Parallelization of Quantifier Elimination on Workstation Network *

Hoon Hong

September 10, 1991

Research Institute for Symbolic Computation
Johannes Kepler University
A-4040 Linz, Austria
e-mail: hhong@risc.uni-linz.ac.at

Abstract

This paper reports our effort to parallelize on a network of workstations the quantifier elimination algorithm over the reals which was devised by Collins and improved by the author. The preliminary experiments show both *sub-linear* and *super-linear* speedups due to speculative parallelism. On the problems we have tested so far, the efficiencies range between 70% and 320%.

1 Introduction

This paper reports our effort to parallelize on a network of workstations the quantifier elimination algorithm over the reals which was devised by Collins [9] and improved by the author [20, 11]. The preliminary experiments show both *sub-linear* and *super-linear* speedups due to speculative parallelism. On the problems we have tested so far, the efficiencies range between 70% and 320%.

Quantifier elimination in a given theory is a procedure that constructs from a quantified formula an equivalent quantifier–free formula. The decision problem is, then, a special case where there is no free variable in the quantified formula. In this paper we are concerned with the first order theory of the reals where a

*Supported in part by a grant from Digital Equipment Cooperation under the contract EERP-AU021 and also by grant from Austrian Science Foundation on parallel symbolic computation.

formula is constructed from real polynomial equations and inequalities by using logical connectives and quantifiers. For example, $(\forall x) x^2 + bx + c > 0$ is a such formula. An equivalent quantifier–free formula is $b^2 - 4c < 0$.

Many important and interesting problems from diverse areas can be expressed as quantifier elimination in this theory: robot motion planning, geometric theorem proving and discovery, termination proof of term rewrite systems, constraint solving in logic programming, etc.

During 1930–1950, Tarski [34] found the first quantifier elimination algorithm in this theory. Since then various improvements and new methods have been devised and analyzed [33, 8, 17, 5, 9, 1, 28, 2, 4, 13, 15, 14, 7, 16, 29, 30, 31, 18, 19, 26, 20, 11, 25, 6, 23, 22]. In this paper we investigate the parallelization of the algorithm which was originally devised by Collins [9] and improved by the author [20].

In [32] Saunders, Lee, and Abdali report their work on parallelizing Collins' original algorithm on a shared memory machine, achieving about 50% efficiency. They observed two main bottlenecks: (1) A few tasks (stack constructions) took very long, causing several other processors to be idle. (2) The lock operations on the shared heap caused heavy overhead (16%). We tackled these two problems as follows:

- We parallelized the author's *improved sequential algorithm* which, unlike the original algorithm, allows stacks in any level to be constructed even when some stacks in lower levels are not constructed yet. Therefore as soon as a processor finishes constructing a stack it can immediately begin to construct another stack in a higher level, without having to wait until another processor finishes constructing an expensive stack. Therefore the effect of expensive stacks is greatly reduced, resulting in better load balance.

- We used a *network of workstations* where each processor accesses its own local memory and carries out garbage collection also independently of each other. Thus the overhead of lock operation does not exist. The communication overhead was made negligible by ensuring coarse granularity of tasks.

However a new problem arises: One important feature of the improved sequential algorithm is that it very often completes quantifier elimination by a partially built cylindrical algebraic decomposition. This feature can interfere with parallelism, either *for* or *against*. Specifically it can happen that the parallel program does not construct (or aborts constructing) stacks which the sequential program constructs, but it can also happen that the parallel program constructs stacks which the sequential program does not construct.

The proportion of these two cases depends on the order in which the stacks are constructed. Therefore we need, if possible, to devise a strategy (for ordering stack constructions) that generates the first case more often. Currently we do not know any general strategy which works well for all inputs, thus in these preliminary investigation we used a naively chosen strategy, and accordingly the experimental results show both cases: *super*-linear and *sub*-linear speedups.

The structure of this paper is as follows: In Section 2 we give a brief description of the improved quantifier elimination algorithm. In Section 3 we describe a parallel version of the sequential algorithm. In Section 4 we describe the implementational issues. In Section 5 we present various experimental results along with some discussions.

2 Sequential Algorithm

In this section, we give a very brief overview of the sequential algorithm. For a detailed exposition of the algorithm, see [20, 10]. The algorithm takes a quantified formula and produces an equivalent quantifier-free formula[1] by carrying out the following three steps:

1. *Projection.* The input n–variate polynomials are 'projected" into a set of $(n-1)$–variate polynomials in a way that a CAD of n–space can be built on top of a CAD of $(n-1)$–space. This process is successively carried out until univariate projection polynomials are obtained.

2. *Truth-invariant partial CAD construction.* The projection polynomials are used in constructing a (partial) decomposition of free variable space such that (1) the cells are "cylindrically" arranged, (2) the input formula has an invariant truth value throughout each cell, and (3) the constant truth value is determined for each cell.

3. *Solution formula construction.* A quantifier-free formula equivalent to the input formula is inferred from the truth-invariant partial CAD.

In this paper we parallelize the second step, thus we will elaborate the second step a little further. We begin by introducing the terminology which will be used throughout the remaining of this paper.

- A *CAD tree* is a tree such that the root node is the hypothetical single cell of a decomposition of 0-dimensional space, the children of the root node are the cells of a decomposition of 1-dimensional space, and the children of a non-root node c are the cells in the stack over the cell (node) c. Each node in a tree is labeled with the associated truth value which at any time can be one of the three: `true`, `false`, and `undetermined`. We will denote the truth value of a node c by $v(c)$.

[1] Actually in rare cases the algorithm fails to produce an equivalent formula and instead produces both a necessary condition and a sufficient condition to the input formula. For the details, see [20].

- A *candidate node* is a leaf whose truth value is `undetermined`.
- The *initial CAD tree* is a CAD tree with the single root node labeled `undetermined`.
- A *final CAD tree* is a CAD tree without any candidates nodes. In fact it is the truth-invariant CAD that we desire to construct.

From now on, we will drop the word "CAD" when no ambiguity arises; i.e. we will say *the initial tree* instead of *the initial CAD tree*. We will also interchangeably use the terms: "cells" and "nodes".

Roughly the algorithm works as follows: it begins with the initial tree, and repeatedly extends the tree by constructing children (stack) of candidate nodes, until the final tree is obtained. However the actual algorithm is a bit more complicated since sometimes certain portion of the tree might be pruned. Thus during the execution of the algorithm, the tree sometimes grows and other times shrinks. Here is the algorithm.

Truth Invariant CAD

(1) [Initialize.] Set the current tree to be the initial tree.

(2) [Choose.] Choose a candidate node c from the current tree.

(3) [Extend.] Construct the children nodes of the chosen node c, and attempt to determine the truth values of some of its children.

(4) [Prune.] Determine the truth values of as many ancestors of c as possible, referring to the quantifiers, removing from the tree the subtree of each node whose truth value is thus determined.

(5) [Done?] If there is no candidate node in the current tree, we are done. Otherwise go to Step (2). ■

Note that Step (2) of the algorithm is *non-deterministic* in the sense that it does not specify which candidate node to be chosen in case there is more than one in the current tree. It can be shown that the output of this algorithm does not depend on the choices made. But the computing time does depend on the choices. Since it is difficult (impossible) to determine the optimal choices, usually various strategies are used instead. For the experiments reported in this paper, we used a strategy which seems to work well for most inputs: roughly it chooses a node whose stack construction is guessed to be cheapest. See [20] for the details.

Step (4) of the algorithm is based on the following theorem:

Theorem 1 *Let $(Q_{f+1}x_{f+1})\cdots(Q_r x_r)F(x_1,\ldots,x_n)$ be an input quantified formula. Let c be a node of level k, $f \le k \le r$, and let $c_1,\ldots,c_n$ be the children nodes of the node c. If $Q_{k+1} = \exists$, $v(c) = \bigvee_{i=1}^{n} v(c_i)$. If $Q_{k+1} = \forall$, $v(c) = \bigwedge_{i=1}^{n} v(c_i)$.*

This theorem provides a way to determine the truth value of a node from the truth values of its children. Step (4) applies this theorem repeatedly, determining the truth value of a parent node from the truth values of its children, until it cannot be applied any more.

3 Parallelized Algorithm

In this section we describe the successive modifications we made on the sequential algorithm for coarse-grain parallelism. The most time consuming part of the sequential algorithm is Step (3) where the children (stack) of a chosen candidate node is constructed. Thus we will try to parallelize the executions of this step.

We begin by noting that in Step (2) of the sequential algorithm there can be more than one candidate nodes. So the main idea is to construct the children of as many candidate nodes as possible in parallel. A naive approach may be to carry out in parallel Step (3) for all candidate nodes, and wait until all are finished, then carry out Step (4) all together. In other words the parallel processes are synchronized at the beginning of Step (4). However, this approach has two main problems:

- The computation times for Step (3) can be very *heterogeneous*. For example, it might take only a few milli-seconds for one candidate node, but a few minutes for another candidate node. Thus the load might not be balanced, making some processors idle.
- It might turn out that during Step (4) a sub-tree built by a processor is pruned out. But this fact will be discovered *only after* that sub–tree has been already completely built.

A better way is to use the model of *a manager and workers*, in which the manager repeatedly assigns a task to the workers and collects the results from the workers. The manager might *abort* the computation of a worker, if the results from other workers indicate that the outcome of the computation carried out by that worker is no longer needed.

The manager keeps track of the current tree. In particular it records whether a node has been assigned to a worker or not. This is done by attaching to each node one more boolean label. The manager also keeps track of the status of each worker: working or idle. Obviously, a worker goes into the status of working when a task is assigned, and becomes idle when the assigned task has been completed or aborted.

Manager (Version 1)

(1) [Initialize.] Set the current tree to be the initial tree.

(2) [Assign tasks.] Simultaneously assign an unassigned candidate node to each idle worker.

(3) [Wait and Extend.] Wait for a result from a worker. (A result from a worker is a stack (children) over the node c assigned to the worker.) Attach the children nodes to the current tree.

(4) [Prune.] Determine the truth values of as many ancestors of c as possible, referring to the quantifiers, removing from the tree the subtree of each node whose truth value is thus determined.

(5) [Abort.] Abort the computation of each worker whose result is not needed any more. (This can be determined by checking whether the candidate node assigned to a worker has been pruned out.)

(6) [Done?] If there is no candidate node in the current tree, we are done. Otherwise go to Step (2). ■

Worker (Version 1)

(1) [Wait.] Wait for a candidate node from a manager. Let c be the node received.

(2) [Extend.] Construct the children nodes of the node c, and attempt to determine the truth values of some of its children.

(3) [Report.] Report the result to the manager. Go to Step (1). ■

When the manager aborts the computation of a worker, the worker immediately jumps to Step (1). This parallel algorithm helps to reduce the problems of the naive approach mentioned above. Specifically

- It balances the load much better since there is no explicit synchronization that is done. While a worker is working on a big task, other workers do not have to wait at the synchronization barrier, but instead can continue with other candidate nodes.

- It detects useless computations early and aborts them.

However this algorithm still has a problem: the *communication overhead* might become dominant, erasing the gain obtained by parallelism. This is so because

- For some candidate nodes, the computation time taken by the worker is so small (a few milliseconds) that the communication overhead cannot be ignored. In fact this is serious since we are using a network of workstations and the communication among them is done via TCP/IP whose typical latency is on the order of tens to hundreds of milliseconds.

- An input formula of a moderate size often requires enormously many stack constructions. (For example, several hundreds of thousands.) Since each of these constructions require at least one message passing, even when the overhead of each message passing is relatively small, the accumulated overhead can be huge.

We overcome this problem by letting a worker construct many stacks (a larger subtree) over a candidate node, not just one stack (the subtree consisting of only the children of the candidate node). The manager controls the size of subtrees by setting two limits:

N_{max}: The maximum number of stacks that will be constructed by the worker.

T_{max}: Roughly the maximum computing time that the worker can spend on constructing stacks. More precisely, before the worker constructs each stack, it checks whether it has already used up the allotted time, and if so, it reports to the manger the subtree it has built thus far.

The purpose of introducing the limit T_{max} is to help load balancing. Here is the algorithm.

Manager (Version 2)

(1) [Initialize.] Set the current tree to be the initial tree.
(2) [Assign tasks.] Simultaneously assign an unassigned candidate node to each idle worker, also setting the two limits: N_{max} and T_{max}.
(3) [Wait and Extend.] Wait for a result from a worker. (A result from a worker is a *subtree* over the candidate node c assigned to the worker.) Attach the subtree to the current tree.
(4) [Prune.] Determine the truth values of as many ancestors of c as possible, referring to the quantifiers, removing from the tree the subtree of each node whose truth value is thus determined.
(5) [Abort.] Abort the computation of each worker whose result is not needed any more.
(6) [Done?] If there is no candidate node in the current tree, we are done. Otherwise go to Step (2). ■

Worker (Version 2)

(1) [Wait and Initialize.] Wait for a candidate node and N_{max} and T_{max} from the manager. Let c_0 be the candidate node it received from the manager, Set the current tree to be the single node c_0. Set $i = N_{max}$. Set $T = T_{max}$.
(2) [Choose.] Choose a candidate node c from the current tree.
(3) [Extend.] Construct children nodes of the chosen node c, and attempt to determine the truth values of some of its children.
(4) [Prune.] Determine the truth values of as many ancestors of c as possible, referring to the quantifiers, removing from the tree the subtree of each node whose truth value is thus determined.
(5) [Done?] Set $i \leftarrow i - 1$ and set $T \leftarrow T$ minus the time consumed by Steps (2), (3) and (4). If one of the following holds, report the current tree to the

manager: (a) there is no candidate node in the current tree, or (b) $i = 0$, or (c) $T \leq 0$. Otherwise goto Step (2). ■

Note that the previous algorithm (version 1) is just a special case of this algorithm (version 2) where $N_{max} = 1$ and T_{max} is arbitrary. In fact the sequential algorithm of the previous section can also be viewed as a special case of this algorithm where $N_{max} = T_{max} = \infty$.

Note also that Step (2) of the manager algorithm does not specify which candidate node to be assigned to which worker. Likewise Step (2) of the worker algorithm does not specify which candidate node to be chosen in case there is more than one candidate node.

Therefore, before executing this algorithm, one needs to decide the following (which we will call a *strategy*): N_{max}, T_{max}, an ordering for manager node choice (scheduling), and an ordering for worker node choice. In devising suitable strategies, one should consider the following conflicting factors which influence the efficiency of the algorithm:

- *Load balancing*: If N_{max} or T_{max} are chosen too big, then the load might not be balanced well.
- *Communication overhead*: If N_{max} or T_{max} are too small, then the communication overhead might become dominant.
- *Speculativeness*: Depending on the chosen strategy the parallel algorithm might construct fewer (or cheaper) stacks than the sequential algorithm, or vice versa.

Currently we do not know any general strategy which works well for all inputs, thus in this preliminary investigation we used the following (naive) strategy.

- $N_{max} = \begin{cases} 1 & \text{if th level of the chosen node} \leq 2 \\ \infty & \text{otherwise} \end{cases}$
  The reason for $N_{max} = 1$ for the lower levels is to produce sufficiently many candidate cells at the beginning in order to help load balancing. The reason for $N_{max} = \infty$ for the higher levels is to reduce communication overhead by making task sizes sufficiently large.
- $T_{max} = 10$ seconds. This choice was made because we thought that 10 seconds of idling of workers is tolerable.
- *Manager node choice*: the depth first and the left hand side first order. This choice was made because it can be efficiently and easily implemented.
- *Worker node choice*: the same strategy as the one used by the sequential algorithm. Roughly it chooses a cell whose stack construction is guessed to be cheapest. For the details see [20].

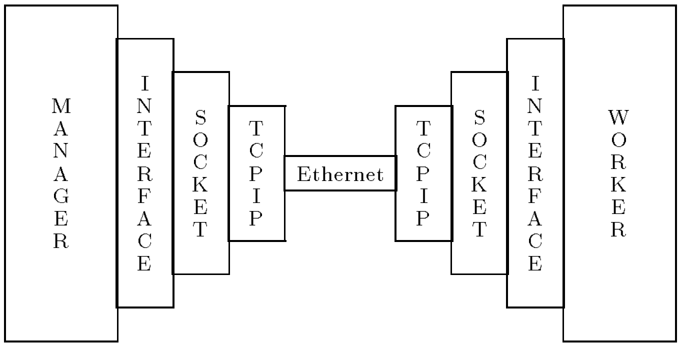


Figure 1: Underlying Communication System

4 Implementation

We have implemented both the sequential and parallel algorithms in C language on the computer algebra library SAC-2 [12]. Originally SAC-2 was written in the programming language ALDES [27], but we translated the whole library into C.

The communication between the workstations was carried out on an ethernet using TCP/IP protocol by means of sockets provided with Berkeley UNIX. On top of the socket, we have built a high level interface which provides the view of a manager and worker model of parallelism (see Figure 1). The interface consists of several primitives. Here are the primitives that can be called by the manager:

- *SendTask:*

  When the manager wishes to assign a task to a worker, it simply makes a call to this primitive passing an object representing the task and the ID of the worker.

  Then the primitive first encodes the object into a bit stream. (In SAC-2, every object is represented as a nested list.)

  Finally it sends the bit stream to the specified worker via a socket.

- *AbortTask:*

  When the manager wishes to abort a task running on a worker, it makes a call to this primitive passing the ID of the worker.

  Then the primitive sends a UNIX–signal to the interface of the worker via a socket.

  The worker interface, upon receiving the signal, aborts the current computation of the worker (except for a few cases explained below) and long jumps to the beginning of the worker algorithm.

There are two exceptional cases: (1) When the worker is sending a result back to the manager, it is not aborted in order to guarantee that all messages arriving at the manager interface are consistent. (2) When the worker is executing a code which is protected by a call to *DisableAbortion* primitive described below.

- *ReceiveResult:*

  When the manager wishes to receive a result from a worker, it makes a call to this primitive.

  Then the primitive waits for a message from a worker, and upon receiving a message it checks whether the message is out-dated; in other words whether it is the result of an aborted task, which was still sent to the manager since the abortion signal arrived while the worker was sending the result back to the manager. If so, it simply discards the message, and waits for another message from a worker.

  When receiving a valid message, it decodes the message (bit stream) into a list object, and return the object and the worker ID to the manager.

Here are the primitives that can be called by a worker.

- *ReceiveTask:*

  When a worker wishes to receive a task from the manager, it makes a call to this primitive.

  Then the primitive waits for a message from the manager, and upon receiving a message, it decodes the message into an object and return the object to the worker.

- *SendResult:*

  When a worker wishes to send a result to the manager it makes a call to this primitive passing the object representing the result.

  Then the primitive encodes the object into a bit stream and sends it to the manager via a socket.

  The whole operation is protected from abortion.

- *DisableAbortion/EnableAbortion:*

  When a worker wishes to protect a certain section of its code from abortion, it wraps the section with these primitives. For example, it is used for protecting the garbage collector from being aborted.

In addition to these, there are several other primitives for housekeeping, error-handling, tracing, statistics gathering, etc. We believe that these primitives can also be used for parallelizing other algebraic algorithms, and we plan to do so.

5 Experiments

5.1 Test Problems

We carried out experiments on both the sequential and the parallel (version 2) algorithms on a number of quantifier elimination problems:

- **Quartic.** Find a necessary and sufficient condition on the coefficients of a quartic polynomial such that it is positive for all real values of the variable.
$$(\forall x) x^4 + ax^3 + bx^2 + cx + d > 0.$$
Variable ordering: (a, b, c, d, x).
A simplified form of this problem ($a = 0$) has been studied by Arnon and Mignotte [3] and Hong [20].

- **Cubic.** Consider a plane curve which is parameterized by a quadratic polynomial and a cubic polynomial such as:
$$\begin{aligned} x &= t^2 + at + b \\ y &= t^3 + ct^2 + dt + e. \end{aligned}$$
We are interested in finding an implicit representation of the curve.
$$(Et)[x = t^2 + at + b \wedge y = t^3 + ct^2 + dt + e].$$
Variable ordering: (a, b, c, d, e, x, y, t).
This problem was communicated to the author by Dennis Arnon. It arose while studying the topological changes on a cubic curve caused by perturbation of the coefficients.

- **McPhee.** This problem was communicated to the author by McPhee.
$$\begin{array}{l} (\exists x)(\exists y)(\exists z) \\ [[ay^2 + by + c = 0 \wedge dz + e \neq 0 \wedge fz + g \neq 0] \\ \Longrightarrow \\ [ax^2 + bx + c = 0 \wedge dx + e \neq 0 \wedge fx + g \neq 0]]. \end{array}$$
Variable ordering: $(a, b, c, d, e, f, g, x, y, z)$.

- **Torus.** Find a defining formula for the torus, centered at the origin, lying on the $y - z$ plan, with the inner radius 1, and with the outer radius 2.
$$(\exists b)(\exists c)\ \ (b^2 + c^2 = 9/4\ \wedge\ x^2 + (y - b)^2 + (z - c)^2 = 1/4).$$
Variable ordering: (y, z, x, b, c). This problem has been studied by Kalkbrenner and Stifter [24] and also by Hong [21].

We should remark here that some of the above problems can be solved more efficiently (even by a pencil and paper) if one uses different formulations from the ones given above. Though this is an interesting research topic, in this paper we are more interested in studying the effect of parallelism, and thus we will be satisfied with the formulations above.

5.2 Performance Measures

Since a network might consist of workstations of various models which differ from each other in their speeds, memory sizes, disk spaces, paging speeds, etc[2], the performance analysis of a parallel program should take account of such non-homogeneity. We will spend some time here developing one such framework for performance analysis.

In the following, let N be the number of the workstations used for executing the parallel program. We index the workstations by the first N positive integers. We assume that on each workstation a worker is executed, and that on the first workstation the manager is also executed. For each experiment, we measure the following quantities. From now we call them *measured quantities*.

- $T_{\mathrm{seq},i}$ — the elapsed time (in seconds) taken by the sequential program on the i–th workstation.[3]
- T_{par} — the elapsed time (in seconds) taken by the parallel program on the network of workstations.
- $T_{\mathrm{run},i}$ — the running time (in seconds) of the i–th worker. This is the same as T_{par} minus the idle time of the worker.
- $T_{\mathrm{run,man}}$ — the running time (in seconds) of the manager. This is the same as T_{par} minus the idle time of the manager.
- $T_{\mathrm{com},i}$ — the time (in seconds) taken by communication between the manager and the i–th worker.

From the measured quantities we will compute several informative *derived quantities*. But before we list them, we need to define several notations.

- S_i — the speed of the i–th workstation. This is relatively defined as

$$S_i = \frac{K}{T_{\mathrm{seq},i}}$$

where K is a constant which only depends on the input.

[2] In fact it is usual that workstations are acquired incrementally, and that each time different models or different types of machines are acquired.

[3] The elapsed time usually consists of the cpu time and the page wait time. Therefore more precise performance analysis might require the measurements of both the cpu time and the page wait time. But for the sake of simpler analysis, we will ignore the distinction between cpu time and elapsed time. In fact, in all our experiments we made sure that the page wait time was negligible for both the sequential and parallel programs.

S the average speed of the workstations, defined as

$$S = \frac{\sum_{i=1}^{N} S_i}{N}.$$

C_{seq} the complexity of the sequential program, defined as (for any i)

$$C_{\text{seq}} = S_i T_{\text{seq},i}.$$

C_{par} the complexity of the parallel program, defined as

$$C_{\text{par}} = T_{\text{run,man}} S_1 + \sum_{i=1}^{N} T_{\text{run},i} S_i.$$

Now we list the derived quantities.

T_{seq} the *average sequential time*, which is defined as

$$T_{\text{seq}} = \frac{C_{\text{seq}}}{S},$$

This can be computed from the measured quantities by the following relation:

$$T_{\text{seq}} = \frac{N}{\sum_{i=1}^{N} \frac{1}{T_{\text{seq},i}}}.$$

σ the *speedup* obtained by the parallel program, which is defined as

$$\sigma = \frac{T_{\text{seq}}}{T_{\text{par}}}.$$

ϵ the *efficiency* of the parallel program, which is defined as

$$\epsilon = \frac{\sigma}{N}.$$

This can be computed from the measured quantities by the following relation:

$$\epsilon = \frac{1}{\sum_{i=1}^{N} \frac{T_{\text{par}}}{T_{\text{seq},i}}}.$$

ρ the *complexity ratio* indicating how much the complexity is changed by using the parallel program, which is defined as

$$\rho = \frac{C_{\text{par}}}{C_{\text{seq}}}.$$

This can be computed from the measured quantities by the following relation:

$$\rho = \frac{T_{\text{run,man}}}{T_{\text{seq},1}} + \sum_{i=1}^{N} \frac{T_{\text{run},i}}{T_{\text{seq},i}}.$$

υ_i the *individual utilization* of the i–th worker, which is defined as

$$\upsilon_i = \frac{T_{\mathrm{run},i} + T_{\mathrm{com},i}}{T_{\mathrm{par}}}.$$

Note that we include the communication overhead here. Thus it measures the utilization of a worker from the manager's point of view: the manager cannot tell the difference between the communication time and the actual run time.

υ_{man} the *individual utilization* of the manager, which is defined as

$$\upsilon_{\mathrm{man}} = \frac{T_{\mathrm{run,man}}}{T_{\mathrm{par}}}.$$

υ the *average utilization* of the workstations, which is defined as

$$\upsilon = \frac{\upsilon_{\mathrm{man}} S_1 + \sum_{i=1}^{N} \upsilon_i S_i}{\sum_{i=1}^{N} S_i}.$$

This can be computed from the measured quantities by the following relation:

$$\upsilon = \frac{\frac{T_{\mathrm{run,man}}}{T_{\mathrm{seq},1}} + \sum_{i=1}^{N} \frac{T_{\mathrm{run},i}}{T_{\mathrm{seq},i}} + \sum_{i=1}^{N} \frac{T_{\mathrm{com},i}}{T_{\mathrm{seq},i}}}{\sum_{i=1}^{N} \frac{T_{\mathrm{par}}}{T_{\mathrm{seq},i}}}.$$

μ_i the *individual communication overhead* of the i–th worker, which is defined as

$$\mu_i = \frac{T_{\mathrm{com},i}}{T_{\mathrm{par}}}.$$

μ the *average communication overhead*, which is defined as

$$\mu = \frac{\sum_{i=1}^{N} \mu_i S_i}{\sum_{i=1}^{N} S_i}.$$

This can be computed from the measured quantities by the following relation:

$$\mu = \frac{\sum_{i=1}^{N} \frac{T_{\mathrm{com},i}}{T_{\mathrm{seq},i}}}{\sum_{i=1}^{N} \frac{T_{\mathrm{par}}}{T_{\mathrm{seq},i}}}.$$

The following easily proven relation shows how the efficiency of a parallel program depends on its utilization, communication overhead, and complexity ratio:

$$\epsilon = \frac{\upsilon - \mu}{\rho}.$$

Clearly one wants to increase the efficiency ϵ, and for that purpose, one hopes to increase the utilization υ and decrease the complexity ρ and the communication overhead μ. But it might happen that while trying to increase υ, to one's dismay, μ or ρ increases, and vice versa.

Table 1: Network Configuration

Model	Processor	Memory	# of units
DECstation 5000	mips R3000	32 MB	1
DECstation 2000	mips R2000	8 MB	5

Table 2: Timings for Sequential Program (seconds)

	DECstation 5000	DECstation 2000	Ratio
quartic	3145	7697	2.4
cubic	299	709	2.4
mcphee	649	1528	2.4
torus	2653	5961	2.3

5.3 Network Configuration

The preliminary experiments were done on a network of 6 workstations as shown in Table 1.[4] We ran the sequential program on one workstation of each model. We also ran the parallel program on the network with varying number of workstations, in particular the first i workstations listed in Table 1 for $i = 2, \ldots, 6$.

On each workstation, one worker program was executed, and on the first workstation in the table (DECstation 5000), the manager program was also executed. All the experiments were carried out after midnight in order to reduce the effect of our users' processes on the effective speeds of the workstations and of the network.

5.4 Experimental data and Discussion

Table 2 reports the elapsed time taken by the sequential program on a workstation of each model. Note that DECstation 2000 is about 2.4 times slower than DECstation 5000. As mentioned previously, the paging times were negligible (less than 1%).

Tables 3 through 6 report various performance measures obtained from the experiments with the parallel program. For the definitions of the terminology used in the tables and the figures, see the previous subsection 5.2. Now we make several remarks on the experimental data.

Quartic Problem

The speedup is almost linear until 5 workers are used, but then it drops when 6 workers are used. It is mainly because the utilization drops to 75%.

[4] We are currently preparing for another experiment with 250 Sun Sparc workstations.

Note that for each column of the table, there is exactly one worker with 100% utilization. We conjecture that this happens because while one worker is finishing up the last task (stack construction), the other workers are idle. This could be improved by using a strategy which constructs expensive stacks first so that the last stack construction is cheap, thus reducing the idling times of other workers. But then this might result in constructing expensive stacks which might not be constructed by the sequential algorithm. Thus it is difficult to predict what the net outcome would be. We plan to carry out more experiments to obtain better understanding.

Note that the communication overheads are shown as 0.00 in the table because they are less than 0.5%. Note also that the complexity ratios are slightly less than 1, which means that the parallel algorithm constructs a little fewer or cheaper stacks than the sequential algorithm.

Cubic Problem and McPhee Problem

We discuss the cubic and the mcphee problems together since they show similar behavior. The most interesting aspect of these two test results is that they show *super–linear* speedups. The cubic problem gives about 140% efficiency and the mcphee problem about 300% efficiency. This is mainly due to the small complexity ratios. For the cubic problem, it is about 65%, and for the mcphee problem about 30%. Thus on these problems, the parallel algorithm constructs far fewer or cheaper stacks than the sequential algorithm.

Note that there are some communication overheads (in the range of 3–13%), but their effects on the efficiency is mitigated by the huge drop in the complexity ratios.

Note also that the efficiency seems to be dropping slowly as the number of workers are increased. One might conjecture that the efficiency will eventually drop below 1 when too many workers are involved, thus causing many of them to be idle.

Torus Problem

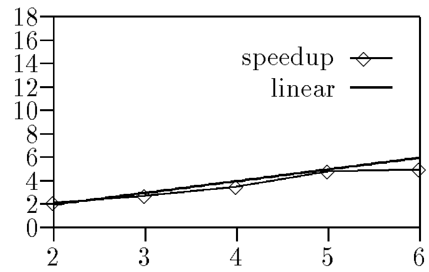
The torus problem shows the opposite behavior to the cubic and the mcphee problems: in spite of very high utilization and negligible communication overhead, it shows relatively low efficiency (about 75%). It is mainly due to the large complexity ratios, which means that the parallel algorithm constructs more (expensive) stacks than the sequential algorithm.

6 Conclusion

In this paper we reported our effort to parallelize on a network of workstations the quantifier elimination algorithm over the reals which was devised by Collins

Table 3: Quartic

# of workstations (N)	2	3	4	5	6
T_{seq}	4465	5191	5651	5968	6200
T_{par}	2117	1903	1615	1233	1244
Speedup (σ)	2.11	2.73	3.50	4.84	4.99
Efficiency (ϵ)	1.05	0.91	0.88	0.97	0.83
Communication (μ)	0.00	0.00	0.00	0.00	0.00
Complexity (ρ)	0.94	0.92	0.92	0.91	0.90
Utilization (υ)	0.99	0.84	0.80	0.88	0.75
υ_1	1.00	0.75	0.78	0.90	0.67
υ_2	0.98	1.00	0.74	1.00	1.00
υ_3		0.88	1.00	0.85	0.79
υ_4			0.74	0.82	0.68
υ_5				0.82	0.69
υ_6					0.81



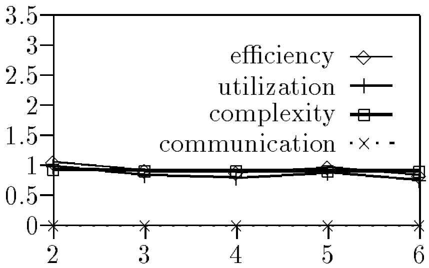
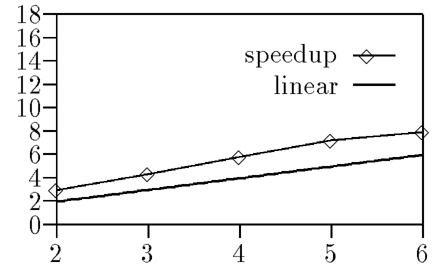


Table 4: Cubic

# of workstations (N)	2	3	4	5	6
T_{seq}	421	487	528	557	577
T_{par}	143	113	92	78	73
Speedup (σ)	2.95	4.33	5.79	7.22	7.93
Efficiency (ϵ)	1.48	1.44	1.45	1.44	1.32
Communication (μ)	0.02	0.02	0.02	0.03	0.03
Complexity (ρ)	0.64	0.64	0.65	0.65	0.64
Utilization (υ)	0.97	0.95	0.96	0.96	0.88
υ_1	0.95	0.93	0.97	0.99	0.90
υ_2	1.00	0.99	0.96	0.98	0.89
υ_3		0.94	1.00	0.99	0.99
υ_4			0.83	0.86	0.76
υ_5				0.86	0.83
υ_6					0.81



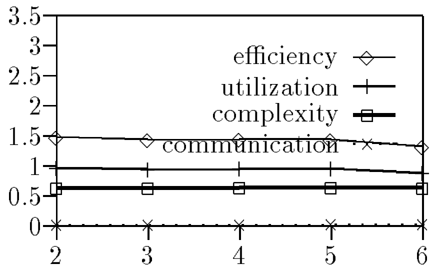
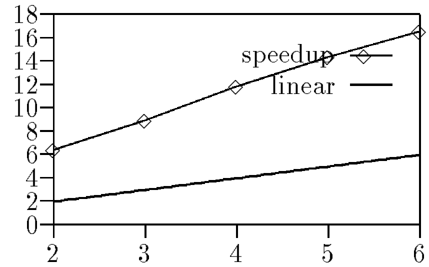


Table 5: McPhee

# of workstations (N)	2	3	4	5	6
T_{seq}	911	1053	1141	1202	1247
T_{par}	142	119	97	84	76
Speedup (σ)	6.41	8.91	11.84	14.34	16.55
Efficiency (ϵ)	3.21	2.97	2.96	2.87	2.76
Communication (μ)	0.05	0.07	0.09	0.10	0.13
Complexity (ρ)	0.29	0.30	0.30	0.30	0.31
Utilization (υ)	0.99	0.96	0.99	0.98	0.98
υ_1	0.95	0.90	0.93	0.92	0.95
υ_2	0.98	0.95	0.98	0.97	0.94
υ_3		0.93	0.93	0.94	0.90
υ_4			0.95	0.94	0.91
υ_5				0.92	0.94
υ_6					0.91



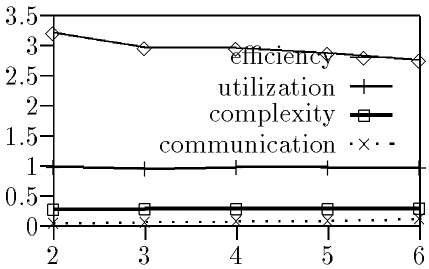
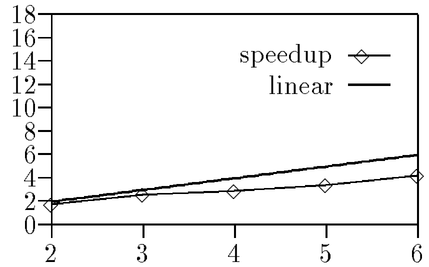
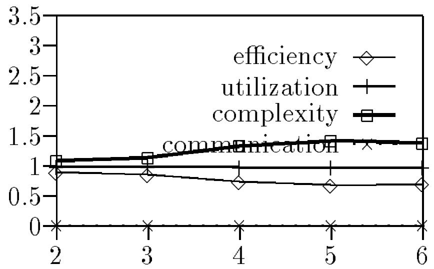


Table 6: Torus

# of workstations (N)	2	3	4	5	6
T_{seq}	3672	4211	4545	4772	4936
T_{par}	2071	1653	1567	1406	1176
Speedup (σ)	1.77	2.55	2.90	3.40	4.20
Efficiency (ϵ)	0.89	0.85	0.73	0.68	0.70
Communication (μ)	0.01	0.01	0.01	0.01	0.01
Complexity (ρ)	1.10	1.15	1.34	1.43	1.38
Utilization (υ)	0.99	0.99	0.98	0.98	0.98
υ_1	0.98	0.98	0.97	0.97	0.97
υ_2	1.00	0.99	0.98	0.97	0.98
υ_3		1.00	0.99	0.99	0.99
υ_4			0.99	0.99	0.99
υ_5				0.99	0.98
υ_6					0.99





and improved by the author. The parallel algorithm was based on speculative parallelism. An interesting experimental finding was that the speculative parallelism does affect the net speedup, sometimes drastically. On the problems we have tested so far, the efficiencies range between 70% and 320%.

One could imagine a speculative sequential algorithm which *simulates* the speculative parallel algorithm. For example this sequential algorithm begins to build one stack, then after a certain time interval, it temporarily stops working on the stack, and begins to build other stack, but again after a certain time interval, it goes back to the previous stack and continues to build it, and so forth.

In this way, the sequential algorithm could also enjoy (or suffer) the benefits (or the penalty) of speculativeness. One might even argue that the speculative parallel algorithm must be compared to the imagined speculative sequential algorithm in order to be fair.

It is an elegant idea, and we will be very interested if such sequential algorithm can be efficiently realized in practice. However at this point we do not see how this can be done efficiently, mainly because of the expected overhead of context switching and that of paging due to the need of more memory, let alone the difficulty of programming.

Recall that the experimental results obtained in this paper are based on a (naively) chosen strategy. We plan to carry out more experiments in order to obtain deeper understanding of the behavior of the parallel algorithm and hope to discover better strategies. We also hope that the techniques used in this paper are applied to parallelizing other symbolic and algebraic algorithms.

I thank George Collins and Bruno Buchberger for making various suggestions on the draft of this paper. I also would like to thank my student Dan Teodosiu for implementing the underlying communication primitives.

References

[1] D. S. Arnon. Algorithms for the geometry of semi-algebraic sets. Technical Report 436, Computer Sciences Dept, Univ. of Wisconsin-Madison, 1981. Ph.D. Thesis.

[2] D. S. Arnon. A cluster-based cylindrical algebraic decomposition algorithm. *Journal of Symbolic Computation*, 5(1,2):189–212, 1988.

[3] D. S. Arnon and M. Mignotte. On mechanical quantifier elimination for elementary algebra and geometry. *Journal of Symbolic Computation*, 5(1,2):237–260, 1988.

[4] M. Ben-Or, D. Kozen, and J. H. Reif. The complexity of elementary algebra and geometry. *J. Comput. System Sci.*, 32(2):251–264, 1986.

[5] W. Böge. Decision procedures and quantifier elimination for elementary real algebra and parametric polynomial nonlinear optimization. Manuscript in preparation, 1980.

[6] B. Buchberger and H. Hong. Speeding-up quantifier elimination by Groebner bases. Technical Report 91-06.0, Research Institute for Symbolic Computation, Johannes Kepler University A-4040 Linz, Austria, 1991.

[7] J. Canny. Some algebraic and geometric computations in PSPACE. In *Proceedings of the 20th annual ACM symposium on the theory of computing*, pages 460–467, 1988.

[8] P. J. Cohen. Decision procedures for real and p-adic fields. *Comm. Pure and Applied Math.*, 22:131–151, 1969.

[9] G. E. Collins. Quantifier elimination for the elementary theory of real closed fields by cylindrical algebraic decomposition. In *Lecture Notes In Computer Science*, pages 134–183. Springer-Verlag, Berlin, 1975. Vol. 33.

[10] G. E. Collins and H. Hong. Computing coarsest squarefree bases and multiplicities. Technical Report OSU-CISRC-5/90 TR13, Computer Science Dept, The Ohio State University, 1990.

[11] G. E. Collins and H. Hong. Partial cylindrical algebraic decomposition for quantifier elimination. *Journal of Symbolic Computation*, 12(3):299–328, September 1991.

[12] G. E. Collins and R. Loos. *The SAC-2 Computer Algebra System*. Department of Computer and Information Sciences, Ohio State University.

[13] N. Fitchas, A. Galligo, and J. Morgenstern. Algorithmes repides en séquential et en parallele pour l'élimination de quantificateurs en géométrie élémentaire. Technical report, UER de Mathématiques Universite de Paris VII, 1987. To appear in: Séminaire Structures Algébriques Ordonnées.

[14] D. Yu. Grigor'ev. The complexity of deciding Tarski algebra. *Journal of Symbolic Computation*, 5(1,2):65–108, 1988.

[15] D. Yu. Grigor'ev and N. N. Vorobjov (Jr). Solving systems of polynomial inequalities in subexponential time. *Journal of Symbolic Computation*, 5(1,2):37–64, 1988.

[16] J. Heintz, M-F. Roy, and P. Solernó. On the complexity of semialgebraic sets. In *Proc. IFIP*, pages 293–298, 1989.

[17] C. Holthusen. *Vereinfachungen für Tarski's Entscheidungsverfahren der elementaren reellen Algebra*. PhD thesis, University of Heidelberg, January 1974.

[18] H. Hong. An improvement of the projection operator in cylindrical algebraic decomposition. Technical Report OSU-CISRC-12/89 TR55, Computer Science Dept, The Ohio State University, 1989.

[19] H. Hong. An improvement of the projection operator in cylindrical algebraic decomposition. In *International Symposium of Symbolic and Algebraic Computation*, pages 261–264, 1990.

[20] H. Hong. *Improvements in CAD–based Quantifier Elimination*. PhD thesis, The Ohio State University, 1990.

[21] H. Hong. Collision problems by partial CAD construction. Technical report, Research Institute for Symbolic Computation, Johannes Kepler University A-4040 Linz, Austria, January 1991.

[22] H. Hong. Comparison of several decision algorithms for the existential theory of the reals. Technical Report 91–41.0, Research Institute for Symbolic Computation, Johannes Kepler University A-4040 Linz, Austria, 1991.

[23] J. R. Johnson. *Algorithms for Polynomial Real Root Isolation*. PhD thesis, The Ohio State University, 1991. Dissertation in preparation.

[24] M. Kalkbrenner and S. Stifter. Some examples for using quantifier elimination for the collision problem. Technical Report 87-22, Research Institute for Symbolic Computation, 1987.

[25] L. Langemyr. The cylindrical algebraic decomposition algorithm and multiple algebraic extensions. In *Proc. 9th IMA Conference on the Mathematics of Surfaces*, September 1990.

[26] D. Lazard. An improved projection for cylindrical algebraic decomposition. Unpublished manuscript, 1990.

[27] R. G. K. Loos. The algorithm description language ALDES (Report). *ACM SIGSAM Bull.*, 10(1):15–39, 1976.

[28] S. McCallum. *An Improved Projection Operator for Cylindrical Algebraic Decomposition*. PhD thesis, University of Wisconsin-Madison, 1984.

[29] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals (part I). Technical Report 853, Cornell University, Ithaca, New York 14853-7501 USA, July 1989.

[30] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals (part II). Technical Report 854, Cornell University, Ithaca, New York 14853-7501 USA, July 1989.

[31] J. Renegar. On the computational complexity and geometry of the first-order theory of the reals (part III). Technical Report 856, Cornell University, Ithaca, New York 14853-7501 USA, August 1989.

[32] B. D. Saunders, H. R. Lee, and S. K. Abdali. A parallel implementation of the cylindrical algebraic decomposition algorithm. In *International Symposium of Symbolic and Algebraic Computation*, pages 298–307, 1990.

[33] A. Seidenberg. A new decision method for elementary algebra. *Ann. of Math*, 60:365–374, 1954.

[34] A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* Univ. of California Press, Berkeley, second edition, 1951.