Component-level Parallelization of Triangular Decompositions

Marc Moreno Maza
ORCCA, University of Western Ontario (UWO)
London, Ontario, Canada
moreno@orcca.on.ca

Yuzhen Xie
ORCCA, University of Western Ontario (UWO)
London, Ontario, Canada
yxie@orcca.on.ca

ABSTRACT

We discuss the parallelization of algorithms for solving polynomial systems symbolically by way of triangular decompositions. We introduce a component-level parallelism for which the number of processors in use depends on the geometry of the solution set of the input system. Our long term goal is to achieve an efficient multi-level parallelism: coarse grained (component) level for tasks computing geometric objects in the solution sets, and medium/fine grained level for polynomial arithmetic such as GCD/resultant computation within each task.

Component-level parallelization of triangular decompositions belongs to the class of dynamic irregular parallel applications, which leads us to address the following question: How to exploit geometrical information at an early stage of the solving process that would be favorable to parallelization? We report on the effectiveness of the approaches that we have applied, including "modular methods", "solving by decreasing order of dimension", "task pool with dimension and rank guided scheduling". We have extended the Aldor programming language to support multiprocessed parallelism on SMPs and realized a preliminary implementation. Our experimentation shows promising speedups for some well-known problems and proves that our component-level parallelization is practically efficient. We expect that this speedup would add a multiplicative factor to the speedup of medium/fine grained level parallelization as parallel GCD and resultant computations.

1. INTRODUCTION

Symbolic solvers of polynomial systems are powerful tools in scientific computing: they are well suited for problems where the desired output must be exact and they have been applied successfully in areas like digital signal processing, robotics, theoretical physics, cryptology with many important outcomes. See [20] for an overview of these applications. The implementation of symbolic methods is, however, a highly difficult task. Indeed, they are extremely time consuming when applied to large examples. Moreover, intermediate expressions can grow to enormous size and may halt the computations, even if the result is of moderate size.

The increasing availability of parallel computer architectures, from SMPs to multi-core laptops, has revitalized the need for developing mathematical algorithms and software capable of exploiting these new computing resources. This need is even more dramatic in the case of symbolic computations which offer exciting, but highly complex challenges to computer scientists. This paper aims at investigating new directions in the parallelization of symbolic solvers for polynomial systems.

Ideally, one would like that each component of the solution set of a polynomial system could be produced by an independent processor, or a set of processors. In practice, the input polynomial system, which is hiding those components, requires some transformations in order to split the computations into subsystems and, then, lead to the desired components. The efficiency of this approach depends on its ability to detect and exploit geometrical information during the solving process. Its implementation, which must naturally involve parallel symbolic computations, is yet another challenge.

Several symbolic algorithms provide a decomposition of the solution set of any system of algebraic equations into components (which may be irreducible or with weaker properties): primary decompositions [19, 41], comprehensive Gröbner bases [46] triangular decompositions [47, 25, 27, 34, 44] and others. These algorithms tend to split the input polynomial system into subsystems and, therefore, seem to be natural candidates for a *component-level* parallelization.

Unfortunately, such a parallelization is very likely to be unsuccessful, bringing no practical speedup w.r.t. comparable sequential implementations of the same algorithms. Indeed, even if computations split into sub-problems which can be processed concurrently, the computing cost of the corresponding tasks are extremely irregular. Even worse: for input polynomial systems with coefficients in the field $\mathbb{Q}$ of rational numbers, a single heavy task may dominate the whole solving process, leading essentially to no opportunities for component-level parallel execution. This phenomenon follows from the following observation. For most polynomial systems with coefficients in $\mathbb{Q}$ that arise in theory or in practice, see for instance `www.SymbolicData.org`, the solution set can be described by a single component! The theoretical justification is given by the celebrated *Shape Lemma* [5] for systems with finitely many solutions.

We show, in this paper, how to achieve a successful *component-level* parallelization for polynomial systems including for the case of rational number coefficients. Among the algorithms that decompose the solution set of a polynomial

ACM 0-89791-88-6/97/05.

system into components, we consider one computing triangular decompositions, called Triade [34]. The first reason for this choice is that, triangular decompositions of polynomial systems with coefficients in $\mathbb{Q}$ can be reduced to triangular decompositions of polynomial systems modulo a prime number [14], bringing rich opportunities for parallel execution. We discuss in Section 2 the main features of this algorithms that are relevant to parallelism. The second reason is that this algorithm has been implemented in the ALDOR language [2] and in the computer algebra systems AXIOM [24] and MAPLE [33] as the `RegularChains` library [30]. This provides us with useful tools for our experimentation work. The third and main reason is that the Triade algorithm can generate the (intermediate or output) components by decreasing order of dimension. As we show in Section 3, this allows us to exploit the opportunities for parallel execution created by modular techniques, leading to successfully component-level parallel execution.

Our objective is to develop a parallel solver for which the number of processors in use depends on the *intrinsic complexity* of the input system, that is on the geometry of its solution set. We do not aim at replacing the previous approaches for parallelizing algorithms. On the contrary, we aim at adding an extra level of parallelism.

The parallelization of two other algorithms for solving polynomial systems symbolically have already been actively studied. First, Buchberger's algorithm for computing Gröbner bases, see for instance [6, 9, 12, 3, 32]. Second, the Characteristic Set Method of Wu [47], see [1, 48, 49]. In all these works, the parallelized operation is polynomial reduction (or simplification). More precisely, given two polynomial sets A and B (with some conditions on A and B, depending on the algorithm) the reductions of the elements of A by those of B are executed in parallel.

The Triade algorithm also has a *polynomial simplification level* which relies on polynomial GCDs and resultants. The parallelization of such computations is reported in [37, 23]. The addition of this second level to the Triade algorithm is work in progress.

In Section 2, we present a *task model* associated with each triangular decomposition computed by the Triade algorithm. We review also how this algorithm makes use of geometrical information discovered during the computations. The techniques that are applied to create the component-level parallelism and to control the feature of tasks in favor of parallelization are introduced in Section 3. Our heuristically efficient *Task Pool with Dimension and Rank Guided scheduling* (TPDRG) is reported in Section 4. In the remaining sections, we report our preliminary implementation and experimentation. We have extended the ALDOR programing language for multi-processed parallel programing on SMPs and realized a preliminary implementation of this component-level parallel algorithm based on the `BasicMath` library [22]. We have conducted an intensive experimentation on some well-known problems. A comparison on the practical efficiency between our TPDRG scheduling and the generally good *Greedy* scheduling has also been performed. These help in evaluating the efficiency of our implementation and reveal its limitation as well. In the conclusion, we discuss the potential to extend this work to achieve efficient multi-level parallelization for triangular decompositions.

2. TASK MODEL

We discuss in this section the main features of the Triade algorithm that are relevant to parallelism. After some notations, we recall in Definition 1 the notion of a *regular chain*, which appears in most algorithm computing triangular decompositions. Then, we review the notions that are specific to the Triade algorithm such as that of a *Task*, Definition 2 and that of a *delayed split*, Definition 3. They are well-adapted to describe the relations between the intermediate computations during the solving of a polynomial system. Algorithm 1 is the top-level procedure of the Triade algorithm: it manages a *task pool*. The tasks are transformed by means of a sub-procedure (Algorithms 2 and 3) which is dedicated to "simple tasks". The execution of such a simple task can be highly irregular and dynamic. It can also generate other tasks. Therefore, Algorithms 1, 2 and 3 may not lead to successful parallel execution. In fact, we will adapt them in Section 3 for this purpose.

NOTATION 1. *Let $\mathbb{K}$ be a field and $X = x_1 < \cdots < x_n$ be n ordered variables. For a subset $F \subset \mathbb{K}[X]$, we denote by $V(F)$ the zero set of F in the affine space $\overline{\mathbb{K}}^n$ where $\overline{\mathbb{K}}$ is an algebraic closure of $\mathbb{K}$. The polynomial p is said* regular *modulo the ideal $\langle F\rangle$ if it is neither zero, nor a zero-divisor modulo $\langle F\rangle$. For a subset $W \subset \overline{\mathbb{K}}^n$, we denote by $\overline{W}$ the* Zariski closure *of W w.r.t. $\mathbb{K}$, that is, the intersection of the $V(G)$ containing $W(T)$ for all $G \subseteq \mathbb{K}[X]$.*

Let $p \in \mathbb{K}[X]$ be a non-constant polynomial. We denote by $mvar(p)$ the main variable *(or largest variable) of p, by $init(p)$ the* initial *(or leading coefficient w.r.t. $mvar(p)$) of p and by $rank(p)$ the* rank *of p, that is, v^d where $v = mvar(p)$ and d is the degree of p w.r.t. v. For two ranks $v_1^{d_1}$ and $v_2^{d_2}$ we write $v_1^{d_1} \prec v_2^{d_2}$ whenever $v_1 < v_2$ or, $v_1 = v_2$ and $d_1 < d_2$ hold.*

Let $T \subset \mathbb{K}[Y]$ be a triangular set, *that is, a set of non-constant polynomials with pairwise different main variables. Let h_T be the product of the initials of T. We denote by $W(T)$ the* quasi-component *of T, that is, $V(T) \setminus V(h_T)$, in other words, the set of the points in $V(T)$ which do not cancel any of the initials of T. For $F \subset \mathbb{K}[X]$, we denote by $Z(F,T)$ the intersection $V(F) \cap W(T)$. We denote by $mvar(T)$ (resp. $rank(T)$) the set of the $mvar(t)$ (resp. $rank(t)$) for all t in T. A variable from X is said* algebraic *w.r.t. T if it belongs to $mvar(T)$.*

Finally, we denote by $Sat(T)$ the saturated ideal of T, *which is defined as follows. If T is empty then $Sat(T)$ is defined as the trivial ideal $\langle 0\rangle$ otherwise it is the ideal of all $p \in \mathbb{K}[Y]$ such that there exists an integer e such that $h_T^e p$ belongs to $\langle T\rangle$. The ideal $Sat(T)$ has two important properties. First, its zero set satisfies the following: $V(Sat(T)) = \overline{W(T)}$. Secondly, if $Sat(T)$ is a proper ideal, then it is equidimensional and its dimension is equal to $n - |T|$ [25].*

DEFINITION 1. *The triangular set T is a* regular chain *if either T is empty or: T is not empty, $T \setminus \{T_{\max}\}$ is a regular chain, and the initial of $T_{\max}$ is regular w.r.t. $Sat(T \setminus \{T_{\max}\})$, where $T_{\max}$ is the polynomial in T with maximum rank. A finite family $\mathcal{X}$ of regular chains of $\mathbb{K}[X]$ is a* triangular decomposition *of $V(F)$ if we have*

$$V(F) = \cup_{T \in \mathcal{T}}\ W(T).$$

DEFINITION 2. *We call a* task *any couple $[F,T]$ where F is a finite subset of $\mathbb{K}[X]$ and $T \subset \mathbb{K}[X]$ is a regular chain. The task $[F,T]$ is* solved *if F is empty, otherwise it* *is* unsolved. *By* solving *a task, we mean computing regular chains $T_1, \ldots, T_e$ such that we have:*

$$V(F) \cap W(T) \ \subseteq\ \cup_{i=1}^{e} W(T_i) \ \subseteq\ V(F) \cap \overline{W(T)}. \quad (1)$$

Most algorithms computing triangular decompositions consist of procedures that take a task $[F_0, T_0]$ as input and returns zero, one or several tasks $[F_1, T_1], \ldots, [F_e, T_e]$. Then, solving an input polynomial system F_0 is achieved by calling one of these procedures with $[F_0, \emptyset]$ as input and obtaining "solved tasks" $[\emptyset, T_1], \ldots, [\emptyset, T_e]$ as output, such that $T_1, \ldots, T_e$ solves $[F_0, \emptyset]$ in the sense of Definition 2.

Therefore, given an algorithm A for computing triangular decompositions, it is natural to associate with each input polynomial system F_0 a *task tree* $G_A(F_0)$ whose vertices are tasks such that there is an arrow from any task $[F_i, T_i]$ to any task $[F_j, T_j]$ if task $[F_j, T_j]$ is among the output tasks of a procedure called on $[F_i, T_i]$; moreover, each internal node $[F_i, T_i]$ has a weight equal to the the (estimated) running time for computing the children of $[F_i, T_i]$. The longest path (summing the weights along the path) from the root to a leaf, called *critical path of* $G_A(F_0)$ and often denoted by T_∞, represents the minimum running time for a parallel execution of $A(F_0)$ on infinitely many processors. (Here we do not consider communication costs and scheduling overheads, for simplicity.) The sum of the all weights in $G_A(F_0)$, called the *work of* $G_A(F_0)$ and often denoted by T_1, represents the minimum running time for a sequential execution of $A(F_0)$.

It is well known that most of algorithms decomposing polynomial systems into components (irreducible, equidimensional, ...) have to face the problem of *redundant components*, which may occur in the output or at intermediate stages of the solving process. This is a central question when computing triangular decompositions, see [4] for a discussion of this topic. Removing redundant components is also an important issue in other symbolic decomposition algorithms such as the one of [28] and also for numerical ones [42]. Being able to remove redundant components at an early stage of the computations helps reducing the work of $G_A(F_0)$ and, possibly its critical path. One of the motivations in the design of the Triade algorithm [34] is to handle efficiently redundant components.

For any input task $[F, T]$ the main procedure of the Triade algorithm, called Triangularize(F, T), solves $[F, T]$ in the sense of Definition 2. This procedure reduces to the situation where F consists of a single polynomial p. One could expect that such an operation, say Decompose(p, T), should return regular chains $T_1, \ldots, T_\ell$ solving the task $[\{p\}, T]$. In fact, we shall explain now why this would not meet our requirement of handling redundant components efficiently.

Observe that $W(T_1) \subseteq W(T_2)$ implies $|T_2| \leq |T_1|$. It follows that, during the solving process, all the (final or intermediate) regular chains should be generated by increasing order of their cardinality, that is, by decreasing order of the dimension of their saturated ideals, in order to remove the redundant ones as soon as possible. Returning to the specifications of the operation Decompose(p, T), observe that $V(p) \cap W(T)$ could contain components of different dimension. (This will happen when $V(p)$ contains some of the irreducible components of $\overline{W(T)}$, but not all of them.) Therefore, it is not desirable for the operation Decompose(p, T) to solve the task $[\{p\}, T]$ in one step. Instead, Decompose(p, T) should compute the quasi-components of $V(p) \cap W(T)$ of maximum dimension and postpone the computation of the other quasi-components. This is made possible by a form of lazy evaluation, formalized by Definition 3, after Notation 2.

NOTATION 2. *Let T_1, T_2 be two regular chains. We write $rank(T_1) \prec rank(T_2)$ whenever $rank(T_2)$ is a proper subset of $rank(T_1)$, or when $v_1^{d_1} \prec v_2^{d_2}$, where $v_1^{d_1}$ (resp. $v_2^{d_2}$) is the smallest element of $rank(T_1) \setminus rank(T_2)$ (resp. $rank(T_2) \setminus rank(T_2)$). When neither $rank(T_1) \prec rank(T_2)$ nor $rank(T_2) \prec rank(T_1)$ hold we write $rank(T_1) \simeq rank(T_2)$. Let F_1, F_2 be finite subsets of $\mathbb{K}[X]$. We write $[F_1, T_1] \prec [F_2, T_2]$ either if $rank(T_1) \prec rank(T_2)$ holds, or if $rank(T_1) \simeq rank(T_2)$ holds and there exists $f_1 \in F_1$ such that $rank(f_1) \prec rank(f_2)$ for all $f_2 \in F_2$. Clearly, any sequence of tasks $[F_0, T_0], \ldots,$ such that $[F_i, T_i] \prec [F_{i+1}, T_{i+1}]$ holds for all i, is finite.*

DEFINITION 3. *The tasks $[F_1, T_1], \ldots, [F_e, T_e]$ form a* delayed split *of the task $[F, T]$ and we write*

$$[F, T] \longmapsto_D [F_1, T_1], \ldots, [F_e, T_e]$$

if for all $1 \leq i \leq e$ we have $[F_i, T_i] \prec [F, T]$ and the following holds

$$V(F) \cap W(T) \ \subseteq\ \cup_{i=1}^{e} Z(F_i, T_i) \ \subseteq\ V(F) \cap \overline{W(T)}.$$

Below, we highlight the main features of the procedure Decompose(p, T) that are relevant to the rest of the paper. First, for a polynomial p and a regular chain T, such that p is not zero modulo $\mathrm{Sat}(T)$, the procedure Decompose(p, T) returns a delayed split of the task $[\{p\}, T]$. Algorithm 1 implements the procedure Triangularize by means of the procedure Decompose. Based on the specifications of Decompose, the validity of this algorithm is easy to check and is established in [34]. Note that our pseudo-code uses the syntax of the Computer Algebra System AXIOM [24]. In particular, we use indentation to denote blocks. Moreover, each of our algorithms generates a sequence of items which are returned one by one in the flow of control by means of **yield** statement.

ALGORITHM 1.

Input: *a task $[F, T]$*

Output: *regular chains $T_1, \ldots, T_e$ solving $[F, T]$ in the sense of Definition 2*

Triangularize(F, T) == **generate**
$R := [[F, T]]$
 # R is a list of tasks
while $R \neq []$ **repeat**
  choose and remove a task $[F_1, U_1]$ from R
  $F_1 = \emptyset \Longrightarrow$ **yield** U_1
  choose a polynomial $p \in F_1$
  $G_1 := F_1 \setminus \{p\}$
  $p \equiv 0 \mod Sat(U_1) \Longrightarrow R :=$ cons $([G_1, U_1], R)$
  for $[H, T] \in$ Decompose(p, U_1) **repeat**
    $R :=$ cons $([G_1 \cup H, T], R)$

The key notion used by the procedure Decompose is that of a polynomial GCD modulo a regular chain, see Definition 4. This notion strengthens that introduced by Kalkbrener [25] and extends that of a polynomial GCD modulo a triangular set, introduced in [35].

DEFINITION 4. *Let p, t, g be non-zero polynomials and T be a regular chain. Assume that p and t are non-constant* *and have the same main variable v. Assume that $v \notin mvar(T)$, that $init(p)$ is regular w.r.t. $Sat(T)$ and that $T \cup \{t\}$ is a regular chain. Then, the polynomial g is a* GCD *of* p and t w.r.t. T *if the following properties hold:*

(G_1) *g belongs to the ideal generated by p, t and $Sat(T)$,*

(G_2) *the leading coefficient h_g of g w.r.t. v is regular w.r.t. $Sat(T)$,*

(G_3) *if $mvar(g) = v$ then p and t belong to $Sat(T \cup \{g\})$.*

More generally, a sequence of pairs $G = (g_1, T_1), \ldots, (g_e, T_e)$, where $g_1, \ldots, g_e$ are polynomials and $T_1, \ldots, T_e$ are regular chains, is a GCD sequence *of* p and t w.r.t. T *if the following properties hold:*

(G_4) *for all $1 \le i \le e$, if $|T_i| = |T|$ then g_i is a GCD of p and t modulo T_i,*

(G_5) *we have $W(T) \subseteq \cup_{i=1}^{e} W(T_i) \subseteq \overline{W(T)}$.*

Four procedures of Triade are essential to the rest of the paper. Their specifications are reviewed in Notation 3.

Notation 3. *Let p, t, T be as in Definition 4. The procedure* $\mathsf{GCD}(p, t, T)$ *computing a GCD sequence of* p and t w.r.t. T. *These GCD computations allow testing whether a polynomial f is regular modular $Sat(T)$. Given a polynomial f, the procedure $RegularizeInitial(f, T)$ returns regular chains $T_1, \ldots, T_e$ such that for all $1 \le i \le e$, either the polynomial f is constant modulo $Sat(T_i)$, or its initial is regular modulo $Sat(T_i)$, and we have $W(T) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq \overline{W(T)}$. Given a triangular set $S \subset \mathbb{K}[X]$ the procedure* $\mathsf{Extend}(S)$ *returns regular chains $T_1, \ldots, T_e$ satisfying $W(S) \subseteq W(T_1) \cup \cdots \cup W(T_e) \subseteq \overline{W(S)}$. Finally, we denote by* $\mathsf{Reduce}(f, T)$ *the polynomial defined as follows: if $f \in \mathbb{K}$ or $f \in Sat(T)$ then* $\mathsf{Reduce}(f, T)$ *is f otherwise it is* $\mathsf{Reduce}(init(f), T) mvar(f) + \mathsf{Reduce}(g)$. *where $g = f - init(f) rank(f)$.*

Algorithm 3 states the algorithm for $\mathsf{Decompose}(p, T)$. It relies on a sub-procedure given by Algorithm 2

Algorithm 2.

Input: *p, T, t as in Definition 4.*

Output: *a delayed split of $[\{p\}, T \cup \{t\}]$.*

$\mathsf{AlgebraicDecompose}(p, T, t)$ == **generate**
  Let h_T be the product of the initials in T
  $f := t\, h_T$
  for $[g_i, T_i] \in \mathsf{GCD}(t, p, T)$ **repeat**
    $|T_i| > |T| \Longrightarrow$
      for $T_{i,j} \in \mathsf{Extend}(T_i \cup \{f\})$ **repeat**
        yield $[p, T_{i,j}]$
    $g_i \in \mathbb{K} \Longrightarrow$ **iterate**
    $mvar(g_i) < v \Longrightarrow$ **yield** $[\{g_i, p\}, T_i \cup \{t\}]$
    $deg(g_i, v) = deg(t, v) \Longrightarrow$ **yield** $[\emptyset, T_i \cup \{t\}]$
    yield $[\emptyset, T_i \cup \{g_i\}]$
    yield $[\{init(g_i), p\}, T_i \cup \{t\}]$

Algorithm 3.

Input: *a polynomial p and a regular chain T such that $p \notin Sat(T)$.*

Output: *a delayed split of $[\{p\}, T]$.*

$\mathsf{Decompose}(p, T)$ == **generate**
  for $C \in RegularizeInitial(p, T)$ **repeat**
    $f := \mathsf{Reduce}(p)$
    $f = 0 \Longrightarrow$ **yield** $[\emptyset, C]$
    $f \in \mathbb{K} \Longrightarrow$ **iterate**
    $v := mvar(f)$
    $v \notin mvar(C) \Longrightarrow$
      yield $[\{init(f), p\}, C]$
      for $D \in \mathsf{Extend}(C \cup \{f\})$
        repeat yield $[\emptyset, D]$
    for $[F, E] \in \mathsf{AlgebraicDecompose}(f, C_{<v} \cup C_{>v}, C_v)$
      repeat yield $[F, E]$

The following fundamental proposition is easily checked from the pseudo-code of Algorithm 2 and Algorithm 3.

Proposition 1. *Let $[F, E]$ be any task returned by Algorithm 3. Then we have:*

(H_1) *either $|E| = |T|$ and $F = \emptyset$,*

(H_2) *or $|E| = |T|$ and F contains a polynomial which is regular w.r.t. $Sat(T)$,*

(H_3) *or $|E| > |T|$.*

Corollary 1. *A task $[F, E]$ returned by Algorithm 3 is solved if and only $\overline{W(E)}$ is a component of $V(p) \cap \overline{W(T)}$ with maximum dimension.*

Solving by decreasing order of dimension. It follows from Corollary 1 that the tasks in Algorithm 1 can be chosen such that the regular chains output by this algorithm are generated by increasing size. To do so, we assign to each task $[F, T] \in R$ in Algorithm 1 an upper bound $m([F, T])$ for the height of the regular chains solving $[F, T]$ in the sense of Definition 2. This upper bound is simply computed as follows. If a polynomial $f \in F$ has been shown to be regular w.r.t. T (See Proposition 1) then $m([F, T]) := |T| + 1$ otherwise $m([F, T]) := |T|$. Then, we say that a task $[F_1, T_1]$ has a *higher priority* than a task $[F_2, T_2]$ if either $m([F_1, T_1]) \le m([F_2, T_2])$ holds, or $m([F_1, T_1]) = m([F_2, T_2])$ and $[F_1, T_1] \prec [F_2, T_2]$ hold. Sorting the tasks in the list R w.r.t. this ordering allows us to solve by decreasing order of dimension and therefore to handle redundant components efficiently. The performances of our inclusion test are reported in [13].

3. PARALLELIZATION

Triangular decomposition algorithms tend to split the input polynomial system into subsystems and, therefore, seem to be natural candidates for parallelization. One could think of deriving a parallel scheme from Algorithm 1 by running the procedure $\mathsf{Triangularize}(F, T)$ on one processor and running each call to Decompose on any other available processor, following a greedy scheduling.

Such a parallelization is very likely to be unsuccessful, bringing no practical speed-up w.r.t. comparable sequential implementations of the same algorithms. Indeed, even if computations split into sub-problems which can be processed concurrently, the computing resources consumption of the corresponding tasks are extremely irregular. Even worse: for input polynomial systems with coefficients in the field $\mathbb{Q}$ of rational numbers, a single heavy task may dominate the whole solving process, leading essentially to no opportunities for component-level parallel execution. This phenomenon follows from the following observation. For most polynomial systems with coefficients in $\mathbb{Q}$ that arise in theory or in practice, see for instance `www.SymbolicData.org`, the solution set can be described by a single component! The theoretical justification is given by the celebrated *Shape Lemma* [5] for systems with finitely many solutions, and, for instance, by the results of [10] for systems with infinitely many solutions.

The situation is, however, quite different for polynomial systems with coefficients modulo a prime number. Indeed, the probability q_n for a monic uniformly random univariate polynomial of degree n to be irreducible modulo a prime number p with $p^n \geq 16$ satisfies $1/2n \leq q_n \leq 1/n$ [18]; hence a polynomial of large degree irreducible over $\mathbb{Q}$ is likely to factor modulo p.

Using modular techniques. The previous remark suggests the use of modular techniques for computing triangular decompositions. We have chosen to rely on the algorithm proposed in [14]. For a given input square system $F \subset \mathbb{Q}[x_1, \ldots, x_n]$ this algorithm computes the simple points of $V(F)$ in four steps:

(S_1) computes a prime number p such that zero-set $V(F)$ can be reconstructed from $V(F \mod p)$ (that is the zero-set of F regarded as subset of $Z/pZ[x_1, \ldots, x_n]$) with a high probability,

(S_2) computes a triangular decomposition of $V(F \mod p)$,

(S_3) computes the equiprojectable decomposition of $V(F \mod p)$,

(S_4) reconstructs by Hensel lifting the equiprojectable decomposition of $V(F)$.

As reported in [14], the second step has the dominant cost. Therefore, we choose to focus on computing triangular decompositions of polynomial systems with coefficients in a finite field.

Our test suite. Table 1 contains data about 7 well-known test systems that we use through the experiments reported in this article. All of them are polynomial systems over $\mathbb{Q}$: for each we give its number of n of equations, its total degree d. the prime number p provided by the above Step (S_1) and the list of the degrees of the triangular decomposition computed at Step (S_2) by the Triade algorithm. We stress the fact that each of these systems, except Cohn2, is equiprojectable, that is, its equiprojectable decomposition consists of a single component. Hence, for a direct computation $\mathbb{Q}$, the computations may not split. Therefore, our modular approach has created opportunities for parallel execution.

Sys	Name	n	d	p	*Degrees*
1	eco6	6	3	105761	[1,1,2,4,4,4]
2	eco7	7	3	387799	[1,1,1,1,4,2, 4,4,4,4,4,2]
3	CassouNogues2	4	6	155317	[8]
4	CassouNogues	4	8	513899	[8,8]
5	Nooburg4	4	3	7703	[18,6,6,3,3,4, 4,4,4,2,2,2, 2,2,2,2,2,1, 1,1,1,1]
6	UteshevBikker	4	3	7841	[1,1,1,1,2,30]
7	Cohn2	4	6	188261	[3,5,2,1,2,1,1, 16,12,10,8,8, 4,6,4,4,4,4,2, 1,1,1,1,1,1,1, 1,1,1,1,1,1,1]

Table 1: Features of the polynomial systems

Regularizing initials for controlling task irregularity. The procedure Decompose as given by Algorithm 3 may result in unpredictable amount of work. Indeed, since the initial of p may not be regular w.r.t. $\text{Sat}(T)$, the polynomial f computed at Line 2 may have a different main variable than p. Hence we cannot predict the main variables and degrees of the input polynomials in the calls to AlgebraicDecompose and Extend. It could be the case these calls lead to inexpensive operations, say polynomial GCDs of univariate polynomials of low degrees, whereas the regular chain contains very large polynomials in many variables. Therefore, Decompose(p, T) may lead to inexpensive computations but expensive data communication. In order to control this phenomenon, we strengthen the notion of a task in Definition 5: the **initial** of every polynomial $f \in F$ in a task $[F, T]$ must be **regular** w.r.t. $\text{Sat}(T)$. The motivation of this Definition is twofold. First, we want to anticipate which operations will be performed by Algorithm 3. Second, we want to force light-load calls to Decompose(p, T) to be performed inside heavy-load calls. Reaching the former goal is discussed after Definition 5 while the latter one is achieved by the *Split-by-height* strategy presented at the end of this section.

DEFINITION 5. *The task* $[F, T]$ *is* standard *if it satisfies the following conditions*

1. *Every polynomial* $f \in F$ *is not constant modulo* $Sat(T)$
2. *For all* $f \in F$ *the initial of* f *is regular w.r.t.* $Sat(T)$.

Estimating the cost of tasks. Assume from now on that every task in Algorithm 1 is standard. When the polynomial p is chosen at Line 6, we know which operations will be performed by the call Decompose(p, U_1). Indeed, when the initial of p being regular w.r.t. $T := U_1$, Line 1 in Algorithm 3 can be removed and we know that $f = p$ holds. Let v be $\text{mvar}(p)$. Therefore, two cases arise:

- either v is algebraic w.r.t. T and $\mathsf{GCD}(C_v, p, C_{<v})$ is called and its costs can be estimated (see for instance [15] for complexity estimates);
- v is not algebraic w.r.t. T and $\mathsf{Extend}(T \cup \{p\})$ is called, leading again to GCD computations with predictable costs.

The *Split-by-height* strategy. Let $[F, E]$ be a task. We introduce a new procedure, called SplitByHeight(F, E), returning a delayed split of $[F, E]$ with the following requirement: If $[G, U]$ is a task returned by SplitByHeight(F, E) and $|U| = |E|$ holds then $G = \emptyset$ holds. An algorithm for

$\mathsf{SplitByHeight}(F, E)$ is easily derived from Algorithm 1, leading to Algorithm 4 below.

Algorithm 4.

Input: *a task* $[F, T]$

Output: *a delayed split of* $[F, T]$ *such that for all output task* $[G, U]$ *we have:* $|U| = |T| \implies G = \emptyset$.

$\mathsf{SplitByHeight}(F, T)$ == **generate**
$R := [[F, T]]$
 # R is a list of tasks
while $R \neq [\,]$ **repeat**
    choose and remove a task $[F_1, U_1]$ *from* R
    $|U_1| > |T| \Longrightarrow$ **yield** $[F_1, U_1]$
    $F_1 = \emptyset \Longrightarrow$ **yield** $[F_1, U_1]$
    choose a polynomial $p \in F_1$
    $G_1 := F_1 \setminus \{p\}$
    $p \equiv 0 \mod \mathit{Sat}(U_1) \Longrightarrow R := \mathsf{cons}\ ([G_1, U_1], R)$
    for $[H, T] \in \mathsf{Decompose}(p, U_1)$ **repeat**
        $R := \mathsf{cons}\ ([G_1 \cup H, T], R)$

Then we derive a new implementation of $\mathsf{Triangularize}(F, T)$ based on $\mathsf{SplitByHeight}$ and given as Algorithm 5.

Algorithm 5.

Input: *a task* $[F, T]$

Output: *regular chains* $T_1, \ldots, T_e$ *solving* $[F, T]$ *in the sense of Definition 2*

$\mathsf{Triangularize}(F, T)$ == **generate**
$R := [[F, T]]$
 # R is a list of tasks
while $R \neq [\,]$ **repeat**
    choose and remove $[F_1, U_1] \in R$ *with max priority*
    $F_1 = \emptyset \Longrightarrow$ **yield** U_1
    for $[H, T] \in \mathsf{SplitByHeight}(F_1, U_1)$ **repeat**
        $R := \mathsf{cons}\ ([H, T], R)$
    sort R by decreasing priority

Two benefits are obtained from Algorithm 5 in view of parallelization. Assume that at each iteration of the **while** loop all tasks with maximum priority are executed concurrently. Then, n (the number of variables) iterations at most will be needed. Secondly, each processor running a call to SplitByHeight will contribute a substantial work to the whole solving process. Indeed, after each call to SplitByHeight, and thus after each parallel step, the minimum height of a regular chain in any tasks of R has increased at least by one.

4. HEURISTICALLY GOOD SCHEDULING

In this section we introduce our heuristically efficient scheduling scheme: *Task Pool with Dimension and Rank Guided dynamic scheduling* (TPDRG), for achieving both dynamic load balancing and removing redundant components. To our application's interest, load balancing means to use efficiently a given number of processors to obtain minimum execution time.

As described in Section 2, computing the triangular decompositions based on the Triade algorithm can be modeled as a task tree. The input problem forms the root of the tree. Processing the root task generates child tasks. The child tasks can further produce more child tasks, leading to the many solution sets of the input system. Among the child tasks, there are no data dependency. This task tree is dynamic and irregular.

This dynamic irregular task tree is a special form of the computation acyclic directed graph (DAG) [8]. Parallel scheduling is to map a computation graph to multiple processors dynamically in order to minimize running time. Unfortunately this problem is *NP*-hard [8]. Scheduling irregular applications is even more difficult. Studies by Yelic [11, 38] show that the difficulty of parallelizing a given program is strongly correlated to the rank of irregularity in the program's structures. With the increasing significance of parallel programming, many researchers have been seeking heuristically good methods on the application/compiler level for different types of applications on many important parallel architectures [39, 38, 12, 36, 43, 26]. In the successful applications, they combine the heuristically efficient methods into their algorithms to gain better results, for example, the task scheduling using a block dependency DAG for block-oriented sparse Cholesky factorization [29], and the pyramid DAG scheduling and phase-parallel scheduling for real root isolation in [16].

In Section 3, we have exploited the geometrical information of the tasks , that is the *regularity*, the *dimension* and the *rank*, to characterize the tasks. Based on this information, we adapt the *task pool* [26] method and the *trapezoid self-scheduling* method [43] into a *task pool with dimension and rank guided dynamic scheduling*.

Task pools are data structures which are used to distribute data dynamically to different processors. The work in [29] has shown that they are effective ways for data distribution in parallel irregular algorithms. The *trapezoid self-scheduling* approach [43] is an improved *guided self-scheduling*, where the best tradeoff between the scheduling overhead and balanced workload can be obtained by linearly decreasing the task size at run time. We use the *dimension* and the *rank* information to estimate the cost of the tasks at run time, and apply these two scheduling techniques to our parallel computation.

Our *task pool with dimension and rank guided dynamic scheduling* is depicted in Figure 1. The *task pool* (which can be seen as an implementation of the list R in Algorithms 5) is managed by a *Manager* processor. The Manager first preprocesses the input task, which will generate child tasks. The Manager then selects unsolved tasks with maximum priority (See the last paragraph in Section 2.) determined by the dimension information of the tasks, say $Task1.1$, $Task1.2$ and $Task1.3$, and estimates the cost of each of the selected tasks (See Section 3), denoted by $Cost1.1$, $Cost1.2$ and $Cost1.3$, and sorts them decreasingly, say $Cost1.1 \geq Cost1.2 \geq Cost1.3$. Now the manager will schedule first $Task1.1$ to an available processor, we call it a *worker*, next $Task1.2$, and $Task1.3$ at the last. When there is only one task in the selection, the Manager will process it by itself. For those tasks with very low estimated cost, the Manager also process them by itself.

The Manager then proceeds to step 2 to receive the results from the workers for removing redundant components by inclusion test, and start another schedule again, say the $Task2$s shown in Figure 1. The overall solving process follows the decreasing order of dimension, which indicates that

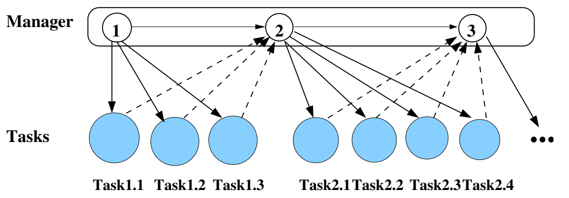

Figure 1: Task pool with dimension and rank guided dynamic scheduling

the dimension of $Task2$s is lower than the dimension of $Task1$s. The Manager repeats this scheduling rule until all the tasks are solved. Other than scheduling, the work load of the Manager is very light and it can not be a bottleneck. The benefit of solving by decreasing order of dimension and how to split a task to maintain properties for cost estimation has been discussed near the end of Section 3.

To evaluate the effectiveness of our *task pool with dimension and rank guided dynamic scheduling*, we compare its practical efficiency in our implementation with the *Greedy scheduling* method [21]. Therefore, we briefly introduce it here. A greedy scheduler attempts to do as much work as possible at every step for a given P number of processors. If there are at least P tasks ready to run, it selects any P of them and runs them. When there are strictly less than P tasks that are ready to run, the greedy scheduler runs them all. Given P processors, a greedy scheduler executes any computation DAG in time: $T_p \leq T_1/P + T_\infty$, where T_p is the minimum running time on P processors, T_1 is the minimum running time on 1 processor, and T_∞ is the critical path length of a DAG. A greedy scheduler is always within a factor of 2 of optimal. It is generally a good scheduler.

5. PRELIMINARY IMPLEMENTATION

In the previous sections, we showed how to create rich opportunities for a component-level parallel execution, and to schedule for load balancing. However, another big challenge remains: the parallel implementation. First of all, solving non-linear polynomial systems symbolically by way of triangular decompositions involves sophisticated algorithms and complex mathematical structures. This requires a very high-level programming language. In addition, this language should be suitable for high performance computing, and permits efficient implementation of fast polynomial arithmetic, as discussed in [17]. Secondly, our parallel scheme includes dynamic task management and heavy data communication along with intensive computations. Therefore, the parallel architecture that we run on will significantly influence the implementation scheme and its overall performance.

5.1 A framework for parallel symbolic computations

We choose the ALDOR programming language and symmetric multiprocessor machines (SMP and multi-core) to build our parallel programming model. In our first attempt a multi-processed parallel framework using ALDOR targeting SMPs and multi-cores has been established.

ALDOR has been designed to express the extremely rich and complex variety of structures and algorithms in computer algebra with focuses on interoperability with other languages and high-performance computing. This language has a two-level object model of *categories* and *domains*, which is similar to *interfaces* and *classes* in Java. They provide a type system that allows the programmer the flexibility to extend or build on existing types, or create new categories and domains, as usually required in algebra.

In addition, an ALDOR program can be compiled into: stand-alone executable programs; object libraries in native operating system formats (which can be linked with one another, or with C or Fortran code to form application programs); portable byte code libraries; and C or Lisp source [7]. Aggressive code optimizations by techniques such as program specialization, cross-file procedural integration and data structure elimination, are performed at intermediate stages of compilation [45]. This produces code that is comparable to hand-optimized C.

For these reasons a sequential implementation [31] of the `Triade` algorithm in ALDOR has been developed together with the BasicMath library for high performance computing. Many of the categories, domains and packages of this sequential implementation (such as polynomial arithmetic, polynomial GCD and resultant, inclusion test, single integer prime field) can be reused or extended for our purpose. These provide us qualified support for realizing a preliminary implementation of the parallel algorithm in a reasonable period of time.

Before our work, however, like many other computer algebra systems, ALDOR programming language did not have any support for parallel programming, not to mention MPI binding, or OpenMP binding. Fortunately, an ALDOR program can be compiled into stand-alone executable programs. This allows us to build separated executable modules to run as independent parallel processes. ALDOR also has a primer `exec()`, for initiating a program `P` within a program `Q`, a wrapper of the *system()* function in C for launching independent processes. These give us the basic functionalities for dynamic task management.

Now comes the challenge of establishing interprocess communication (IPC) in ALDOR, which is critical for all parallel programming environment. We rely on *shared memory segments* for System V IPC and develop a `SharedMemorySegment` domain in ALDOR, based on the interoperability of ALDOR with the C programing language. The `SharedMemorySegment` domain has methods for creating a segment and connecting to it, attaching i.e. getting a pointer to the segment, reading and writing, and detaching from and deleting the segment. An element of the domain `SharedMemorySegment` can be either a string or a primitive array of machine integers.

Another difficult issue for data communication is the many complex data types in our program. For example, sparse multivariate polynomial and dense multivariate polynomial are all valid types in our program in ALDOR. In our package, a `Triade` task (see Definition 2) is a list of lists of polynomials with special properties (including the list of variables and the ring characteristic). It is given by $[F, T]$, where, F is a list of unsolved polynomials; and, T is a list of polynomials forming a regular chain.

To achieve effective data communication, we convert any multivariate polynomial to a primitive array of machine integers via Kronecker substitution [18].

5.2 Synchronization

Another main concern is the concurrency (synchronization) control for multiprocessing in ALDOR, since there was no such mechanisms before. For our component-level parallel solving reported in Sections 3 and 4, we apply a "process farm" parallel scheme consisting of a *Manager* process and *worker* processes initiated by the Manager when needed.

Between the Manager and a worker process, data communication is synchronized by three shared memory segments defined by a protocol related to the task ID. Let the task ID be i. The three segments are named as $task\_i$, $result\_tag\_i$, and $result\_i$ respectively. The main strategy is described in Figure 2. At a time the Manager selects tasks with highest

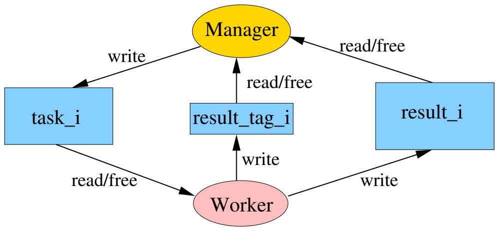

Figure 2: Synchronizing Manager with a worker

priority for processing. Let tasks with ID of i, j, k be chosen and they are sorted by decreasing order with respect to their costs. The manager first writes the content of the task with $ID = i$ into a shared memory segment named $task\_i$. Then it launches a worker and passes the value i (the task ID) and the size of the task as command line arguments to the worker process. Continually, the Manager process will do the same to task j and task k. Then for tasks i, j, k, the Manager will check in turn their result tag segments named $result\_tag\_i$, $result\_tag\_j$ and $result\_tag\_k$. If there is a result in any one of the result tag segments, for instance j, it will read the size of the result for task j from this segment named $result\_tag\_j$, and free it. Now the Manager knows the result of task j is returned and its size, then it will read the result for task j from the segment named $result\_j$, and free this shared memory segment at the end. The same operation will be done for both task i and task k.

When a worker is launched, it will get the task ID and task size from the command line argument. Let this ID be m. By the protocol, the worker will know the names of the three shared memory segments to work with, and the order and permission to access them. It will first read the task from the segment named $task\_m$. Then it can free the above two segments. After finishing its job, the worker will write first the result into the segment named $result\_m$, then the size of the result into the segment named $result\_tag\_m$. At last, it terminates by itself.

6. EXPERIMENTATION

Our experimentation was conducted on Silky in Canada's Shared Hierarchical Academic Research Computing Network (SHARCNET). Silky is the SHARCNET Mid-Range Symmetric Multiprocessing (SMP) cluster. Its model is SGI Altix 3700 Bx2, equipped with 128 Itanium2 Processors (1.6GHz). It is a 64-bit architecture with 256 GB memory and 6 MB cache. The operating system running there is SUSE Linux Enterprise Server 10 (ia64). It is a highly shared supercomputing facility, widely used in Canada. All the jobs are submitted to the queues and the system schedule them. There are four job queues: mpi queue, threaded queue, serial queue and test queue. Our parallel benchmark runs were submitted to the threaded queue, and the sequential runs were submitted to the serial queue. The threaded queue has a limit of 96 max threaded cpus per user and maximum single job which can be submitted is 64, whereas the serial queue permits a single user to run 4 serial jobs. Jobs running in the serial queue are no longer pre-emtable by all other queues, except for the test queue. While we were running our programs, we noticed that there were 84 processors (out of 128) which were exclusively running by other five jobs.

The problems tested in this experimentation are listed in Table 1. Their detail features have been described in Section 3. These problems are either small or medium-sized.

For each problem, its sequential running time with and without the *regularized initial* condition, imposed by the use of standard tasks (See Definition 5), are listed in column `noregSeq` and colmn `regSeq` respectively in Table 2. The sequential runs are given by the `Triade` solver [31]. Column 4 `slowBy` is the ratio between these two timings. This result shows that the cost for maintaining the property of standard tasks is negligible (0.01%). Column #`CPUs` records the number of processors which can give significant speedup to the example's run, that is, beyond it, the increase in the number of processors can hardly influence its execution time any more. The parallel execution time using this number of processors is recorded in column `SigPar` The speedup ratio (`SPD`) is calculated by comparing the parallel execution time with respect to the comparable sequential running time `noregSeq`.

Table 3 reports on the parallel execution time (wall time) of each problem on a varied number of processors, from 3 to 21. For each run, one processor is always used by the Manager. Thus, given P processors, there are actually $P-1$ which can be scheduled for workers. Their speedups (on the vertical axis) with respect to the number of processors are illustrated in Figure 3. Bottom up, the lines correspond to the problems $1, 6, 3, 4, 5, 7, 2$, respectively.

These results demonstrate that, for these small or medium-sized problems, our component-level (coarse grained) parallel triangular decompositions implemented in a high-level computer algebra system can gain a speedup from 2 to 6, using a considerably small number of processors (from 5 to 13). This is very promising comparing with the parallel implementation (coarse grained) of the cylindrical algebra decomposition algorithm in Fortran by Saunders et al [40], where they worked only on small problems, and their best speed up is 3 and the number of processors for obtaining it is 7. The fine grained parallel Gröbner basis implemented in C in [11] reported a speed up from 2 to 15 for small examples.

However, for all these small examples, beyond some limit, the speedup cannot increase when adding more number of processors. This result agrees with the observations in [40] and [11] for small problems. This is due to the fact that, given a computation graph, the minimum execution time is at least $max(T_1/P, T_\infty)$. Large examples are expected to gain better scalability.

In Table 4, for each of the problems, we show the minimum parallel running time (in column `TPDRG`) of our parallel implementation using our *TPDRG* scheduling method and the number of processors used for gaining it, denoted by column #`CPUs (A)`. To evaluate the efficiency of our *TPDRG*

Sys	$noregSeq$ (s)	$regSeq$ (s)	$slowBy$	#CPUs	$SigPara$ (s)	SPD
1	3.626	3.997	0.01	5	1.942	1.9
2	707.533	727.954	0.01	9	119.435	5.9
3	463.016	476.161	0.01	9	207.287	2.3
4	2132.871	2162.39	0.01	11	894.267	2.5
5	4.102	4.140	0.01	11	1.632	2.6
6	866.271	866.199	-	13	451.927	2.0
7	298.329	305.243	0.01	11	96.376	3.1

Table 2: Wall time (s) for sequential (with vs without *regularized initial*) vs parallel

#P	Sys 1	Sys 2	Sys 3	Sys 4	Sys 5	Sys 6	Sys 7
3	3.142	355.08	278.703	1401.432	2.101	622.878	104.979
5	1.942	225.286	214.241	1004.694	2.101	481.734	98.436
7	1.911	142.740	209.171	939.403	1.905	470.175	97.191
9	1.913	119.435	207.287	905.249	1.778	455.207	96.704
11	1.952	119.477	207.083	894.267	1.632	453.125	96.376
13	-	119.093	206.375	874.531	1.611	451.927	96.416
17	-	120.009	211.702	865.509	1.626	451.570	96.203
21	-	119.173	-	852.488	-	451.357	96.539

Table 3: Parallel timing (s) vs #processor

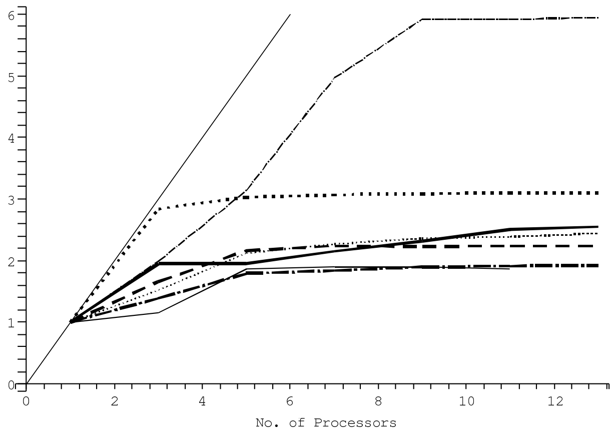

Figure 3: Speedup vs number of processors

scheduling, we also implemented a parallel version using the *Greedy scheduling* for comparison. To reveal the influence of the number of processors, we investigate two timings for the *Greedy scheduling* technique. One is using the same number of processors as used in our best parallel run that we noticed. We record one more to use 2 more processors than that in column `#CPUs (A)`. Except for the very small example *eco6*, all other examples show a better timing for the *TPDRG* scheduling method. This proves that our *TPDRG* scheduling is heuristically efficient.

System	$TPDRG$ (best)	#CPUs (A)	Greedy (A)	#CPUs (B)	Greedy (B)
1	1.911	7	1.792	9	1.778
2	119.093	13	120.505	15	120.516
3	206.375	13	213.206	15	213.354
4	852.488	20	896.787	22	939.618
5	1.611	13	1.631	15	1.632
6	451.357	20	500.504	22	469.354
7	96.203	17	100.779	19	96.169

Table 4: *Best* TPDRG *timing vs* Greedy scheduling (s)

To get the knowledge of the effectiveness of our data communication techniques, we measured the total data size that the workers have read and written for data exchanging. This data size is the number of integers in the array allocated in the shared memory segments at run time. An integer has 64 bits on Silky. These allocations are dynamic and freed after each use. The cpu time for reading covers the conversion back to lists of multivariate polynomials in Aldor by reversing Kronecker substitution from an array of integers. The cpu time for writing covers the conversion to an array of integers from lists of multivariate polynomials in Aldor by Kronecker substitution.

System	DataSize (#integer)	Read (ms)	Write (ms)	Workers (ms)	$Read + Write$ vs Total
1	7717	80	56	1031	13.2%
2	56689	383	172	174329	0.3%
3	112802	4394	301	125830	3.7%
4	430217	33760	426	585112	5.8%
5	13307	123	55	1121	15.9%
6	254145	9773	386	233582	4.4%
7	77426	819	317	31858	3.6%

Table 5: CPU time in read and write for data communication vs workers' total

The result reveals the time efficiency of these techniques, for the time spent in data exchanging for relatively large examples, such as CassouNogues and UteshevBikker, they only take less than 6% percent of the workers total cpu time. However, the data size is quite large. Our tests on large examples such as Virasoro ($n = 8$, $d = 2$, $p = 58309$) and Reimer-4 ($n = 4$, $d = 5$, $p = 55249$) failed by memory allocation failure. We have noticed rich parallelism having shown in these examples. The memory problem also suggests the insufficiency of the conversion methods for handling large polynomial systems for data communication. This investigation reveals the main limitation of our preliminary implementation: lack of effective method for large data communication.

7. TOWARD EFFICIENT MULTI-LEVEL PARALLELIZATION

We introduce a component-level parallel algorithm for solving non-linear polynomial systems symbolically by way of triangular decompositions. This algorithm employs techniques of solving by decreasing order of dimension, lazy evaluation and modular methods to obtain rich coarse grained parallel opportunities. With the discovery of geometrical information during the solving process, a *task pool with dimension and rank guided scheduling* scheme is designed to allow us a heuristically efficient parallelization. We have developed a parallel framework by extending the Aldor programming language to support multi-processed parallelism targeting symmetric multiprocessor machines. This framework can benefit to other algorithms in adapting to the new parallel architectures, such as SMPs and multicores. Our preliminary implementation and experimentation demonstrate good performance gain with respect to the comparable sequential solver.

Our long term goal is to achieve an efficient multi-level parallelism: *coarse grained (component)* level for tasks computing geometric objects in the solution sets, and *medium/fine grained* level for polynomial arithmetic such as GCD/resultant computation within each task. We expect that the speedup on the component level would add a multiple factor to the speedup of medium/fine grained level parallelization as parallel GCD/resultant computations.

Through this work we have noticed that our coarse-grained tasks are highly irregular and problem dependent. In many occasions a few heavy tasks dominate the computing time. We can gain higher scalability and more parallel opportunities by applying medium and fine grained parallelization for polynomial arithmetic such as multiplication, GCD/resultant, and factorization involved in each task. Parallel arithmetic for univariate polynomials over fields is well-developed. We need to extend these methods to multivariate case over more general domains with potential of automatic case discussion.

It has shown that the heavy data communication and dynamic process management can be a bottleneck for an efficient parallel execution. In fact, this is an important challenge for parallel symbolic computations in general. Our another objective is to develop a model for *threads* in ALDOR to support parallelism for symbolic computations targeting SMP and multi-cores. Emerging SMPs and multi-cores fully support thread-level parallelism. ALDOR's interoperability with C and the machine resources provides a feasible way to construct ALDOR threads.

8. REFERENCES

[1] I. A. Ajwa. *Parallel Algorithms and Implementations for the Gröbner Bases Algorithm and the Characteristic Set Method*. PhD thesis, Kent State University, Kent, Ohio, 1998.

[2] aldor.org. *The* Aldor *compiler web site*. University of Western Ontario, Canada, 2002.

[3] G. Attardi and C. Traverso. Strategy-accurate parallel Buchberger algorithms. *Journal of Symbolic Computation*, 21(4):411–425, 1996.

[4] P. Aubry and M. Moreno Maza. Triangular sets for solving polynomial systems: A comparative implementation of four methods. *J. Symb. Comp.*, 28(1-2):125–154, 1999.

[5] E. Becker, T. Mora, M. G. Marinari, and C. Traverso. The shape of the shape lemma. In *Proc. of the international symposium on Symbolic and algebraic computation*, pages 129–133, New York, NY, USA, 1994. ACM Press.

[6] R. Bradford. A parallelization of the buchberger algorithm. In *Proc. of the international symposium on Symbolic and algebraic computation*, New York, NY, USA, 1990. ACM Press.

[7] Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morisson, Jonathan M. Steinbach, Robert S. Sutor, and Stephen M. Watt. *AXIOM Library Compiler User Guide*. NAG, The Numerical Algorithms Group Limited, Oxford, United Kingdom, 1st edition, November 1994. AXIOM is a registred trade mark of NAG.

[8] P. Brucker. *Scheduling Algorithms*. Springer, Berlin, 3rd edition, 2001.

[9] R. Bündgen, M. Göbel, and W. Küchlin. A fine-grained parallel completion procedure. In *ISSAC '94: Proceedings of the international symposium on Symbolic and algebraic computation*, pages 269–277, New York, NY, USA, 1994. ACM Press.

[10] L. Busé and C. D'Andrea. On the irreducibility of multivariate subresultants. *C. R. Math. Acad. Sci. Paris*, 338(4):287–290, 2004.

[11] S. Chakrabarti and K. Yelick. Implementing an irregular application on a distributed memory multiprocessor. *ACM SIGPLAN NOTICES*, 28(7), 1993.

[12] S. Chakrabarti and K. Yelick. Distributed data structures and algorithms for Gröbner basis computation. *LISP AND SYMBOLIC COMPUTATION: An International Journal*, 7:147–172, 1994.

[13] C. Chen, F. Lemaire, M. Moreno Maza, W. Pan, and Y. Xie. Efficient computations of irredundant triangular decompositions with the `regularchains` library. Submitted to CASA 2007, 2006.

[14] X. Dahan, M. Moreno Maza, É. Schost, W. Wu, and Y. Xie. Lifting techniques for triangular decompositions. In *ISSAC'05*, pages 108–115. ACM Press, 2005.

[15] X. Dahan, M. Moreno Maza, É. Schost, and Y. Xie. On the complexity of the D5 principle. In *Proc. of* Transgressive Computing 2006, Granada, Spain, 2006.

[16] T. Decker and W. Krandick. Parallel real root isolation using the descartes method. In *Proc. 6th HiPC'99*. ACM Press, 1999.

[17] A. Filatei, X. Li, M. Moreno Maza, and É Schost. Implementation techniques for fast polynomial arithmetic in a high-level programming environment. In *Proc. ISSAC'06*, pages 93–100. ACM Press, 2006.

[18] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 1999.

[19] P. Gianni, B. Trager, and G. Zacharias. Gröbner Bases and Primary Decomposition Of Polynomial Ideals. *J. Symb. Comp.*, 6:149–167, 1988.

[20] J. Grabmeier, E. Kaltofen, and V. Weispfenning, editors. *Computer Algebra Handbook*. Springer, 2003.

[21] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(3):416–429, March 1969.

[22] The Computational Mathematics Group. The basicmath library. NAG Ltd, Oxford, UK, 1998. http://www.nag.co.uk/projects/FRISCO.html.

[23] H. Hong and H. W. Loidl. Parallel computation of modular multivariate polynomial resultants on a shared memory machine. In B. Buchberger and J. Volkert, editors, *Proc. of CONPAR 94–VAPP VI, Springer LNCS 854.*, pages 325–336. Springer Verlag, September 1994.

[24] R. D. Jenks and R. S. Sutor. *AXIOM, The Scientific Computation System*. Springer-Verlag, 1992. AXIOM is a trade mark of NAG Ltd, Oxford UK.

[25] M. Kalkbrener. A generalized euclidean algorithm for computing triangular representations of algebraic varieties. *J. Symb. Comp.*, 15:143–167, 1993.

[26] M. Korch and T. Rauber. Evaluation of task pools for the implementation of parallel irregular algorithms. In *Proc. IEEE ICPPW'02*. IEEE, 2002.

[27] D. Lazard. A new method for solving algebraic systems of positive dimension. *Discr. App. Math*, 33:147–160, 1991.

[28] G. Lecerf. Computing the equidimensional decomposition of an algebraic closed set by means of lifting fibers. *J. Complexity*, 19(4):564–596, 2003.

[29] H. Lee, J. Kim, S. Hong, and S. Lee. Task scheduling

using a block dependency dag for block-oriented sparse cholesky factorization. In *Proc. SAC*, Italy, 2000. ACM Press.

[30] F. Lemaire, M. Moreno Maza, and Y. Xie. The `RegularChains` library. In *Maple 10*, Maplesoft, Canada, 2005. Refereed software.

[31] F. Lemaire, M. Moreno Maza, and Y. Xie. Making a sophisticated symbolic solver available to different communities of users. In *Proc. of* Asian Technology Conference in Mathematics*'06*, 2006.

[32] A. Leykin. On parallel computation of Gröbner bases. In *ICPP Workshops*, pages 160–164, 2004.

[33] Maplesoft. *Maple 10*. http://www.maplesoft.com/, 2005.

[34] M. Moreno Maza. On triangular decompositions of algebraic varieties. Technical Report TR 4/99, NAG Ltd, Oxford, UK, 1999. Presented at the MEGA-2000 Conference, Bath, England. http://www.csd.uwo.ca/∼moreno.

[35] M. Moreno Maza and R. Rioboo. Polynomial gcd computations over towers of algebraic extensions. In *Proc. AAECC-11*, pages 365–382. Springer, 1995.

[36] C. Polychronopoulos and D. Kuck. Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers. *IEEE Transactions on Parallel and Distributed Systems*, C-36:1425–1439, December 1987.

[37] M. O. Rayes, P. S. Wang, and K. Weber. Parallelization of the sparse modular gcd algorithm for multivariate polynomials on shared memory multiprocessors. In *Proc. of the international symposium on Symbolic and algebraic computation*, pages 66–73, New York, NY, USA, 1994. ACM Press.

[38] A. Ranade S. Chakrabarti and K. Yelic. Randomized load balancing for tree-structured computations. In *IEEE Scalable High Performance Computing Conference*, 1994.

[39] s. Lucco. A dynamic scheduling method for irregular parallel programs. In *Proc. ACM SIGPLAN'92 PLDI-6*, CA, 1992. ACM Press.

[40] B. David Saunders, Hong R. Lee, and S. Kamal Abdali. A parallel implementation of the cylindrical algebraic decomposition algorithm. In *ISSAC*, pages 298–307, 1989.

[41] T. Shimoyama and K. Yokoyama. Localization and primary decomposition of polynomial ideals. *J. Symb. Comput.*, 22(3):247–277, 1996.

[42] A.J. Sommese, J. Verschelde, and C.W. Wampler. Numerical decomposition of the solution sets of polynomial systems into irreducible components. *SIAM J. Numer. Anal.*, 38(6):2022–2046, 2001.

[43] L. Ni T. Tzen. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1), 1993.

[44] D. M. Wang. *Elimination Methods*. Springer, Wein, New York, 2000.

[45] Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. A first report on the a# compiler. In *ISSAC '94: Proceedings of the international symposium on Symbolic and algebraic computation*, New York, NY, USA, 1994. ACM Press.

[46] V. Weispfenning. Canonical comprehensive grobner bases. In *ISSAC 2002*, pages 270–276. ACM Press, 2002.

[47] W. T. Wu. A zero structure theorem for polynomial equations solving. *MM Research Preprints*, 1:2–12, 1987.

[48] Y.W. Wu, W.D. Liao, D.D. Lin, and P.S. Wang. Local and remote user interface for ELIMINO through OMEI. Technical report, Kent State University, Kent, Ohio, 2003. http://icm.mcs.kent.edu/reports/.

[49] Y.W. Wu, G.W. Yang, H. Yang, W.M. Zheng, and D.D. Lin. A distributed computing model for wu's method. *Journal of Software (in Chinese)*, 16(3), 2005.