions of different forms.

4.2.1 Accumulation loop

The algorithm for transforming accumulation loops is present in Algorithm 2. The correctness of the loop transformation is provided by the map-reduce programming model. Parallel mapping computes intermediate results which are consumed by parallel reduction. The transformation is done only if the accumulation operator is associative.

An accumulation loop with accumulation assignments in the body are first transformed to several accumulation loops. Each loop carries only one accumulation assignment. This is achieved through loop fission [2]. For instance, the loop on the left of the codes below is transformed to the two loops on the right:

Algorithm 2 *transformAccumLoop*($\mathtt{l}$: *Loop*, $\mathtt{E}$: *PropEnv*, Γ : *TypeEnv*)

Require: $\mathtt{l}$ is an accumulation loop with one accumulation assignment.
$\nu \leftarrow$ *getIterationVariable*($\mathtt{l}$)
$\mathtt{s} \leftarrow$ *getIterationSequence*($\mathtt{l}$)
$\mathtt{b} \leftarrow$ *getLoopBody*($\mathtt{l}$)
$\mathtt{a} \leftarrow$ *getAccumulationVariable*($\mathtt{b}$)
$\mathtt{f} \leftarrow$ *getAccumulationOperator*($\mathtt{b}$)
Let $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ be fresh variables
$\mathtt{let}_1 \leftarrow$ *genLet*($\alpha_1, \mathtt{s}$)
$\mathtt{g} \leftarrow$ *genAnonymousFunction*($\mathtt{b}, \nu$)
$\theta_2 \leftarrow$ *genParallelMap*($\mathtt{g}, \alpha_1, \Gamma$)
$\mathtt{let}_2 \leftarrow$ *genLet*(α_2, θ_2)
$\mathtt{id} \leftarrow$ *getNeutralValue*($\mathtt{f}, \mathtt{E}, \Gamma$)
if *leftAssociative*?($\mathtt{b}$) **then**
  $\theta_3 \leftarrow$ *genLeftParallelReduce*($\mathtt{f}, \alpha_2, \mathtt{id}, \Gamma$)
else
  $\theta_3 \leftarrow$ *genRightParallelReduce*($\mathtt{f}, \alpha_2, \mathtt{id}, \Gamma$)
$\mathtt{let}_3 \leftarrow$ *genLet*(α_3, θ_3)
$\theta_4 \leftarrow$ *genFunCall*($\mathtt{f}, [\nu, \alpha_3], \Gamma$)
$\mathtt{let}_4 \leftarrow$ *genLet*(ν, θ_4)
return *forwardSubstitution*($[\mathtt{let}_1, \mathtt{let}_2, \mathtt{let}_3, \mathtt{let}_4]$)

```
x := 0                          x := 0
y := 1                          y := 1
for i in 1..10 repeat           for i in 1..10 repeat
  x := x + i*i + i                x := x + i*i + i
  y := y * i                    for i in 1..10 repeat
                                  y := y * i
```

An accumulation loop is then transformed to a sequence of four assignments following Algorithm 2. The first assignment computes the list of elements produced by the loop iterator. The right hand side of the assignment is generated corresponding to iterators of different forms:

$$\alpha_1 := \begin{cases} e & \text{iterator: } \mathtt{for}\ \nu\ \mathtt{in}\ e \\ [\alpha_0\ \mathtt{for}\ \alpha_0\ \mathtt{in}\ e_1..e_2] & \text{iterator: } \mathtt{for}\ \nu\ \mathtt{in}\ e_1..e_2 \end{cases}$$

We use α_i to represent a fresh variable. Variable α_1 is passed to the next assignment. The right hand side of the next assignment calls parallelMap operator:

$$\alpha_2 := \text{parallelMap}(g, \alpha_1),$$

operator g is an unary anonymous function. The mapping of g over α_1 returns a list. The value of each element in the list is combined to the accumulation variable in the original accumulation loop. The parallel mapping is generated by function *genParallelMap*.

The third assignment reduces the binary accumulation operator f over the value of variable α_2. The result is assigned to variable α_3:

$$\alpha_3 := \begin{cases} \text{parallelRightReduce}(f, \alpha_2, c_i) & \text{for right reduction} \\ \text{parallelLeftReduce}(f, \alpha_2, c_i) & \text{for left reduction} \end{cases}$$

parallel reduction should be chosen following the associativity used in the original accumulation loop. The function *leftAssociative*? returns true, if the reduction is left associative. The operator f is given by function *getAccumulationOperator*. The constant c_i is the neutral element of operator f. The neutral value is obtained by calling function *getNeutralValue*, which looks up the assumption environment. The generations of function calls to the two parallel reductions are done using functions *genLeftParallelReduce* and *genRightParallelReduce*, respectively.

The last assignment combines the value of α_3 to the accumulation variable:

$$v := f(v, \alpha_3).$$

The accumulation variable v is extracted from the accumulation loop via function *getAccumulationVariable*.

At the final step, we call subroutine *forwardSubstitution* over the assignment sequence. The function implements forward expression substitutions. This transformation simplifies the four generated assignments into an single assignment.

4.2.2 Library function call and built-in reduce form

The other two kinds of reduction are transformed using Algorithm 3. Both reductions are function applications, using left associative reductions by default. Therefore, the operators of these reductions are simply replaced with `parallelLeftReduce`.

Algorithm 3 *transformReduce*(e : *Funcall*, E : *PropEnv*, Γ : *TypeEnv*)

Require: e is a function application of `reduce` or the built-in reduce operator.
op ← the operand of e which is a binary operator
s ← the operand of e which is a sequence
id ← *getNeutralValue*(op, E, Γ)
return *genLeftParallelReduce*(op, s, id, Γ)

5. IMPLEMENTATION

The entire framework is implemented as a library in the OpenAxiom computer algebra system. A graphical description of the framework is shown in Figure 2.

5.1 Concurrency in OpenAxiom

The parallel mapping and reduction package is implemented using the concept of *futures* [10]. The interfaces of *futures* are follows:

```
Future(T: Type): Public == Private where
  Public ==
    future: (() -> T) -> %
    get: % -> T
    ...
```

`future(t)` creates a future by taking in a function `t`. Function `t` will be executed in the background. Type `Future` is parameterized by type `T`. Variable `T` is the return type of function `t`. Once a future value is created, `t` starts execution. `get(f)` retrieves the computed value from future `f`. If the function wrapped by `f` terminates when `get` is called, result will be returned immediately. Otherwise `get` waits until the function finishes.

6. EXPERIMENTS

The automatic parallelization framework was tested in several configurations, including a software regression test, a set of algebra library functions, and a polynomial homotopy continuation package. We measured the performance of the sequential programs and their parallelized versions on a desktop PC and a computation node of the Brazos cluster [1]. Both machines use a GNU/Linux operating system. The tests were conducted wit an SBCL-based build of the OpenAxiom system. The desktop PC has one Intel Core2 2.4GHz dual-core processor with 4GB memory. The cluster computation node we used has two quad-core Intel Xeon E5420 2.5GHz processors with 32GB memory. The sequential versions used as references are the original source codes. Each sequential program is parallelized using two, four and eight threads, respectively.

6.1 A software installation test

We started with parallelizing a software regression test which takes long time to complete. There are five reduce calls in the test. The first four compute simple algebraic extensions of lists of polynomials. The last one multiplies a list of five univariate polynomials of simple algebraic extension.

The sequential version took 684s to complete. The last reduction alone accounts for 646s. Since the last reduction only multiplies five polynomials, we only use two threads in parallelization. The parallelized code takes 592s, which is improved by (only) 15% comparing to the sequential version. The parallelized long running reduction costs 527s which is 19% improvement over its sequential version. For the first four reductions, it is unnecessary to generate parallel codes which introduce overheads. To avoid this, we plan an adaptive framework in the future which uses more analysis and parallelize codes selectively.

6.2 Algebra library functions

We applied the framework to a set of 22 library functions. The semantics of the functions is described in Table 1. Functions `vdet`, `Chebyshev1` and `Chebyshev2` are implemented by us, and others are from the algebra library shipped with the AXIOM system family. All the functions are tested with randomly generated inputs. Each of these functions directly or indirectly uses at least one reduction in any of the three forms. Figure 3 and Figure 4 show the execution times of both the sequential and parallel versions of the library functions on the desktop PC and cluster, respectively. The computed speed-ups are in Table 2.

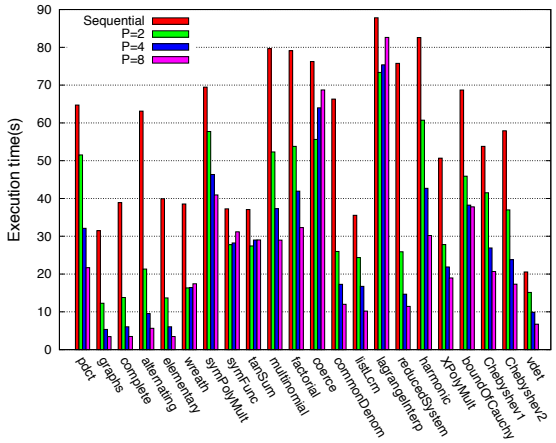

Figure 3: Execution times of library functions on the PC.

The performance of a parallel reduction may be dominated by the administrative cost of spawning threads amd collecting intermediate results. Such cases include functions `pdct`, `commonDenom`, `listLcm` on cluster, and `symFunc`, `tanSum` and `coerce` on both machines. *e.g.*, in the parallel reduction of `pdct`, the final combination performs multiplications of large positive integers. The reductions too respectively 15s (38% of total execution time), 23s (65%) and 27s (79%) with two, four and eight threads on the cluster. The speed-ups of `pdct` are better on the desktop PC. The reason is that the large number multiplication is faster on desktop PC, which makes the final combination step less dominant. For instance, Multiplying 4000-bit positive integers is 6 times faster on the desktop machine than on the cluster node.

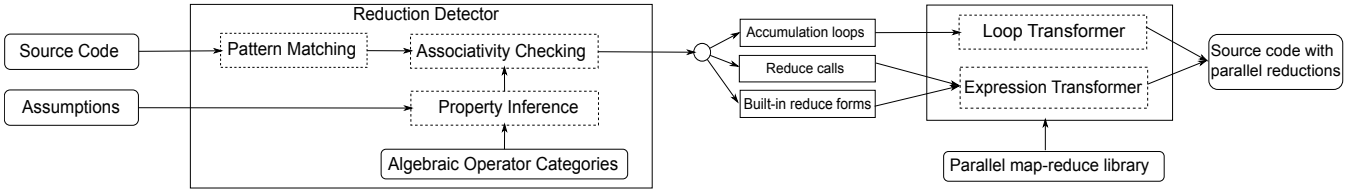

Figure 2: The workflow of the automatic parallelization framework.

Function name	Description
`pdct`	`pdct` takes in a partition of form $[a_1^{n_1}, ..., a_k^{n_k}]$, and computes $\prod_{i=1}^{k} n_i! a_i^{n_i}$.
`graphs`	`graphs(n)` is the cycle index of the group induced on the edges of a graph by applying the symmetric function to the `n` nodes.
`complete`	`complete(n)` computes the cycle index of the symmetric group of degree `n`.
`alternating`	`alternating(n)` computes the cycle index of the alternating group of degree `n`.
`elementary`	`elementary(n)` computes the `n`-th elementary symmetric function expressed in terms of power sums.
`wreath`	`wreath(s1, s2)` computes the cycle index of the wreath product of the two groups whose cycle indices are `s1` and `s2`.
`symPolyMult`	multiplication between two symmetric polynomials.
`symFunc`	`symFunc` takes in a list of elements $[r_1, ..., r_n]$ of a ring. It returns the vector of the elementary symmetric functions in the r_i's: $[r_1 + ... + r_n, r_1 r_2 + ... + r_{n-1} r_n, ..., r_1 r_2 ... r_n]$.
`tanSum`	computes expansions of tangets of sums.
`multinomial`	`multinomial(n,[m1,m2,...,mk])` computes the multinomial coefficient `n!/(m1!m2!...mk!)`
`factorial`	computes factorial `n`.
`coerce`	creates a permutation from a list of cycles.
`reducedSystem`	`reducedSystem(A)` returns a matrix `B` *s.t.* `Ax = 0` and `Bx = 0` have the same solutions in a ring.
`commonDenom`	computes a common denominator for a list of fraction integers.
`listLcm`	computes the least common multiply of a list of univariate polynomial integers.
`harmonic`	`harmonic(n)` compute the `n`-th harmonic number.
`lagrangeInterp`	computes the Largrange interpolation of a list of points.
`XPolyMult`	multiplies two generalized polynomials whose coefficients are not required to form a commutative ring.
`boundOfCauchy`	computes the Cauchy bound for the roots of the input polynomial.
`Chebyshev1`	evaluation of Chebyshev's first function $\theta(x) = \ln[\prod_{i=1}^{\pi(x)} p_i]$, where p_i is a prime, and π is the prime counting function.
`Chebyshev2`	evaluation of Chebyshev's second function $\psi(x) = \ln[\mathrm{lcm}(1, 2, 3, ..., \lfloor x \rfloor)]$.
`vdet`	`vdet(m)` computes the determinant for a Vandermonde matrix `m`.

Table 1: OpenAxiom library functions used in the experiments.

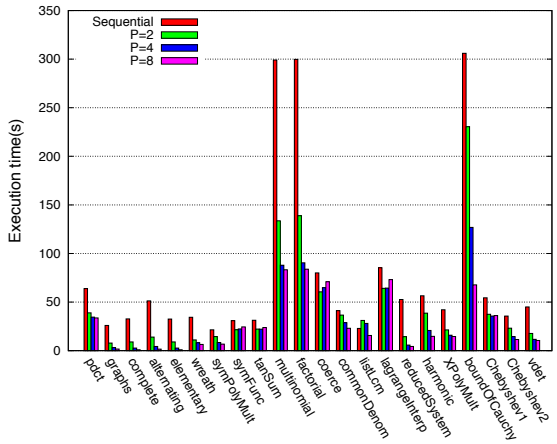

Figure 4: Execution times of library functions on the cluster.

Better speed-ups are obtained for a reduction if its sequential execution time increases rapidly with the size of the input list. Such as the functions `graphs`, `complete`, `alternating`, `elementary`, `reducedSystem` on both machines, and `commonDenom` on the desktop machine. Consider the super linear speed-ups of `graphs`. The reduction in this case uses addition over symmetric polynomials to sum elements in list of monomials. The addition operator first converts the two input polynomials to their internal representation. The internal representation is a list of monomials. The addition merges the two lists together in a sorted order. In the sequential code, merging two lists in sorted order iteratively has quadratic complexity. In the parallelized version, the monomial list obtained by each thread is much shorter. The local merge sort performed by each thread significantly saves the time as the execution time decreases quadratically when the input list becomes shorter. Moreover, the final combination adds up polynomials returned by the threads. Since each polynomial is a monomial list already in *sorted* order, this also significantly saves the computation time.

The execution of a function may be accelerated both by direct calls to parallel reductions functions, and by reductions present in

Function name	2_{PC}	4_{PC}	8_{PC}	2_{CL}	4_{CL}	8_{CL}
pdct	1.26	2.02	2.98	1.64	1.852	1.89
graphs	2.56	5.88	9.14	3.33	8.29	15.32
complete	2.82	6.43	11.77	3.63	12.28	34.92
alternating	2.96	6.62	11.16	3.66	11.86	32.59
elementary	2.92	6.61	11.49	3.61	12.13	34.19
wreath	2.36	2.34	2.21	3.10	4.23	5.34
symPolyMult	1.20	1.50	1.70	1.48	2.57	3.19
symFunc	1.34	1.32	1.20	1.43	1.38	1.26
tanSum	1.35	1.28	1.28	1.41	1.41	1.32
multinomial	1.52	2.13	2.75	2.24	3.40	3.59
factorial	1.47	1.89	2.45	2.16	3.32	3.57
coerce	1.37	1.19	1.11	1.32	1.23	1.13
reducedSystem	2.92	5.16	6.62	3.67	9.22	12.21
commonDenom	2.55	3.84	5.53	1.13	1.23	1.13
listLcm	1.46	2.12	3.48	0.74	0.82	1.46
harmonic	1.20	1.17	1.06	1.46	2.71	3.83
lagrangeInterp	1.36	1.93	2.73	1.33	1.33	1.17
XPolyMult	1.82	2.32	2.67	1.97	2.65	2.89
boundOfCauchy	1.50	1.80	1.82	1.33	2.41	4.52
Chebyshev1	1.30	2.00	2.6	1.45	1.53	1.51
Chebyshev2	1.57	2.43	3.34	1.54	2.46	3.11
vdet	1.56	2.45	3.73	2.54	3.88	4.29

Table 2: Speed-ups of the parallelized library functions on the PC and cluster with 2, 4 and 8 threads, respectively.

called functions. For instance, the function `wreath` contains a reduction using symmetric polynomial multiplication. The speed-ups shown in the results come from two sources: the parallel reduction itself, and the parallelized symmetric polynomial multiplication as the binary reduction operator. The latter contributes in a more significant way since multiplication is the most frequent computation in function `wreath`.

6.3 An application: concurrent homotopy continuation

We also developed a sequential polynomial homotopy library for solving polynomial systems. The parallelization framework is able to parallelize the path-following part of the source code. The homotopy library implements cheater's homotopy [11]. Linear system solving and evaluation in Newton's method use functions provided by OpenAxiom's algebra library.

This application is inspired by a recent work of Verschelde and Yoffe [18] on manually parallelizing various polynomial homotopy methods. Following a solution path is a task that can be done in parallel. We measured the performance and computed the speed-ups of the parallelized library on solving the 7-cyclic root problem. The system has 924 solution paths [6, 5]. Results on the desktop PC and the cluster node are shown in Table 3 and Table 4, respectively. On

Thread #	Execution Time(s)	Speedup(x)
Sequential	1370.47	1.00
2	727.82	1.88
4	790.19	1.73
8	778.32	1.76

Table 3: Performance of parallel homotopy continuations on the PC.

Thread #	Execution Time(s)	Speedup(x)
Sequential	1586.06	1.00
2	805.20	1.97
4	491.85	3.22
8	314.76	5.04

Table 4: Performance of parallel homotopy continuations on the cluster.

the desktop PC, we obtain good speed-up using two threads. We observed that the performance does not scale with more threads. This is due to contentions on limited computation resource. On the cluster node, we obtained good speed up using two and four threads. Speed up with eight threads is still far from ideal (8x). We believe this is because our preliminary parallel mapping and reduction library does not support advanced workload balancing strategies such as work-stealing, and thread pool. In the experiments where we use 8 threads, there are two threads which finish local computations much earlier than others do. The early finished threads return and exit instead of helping others to do more work.

7. RELATED WORK

The parallelization strategy presented in this paper is a map-reduce model [8]. In this model, programs are expressed in a functional style using a mapping followed by a reduction. Mapping and reduction are then automatically parallelized.

Most automatic parallelizations frameworks use program analysis, which are done at compile time, or at run time, or both. These analysis mainly focus on determining the parallelizability of loops. Static analysis such as GCD test [19] and Omega test [14] convert loops into linear systems. Dependence information is computed through linear system solving. Computation pattern matching is another way of parallelizing codes with specific formats. For instance, the range test [7] computes dependence information for loops whose iteration bounds are symbolic non-linear expressions. When the dependence information can not be determined at compile time, *e.g.*, the behavior of loops depends on the values computed at run time, dynamic analysis is used to make decision at run time. For instance, the work of Salz and Mirchandaney [17] records the memory reference right before the execution of a loop. The information is used for computing dynamic dependence information. More recently, the hybrid analysis [16] and the sensitivity analysis [15] combine the information both from compile and run time, which proves to be a more precise way for capturing dependence information. The program analysis present in this paper is based on semantics pattern matching at compile time. This is due to the simple loop pattern used by reductions.

It is feasible to adapt our automatic parallelization framework to work for algebraic systems using other programming languages such as C++, Java and Aldor. One way to achieve the adaptation is through a type system or an annotation system which is able to provide algebraic semantic information from users. For instance, in C++ properties can be defined using *concepts* [9]. Java programmers can use *annotation* to provide extra information before declarations and definitions. Annotation in Java, however, must be written before definitions and declarations. Therefore, the user has to modify the original source code. By contrast, in our work, user assumptions may be written separately from the source code. The Fortress programming language [4] provides parallel execution by default for implicit parallelizable structures. Furthermore, properties may be specified using *traits*. At some point, a draft [3] of

Fortress specification allowed user-defined axioms. That capability was removed from recent Fortress language specification [4], and it does not seem to have even been implemented in released versions. Therefore, we cannot offer tangible point of comparison with Fortress.

8. CONCLUSION AND FUTURE WORK

We presented a set of algorithms and implementation of an automatic parallelization framework. The framework parallelizes reductions using algebraic semantics information in form of user-provided axioms. Experimental results show that the framework is capable of speeding up algebraic library functions as well as user applications which contain parallel reductions.

In the future, we would like to implement workload balancing such as work-stealing to improve the performance of parallel libraries. We plan on testing the framework with more real-world applications. Being able to parallelize reductions is only a start. It is important for us to support more implicit parallel structures *e.g.* recursions and nested loops, as well as to integrate more advanced automatic parallelization algorithms.

9. ACKNOWLEDGEMENTS

The authors thank the Texas A&M University Brazos HPC cluster for provinding computing resources to support the research reported here. This work was partially supported by NSF grant CCF-1035058.

10. REFERENCES

[1] Brazos HPC cluster. `http://brazos.tamu.edu`, Texas A&M University, 2011.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[3] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, JanWillem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification version 1.0α. Technical report, Sun Labs, Oracle co., 2006.

[4] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, JanWillem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The fortress language specification version 1.0. Technical report, Sun Labs, Oracle co., 2008.

[5] Jörgen Backelin and Ralf Fröberg. How we proved that there are exactly 924 cyclic 7-roots. In *Proceedings of the 1991 international symposium on Symbolic and algebraic computation*, ISSAC '91, pages 103–111, New York, NY, USA, 1991. ACM.

[6] Göran Björck. Functions of modulus one on zp whose fourier transforms have constant modulus. In *Proceedings of Alfred Haar Memorial Conference*, 49, pages 193–197. Colloquia Mathematics Societatis János Bolyai, 1985.

[7] William Blume and Rudolf Eigenmann. The range test: a dependence test for symbolic, non-linear expressions. In *Proceedings of the 1994 conference on Supercomputing*, Supercomputing '94, pages 528–537, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[9] Gabriel Dos Reis and Bjarne Stroustrup. Specifying c++ concepts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 295–308, New York, NY, USA, 2006. ACM.

[10] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[11] T. Y. Li, T. Sauer, and J. A. Yorke. The cheater's homotopy: an efficient procedure for solving systems of polynomial equations. *SIAM J. Numer. Anal.*, 26:1241–1251, October 1989.

[12] Yue Li and Gabriel Dos Reis. A quantitative study of reductions in algebraic libraries. In *PASCO '10: Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, pages 98–104, New York, NY, USA, 2010. ACM.

[13] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[14] William Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[15] Silvius Rus, Maikel Pennings, and Lawrence Rauchwerger. Sensitivity analysis for automatic parallelization on multi-cores. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 263–273, New York, NY, USA, 2007. ACM.

[16] Silvius Rus, Lawrence Rauchwerger, and Jay Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *Int. J. Parallel Program.*, 31:251–283, August 2003.

[17] Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. Comput.*, 40:603–612, May 1991.

[18] Jan Verschelde and Genady Yoffe. Polynomial homotopies on multicore workstations. In *Proceedings of the 4th International Workshop on Parallel and Symbolic Computation*, PASCO '10, pages 131–140, New York, NY, USA, 2010. ACM.

[19] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.